Proposition of multi-agent conflict situation simulation and reinforcement learning toolset with demonstration of early stage implementation

R. JAROSZ

robert.jarosz@wat.edu.pl

Military University of Technology, Faculty of Cybernetics
Kaliskiego Str. 2 00-908 Warsaw, Poland

This paper introduces conceptual approach to modelling conflicts. A flexible framework compatible in development phase is presented. Model scalability, possibility of parallelization and computational distribution over network is discussed. As example of application there are presented two variants of classic game theory problems. At the end of the paper current problems are briefly stated and future work direction is presented.

Keywords: artificial intelligence, game theory, situation modeling, multi-agent conflict situations, rust.

DOI: 10.5604/01.3001.0054.6282

1. Introduction

This paper presents early stage of toolset development aimed for modeling conflict situation. Under game theory consideration sides of conflict traditionally called as players are interested to solve conflicting situations at their favor. Due to limited resources and reward pool, it is usually difficult or impossible to select a set of players' decision leading to outcome optimal for everyone. Often one's decision producing good outcome for him at the same time leads to worse outcome for another player.

Conflicting games are almost certainly impossible to avoid. They happen in almost every aspect of life and cover situations from not material stakes like quality of entertainment through games with small material stakes to finally global conflicts deciding life and prosperity of whole nations. Some of them are solved with strength or dexterity and some with intelligence, knowledge or some different trait, possibly with mix of them. There is something exciting in rivalry, it is observable in sports or in popularity of card, board and computer games.

This paper presents approach to modeling conflicting situations using game theory conceptions, computer simulation and machine learning. Game theory is mathematical study on modelling such situations with respect to made decisions and used strategies. Lately numerous games are the field of research of artificial intelligence leading to some serious optimizations and over performing human capabilities in solving these problems. Some of many interesting research is presented in: [1], [2], [3], [4], [5], [6], [7], [8].

This paper is organized as follows:

1. Brief introduction to game theory, it is concepts and definitions.
2. Collection of requirements demanded from the tool.
3. Presentation of current state of toolset.
4. Appliance of toolset for two traditional game theory problems.
5. Conclusion and indication of future directions of development.

2. Game theory model

Many game theory terms used in game theory are introduced in [9], [10], a subset of them will be used in this paper, therefore these terms are clarified below.

- Set of *players*: denoted with $\mathcal{N}$ where $|\mathcal{N}| = \boldsymbol{N}$
- Set of *game states*: set of possible game states, denoted with S. A subset of states with active player $n \in \mathcal{N}$ is denoted S_n.
- A partition of set S into family $\hat{S}$ consisting of $N + 2$ subsets:
  - $S_n \neq \emptyset$ a subset of game states when player n is in position to take an action defined for $n \in \mathcal{N}$;
  - S_e is the set of leaf nodes, representing final state of the game with no further actions are possible;
  - S_c is a set of states in which decision is made by model player *nature* or

chance – implementing randomness in the game, actions of *chance* are selected with some random distribution. Chance is rather part of game rules than actual player.

- The partition hold properties:
  - $S = S_e \cup S_1 \cup \ldots \cup S_{\overline{N}} \cup S_c$;
  - $\forall \mathrm{a}, \mathrm{b}: S_a \cap S_b = \emptyset$;
  - $\forall n \in \mathcal{N}: S_n \neq \emptyset$, otherwise, game would not involve that player making game less than N player.

When game itself does not introduce randomness and its result is deterministic by action of regular players $\mathrm{S_c} = \emptyset$. Generally, S_e could be empty for problems like infinite prisoner's dilemma. However, endless games cannot be completely simulated. Many problems are defined to guarantee to reach finished state or there is explicit time or step limit for game when game is terminated in its current state.

- Action set: a set of possible moves in the game. Set of all possible actions will be denoted $\mathcal{A}$. The set of possible actions may differ (be limited) depending on the current game state. Set of actions possible set $s \in S$ will be given by function: $A: S \rightarrow 2^{\mathcal{A}}$
- *Information set*: A non-empty subset $W_{n,k} \subseteq S_n$ of game states which are indistinguishable among each other by active player. If every information set has only one element, then player has the complete knowledge of the game – knows exactly in which state the game is in the moment of taking action. If information set has more than one element player lacks information to precisely indicate in which state the game is. Obviously: $\forall s_i, s_j \in W_{n,k}: A(s_i) = A(s_j)$
- *Payoff:* A value $r \in R$ measuring score of players in game. Usually *payoff* is simply a number, which out of the box provides properties such as:
  - total ordering of possible values;
  - ability to sum final payoff from partial (step) payoffs;
  - norm – estimation how much one payoff is better than other.

  In some specific game models *payoff* can be more complicated, for example it could be tuple of numeric values. Such models introduce additional complications, namely multi-objective optimization. Such cases demand custom defined preference model for *payoffs*, and relation of total ordering is not guaranteed. Generally, players can use different space of *payoffs.* In specific cases one player can use single numeric value and another consider several factors in tuple. Therefore, formally R_n will denote space of *payoffs* for player n.
- *Policy:* A function selecting an action (move) based on the state in which game is. Policy π_n defined on the whole state set is: $\pi_\mathrm{n}: S \rightarrow \mathcal{A} \cup \perp$. No action is returned if player n can not move in current game state. If domain is restricted to states in which player can move, then it is $\pi_\mathrm{n}: S_n \rightarrow \mathcal{A}$. A set of possible policies of player n will be denoted P_n.

3. Extensive-form games

A N-player game in extensive form is defined with:

- A set $\mathcal{N}$ of N players.
- A *game tree* – directed graph with root node representing initial state of the game. Nodes in the graph represent game states. Edges of the graph represent moves (actions) transforming game state to another. Graph is directed from root node to leaf nodes.
- Partition of tree nodes subsets: $S_c, S_1, \ldots S_N, S_e$
- For each terminal node $s_t \in S_e$ there is defined a tuple $H_t \in R_1 \times R_2 \times \ldots \times R_\mathrm{N}$, containing rewards for respective players.

4. Normal-form games

Games in normal form do not define game tree, they do not recognize separate states of the game nor moves made by players in these states. Rather than that, games in normal form operate on higher level of abstraction using player policies as primitive component. While in extensive-form game *payoffs* were based on game states, and policies used to select actions leading to game evaluation states, in normal-form games *payoffs* are defined based on policies. This is a different view, not as a player but rather from over the game, where the game and its properties are the target of research.

A *situation* is tuple of policy selection by every player. A set of possible situations is given:

$$P = P_1 \times P_2 \times \ldots \times P_\mathrm{N}$$

A *payoff* for player n is given with function:

$$H_n: P_1 \times P_2 \times \ldots \times P_\mathrm{N} \rightarrow R_n$$

And $H = (H_1, \ldots, H_N)$ is tuple of each player's reward.

The game itself is a tuple:

$$T = \langle \mathcal{N}, P, H \rangle$$

5. Conception of multi agent situation modelling tool

Presented approach aims at delivering toolset for modelling games between numerous players. The following principles are considered in tool development process:

- Support games with asymmetric players – with different set of actions and representation form of information set.
- Compatibility with game theory terms.
- Modular construction allowing elastic replacement of information set representation forms and policies.
- Scalability for growing number of modelled players.
- Possibility of building distributed simulation.
- Delivery of reinforcement learning oriented features.
- Error awareness – with maximalizing type and memory error detection during compile time and detection of errors occurring during run time.
- Implementations should be possibly generic or dynamic.
- Attention for performance with respect for scalability.
- Support arbitrarily delivered *payoffs* calculated with shared rules and custom score calculations for individual players using nonstandard scoring method.

6. Overview of proposed solution

For development of proposed toolset, the Rust language was selected. Rust is often used in backend solutions in data science, due to its performance and safety. Language supports generic types for template implementations and late-binding types for situations where run-time flexibility is needed. Type compatibility and instance lifetime is checked during compilation, therefore model execution can be effectively secured from null-pointer and method not found errors. Moreover Rust allows pattern matching against enumeration types and asserts that every pattern branch is checked, minimalizing risk of forgotten case in implementation.

Recently described elements of game theory are usually represented by types implementing respective *traits*. Implementing *trait* for type enforces implementation of methods listed in trait, creating compatibility interface for that type. Shared behavior is implemented for generic types with annotations to implement needed traits.

The workflow model consists of the environment and a number of players – called agents. Environment and agents run in separate threads. The role of environment is to be game order enforcer and to keep track of actual game state. Agents communicate with the environment informing it about selected actions or occurred errors. Environment receives action selection from agents and transforms game state to new one. After resolving agent's actions environment issues updates for selected agents. Important role for environment is also reaction to errors. Depending on implementation it could propagate errors to every agent and stop the game or try to recover from error.

Communication between agents and environment is made with generic communication medium. Different agents are not bound to use the same communication form – some can use inter thread channels and some (remotely connected) use some form of network communication.

Agents and environment work in concurrent model, not necessarily working in parallel. The environment usually waits for a message from any agent to process idling between. Agents process observations received from the environment (parallelly). When one agent is selected to play, he evaluates his next move, in the meantime other players may stay idle or could be finishing recent updates and potentially doing some heavy pre-computation for next move. Current implementation uses Rust's built in synchronous concurrency model, however for future works may lead to development of asynchronous model.

7. Domain parameters

Model implements workflow with generic types, allowing players to use different communication mediums, information set representation and policy types. However, in order to communicate with environment a set of agreed parameters are needed to be introduced for protocol purpose. Such domain parameters are represented as `struct` implementing following `trait`:

```
pub trait DomainParameters : Clone +
Debug + Send + Sync + 'static
  + Send + 'static {
    type ActionType: Action;
    type GameErrorType:
      InternalGameError;
    type UpdateType: Send + Clone;
    type AgentId: AgentIdentifier ;
    type UniversalReward: Reward ;
}
```

This trait itself does not introduce any methods, however, requires pointing in types used in communication in framework. The code snippet may be confusing for readers not familiar with Rust. The code introduces a `trait` (interface) named `DomainParameters` with requirement to also implement traits `Clone`, `Debug`, `Send`, `Sync` – mainly because further defined types must also implement these. Domain parameters include following types:

1. `ActionType` the data structure that will be sent by agent to environment. This does not necessary mean that every agent has the same set of actions, only that every possible action produced by agent will be understood by environment. For compilation level assurance it is recommended to implement it statically using `enum` structure. Another option would be using data structure that could be parsed by environment e.g. `String` containing JSON serialized action, this approach while giving more flexibility introduces possibility of desterilizing errors during applying action that cannot be prevented during compilation time. `ActionType` must implement `Action` trait, this trait does not introduce new methods, however it combines required super-traits.
2. `GameErrorType` type used in specific modelled game to represent internal error. These types of errors are caused during processing game rules and thus must be implemented alongside game rules.
3. `UpdateType` represents data that is sent by environment to agents. And analogous rule applies that it must understandable by every agent. Similarly static `enum` or `struct` type is preferred, and serialized data possible with awareness that it could potentially cause problems when agent cannot deserialize correctly during applying update to his information set.
4. `AgentId` represents identification token for player, among other typical in this context traits `AgentId` must implement `Hash` and `Eq` traits so they can be stored in `HashMap`. IDs should be unique in one game, this is required proper communication channel is selected by these IDs.
5. `UniversalReward` represent payoff type that is distributed by environment to agents. In certain use cases, agents would like to track their score during game progress, therefore this type need to be addable.

   Traits `Send` and `Sync` are traits marking types safe to send across threads respectively with transferring ownership and by reference. Usually, these traits are automatically implemented for created structs if their internal attributes are `Send` and `Sync` respectively. Data types usually implement these traits. Trait `Clone` requires method `clone()` and represents type that can be duplicated. Usually, this trait can be automatically derived via macro and similarly `Debug` trait allowing verbose text representation of structure.

8. Error handling

Errors types are organized in tree structure using enumeration types. Ideally errors should cover every incorrect situation in the model. Some errors may be caused by specific game rules – what suggest that state transition or policy could be implemented incorrectly. Other kind of errors are caused by malfunction of the model itself – wrong data serialization, early exit in one of the thread, wrong condition of communication channel. Top level categories of errors are:

1. communication misfunction,
2. violating game protocol,
3. data conversion,
4. game rule error.

Currently errors are handled by logging the event, propagation to other entities and exiting model execution. In future cases with recoverable errors are planned to be handled. For example, playing illegal action can be noted in dedicated event stash and fouling player asked to choose different action. This may be especially useful for agents learning rules of the game via reinforcement learning – episode could run forward with correct step while wrong step along with penalty can be added later to learning data set.

Proper error definition, along with verbose description of cause helps building and debugging models, therefore the list of possible errors will likely expand in the future.

9. Communication channels

Basic form of communication is channel between two structures implementing symmetrically trait `BidirectionalEndpoint`. For every player one port for communication with environment must be constructed with paired port for environment to communicate with this agent. Every endpoint has specified Outward and Inward types that determine what messages are sent and received with it. Agent will have port with inward type matching environments message and outward type matching his message to environment. Paired endpoint in environment will have these types reversed. Environment and agent messages are generic types specified with domain parameters. Environment message is `enum` with the most important variants:

1. `YourMove` – to inform player that it is his turn to move.
2. `GameOver` – to inform player that the game is over.
3. `StateUpdate(UpdateType)` – to inform player about his game state change with regard to that player perspective. Within this message there is domain's update structure aggregate.
4. `ErrorNotify(AmfiError)` – to communicate error (main error type is described in previous section). Error is aggregated with this message.

There are several other variants, but these make core of game flow. Analogously the agent's message is `enum` consisting of variants:

1. `TakeAction(ActionType)` – message sent by agent with aggregated game action.
2. `NotifyError(AmfiError)` – information about occurred error sent by agent to environment.

Included standard implementation uses Rust's standard `mpsc::channel` for efficient communication between threads. There exists experimental TCP/IP implementation using binary data serialization, and in future there are planned http-based ports (using JSON or grpc) data serialization.

Network communication allows federating players to separate physical machines increasing scaling potential. The drawback however is that data needs to be serialized by sender and desterilized by receiver, unlike as in single program communication where structures can be sent between threads (provided that they implement `Send` trait which is automatically derived for majority of types). This additional data operation introduces additional processing and combined with network latency slows down the model. Therefore, local model launch is preferred for performance purposes, if machine can run all players.

Currently the following three concepts of orchestrating communication are considered:

1. Environment is connected to every agent with separate bidirectional communication channel. The advantage of this solution is it's conceptual simplicity, as environment has set of independent channels which can be randomly accessed on demand, moreover channels can be wrapped in dynamic structures making it easier to communicate with different agents using different types of channels. The drawback of this solution is problem of selecting listening channel in environment. Environment should listen to all agents (any one can communicate some error message), this requires switching channels to listen on when idle – introducing potential performance problems. Current default switch strategy is round robin, checking if there is message queued from certain agent and switching to next if none was found.
2. Environment groups agents and use single port for listening messages from them and individual ports to send to them. This conception is optimized for local agents as it utilizes Rust's natural `mpsc::channel` (multi producer – single consumer). The advantage of this concept is performance in local environments and simplicity in implementation as long as `mpsc` channel is the only one used. The drawback is difficulty to use it alongside bidirectional channels (e.g. TCP streams) – in this case listening switching must be performed – similarly to previous conceptions with separate bidirectional channels.
3. Another conception – however not yet implemented is using listening dispatch thread. Listening dispatch would run along environment and collect messages from agents and then queuing them into single queue. Saving operational time environment would use to switch between channels. The concept will be object of future work.

10. Policy

Automatic player implementation requires implementing some policy to choose actions based on current game state (actually information set as player does not have complete

information about state in general). The trait for such type is given:

```
pub trait Policy
<DP:DomainParameters>: Send{
  type InfoSetType:
    InformationSet<DP>;

  fn select_action(&self,
      state: &Self::InfoSetType) ->
    Option <DP:: ActionType>;

}
```

Policy has one associated type, which is information set it works on and one method that given the instance of information set selects proper action as `Some(action)` or `None` where no action is available. The important detail about policy trait is that it is compatible only with declared information set type. However, this type may be set generic as long as said generic type has proper interface for policy implementation.

```
pub struct ActorCriticPolicy<
  DP: DomainParameters,
  InfoSet: InformationSet <DP> + Debug
    + ConvertToTensor <InfoSetWay>,
  InfoSetWay : WayToTensor>{
  network : A2CNet ,
  convert_way : InfoSetWay ,
  /*
  ...
  */
}
```

Such structure allows generic implementation of actor-critic policy for information sets that can be translated to tensor and `ActionType` specified in domain parameters can be constructed using integer index (actor network outputs integer that must be interpreted as action).

Ensuring that used information set can be converted to tensor could be done by requiring from it to implement `Into<Tensor>`. However, for research purpose one may implement one information set and experiment with different conversion ways to `Tensor`. Therefore, requirement is made to implement `ConvertToTensor<InfoSetWay>` where `InfoSetWay` represents a method of conversion (what data is considered, how it is structured and potentially normalized). For example, given some information set s_1 that gives player information about strict observable at the moment facts, some facts observed in the past and some synthetic probabilistic assumptions about not known state parameters. Researcher might be interested in comparing effectiveness of policy using certain observable facts with policy using additionally previous observations or even analytically processed data for example "considering enemy previous actions $a_1, a_2, \ldots$ probability that in current game state enemy has asset x is p". In this case agent is interested in developing different conversion forms for the same information set, and to feed differently shaped neural network.

11. Payoffs

Payoffs in the model represent two independent concepts:

1. Payoff provided by central environment.
2. Payoff provided by agent's information set.

First type of payoff is provided with definition of game rules. The central environment can calculate players' payoffs and communicate it to agents. It can be implemented in two approaches:

1. Play the game to one of the final states and then send whole payoffs to agents – this is intuitive for general games (when intermediate rewards are not necessarily defined).
2. Provided that in the game partial payoffs are defined it can be partially distributed during game, and the final payoff is sum of intermediate rewards [12], [13]. This is often used in reinforcement learning techniques like Q-learning [14] policy gradient methods or actor–critic based methods [15].

The subjective payoff is calculated by the agent itself based on his information set. This approach is strictly technical because different calculation method actually means the different game was played. This is used to model some irrationality of the player or the fact that he actually plays different game. An example of an application could be the following:

Let there be a board game for 3 players with strict rules on how to count points at the end of the game. Player A is not interested in winning, rather he would like see player B lose spectacularly so he does not mind playing in the way to help player C. As player's A objective is different than described in game rules, he plays a different game, the game is different also for B and C. However rather than implementing different games for each interesting irrationality it would be easier to change the agent's payoff perspective.

This self calculated payoff is represented by trait `SelfAssessment` and as stated before

I is not distributed by environment and must be calculated by agent based on current information set state. For models that do not need such evaluation it could be set to empty type and not be considered at all.

12. Using reinforcement learning to optimize policies

Reinforcement learning methods optimize action selection with respect to current state. Practically agents choose action and remember how certain actions change the final payoff after the game. Some policies, commonly known as *critic* type policies try to evaluate action by predicting how will it impact the final payoff. Popular critic policy is DQN – deep Q-learning network that learns to predict the final outcome of action given with equation:

$$Q^*(s,a) = E\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

Where s is current state (information set) and a is an evaluated action. The outcome of function is prediction of summed next instant reward (change in payoff) and discounted sum of future rewards – usually reduced with discount factor $\gamma < 1$ ensuring that algorithm converge to finish game due to inflation of payoffs. The action with the best expected payoff is chosen.

Another type of policies are *actor* type policies that selects action internally, that is given the state s alone output selected action. Usually, such policies implemented with neural networks output some categorical distribution of actions to be taken, and then the action is sampled (or the one with the greatest probability is chosen).

A hybrid methods are also possible, the popular option is *actor–critic* method introducing two neural networks – one generating actions distribution and the other one evaluating selected action. Example of such method is Actor Critic with advantage baseline (A2C).

Regardless of the choice of learning algorithm, agent must collect history of the game played, this history must contain information set (state) in which he has taken action, the taken action and data to assess taken action. In presented framework a following structure is proposed: Single episode (one game) is saved as a trajectory – a list of trace steps. Each trace step consists of:

- Information set;
- Taken action;
- Reward from environment;
- Reward based on self evaluation.

With steps ordered chronologically one can calculate future payoff by summing and discounting rewards for the next steps. Several episode trajectories are collected into batch, which is after certain number of episodes used to update policy. The library implements generic agent that collects the trajectory. Without collected trace agent cannot update policy, however it may be optimal solution for not learning agents as trajectory collection consume compute resources and memory.

In the current state of toolset two generic learning policies are implemented (DQN and A2C) with more to be in the future. The policies are generic however some constraints must be met to use them. Firstly, one must provide neural network shape, and secondarily some type conversions are needed. For both of these policies information set must be convertible to Tensor type, for Q-learning also an action type must be convertible to Tensor. For the A2C policy action must be constructed from integer index, as policy chooses action number, then this number is to be converted to action.

13. Application in prisoner dilemma

Prisoner's dilemma is well known problem in game theory. It is two-player game, in which both players simultaneously choose to defect the other one or cooperate with him. In this game choosing to defect other player pays better than cooperating in single game regardless of the other playerschoice.

Tab. 1. Prisoner dilemma payoff table

	cooperate	defect
cooperate	5 5	10 1
defect	1 10	3 3

There is a variant of this problem that repeats choice for n rounds. While in single game episode it is better for player to make defect move, in iterated problem it may be better for both players to agree on cooperation. Yet when the number of iterations is constant and known, player may decide to defect in the last round as it cannot be punished in the future rounds. As the other player may think in similar way, both can conclude that the last round will be defect on both sides and similarly think about previous round. If in the last round player is being betrayed anyway, he may choose to betray in previous round because it is now last relevant

round. Iterating this logic to backwards players may conclude that they should start moving with defect in first round. This analytic method is called reverse induction.

Practically there exist strategies that score better in iterated prisoner's dilemma. These strategies assume that the other player can be influenced to select cooperative actions and then both players could benefit from it. Such strategies feature punishment for other player's defect moves, however, to avoid remaining in defect-defect equilibrium punishing player under certain circumstances forgives other. This way the second player is taught that he can score more if he cooperates. To demonstrate this feature of the game three models have been built and simulated:

1. Two agents playing against each other, learning their policies.
2. Two agents playing against each other, with one of them learning and the other one using fixed policy to start punishing if opponent defected in last two subsequent rounds and forgive when opponent cooperated for two previous rounds.

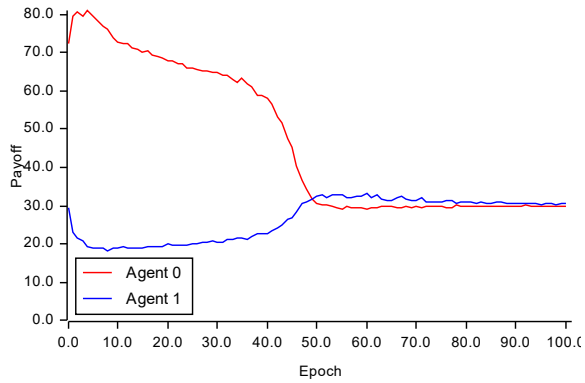

Chart 1. Two agents learning simultaneously

The experiment with two learning agent converges to Nash's equilibrium in defect–defect, and both of players scoring about 30 in 10 rounds.

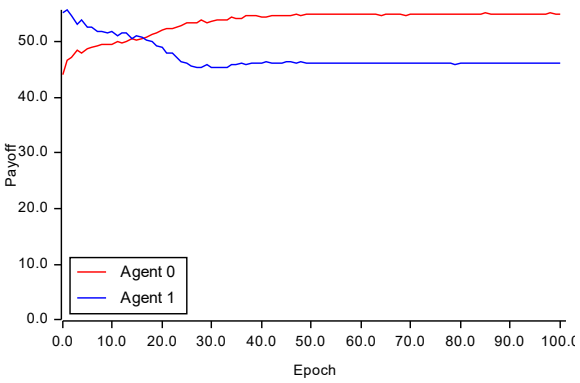

Chart 2. Learning against agent punishing for defect and waiting for two subsequent cooperations to forgive

When one agent does not learn and instead uses certain punishing strategy it is possible for model to converge to different results. On Chart 2 it may be seen that punishing agent achieves scores above 40 and learning above 50.

These models utilize classic reinforcement learning conception that reward is provided by environment. As it can be read from charts two learning agents converge to *always defect* policy. On the other hand, models with one fixed policy show that by keeping some punishing-forgiving policy it is possible achieve payoffs better for both players. This observation may lead to question if it possible for agents to achieve these strategies with learning. It may be possible that modifying the reward function and therefore optimalization target would lead to achieving better global payoff.

An attempt to construct model with second agent learning with optimizing different function is proposed. Two models have been built, in both first agent is normal learning agent using rewards distributed by environment. The second agent learns, but maximize different payoff function:

1. a combination of table payoff with count of cooperations made by first player; $\hat{r} = r + ac$
2. a combination of table payoff with heuristic assessment of punishment effectiveness. $\hat{r} = r + ah$

Unfortunately, these models did not result in second agent learning punishing strategy. Yet the experiment allowed to construct a hypothesis about necessary condition to construct punishing strategy via learning. In the proposed model reward $\hat{r} = r + ac$ is historyless, meaning it does not include historic data and therefore it can be used to construct new single round payoff table, as every time the opponent makes

cooperate action it adds to defined payoff – such defined reward leads to new Nash's equilibrium but actually does not introduce any new information to develop punishing strategy.

Tab. 2. Modified payoff table for prisoner dilemma

	cooperate	defect
cooperate	5 5+a	10 1
defect	1 10+a	3 3

The second model introduces new problem, namely, how to measure effectiveness of punishment. Intuitively punishment is successful when after enemy defected, player defects and in the subsequent round opponent switches to cooperation. This assessment can be combined with game defined rewards and produce new self-assessed payoff. Unfortunately, these models does not introduce successful assessments. Future experiments will use different punishment evaluations and hopefully punishing strategy could be achieved via learning.

14. Application in replicator dynamic problem

To demonstrate game with larger number of players the following experiment is proposed. Prepare game for N players (10, 100, etc). In the game in M rounds players are paired randomly and each pair plays chicken (hawk-dove) game with the following payoff table:

Tab. 3. Chicken game payoff table

	dove	hawk
dove	2 2	4 1
hawk	0 4	0 0

In single game every player should face several encounters with random other player to play a round of chicken game. His final payoff is sum of round payoffs. Players will use adaptive policies – learnt by repeating main game many times. Theoretically this particular game variant played in adaptive population should result in 2/3 players playing hawk and 1/3 playing dove. Learning agents use advantage actor critic method. Model can be built with only learning agents or can be populated with some not learning agents – e.g. realizing pure dove and hawk strategies. Several population models have been run. The following charts depict learning process in model constructed with 20 learning agents, 5 fixed on pure hawk strategy and 5 pure dove strategy. In this example model should converge to have average 20 hawks and 10 doves, and with 20 learning agents 5 of them should become doves and 15 hawks (ratio is 1/3).

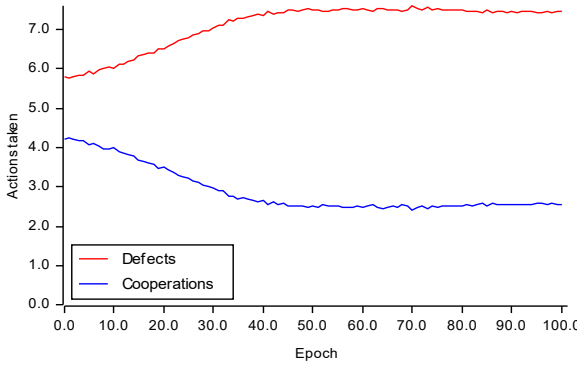

Chart 3. Evolution of taken actions by learning agents in multi-agent chicken game

As seen on the chart learning agents tend to choose dove about 2.5 times in ten rounds and 7.5 hawk. That result is coherent with expectation. The following chart represents evolution of average payoffs per group of agents.

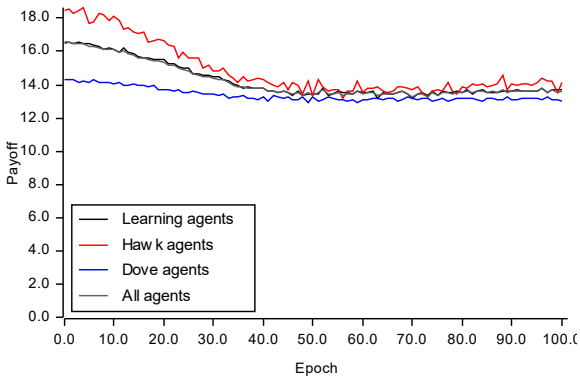

Chart 4. Evolution of payoffs in multi-agent chicken game

15. Future works

The simulation and learning framework is in early stage of development. Future works will be focused on several aspects:

- improving communication model to improve performance of models with many agents, optionally proposition of asynchronous solution to be provided, benchmarking performance of implemented communication algorithms;

- reviewing trajectory tracking structure to optionally include information about information set transition;
- implementing more learning algorithms;
- developing models depicting more complex conflicting problems;
- implementing features making model construction and data collection easier.

16. Summary

The toolset is ready to build models, which was demonstrated with classic game theory problems. Learning models utilizing classic reinforcement learning approach are supported.

Decentralization of models is not yet supported out of the box. There is *trait* system allowing generic communication channels using round robin polling for messages. However no stable network communication channels are ready.

The code of the library and examples are published under MIT license and can be viewed at [11]. The version of software in the moment is 0.1.0.

Presented models while not introducing any novelty to common knowledge are useful to test current state of work and expose some problems that might be addressed in the future.

17. Bibliography

[1] Mnih V., et al., "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, 529–533 (2015).
[2] Mnih V., et al., "Asynchronous methods for deep reinforcement learning", *Proceedings of the 33rd International Conference on Machine Learning*, 1928–1937, NY USA 2016.
[3] Lüth C., "A Review of: Human-Level Control through deep Reinforcement Learning", *Seminar Paper Artificial Intelligence for Games*, University of Heidelberg 2019.
[4] Bellemare M.G., Naddaf Y., Veness J., Bowling M., "The arcade learning environment: An evaluation platform for general agents", *J. Artif. Intell. Res.*, vol. 47, 253–279 (2013).
[5] Bellemare M.G., Veness J., Bowling M., "Investigating contingency awareness using Atari 2600 games", *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26(1), 864–871, 2012.
[6] Rong J., Qin T., An B., "Competitive bridge bidding with deep neural networks", *arXiv Prepr. arXiv1903.00900*, 2019.
[7] Gong Q., Jiang Y., Tian Y., "Simple is better: Training an end-to-end contract bridge bidding agent without human knowledge", *Real-world Sequential Decision Making, Workshop at ICML*, June 14, 2019, Long Beach, USA.
[8] Tian Y., Gong Q., Shang W., Wu Y., Zitnick C.L., "Elf: An extensive, lightweight and flexible research platform for real-time strategy games", *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017.
[9] Binmore K., *Game theory: a very short introduction*, OUP Oxford, 2007.
[10] Ameljańczyk A., *Teoria gier*, WAT, 1978.
[11] Jarosz R., "Amfiteatr", Warsaw, 2024.
[12] Sutton R.S., Barto A.G., *Reinforcement learning: An introduction*, MIT Press, 2018.
[13] Arulkumaran K., Deisenroth M.P., Brundage M., Bharath A.A., "A brief survey of deep reinforcement learning", *arXiv Prepr. arXiv1708.05866*, 2017.
[14] Gaskett C., Wettergreen D., Zelinsky A., "Q-learning in continuous state and action spaces", *Australasian Joint Conference on Artificial Intelligence*, LNAI 1747, 417–428, Springer 1999.
[15] Grondman I., Busoniu L., Lopes G.A.D., Babuska R., "A survey of actor-critic reinforcement learning: Standard and natural policy gradients", *IEEE Trans. Syst. Man, Cybern. Part C: Applications Rev.*, vol. 42, no. 6, 1291–1307, 2012.

Propozycja narzędzi do symulacji i uczenia maszynowego w sytuacjach konfliktowych z udziałem wielu stron wraz z demonstracją wczesnej wersji implementacji

R. JAROSZ

W artykule opisano koncepcję biblioteki do symulacji interakcji między graczami rozpatrującymi sytuacje konfliktowe. Opisano w nim założenia, w tym wymagania funkcjonalne dla oprogramowania. Przedstawiono proponowane rozwiązanie i wstępną wersję implementacji wraz z przykładami zastosowania koncepcji w klasycznych problemach teorii gier.

Słowa kluczowe: teoria gier, symulacja, uczenie ze wzmocnieniem, optymalizacja strategii, gry wieloosobowe.