ASMACAG Documentation

Contents

Namespace `ASMACAG`

Sub-modules

- ASMACAG.Game
- ASMACAG.Heuristics
- ASMACAG.Players
- ASMACAG.play_game
- ASMACAG.play_n_games

Module `ASMACAG.Game`

Module containing the logic to play an ASMACAG Game and the classes needed for it.

Sub-modules

- ASMACAG.Game.Action
- ASMACAG.Game.Card
- ASMACAG.Game.CardCollection
- ASMACAG.Game.CardType
- ASMACAG.Game.Game
- ASMACAG.Game.GameParameters
- ASMACAG.Game.GameState
- ASMACAG.Game.Observation
- ASMACAG.Game.Rules

Module `ASMACAG.Game.Action`

An Action describes the Card played and on what Card it has been played.

Classes

Class `Action`

```
class Action(
    played_card: ASMACAG.Game.Card.Card,
    board_card: ASMACAG.Game.Card.Card = None
)
```

An Action describes the Card played and on what Card it has been played.

Methods

Method `clone`

```
def clone(
    self
) -> ASMACAG.Game.Action.Action
```

Creates a deep copy of the Action and returns it.

Method `copy_into`

```
def copy_into(
    self,
    other: Action
) -> None
```

Deep copies the Action contents into another one.

Method `get_board_card`

```
def get_board_card(
    self
) -> ASMACAG.Game.Card.Card
```

Returns the Card on which the Action.get_played_card() has been played (if the Card.get_type() of the Action.get_played_card() is CardType.NUMBER).

Method `get_played_card`

```
def get_played_card(
    self
) -> ASMACAG.Game.Card.Card
```

Returns the Card played.

Module `ASMACAG.Game.Card`

A Card has a CardType. It also has a number if it is a CardType.NUMBER.

Classes

Class `Card`

```
class Card(
    card_type: ASMACAG.Game.CardType.CardType,
    number: int = None
)
```

A Card has a CardType. It also has a number if it is a CardType.NUMBER.

Methods

Method `clone`

```
def clone(
    self
) -> ASMACAG.Game.Card.Card
```

Creates a copy of the Card and returns it.

Method `copy_into`

```
def copy_into(
    self,
    other: Card
) -> None
```

Copies the Card contents into another one.

Method `get_number`

```
def get_number(
    self
) -> int
```

Returns the number of the Card (if Card.get_type() is CardType.NUMBER).

Method `get_type`

```
def get_type(
    self
) -> ASMACAG.Game.CardType.CardType
```

Returns the type of the Card as a CardType.

Module `ASMACAG.Game.CardCollection`

An ordered collection of Card that can be used to define a deck, hand, table...

Classes

Class `CardCollection`

```
class CardCollection
```

An ordered collection of Card that can be used to define a deck, hand, table...

Methods

Method `add_card`

```
def add_card(
    self,
    card: ASMACAG.Game.Card.Card
    ) -> None
```

Adds a Card to the CardCollection.

Method **`add_cards`**

```
    def add_cards(
        self,
        cards: Iterable[ASMACAG.Game.Card.Card]
    ) -> None
```

Adds any iterable collection of Card to the CardCollection.

Method **`clear`**

```
    def clear(
        self
    ) -> None
```

Empties the CardCollection.

Method **`clone`**

```
    def clone(
        self
    ) -> ASMACAG.Game.CardCollection.CardCollection
```

Creates a deep copy of the CardCollection and returns it.

Method **`copy_into`**

```
    def copy_into(
        self,
        other: CardCollection
    ) -> None
```

Deep copies the CardCollection contents into another one.

Method **`draw`**

```
    def draw(
        self
    ) -> ASMACAG.Game.Card.Card
```

Removes and returns the first Card from the CardCollection.

Method `get_card`

```
def get_card(
    self,
    index: int
) -> ASMACAG.Game.Card.Card
```

Returns the Card contained in the CardCollection at the specified index.

Method `get_cards`

```
def get_cards(
    self
) -> list[ASMACAG.Game.Card.Card]
```

Returns the ordered list of Card contained in the CardCollection.

Method `get_empty`

```
def get_empty(
    self
) -> bool
```

Returns a bool stating whether the CardCollection is empty.

Method `remove`

```
def remove(
    self,
    card: ASMACAG.Game.Card.Card
)
```

Removes the fist occurrence of the specified Card from the CardCollection.

Method `shuffle`

```
def shuffle(
    self
) -> None
```

Shuffles the CardCollection.

Module `ASMACAG.Game.CardType`

enum that describes the different types of Card.

Classes

Class `CardType`

```
class CardType(
    value,
    names=None,
    *,
    module=None,
    qualname=None,
    type=None,
    start=1
)
```

enum that describes the different types of Card.

Ancestors (in MRO)

- enum.Enum

Class variables

Variable `DIV2` A Card that divides the resulting score of using the next Action by 2.

Variable `MULT2` A Card that multiplies the resulting score of using the next Action by 2.

Variable `NUMBER` A Card that contains a number.

Module `ASMACAG.Game.Game`

Contains the logic for playing the ASMACAG game with certain GameParameters according to the rules defined in the ForwardModel they contain.

Classes

Class `Game`

```
class Game(
    parameters: ASMACAG.Game.GameParameters.GameParameters
)
```

Contains the logic for playing the ASMACAG game with certain GameParameters according to the rules defined in the ForwardModel they contain.

Methods

Method `get_winner`

```
def get_winner(
    self
) -> int
```

Returns the index of the Player that is winning the Game.

Method `play_turn`

```
def play_turn(
    self,
    player: ASMACAG.Players.Player.Player,
    budget: float,
    verbose: bool,
    enforce_time: bool
) -> ASMACAG.Game.Action.Action
```

Performs a Player turn.

Method `random_action`

```
def random_action(
    self,
    observation: ASMACAG.Game.Observation.Observation
) -> ASMACAG.Game.Action.Action
```

Returns a random valid Action for the state defined in the given Observation.

Method `reset`

```
def reset(
    self
) -> None
```

Resets the GameState so that is ready for a new Game.

Method **run**

```
def run(
    self,
    player_0: ASMACAG.Players.Player.Player,
    player_1: ASMACAG.Players.Player.Player,
    budget: float,
    verbose: bool,
    enforce_time: bool
)
```

Runs an ASMACAG Game.

Method **set_save_file**

```
def set_save_file(
    self,
    filename: Optional[str]
) -> None
```

Sets the file that the Game is saved to.

Method **think**

```
def think(
    self,
    player: ASMACAG.Players.Player.Player,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Requires the Player to decide, given an Observation, what Action to play and returns it.

Module **ASMACAG.Game.GameParameters**

Contains the parameters for a Game. Note that these are assumed to be static and therefore are always shallow copied. Do not modify them after instatiating.

Classes

Class **GameParameters**

```
class GameParameters(
    amount_cards_on_hand=9,
    amount_cards_on_board=20,
    amount_action_points=3,
    min_number=1,
    max_number=6,
    amount_cards_limit_number=5,
    amount_cards_normal_number=8,
    amount_cards_mult2=6,
    amount_cards_div2=6,
    seed=None,
    randomise_hidden_info=True,
    forward_model: ASMACAG.Game.Rules.ForwardModel.ForwardModel = <Game.Rules.SimpleFor
)
```

Contains the parameters for a Game. Note that these are assumed to be static and therefore are always shallow copied. Do not modify them after instatiating.

Module `ASMACAG.Game.GameState`

Contains the state of a Game.

Classes

Class `GameState`

```
    class GameState(
        game_parameters: ASMACAG.Game.GameParameters.GameParameters
    )
```

Contains the state of a Game.

Methods

Method `get_observation`

```
    def get_observation(
        self
    ) -> ASMACAG.Game.Observation.Observation
```

Gets a Observation representing this GameState with its non-observable parts randomised.

Method `reset`

```
def reset(
    self
) -> None
```

Resets and sets up the GameState so that is ready for a new Game. Must be called by Game.run().

Module `ASMACAG.Game.Observation`

A GameState view for a particular Player where the non-observable parts have been randomized.

Classes

Class `Observation`

```
class Observation(
    game_state: ASMACAG.Game.GameState.GameState,
    randomise_hidden_info: bool = True
)
```

A GameState view for a particular Player where the non-observable parts have been randomized.

Methods

Method `clone`

```
def clone(
    self
) -> ASMACAG.Game.Observation.Observation
```

Creates a deep copy of the Observation and returns it.

Method `copy_into`

```
def copy_into(
    self,
    other: Observation
) -> None
```

Deep copies the Observation contents into another one.

Method **`get_actions`**

```
def get_actions(
    self
) -> list[ASMACAG.Game.Action.Action]
```

Gets a list of the currently possible Action.

Method **`get_random_action`**

```
def get_random_action(
    self
) -> ASMACAG.Game.Action.Action
```

Gets a random Action that is currently valid.

Method **`is_action_valid`**

```
def is_action_valid(
    self,
    action: ASMACAG.Game.Action.Action
) -> bool
```

Checks if the given Action is currently valid.

Method **`randomise`**

```
def randomise(
    self
) -> None
```

Randomises the Observation to get a new possible state of the Game.

Module `ASMACAG.Game.Rules`

Module containing the rules to play an ASMACAG Game, each rule set is a class inheriting from ForwardModel.

Sub-modules

- ASMACAG.Game.Rules.ForwardModel
- ASMACAG.Game.Rules.SimpleForwardModel

Module `ASMACAG.Game.Rules.ForwardModel`

Abstract base class that defines the rules of a Game.

Classes

Class ForwardModel

```
class ForwardModel
```

Abstract base class that defines the rules of a Game.

Ancestors (in MRO)

- abc.ABC

Methods

Method is_terminal

```
def is_terminal(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
) -> bool
```

Tests a GameState or Observation against a finish condition and returns whether it has finished.

Method is_turn_finished

```
def is_turn_finished(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
) -> bool
```

Tests a GameState or Observation against the end turn condition and returns whether the turn has finished.

Method on_turn_ended

```
def on_turn_ended(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
)
```

Moves the GameState or Observation when the Player turn is finished.

Method step

```
def step(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState, ASMACAG.Game.Observation.Observ
```

```
        action: ASMACAG.Game.Action.Action
    ) -> bool
```

Moves a GameState or Observation forward by playing the Action. Returns false if the Action couldn't be played.

Module `ASMACAG.Game.Rules.SimpleForwardModel`

Defines a basic default set of rules for a Game.

Classes

Class `SimpleForwardModel`

```
    class SimpleForwardModel
```

Defines a basic default set of rules for a Game.

Ancestors (in MRO)

- Game.Rules.ForwardModel.ForwardModel
- abc.ABC

Methods

Method `give_min_score`

```
    def give_min_score(
        self,
        game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
    )
```

Calculates the minimum possible score for the GameState or Observation and adds it to the current player.

Method `is_terminal`

```
    def is_terminal(
        self,
        game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
    ) -> bool
```

Tests a GameState or Observation against a finish condition and returns whether it has finished.

Method **is_turn_finished**

```
def is_turn_finished(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
) -> bool
```

Tests a GameState or Observation against the end turn condition and returns whether the turn has finished.

Method **on_turn_ended**

```
def on_turn_ended(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState,ASMACAG.Game.Observation.Observa
)
```

Moves the GameState or Observation when the Player turn is finished.

Method **step**

```
def step(
    self,
    game_state: Union[ASMACAG.Game.GameState.GameState, ASMACAG.Game.Observation.Observ
    action: ASMACAG.Game.Action.Action
) -> bool
```

Moves a GameState or Observation forward by playing the Action. Returns false if the Action couldn't be played.

Module **ASMACAG.Heuristics**

Module containing different Heuristic to evaluate a GameState or an Observation.

Sub-modules

- ASMACAG.Heuristics.Heuristic
- ASMACAG.Heuristics.SimpleHeuristic

Module **ASMACAG.Heuristics.Heuristic**

Abstract base class that defines a reward for the current Player given an Observation.

Classes

Class `Heuristic`

```
class Heuristic
```

Abstract base class that defines a reward for the current Player given an Observation.

Ancestors (in MRO)

- abc.ABC

Methods

Method `get_reward`

```
def get_reward(
    self,
    observation: ASMACAG.Game.Observation.Observation
) -> float
```

Returns a reward for the current Player given an Observation.

Module `ASMACAG.Heuristics.SimpleHeuristic`

Defines a simple reward for the current Player given an Observation by using the current score difference.

Classes

Class `SimpleHeuristic`

```
class SimpleHeuristic
```

Defines a simple reward for the current Player given an Observation by using the current score difference.

Ancestors (in MRO)

- Heuristics.Heuristic.Heuristic
- abc.ABC

Methods

Method `get_reward`

```
def get_reward(
    self,
    observation: ASMACAG.Game.Observation.Observation
) -> float
```

Returns a reward for the current Player given an Observation by using the current score difference.

Module `ASMACAG.Players`

Module containing different Player to evaluate a GameState or an Observation.

Sub-modules

- ASMACAG.Players.HumanPlayer
- ASMACAG.Players.MCTS
- ASMACAG.Players.NTBEA
- ASMACAG.Players.OE
- ASMACAG.Players.OSLAPlayer
- ASMACAG.Players.Player
- ASMACAG.Players.RandomPlayer

Module `ASMACAG.Players.HumanPlayer`

Entity that lets a human player play an Game by using console inputs.

Classes

Class `HumanPlayer`

```
class HumanPlayer
```

Entity that lets a human player play an Game by using console inputs.

Ancestors (in MRO)

- Players.Player.Player
- abc.ABC

Methods

Method **think**

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Requests the user to decide what Action to play using the console.

Module **ASMACAG.Players.MCTS**

Module containing the MCTSPlayer and the auxiliary classes needed fot it.

Sub-modules

- ASMACAG.Players.MCTS.MCTSNode
- ASMACAG.Players.MCTS.MCTSPlayer

Module **ASMACAG.Players.MCTS.MCTSNode**

Node class for the tree used in MCTSPlayer.

Classes

Class **MCTSNode**

```
class MCTSNode(
    observation: ASMACAG.Game.Observation.Observation,
    heuristic: ASMACAG.Heuristics.Heuristic.Heuristic,
    action: ASMACAG.Game.Action.Action,
    parent: MCTSNode = None
)
```

Node class for the tree used in MCTSPlayer.

Methods

Method **add_child**

```
def add_child(
    self,
    child: MCTSNode
) -> None
```

Adds a child to the Node child list.

Method `backpropagate`

```
def backpropagate(
    self,
    reward: float
) -> None
```

Backpropagates the reward to the Node and its parents.

Method `extend`

```
def extend(
    self
) -> None
```

Extends the Node by generating a child for each possible Action.

Method `get_action`

```
def get_action(
    self
) -> ASMACAG.Game.Action.Action
```

Returns the Action of the Node.

Method `get_amount_of_children`

```
def get_amount_of_children(
    self
) -> int
```

Returns the amount of children of the Node.

Method `get_average_reward`

```
def get_average_reward(
    self
) -> float
```

Returns the average reward of the Node

Method **`get_best_child_by_average`**

```
def get_best_child_by_average(
    self
) -> Optional[ASMACAG.Players.MCTS.MCTSNode.MCTSNode]
```

Returns the best child of the Node by average reward.

Method **`get_best_child_by_ucb`**

```
def get_best_child_by_ucb(
    self,
    c_value: float
) -> ASMACAG.Players.MCTS.MCTSNode.MCTSNode
```

Returns the child of the Node with the highest UCB value.

Method **`get_is_terminal`**

```
def get_is_terminal(
    self
) -> bool
```

Returns whether the Node is terminal (as in the game is over or the turn is finished).

Method **`get_is_unvisited`**

```
def get_is_unvisited(
    self
) -> bool
```

Returns whether the Node is unvisited.

Method **`get_random_child`**

```
def get_random_child(
    self
) -> ASMACAG.Players.MCTS.MCTSNode.MCTSNode
```

Returns a random child of the Node.

Method **`rollout`**

```
def rollout(
    self
) -> float
```

Performs a random rollout from the Node and returns the reward.

Method `visit`

```
def visit(
    self,
    reward: float
) -> None
```

Visits the Node by adding to the visit count and adding the reward to the total reward.

Module `ASMACAG.Players.MCTS.MCTSPlayer`

Entity that plays a Game by using the Monte Carlo Tree Search algorithm to choose all Action in a turn.

Classes

Class `MCTSPlayer`

```
class MCTSPlayer(
    heuristic: ASMACAG.Heuristics.Heuristic.Heuristic,
    c_value: float
)
```

Entity that plays a Game by using the Monte Carlo Tree Search algorithm to choose all Action in a turn.

Ancestors (in MRO)

- Players.Player.Player
- abc.ABC

Methods

Method `compute_turn`

```
def compute_turn(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
)
```

Computes a list of Action for a complete turn using the Monte Carlo Tree Search algorithm and sets it as the turn.

Method **think**

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Computes a list of Action for a complete turn using the Monte Carlo Tree Search algorithm and returns them in order each time it's called during the turn.

Module **ASMACAG.Players.NTBEA**

Module containing the NTBEAPlayer and the auxiliary classes needed fot it.

Sub-modules

- ASMACAG.Players.NTBEA.Bandit1D
- ASMACAG.Players.NTBEA.Bandit2D
- ASMACAG.Players.NTBEA.FitnessEvaluator
- ASMACAG.Players.NTBEA.NTBEAPlayer

Module **ASMACAG.Players.NTBEA.Bandit1D**

Class representing a 'bandit' that holds score data for a one-dimensional stat, to be used in the model for the NTBEA algorithm used by NTBEAPlayer.

Classes

Class **Bandit1D**

```
class Bandit1D(
    c: float
)
```

Class representing a 'bandit' that holds score data for a one dimensional value, to be used in the model for the NTBEA algorithm used by NTBEAPlayer.

Methods

Method **get_element_best_score**

```
def get_element_best_score(
    self
) -> int
```

Returns the element with the biggest score.

Method **get_element_best_ucb**

```
def get_element_best_ucb(
    self
) -> int
```

Returns the element with the biggest ucb value.

Method **get_score**

```
def get_score(
    self,
    element: int
) -> float
```

Returns the score of the given element.

Method **get_ucb**

```
def get_ucb(
    self,
    element: int
) -> float
```

Returns the ucb value for a given element.

Method **update**

```
def update(
    self,
    element: int,
    score: float
) -> None
```

Updates the bandit with an element and its score. If it doesn't exist yet, it is added.

Module ASMACAG.Players.NTBEA.Bandit2D

Class representing a 'bandit' that holds score data for a two-dimensional stat (an ordered pair), to be used in the model for the NTBEA algorithm used by NTBEAPlayer.

Classes

Class Bandit2D

```
class Bandit2D(
    c: float
)
```

Class representing a 'bandit' that holds score data for a two-dimensional stat (an ordered pair), to be used in the model for the NTBEA algorithm used by NTBEAPlayer.

Methods

Method get_element

```
def get_element(
    self,
    element1: int,
    element2: int
) -> int
```

Transforms a pair of elements int a unique individual element.

Method get_elements

```
def get_elements(
    self,
    element: int
) -> (<class 'int'>, <class 'int'>)
```

Transforms an individual element back to the pair of elements in encodes.

Method get_elements_best_score

```
def get_elements_best_score(
    self
) -> Tuple[int, int]
```

Returns the pair of elements with the biggest score.

Method **`get_elements_best_ucb`**

```
def get_elements_best_ucb(
    self
) -> Tuple[int, int]
```

Returns the pair of elements with the biggest ucb value.

Method **`get_score`**

```
def get_score(
    self,
    element1: int,
    element2: int
) -> float
```

Returns the score of a given pair of elements.

Method **`get_ucb`**

```
def get_ucb(
    self,
    element1: int,
    element2: int
) -> float
```

Returns the ucb value for a given pair of elements.

Method **`update`**

```
def update(
    self,
    element1: int,
    element2: int,
    score: float
) -> None
```

Updates the bandit with a pair of elements and its score. If it doesn't exist yet, it is added.

Module `ASMACAG.Players.NTBEA.FitnessEvaluator`

Class used to calculate the fitness of a turn decided by NTBEA. It needs to translate between an Action list and the ints the Bandit1D and Bandit2D use.

Classes

Class `FitnessEvaluator`

```
class FitnessEvaluator(
    heuristic
)
```

Class used to calculate the fitness of a turn decided by NTBEA. It needs to translate between an Action list and the ints the Bandit1D and Bandit2D use.

Methods

Method `evaluate`

```
def evaluate(
    self,
    parameters: list[int],
    observation: ASMACAG.Game.Observation.Observation
) -> float
```

Calculates the fitness of a turn given by NTBEA as a parameter list, playing it from the given Observation.

Method `get_action_from_parameter`

```
def get_action_from_parameter(
    self,
    parameter: int
) -> ASMACAG.Game.Action.Action
```

Converts an int parameter from NTBEA to an Action.

Method `get_parameter_from_action`

```
def get_parameter_from_action(
    self,
    action: ASMACAG.Game.Action.Action
) -> int
```

Converts an Action to an int parameter for NTBEA.

Method `ntbea_to_turn`

```
def ntbea_to_turn(
    self,
    ntbea_parameters: list[int]
) -> list[ASMACAG.Game.Action.Action]
```

Converts a list of int parameters from NTBEA to a list of Action representing a turn.

Module `ASMACAG.Players.NTBEA.NTBEAPlayer`

Entity that plays a Game by using the N-Tuple Bandit Evolutionary Algorithm to model fitness and evolve a list of Action based on it, composing a turn.

Classes

Class `NTBEAPlayer`

```
class NTBEAPlayer(
    heuristic: ASMACAG.Heuristics.Heuristic.Heuristic,
    dimensions: list[int],
    c_value: float,
    neighbours: int,
    mutation_rate: float,
    initializations: int
)
```

Entity that plays a Game by using the N-Tuple Bandit Evolutionary Algorithm to model fitness and evolve a list of Action based on it, composing a turn.

Methods

Method `compute_turn`

```
def compute_turn(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float,
    initializations: int
)
```

Computes a list of Action for a complete turn using the N-Tuple Bandit Evolutionary Algorithm it as the turn.

Method `create_bandits`

```
def create_bandits(
    self
) -> None
```

Create the empty 1D and 2D bandits.

Method **`get_best_individual`**

```
def get_best_individual(
    self,
    population: list[list[int]]
) -> list[int]
```

Returns the best individual from a population, by UCB

Method **`get_neighbours`**

```
def get_neighbours(
    self,
    individual: list[int],
    neighbour_amount: int,
    mutation_rate: float
) -> list[list[int]]
```

Generates a list of neighbours from an individual. It changes at least one parameter (randomly chosen). The rest of them can change depending on the mutation rate.

Method **`get_random_individual_valid`**

```
def get_random_individual_valid(
    self,
    observation: ASMACAG.Game.Observation.Observation
) -> list[int]
```

Generates a random turn that is valid for the given observation. Note that the observation state after running this method will be the result of playing the turn.

Method **`get_total_ucb`**

```
def get_total_ucb(
    self,
    individual: list[int]
) -> float
```

Returns the UCB of an individual, being the mean of its UCB for each bandit. If the individual is not in a bandit it will return a big number.

Method **`mutate_gen`**

```
def mutate_gen(
    self,
    individual: list[int],
    j: int
) -> None
```

Mutate the j-th gen of an individual.

Method **`think`**

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Computes a list of Action for a complete turn using the N-Tuple Bandit Evolutionary Algorithm and returns them in order each time it’s called during the turn.

Method **`update_bandits`**

```
def update_bandits(
    self,
    individual: list[int],
    score: float
) -> None
```

Updates the bandits with the given individual and score.

Method **`valid_initialization`**

```
def valid_initialization(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    initializations: int
) -> Tuple[list[int], float]
```

Generates a given amount of complete valid turns randomly and adds their stats to the bandit-based model, returning the best turn found and the score it yielded.

Module `ASMACAG.Players.OE`

Module containing the OEPlayer and the auxiliary classes needed fot it.

Sub-modules

- ASMACAG.Players.OE.OEPlayer
- ASMACAG.Players.OE.TurnGenome

Module `ASMACAG.Players.OE.OEPlayer`

Entity that plays a Game by using the Online Evolution algorithm to evolve a list of Action composing a turn.

Classes

Class `OEPlayer`

```
class OEPlayer(
    heuristic: ASMACAG.Heuristics.Heuristic.Heuristic,
    population_size: int,
    mutation_rate: float,
    survival_rate: float
)
```

Entity that plays a Game by using the Online Evolution algorithm to evolve a list of Action composing a turn.

Ancestors (in MRO)

- Players.Player.Player
- abc.ABC

Methods

Method `compute_turn`

```
def compute_turn(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
)
```

Computes a list of Action for a complete turn using the Online Evolution algorithm and sets it as the turn.

Method **`think`**

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Computes a list of Action for a complete turn using the Online Evolution algorithm and returns them in order each time it's called during the turn.

Module `ASMACAG.Players.OE.TurnGenome`

Genome class representing a list of Action composing a turn for use in OEPlayer.

Classes

Class **`TurnGenome`**

```
class TurnGenome
```

Genome class representing a list of Action composing a turn for use in OEPlayer.

Methods

Method **`clone`**

```
def clone(
    self
) -> ASMACAG.Players.OE.TurnGenome.TurnGenome
```

Returns a clone of this TurnGenome.

Method **`copy_into`**

```
def copy_into(
    self,
    other: TurnGenome
) -> None
```

Copies this TurnGenome into another one.

Method **crossover**

```
def crossover(
    self,
    parent_a: TurnGenome,
    parent_b: TurnGenome,
    observation: ASMACAG.Game.Observation.Observation
)
```

Fills up this TurnGenome with Action from both parents while making sure that the resulting turn is valid. Note that the observation state is not preserved.

Method **get_actions**

```
def get_actions(
    self
) -> list[ASMACAG.Game.Action.Action]
```

Returns the list of Action of this TurnGenome.

Method **get_reward**

```
def get_reward(
    self
) -> float
```

Returns the reward of this TurnGenome.

Method **mutate_at_random_index**

```
def mutate_at_random_index(
    self,
    observation: ASMACAG.Game.Observation.Observation
)
```

Mutates this TurnGenome at a random Action of the turn while keeping the whole turn valid. Note that the observation state is not preserved.

Method **random**

```
def random(
    self,
    observation: ASMACAG.Game.Observation.Observation
)
```

Fills up this TurnGenome with random valid Action composing a turn. Note that the observation state is not preserved.

Method `set_reward`

```
def set_reward(
    self,
    reward: float
) -> None
```

Sets the reward of this TurnGenome.

Module `ASMACAG.Players.OSLAPlayer`

Entity that plays a Game by selecting the best Action found with a greedy one step lookahead search based on an Heuristic.

Classes

Class `OSLAPlayer`

```
class OSLAPlayer(
    heuristic: ASMACAG.Heuristics.Heuristic.Heuristic
)
```

Entity that plays a Game by selecting the best Action found with a greedy one step lookahead search based on an Heuristic.

Ancestors (in MRO)

- Players.Player.Player
- abc.ABC

Methods

Method `think`

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Returns a randomly selected valid Action to play given an Observation.

Module `ASMACAG.Players.Player`

Abstract base class for an entity with a defined behaviour for playing a Game.

Classes

Class `Player`

```
class Player
```

Abstract base class for an entity with a defined behaviour for playing a Game.

Ancestors (in MRO)

- abc.ABC

Methods

Method `think`

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Returns an Action to play given an Observation. It must return an action within the given budget of time (in seconds).

Module `ASMACAG.Players.RandomPlayer`

Entity that plays a Game by selecting random valid Action.

Classes

Class `RandomPlayer`

```
class RandomPlayer
```

Entity that plays a Game by selecting random valid Action.

Ancestors (in MRO)

- Players.Player.Player
- abc.ABC

Methods

Method **`think`**

```
def think(
    self,
    observation: ASMACAG.Game.Observation.Observation,
    budget: float
) -> ASMACAG.Game.Action.Action
```

Returns a randomly selected valid Action to play given an Observation.

Module `ASMACAG.play_game`

Main program that plays a Game between two Player.

Module `ASMACAG.play_n_games`

Main program that plays a set number of Game between any number of pairs of Player.