Reference Manual

Games

class `easyAI.`**`TwoPlayersGame`** ¶ [source]

Base class for… wait for it… two-players games !

To define a new game, make a subclass of TwoPlayersGame, and define the following methods:

- `__init__(self, players, ...)` : initialization of the game
- `possible_moves(self)` : returns of all moves allowed
- `make_move(self, move)`: transforms the game according to the move
- `is_over(self)`: check whether the game has ended

The following methods are optional:

- `show(self)` : prints/displays the game
- `scoring`: gives a score to the current game (for the AI)
- `unmake_move(self, move)`: how to unmake a move (speeds up the AI)
- `ttentry(self)`: returns a string/tuple describing the game.
- `ttrestore(self, entry)`: use string/tuple from ttentry to restore a game.

The __init__ method *must* do the following actions:

- Store `players` (which must be a list of two Players) into self.players
- Tell which player plays first with `self.nplayer = 1 # or 2`

When defining `possible_moves`, you must keep in mind that you are in the scope of the *current player*. More precisely, a subclass of TwoPlayersGame has the following attributes that indicate whose turn it is. These methods can be used but should not be overwritten:

- `self.player` : the current player (e.g. `Human_Player`)
- `self.opponent` : the current Player's opponent (Player).
- `self.nplayer`: the number (1 or 2) of the current player.
- `self.nopponent`: the number (1 or 2) of the opponent.
- `self.nmove`: How many moves have been played so far ?

For more, see the examples in the dedicated folder.

`get_move`() [source]

Method for getting a move from the current player. If the player is an AI_Player, then this method will invoke the AI algorithm to choose the move. If the player is a Human_Player, then the interaction with the human is via the text terminal.

`play`(*nmoves=1000*, *verbose=True*) [source]

Method for starting the play of a game to completion. If one of the players is a Human_Player, then the interaction with the human is via the text terminal.

Parameters: **nmoves:**

The limit of how many moves (plies) to play unless the game ends on it's own first.

verbose:

Setting verbose=True displays additional text messages.

`play_move`(*move*) [source]

Method for playing one move with the current player. After making the move, the current player will change to the next player.

Parameters: **move:**

The move to be played. `move` should match an entry in the `.possibles_moves()` list.

Players

class `easyAI.`**`Human_Player`**(*name='Human'*) [source]

Class for a human player, which gets asked by text what moves she wants to play. She can type `show moves` to display a list of moves, or `quit` to quit the game.

class `easyAI.`**`AI_Player`**(*AI_algo*, *name='AI'*) [source]

Class for an AI player. This class must be initialized with an AI algortihm, like `AI_Player( Negamax(9) )`

AI algorithms

class `easyAI.AI.`**`Negamax`**(*depth*, *scoring=None*, *win_score=inf*, *tt=None*) [source]

This implements Negamax on steroids. The following example shows how to setup the AI and play a Connect Four game:

```
>>> from easyAI.games import ConnectFour
>>> from easyAI import Negamax, Human_Player, AI_Player
>>> scoring = lambda game: -100 if game.lose() else 0
>>> ai_algo = Negamax(8, scoring) # AI will think 8 turns in advance
>>> game = ConnectFour([Human_Player(), AI_Player(ai_algo)])
>>> game.play()
```

Parameters: **depth:**

How many moves in advance should the AI think ? (2 moves = 1 complete turn)

scoring:

A function f(game)-> score. If no scoring is provided

and the game object has a `scoring` method it ill be used.

win_score:

Score above which the score means a win. This will be

used to speed up computations if provided, but the AI will not differentiate quick defeats from long-fought ones (see next section).

tt:

A transposition table (a table storing game states and moves) scoring: can be none if the game that the AI will be given has a `scoring` method.

Notes

The score of a given game is given by

```
>>> scoring(current_game) - 0.01*sign*current_depth
```

for instance if a lose is -100 points, then losing after 4 moves will score -99.96 points but losing after 8 moves will be -99.92 points. Thus, the AI will chose the move that leads to defeat in 8 turns, which makes it more difficult for the (human) opponent. This will not always work if a `win_score` argument is provided.

class `easyAI.AI.`**`NonRecursiveNegamax`**(*depth*, *scoring=None*, *win_score=inf*, *tt=None*) [source]

This implements Negamax without recursion. The following example shows how to setup the AI and play a Connect Four game:

```
>>> from easyAI.games import ConnectFour
>>> from easyAI import NonRecursiveNegamax, Human_Player, AI_Player
>>> scoring = lambda game: -100 if game.lose() else 0
>>> ai_algo = NonRecursiveNegamax(8, scoring) # AI will think 8 turns in advance
>>> game = ConnectFour([Human_Player(), AI_Player(ai_algo)])
>>> game.play()
```

This algorithm also *REQUIRES* that the game class support the `ttentry` and `ttrestore` methods.

This algorithm ignores any optional `unmake_move` method in the game class.

This version of Negamax does not support transposition tables.

Parameters: **depth:**

How many moves in advance should the AI think ? (2 moves = 1 complete turn)

scoring:

A function f(game)-> score. If no scoring is provided

and the game object has a `scoring` method it will be used.

win_score:

Score above which the score means a win.

tt:

A transposition table (a table storing game states and moves). Currently, this parameter is ignored.

class `easyAI.AI.`**`DUAL`**(*depth*, *scoring=None*, *win_score=100000*, *tt=None*) [source]

This implements DUAL algorithm. The following example shows how to setup the AI and play a Connect Four game:

```
>>> from easyAI import Human_Player, AI_Player, DUAL
>>> AI = DUAL(7)
>>> game = ConnectFour([AI_Player(AI),Human_Player()])
>>> game.play()
```

Parameters: **depth:**

How many moves in advance should the AI think ? (2 moves = 1 complete turn)

scoring:

A function f(game)-> score. If no scoring is provided

and the game object has a `scoring` method it ill be used.

win_score:

Score LARGER than the largest score of game, but smaller than inf. It's required to run algorithm.

tt:

A transposition table (a table storing game states and moves) scoring: can be none if the game that the AI will be given has a `scoring` method.

Notes

The score of a given game is given by

```
>>> scoring(current_game) - 0.01*sign*current_depth
```

for instance if a lose is -100 points, then losing after 4 moves will score -99.96 points but losing after 8 moves will be -99.92 points. Thus, the AI will chose the move that leads to defeat in 8 turns, which makes it more difficult for the (human) opponent. This will not always work if a `win_score` argument is provided.

class `easyAI.AI.`**`SSS`**(*depth*, *scoring=None*, *win_score=100000*, *tt=None*) [source]

This implements SSS* algorithm. The following example shows how to setup the AI and play a Connect Four game:

```
>>> from easyAI import Human_Player, AI_Player, SSS
>>> AI = SSS(7)
>>> game = ConnectFour([AI_Player(AI),Human_Player()])
>>> game.play()
```

Parameters: **depth:**

How many moves in advance should the AI think ? (2 moves = 1 complete turn)

scoring:

A function f(game)-> score. If no scoring is provided

and the game object has a `scoring` method it ill be used.

win_score:

Score LARGER than the largest score of game, but smaller than inf. It's required to run algorithm.

tt:

A transposition table (a table storing game states and moves) scoring: can be none if the game that the AI will be given has a `scoring` method.

Notes

The score of a given game is given by

```
>>> scoring(current_game) - 0.01*sign*current_depth
```

for instance if a lose is -100 points, then losing after 4 moves will score -99.96 points but losing after 8 moves will be -99.92 points. Thus, the AI will chose the move that leads to defeat in 8 turns, which makes it more difficult for the (human) opponent. This will not always work if a `win_score` argument is provided.

Transposition tables

class `easyAI.AI.TT`(*own_dict=None*) [source]

A tranposition table made out of a Python dictionnary.

It creates a "cache" of already resolved moves that can, under some circumstances, let the algorithm run faster.

This table can be stored to file, allowing games to be stopped and restarted at a later time. Or, if the game is fully solved, the cache can return the correct moves nearly instantly because the AI alogorithm no longer has to compute correct moves.

Transposition tables can only be used on games which have a method game.ttentry() -> string or tuple

To save the table as a *pickle* file, use the **tofile** and **fromfile** methods. A pickle file is binary and usually faster. A pickle file can also be appended to with new cached data. See python's pickle documentation for secuirty issues.

To save the table as a universal JSON file, use the **to_json_file** and **from_json_file** methods. For these methods, you must explicity pass **use_tuples=True** if game.ttentry() returns tuples rather than strings.

Usage:

```
>>> table = TT()
>>> ai = Negamax(8, scoring, tt = table)
>>> ai(some_game) # computes a move, fills the table
>>> table.tofile('saved_tt.data') # maybe save for later ?

>>> # later (or in a different program)...
>>> table = TT().fromfile('saved_tt.data')
>>> ai = Negamax(8, scoring, tt = table)
```

Transposition tables can also be used as an AI (`AI_player(tt)`) but they must be exhaustive in this case: if they are asked for a position that isn't stored in the table, it will lead to an error.

`from_json_file`(*filename*, *use_tuples=False*) [source]

Loads a transposition table previously saved with `TT.to_json_file`

`fromfile`(*filename*) [source]

Loads a transposition table previously saved with `TT.tofile`

`lookup`(*game*) [source]

Requests the entry in the table. Returns None if the entry has not been previously stored in the table.

`store`(***data*) [source]

Stores an entry into the table

`to_json_file`(*filename*, *use_tuples=False*) [source]

Saves the transposition table to a serial JSON file. Warning: the file can be big (~100Mo).

`tofile`(*filename*) [source]

Saves the transposition table to a file. Warning: the file can be big (~100Mo).

Solving Games

`easyAI.AI.solving.`**`id_solve`**(*game*, *ai_depths*, *win_score*, *scoring=None*, *tt=None*, *verbose=True*) [source]

Solves a game using iterative deepening, i.e. determines if by playing perfectly the first player can force a win, or whether it will always lose against a perfect opponent.

This algorithm explores the game by using several times the Negamax algorithm, always starting at the initial state of the game, but taking increasing depth (in the list ai_depths) until the score of the initial condition indicates that the first player will certainly win or loose, in which case it stops. The use of transposition table leads to speed gain as the results of shallower searches are used to help exploring the deeper ones.

Parameters: **ai_depths:**

List of AI depths to try (e.g. [5,6,7,8,9,10])

win_score:

Score above which a score means a win.

scoring:

Scoring function (see doc of class Negamax)

tt:

An optional transposition table to speed up computations.

verbose:

If set to `True`, will print a summary of the best move after each depth tried.

Returns: (result, depth, move):

As below

result:

Either 1 (certain victory of the first player) or -1 (certain defeat) or 0 (either draw, or the search was not deep enough)

depth:

The minimal number of moves before victory (or defeat)

move:

Best move to play for the first player.

Also returns with `tt` set.

Will be None if `use_tt` was set to false, else will be a transposition table containing all the relevant situations to play a perfect game and can be used with `AI_player(tt)`

`easyAI.AI.solving.`**`df_solve`**(*game*, *win_score*, *maxdepth=50*, *tt=None*, *depth=0*) [source]

Solves a game using a depth-first search: the game is explored until endgames are reached.

The endgames are evaluated to see if there are victories or defeats. Then, a situation in which every move leads to a defeat is labelled as a (certain) defeat, and a situation in which one move leads to a (certain) defeat of the opponent is labelled as a (certain) victory. Situations are evaluated until the initial condition receives a label (victory or defeat). Draws are also possible.

This algorithm can be faster but less informative than `id_solve`, as it does not provide 'optimal' strategies (like shortest path to the victory). It returns simply 1, 0, or -1 to indicate certain victory, draw, or defeat of the first player.

Parameters: **game:**

An Game instance, initialized and ready to be played.

win_score:

Score above which a score means a win.

maxdepth:

Maximal recursion depth allowed.

tt:

An optional transposition table to speed up computations.

depth:

Index of the current depth (don't touch that).

Returns: result

Either 1 (certain victory of the first player) or -1 (certain defeat) or 0 (either draw, or the search was not deep enough)