norms-games documentation

Release 0

Nieves Montes

May 08, 2022

Contents

1 Introduction

norms-games implements a computational model of Elinor Ostrom's Institutional Analysis and Development framework *[1]*. It includes the interpreter of the Action Situation Language (ASL) and the game engine to automatically generate extensive-form games from ASL descriptions.

2 Requirements

`norms-games` requires a working installation of the following:

- Python 3
- SWI-Prolog
- The PySwip package

3 Usage

For the time being, the `norms-games` package requires to download a local copy of the source code (using `git clone`). The path to the package should then be appended to your Python path.

The `examples` directories has some illustrations on how to use the basic functions. Basically, you should create your ASL description in three distinct files:

- `agents.pl`
- `states.pl`
- `rules.pl`

Then, to construct the extensive-form game semantics of your description, it is enough to call:

```
>>> build_full_game(<path_to_ASL_description>, <identifier>, (threshold=..., max_rounds=.
↪..))
```

See the documentation for further details.

4 Contents

4.1 ngames.extensivegames

4.1.1 ngames.extensivegames.ExtensiveFormGame

class ngames.extensivegames.**ExtensiveFormGame**(***kwargs*)

Implementation of a game in extensive form.

The game is initialized 'empty', meaning with minimal attribute assignments. Attributes are then set through the various methods. The extensive form game is modelled as described in the reference, see the chapter on extensive games.

Parameters ****kwargs** – Additional keyword arguments.

game_tree
Game tree, directed graph. Other than the methods and attributes of the class, two additional attributes are set: * root (Any): The root node, initialized to None. * terminal_nodes (List[Any]): The list of terminal nodes, initialized to an empty list. The game tree is initialized as empty.

Type networkx.DiGraph

information_partition
For every player (key), it maps it to the list of the information sets (values).

Type Dict[Any, List[Set[Any]]]

is_perfect_informtion
The game is initialized as being of perfect information.

Type bool, *True*

players
List of players in the game. It is initialized empty.

Type List[Any]

probability
Probability distributions over the outgoing edges at every node where chance takes an action. The keys are the nodes where chance acts. The values are dictionaries mapping every outgoing edge from that node to its probability.

Type Dict[Any, Dict[Tuple[Any, Any], float]]

turn_function
Function that maps every non-terminal node to the player whose turn it is to take an action at the node.

Type Dict[Any, Any]

utility
For every terminal node, it maps the utility that the various players (excluding chance) assign to it.

Type Dict[Any, Dict[Any, float]]

See also:

`networkx.DiGraph`

add_edge(*from_node: Any*, *to_node: Any*, *label: Any*) → None
Add an edge to the game tree between two nodes.

Parameters

- **from_node** (*Any*) – Origin node of the edge.
- **to_node** (*Any*) – Destination node of the edge.
- **label** (*Any*) – The edge label corresponsing to the action being take.

add_information_sets(*player_id: Any*, **additional_info_sets: Set[Any]*) → None
Add an information set to the partition of the given player.

This method does not require that all nodes where `player_id` takes an actions are included in some information set. It does check that all the nodes in the information partition to be added belong to the theta partition of `player_id`, and that they have no been previously included in some other information set.

Parameters

- **player_id** (*Any*) – The game player whose information partition is to be expanded.
- ***additional_info_sets** (*Set[Any]*) – The information sets that are to be added.

add_node(*node_id: Any*, *player_turn: Optional[Any] = None*, *is_root: bool = False*) → None
Add a node the game tree.

If the node is non-terminal and it is not a chance node, perfect information is assumed. A set containing the single node is added to the information partition of the player playing at the node.

Also, if the node is non-terminal (regardless of whether it is a chance node or not), it is added to *turn_function* and its player is assigned.

Parameters

- **node_id** (*Any*) – Node to be added.
- **player_turn** (*Any, optional*) – Whose player has the turn at the node. If None is given, it is assumed that the node is terminal. The default is None.
- **is_root** (*bool, optional*) – Whether the node is the root of the game tree. The default is False.

add_players(**players_id: Any*) → None
Add a lists of players to the game, encoded in any data structure.

Parameters **players_id** (*List[Any]*) – Players to be added to the game. Exclude 'chance'.

Raises **ValueError** – If 'chance' is among the players to be added.

get_action_sequence(*terminal_node: Any*) → Tuple[List[Tuple[str, str]], float]
Get the sequence of actions and probability to a terminal node.

Parameters **terminal_node** (*Any*) – The terminal node to get the sequence of actions from the root.

Returns

- **action_sequence** (*List[Tuple[str,str]]*) – The sequence of action from the root to the terminal node, as a list of tuples of (player, action).
- **probability** (*float*) – The probability of the sequence of actions.

get_available_actions(*node: Any*) → Set[Any]
Get what actions are available at the given node.

Parameters **node** (*Any*) –

Returns Set of available actions according to the game tree.

Return type Set[Any]

get_choice_set(*player_id: Any*, *information_set: Set[Any]*) → Set[Any]
Get the choice set for some player at some information set.

Parameters

- **player_id** (*Any*) –
- **information_set** (*Set[Any]*) – The information set for which the choice set is to be retrieved.

Returns List of edges outgoing from every node in the information set.

Return type List[Tuple[Any]]

get_nonterminal_nodes() → List[Any]
Obtain the list of non-terminal nodes in the game tree.

Returns List of non-terminal nodes.

Return type List[Any]

get_player_utility(*player_id: Any*) → Dict[Any, float]
Return the utility function for the given player.

Parameters **player_id** (*Any*) –

Returns A map from every terminal node to the utility assigned to it by the given player.

Return type Dict[Any, float]

get_theta_partition() → Dict[Any, Set[Any]]
Get the turns partition.

The turns partition (or Θ partition) splits the non-terminal nodes into disjunct sets, according to whose turn it is to play at the node (including the 'chance' player).

Returns For every player in the game, including 'chance', the set of nodes where it is that player's turn to play.

Return type Dict[Any, Set[Any]]

get_utility_table() → pandas.core.frame.DataFrame
Get a pandas dataframe with the utility for every player.

Returns **utility_table**

Return type pandas.DataFrame

set_information_partition(*player_id: Any*, **partition: Set[Any]*) → None
Set the information partition of the given player.

It is only useful to call this method when modeling games with imperfect information, otherwise when nodes are added to the game tree perfect information is assumed by default.

The method checks that all the nodes where it is the player's turn to move are included in the information partition, and viceversa, that at all the nodes in the various information sets it is the player's turn. Also, it checks that all the nodes in any given information set have the same number of outgoing edges, and that they are non-terminal.

Parameters

- **player_id** (*Any*) –
- **partition** (*Set[Any]*) – Information sets making up the player's information partition.

Raises **AssertionError** – If the union of information sets does not correspon to the same nodes where it is the player's turn to play, or If some nodes in the same information set have different amounts of outgoing edges, or If some node is terminal.

Notes

Please note that the method does not check that all the information sets provided are disjunct.

set_node_player(*node_id: Any*, *player_turn: Any*) → None
Set the player at a node after it has been added to the game tree.

If the node had been designated as a terminal, remove it from that list.

Parameters

- **node_id** (*Any*) – The node whose player changes.
- **player_turn** (*Any*) – The new player that takes turn at the node.

set_probability_distribution(*node_id: Any*, *prob_dist: Dict[Tuple[Any], float]*) → None
Set the probabilities over the outgoing edges of a chance node.

Parameters

- **node_id** (*Any*) – Node over whose outgoing edges the probability is given.
- **prob_dist** (*Dict[Tuple[Any], float]*) – Probability distribution over the outgoing edges of the node.

Raises

- **ValueError** – If at the given node, it is not chance's turn, or if one of the provided edges does not have the given node as origin, or if there is some edge going out from the node for which the probability is not specified.
- **AssertionError** – If the sum of the probabilities over all the edges is not close to unity with $10-^3$ absolute tolerance.

set_uniform_probability_distribution(*node_id: Any*) → None
Set a equal probabilities over the outgoing edges of a chance node.

Parameters **node_id** (*Any*) – A node where chance takes its turn.

set_utility(*node_id: Any*, *utilities: Dict[Any, float]*) → None
Set the utility for all players at the given terminal node.

Parameters

- **node_id** (*Any*) – A terminal node.
- **utilities** (*Dict[Any, float]*) – Dictionary that maps every player in the game to the utility it assigns to the terminal node.

4.1.2 ngames.extensivegames.plot_game

class ngames.extensivegames.**plot_game**(*game:* ngames.extensivegames.ExtensiveFormGame, *player_colors: Dict[Any, str]*, *utility_label_shift: float = 0.03*, *fig_kwargs: Optional[Dict[str, Any]] = None*, *node_kwargs: Optional[Dict[str, Any]] = None*, *edge_kwargs: Optional[Dict[str, Any]] = None*, *edge_labels_kwargs: Optional[Dict[str, Any]] = None*, *patch_kwargs: Optional[Dict[str, Any]] = None*, *legend_kwargs: Optional[Dict[str, Any]] = None*, *draw_utility: bool = True*, *decimals: int = 1*, *utility_label_kwargs: Optional[Dict[str, Any]] = None*, *info_sets_kwargs: Optional[Dict[str, Any]] = None*)

Make a figure of the game tree.

Encoded information:

- Node colors encode the turn function at every node.
- Dashed archs between nodes indicate information sets.
- Numbers in parenthesis below terminal nodes indicate utilities (optional).

Parameters

- **game** (ExtensiveFormGame) – A game in extensive form to be plotted.

- **player_colors** (*Dict[Any, str]*) – Dictionary mapping every player in the game to the color to use for the nodes where it is the player's turn. Color white is not recommended, as it is reserved for chance nodes.
- **utility_label_shift** (*float, optional*) – To adjust the utility labels under the terminal nodes. The default is 0.03.
- **fig_kwargs** (*Dict[str, Any], optional*) – Additional keywork arguments related to the rendering of the figure - they are passed to *matplotlib.pyplot.subplots*. The default is None.
- **node_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the game tree nodes - they are passed to *nx.draw_network*. The default is None.
- **edge_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the game tree edges - they are passed to *nx.draw_network*. The default is None.
- **edge_labels_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the edge labels - they are passed to *nx.draw_network_edge_labels*. The default is None.
- **patch_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the legend patches - they are passed to *matplotlib.patches.Patch*. The default is None.
- **legend_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the legend - they are passed to *matplotlib.axes.Axes.legend*. The default is None.
- **draw_utility** (*bool, optional*) – Whether labels should be drawn below the terminal nodes displaying the utilities for all players. The default is True.
- **decimals** (*int, optional*) – The number of decimals places for the utility labels. The default is 1.
- **utility_label_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the utility labels at the terminal nodes - they are passed to *matplotlib.pyplot.text*. The default is None.
- **info_sets_kwargs** (*Dict[str, Any], optional*) – Additional keyword arguments related to the rendering of the archs connecting the information sets - they are passed to *matplotlib.patches.Arch*. The default is None.

Returns **fig**

Return type matplotlib.figure.Figure

4.1.3 ngames.extensivegames.backward_induction

class ngames.extensivegames.**backward_induction**(*game:* ngames.extensivegames.ExtensiveFormGame, *h: Any*, *u_dict: Dict[Any, Dict[Any, float]] = {}*)

Compute the value of node *h* in a subgame by backward induction.

It computes the values of all the nodes in the subgame having *h* as its root node. Only for games with perfect information.

Parameters

- **game** (ExtensiveFormGame) – A game in extensive forms. Must be a perfect information game without any stochastic effects (no *chance* nodes).
- **h** (*Any*) – The root of the subgame where to start computing.
- **u_dict** (*Dict[Any, Dict[Any, float]], optional*) – A dictionary of the values for every player at the nodes that have already been revisited. The default is {}, and it should not be modified. It is necessary to perform the recursion.

Returns A dictionary mapping, for each visited node (all the descendants of *h*), the value assigned to it by every player in the game.

Return type Dict[Any, Dict[Any, float]]

4.1.4 ngames.extensivegames.subgame_perfect_equilibrium

class ngames.extensivegames.**subgame_perfect_equilibrium**(*game: ngames.extensivegames.ExtensiveFormGame*)

Find a subgame perfect equilibrium in pure strategies.

Parameters **game** (ExtensiveFormGame) – The game in extensive form.

Returns **SPE** – A subgame perfect equilibrium in pure strategies, mapping each nodes to the action to be taken.

Return type Dict[Any, Any]

4.1.5 ngames.extensivegames.DFS_equilibria_paths

class ngames.extensivegames.**DFS_equilibria_paths**(*game: ngames.extensivegames.ExtensiveFormGame*, *h: Any*, *pure_strategy: Dict[Any, Any]*, *path: List[Any]*, *probability: float*, *path_store: List[Tuple[List[Any], float]]*)

Find all the equilibrium paths.

This function finds all of the paths given the deterministic strategy and considering the chance nodes, and stores (path, probability) tuples in a *store*.

Parameters

- **game** (ExtensiveFormGame) – The game being played.
- **h** (*Any*) – The node where play starts.
- **pure_strategy** (*Dict[Any, Any]*) – A dictionary mapping every decision node to the (edge, action) pair to be followed.
- **path** (*List[Any]*) – The path played before reaching the current node.
- **probability** (*float*) – The probability of playing the path played before reaching the current node.
- **path_store** (*List[Tuple[List[Any], float]]*) – A store where the computed paths are stores alongside with their probabilities of being played.

Examples

The intended way to call this function is: >>> path_store = [] >>> DFS_equilibria_paths(game, game.game_tree.root, pure_strat, [], 1, path_store) >>> print(path_store)

4.2 ngames.normalgames

4.2.1 ngames.normalgames.NormalFormGame

class ngames.normalgames.**NormalFormGame**(*players: List[Any]*, *actions: List[Tuple[Any, ...]]*, *payoff_function: Dict[Tuple[Any, ...], Tuple[float, ...]]*, ***kwargs*)

A general, n-player general-sum normal-form game.

A finite, n-person normal form game is a tuple (N, A, u)**, where:**

- $N = \{1, 2, ..., n\}$ is the set of players.
- $A = A_1 \times A_2 \times ... \times A_n$ is the set of joint actions, where A_i is the set of actions available to player i.
- $u = (u_1, u_2, ..., u_n)$, where $u_i : A \to R$ are real-valued payoff functions.

Parameters

- **players** (*List[Any]*) – The list of players that participate in the game.
- **actions** (*List[Tuple[Any, ...]]*) – A list of tuples. Each tuple corresponds to the domain of actions (or 'pure' strategies) available to each player in order, *i.e.* actions for player 1, actions for player 2, etc.
- **payoff_function** (*Dict[Tuple[Any, ...], Tuple[float, ...]]*) – A dictionary mapping joint actions to player's payoffs. The dictionary format is: tuple of joint actions (key) : tuple of players' float value payoffs (value).
- ****kwargs** – Arbitrary keyword arguments.

action_to_index
A dictionary of dictionaries. For every player, it stores a dictionary mapping the player's domain of actions to integers.

Type Dict[int, Dict[Any, int]]

num_players
The number of players in the game.

Type int

payoffs
Dictionary mapping every player to their payoff array. The indices of the matrix correspond are stored in *action_to_index*.

Type Dict[int, numpy.ndarray]

player_actions
A dictionary mapping every player to the list of actions she has available.

Type Dict[int, List[Any]]

****kwargs**
Arbitrary keyword arguments.

Raises `ValueError` – If the number of action options does not equal the number of players. If there are outcomes for whom no utility has been provided. If there is some outcome for which the number of payoff elements does not match the number of players in the game.

`actions_to_indices`(**args*) → Tuple[int, ...]
Map a list of joint actions to a list of integers.

Parameters `*args` – Joint actions taken by the players.

Returns **indices** – Tuple of integer indices corresponding to the joint actions.

Return type Tuple[int, . . .]

Raises `ValueError` – If the number of actions does not match the number of players in the game.

`all_outcomes`() → Set[Tuple[Any, ...]]
Generate all the possible outcomes.

Returns **all_outcomes** – A set of all the outcomes, as tuples of player's actions.

Return type Set[Tuple[Any, . . .]]

`completely_random_strat_vector`() → List[float]
Compute the vector of completely random strategies.

Build the list that correponds to the vector of completely random strategies. To be used as the initial guess in an optimization procedure.

Returns The vector of completely random strategies.

Return type List[float]

`incentive_target_function`(*mixed_strategy: Dict[Any, Dict[Any, float]]*) → float
Compute the target function to minimize the incentive to swicth.

Compute the target function that is to be minimised when all players do not have an incentive to switch from the equilibrium mixed strategy.

$$f(s) = \sum_{i \in G} \sum_{j \in A_j} (d_i^j(s))^2$$

Parameters `mixed_strategy` (*`Dict[Any, Dict[Any, float]]`*) – The mixed strategy for which the total switching incentive is being computed.

Returns $f(s)$.

Return type float

References

Shohan, Y., & Leyton-Browm, K. (2009). Computing Solution Concepts of Normal-Form Games. In Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations (pp. 87–112). Cambridge University Press.

`is_zero_sum`() → bool
Check whether the game is zero-sum.

A normal form game is *zero sum* if the payoff functions fulfill the following property: .. math:: sumlimits_{i} u_i(a_1,...,a_n) = 0quadforall iin N,forall a_1,...,a_n in A

Returns Is the game zero-sum(*True*) or not (*False*).

Return type bool

make_linear_constraints() → scipy.optimize._constraints.LinearConstraint
Build the linear constraints for a joint strategies vector.

In a vector that encodes a joint strategy profile, the components that make up the mixed srtategies of any individual must add up to one. This method encodes this requirement.

Return type scipy.optimize.LinearConstraint

make_strat_bounds() → scipy.optimize._constraints.Bounds
Build the bounds [0,1] for a joint strategy profile vector.

In any vector encoding a joint strategy, all components must be between 0 and 1.

Return type scipy.optimize.Bounds

mixed_strategies_rewards(**args*) → Tuple[float, ...]
Return the expected rewards from a mixed strategy profile.

$$u_i(s) = \sum_{a \in A} u_i(a) \prod_{j \in N} s_j(a_j)$$

Parameters **args** – Players' mixed strategies, as dictionaries mapping actions to probabilities. For every mixed strategy, it is checked that it is a valid probability distribution.

mixed_strategy_support(*player: int*, *mixed_strategy: Dict[Any, float]*) → Set[Any]
Get the support set of a given player's mixed strategy.

Parameters

- **player** (*int*) – Who is playing the mixed strategy. Takes values $1, 2, ..., n$
- **mixed_strategy** (*Dict[Any, float]*) – Probability distribution.

Returns The set of actions with non-zero probability in the mixed strategy.

Return type Set[Any]

num_outcomes() → int
Compute how many different outcomes are possible in this game.

Returns The number of possible outcomes.

Return type int

pure_strategies_rewards(**args*) → Tuple[float, ...]
Get the rewards for every player when they play a pure strategy.

Parameters ***args** – Joint actions taken by the players.

Returns **round_rewards** – Rewards obtained by the players.

Return type Tuple[float, . . .]

strategies_vec2dic(*vector: List[float]*) → Dict[Any, Dict[Any, float]]
Turn a vector representing a strategy into dictionary format.

Includes checks that the input vector has the correct size, and that the mixed strategies encoded in the vector are proper probability distributions.

Parameters **vector** (*List[float]*) – A joint strategy encoded as a vector.

Returns The dictionary format of the input strategy vector.

Return type Dict[Any, Dict[Any, float]]

switch_incentive(*player: Any*, *action: Any*, *mixed_strategy: Dict[Any, Dict[Any, float]]*) → float

Compute the incentive for a player to switch to a pure strategy.

Given a general mixed strategy profile, compute the incentive of `player` to switch to a pure strategy where `action` is played. It corresponds to the formula:

$$c_i^j(s) = u_i(a_j^i, s_{-j}) - u_i(s)$$
$$d_i^j(s) = max(c_i^j(s), 0)$$

where the subindex corresponds to the :math`i`-th player and the superindex corresponds to its :math`j`-th action.

Parameters

- **player** (*Any*) – The player for whom the incentive is computed.
- **action** (*Any*) – The action that the player is tempted of swictching to as a pure strategy.
- **mixed_strategy** (*Dict[Any, Dict[Any, float]]*) – The original mixed strategy for all players.

Returns $d_i^j(s)$.

Return type float

References

Shohan, Y., & Leyton-Browm, K. (2009). Computing Solution Concepts of Normal-Form Games. In Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations (pp. 87–112). Cambridge University Press.

4.3 ngames.build

4.3.1 ngames.build.build_game_round

class ngames.build.**build_game_round**(*identifier: str*, *threshold: int*, *root_state_facts: List[str]*, *expand_node: int*, *node_counter: int*, *player_order: List[str]*)

Build a round of the game (i.e. all possible state transitions).

Build a round of the game, i.e. a restricted form of an extensive-form game, that models all the ways by which a pre-transition state might evolve given the joint actions that agents perform.

Parameters

- **identifier** (*str*) – Identifier for the action situation rules that we want to include when building the game.
- **threshold** (*int*) – The rules with priority over this threshold are not considered.
- **root_state_facts** (*List[str]*) – The fluents that hold true at the single pre-transition state.
- **expand_node** (*int*) – The integer for the node that is being expanded, that is the root of the game round.
- **node_counter** (*int*) – To ensure that node numbering in the game round does not clash with that of the general game.
- **player_order** (*List[str]*) – An order of the players to always add player's information sets in the same order.

Returns

- *ngames.evaluation.ExtensiveFormGame* – The game round.
- *Dict[int, bool]* – A dictionary mapping the terminal nodes of the game tree to whether the termination conditions are met at them.
- *int* – The updated node counter.

4.3.2 ngames.build.build_full_game

class ngames.build.**build_full_game**(*folder: str*, *identifier: str*, *threshold: int = 1000*, *max_rounds: int = 10*)

Build a complete game tree from an action situation Prolog description.

This function takes in a complete Prolog description of an action situation according to the if-then-where rules and builds the complete Extensive Form Game that models the situation. It expands the game tree in a breadth-first fashion.

Additional attributes of the game not included in the original implementation are added to the returned game: * roles: a dictionary mapping the participants to the list of roles they assume. * node_rounds: a dictionary mapping every non-intermediate node to the number of rounds performed to get there. * state_fluents: a dictionary mapping every non-intermediate node to the predicates that hold true in that node.

Parameters

- **folder** (*str*) – Folder where the agents.pl, states.pl and rules.pl files are placed.
- **identifier** (*str*) – Identifier for the action situation rules that we want to include when building the game.
- **threshold** (*int, optional*) – if-then-where rules, of any type, whose priority exceeds the threshold are not considered while building the game. Default is 1000.
- **max_rounds** (*int, optional*) – The maximum number of rounds to perform during the game tree expansion. The default is 10.

Returns The resulting Extensive Form Game corresponding to the action situation.

Return type ngames.evaluation.ExtensiveFormGame

4.4 ngames.equilibrium

4.4.1 ngames.equilibrium.build_subgame

class ngames.equilibrium.**build_subgame**(*extensive_game:* ngames.extensivegames.ExtensiveFormGame, *root: int*)

Build a normal game that emanates from a node in an extensive game.

Assumes that the root from which the normal form game is built corresponds to the root of the last round of the extensive game.

Parameters

- **extensive_game** (ExtensiveFormGame) –
- **root** (*int*) – The root from which a last rounds of the extensive game starts.

Raises

- **AssertionError** – If a descendant of a chance node is not terminal.
- **ValueError** – If a path of play does not end at a chance or at a terminal node.

Returns **normal_form_game** – The game in normal form corresponding that starts at the input root node.

Return type *NormalFormGame*

4.4.2 ngames.equilibrium.scalar_function

class ngames.equilibrium.**scalar_function**(*x: List[float]*)

Function to be minimized: total deviation incentives for all players.

Written in a format such that the only input is a mixed strategy encoded as a numpy.array.

Parameters **x** (*List[float]*) – Array encoding a mixed strategy. The game to which it is to be applied is set as an attribute of the function externally

Raises **AttributeError** – If, at the time of being called, no normal-form game has been set as an attribute, to which the mixed strategy array encoded in the array is passed to compute the incentives to deviate.

Returns

f – The function to be minimixed, i.e. the total incentive to deviate from the mixed strategy:

$$f(s) = \sum_{i \in G} \sum_{j \in A_j} (d_i^j(s))^2$$

Return type float

References

Shohan, Y., & Leyton-Browm, K. (2009). Computing Solution Concepts of Normal-Form Games. In Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations (pp. 87–112). Cambridge University Press.

4.4.3 ngames.equilibrium.minimize_incentives

class ngames.equilibrium.**minimize_incentives**(*normal_game:* ngames.normalgames.NormalFormGame)

Compute the mixed strategy that minimizes the incentive to deviate.

Given an arbitrary normal-form game, compute the (in general, mixed) strategy profile that minimizes the incentives to deviate from all players. The target function being minimized is:

$$f(s) = \sum_{i \in G} \sum_{j \in A_j} (d_i^j(s))^2$$
$$c_i^j(s) = u_i(a_j^i, s_{-j}) - u_i(s)$$
$$d_i^j(s) = max(c_i^j(s), 0)$$

It uses the scipy.optimize library to solve the optimization problem. In particular, their implementation of the Trust-Region Constrained algorithm.

Parameters **normal_game** (NormalFormGame) –

Raises **ValueError** – If the optimization (i.e. the call to `scipy.optimize.minimize`) is not successfull.

Returns

- *Dict[Any, Dict[Any, float]]* – The mixed strategy that minimizes the total incentives to deviate, as a dictionary mapping every player in the game to a mixed strategy.
- *float* – The target function at the solution.

4.4.4 ngames.equilibrium.subgame_perfect_equilibrium

class ngames.equilibrium.**subgame_perfect_equilibrium**(*extensive_form_game:* ngames.extensivegames.ExtensiveFormGame, *equilibrium_function: Callable*)

Compute the sequential equilibriums in an Extensive Form Game.

This function works on an Extensive Form Game built as a sequence of, possibly different, Normal Form Games. This function identifies the subgames starting from those closer to the end of the game (i.e. to the root nodes), and passes them to `build_subgame` to build that part of the game tree as a normal form game. Then, it calls the provided equilibrium function to find the mixed strategy incentivized in that subgame, and backtracks the resulting utility up the game tree.

Parameters

- **extensive_form_game** (ExtensiveFormGame) –
- **equilibrium_function** (*Callable*) – The function that computes a solution concept on a (single-rounds) normal form game. It should return the result as a dictionary with the game players as keys, and a dictionary mapping their available actions to the probabilities as values.

Returns

- **subgame_mixed_strategies** (*Dict[Any, Any]*) – The mapping from the nodes that are the roots of the subgames to the mixed equilibrium strategies computed for their subgame.
- **backtrack_utilities** (*Dict[Any, Any]*) – The mapping from the nodes that are the roots of the subgames to the utilities resulting from the computed equilibrium strategies.
- **target_function** (*Dict[Any, Any]*) – Mapping from the nodes that are roots of the sequential normal-form games to the optimized deviation incentive function, which should ideally be ~0 for all nodes.

4.4.5 ngames.equilibrium.outcome_probability

class ngames.equilibrium.**outcome_probability**(*extensive_game:* ngames.extensivegames.ExtensiveFormGame, *rounds_strat: Dict[Any, Any]*, *outcome_node: Any*)

Compute the probability of reaching a terminal node.

Compute the probability that a terminal node is reached given the strategies of players at the consecutive game rounds.

Parameters

- **extensive_game** (ExtensiveFormGame) – The game in extensive form where the path and probability of play is to be computed.
- **rounds_strat** (*Dict[Any, Any]*) – The mixed strategies at every round of the game, equivalent to a normal form game.
- **outcome_node** (*Any*) – The terminal node towards which the path of play is to be computed.

Return type float

genindex

5 References

[1] Ostrom, E. (2005). Understanding Institutional Diversity. Princeton University Press.

A full-length paper outlining the features of this computational model is currently under review. In the mean time, a pre-print is available here.

Index

O

P

S

T

U