Game over

- **Topic:** decision making, Monte Carlo tree search, reinforcement learning
- **Task A:**
  1. Implement a method for exploring the decision tree.
  2. Play a game of tic-tac-toe against the computer. It is such a simple game that a working AI should never misplay.
  3. Once the AI works, try having an AI vs. AI match. You can also try changing game settings. How about playing on a larger grid with a row of 5 needed for victory?
  4. Study how changing the AI thinking time affects performance.
- **Task B:**
  1. Play 4 in a row. This game has simpler decisions than tic-tac-toe (choice of column vs. choice of column *and* row) so the AI should be more efficient. Can it beat you?
- **Template:**
  - game_ai.py
  - tictac.py
  - n_in_a_row.py
- **Further reading:**
  - https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
  - https://deepmind.com/blog/article/alphago-zero-starting-scratch

game_ai.py

`class game_ai.AI(player, thinking_time=5000)` [source]

A Monte Carlo tree search AI.

The AI chooses its actions using a MC tree built out of MC_node objects.

Parameters:
- **player** (int) - the player number for this AI
- **thinking_time** (int) - the number of iterations the AI is allowed when building the MC tree

`pick_move(state)` [source]

Given the state of the game, the AI chooses its next move.

Parameters: **state** - the state of the game in the format defined by the game module
Returns: the chosen play action (move) in the format defined by the game module
Return type: object

`print_progress(step)` [source]

Draws a progress bar.

Parameters: **step** (int) - the number of iterations taken so far

`class game_ai.MC_node(player, state, layer, play, explore=1.0)` [source]

A node in the Monte Carlo tree.

Each node records the win and play statistics and links to the nodes that come before and after it in the tree.

Parameters:
- **player** (int) - player number for the player whose move lead to this node
- **state** - current state of the game
- **layer** (int) - counts how deep in the tree this node is
- **play** - the play action ("move") that leads to this node
- **explore** (float) – for large values the algorithm is more eager to try non-optimal plays

`backpropagate(winner)` [source]

Recursively records play statistics according to who won the simulated game.

The number of plays from this node is increased by one regardless of the outcome of the game.

If the player whose turn it is at this node won the game, the win count is also increased by one. If the game ended in a draw, the win count is increased by one half.

Once the stats have been recorded for one node, the algorithm moves on to the parent node. The process is repeated until the root node (the node with no parent) is reached.

Parameters: **winner** (int) – number of the winning player or 0 if the result was a draw

`bias_factor()` [source]

Calculates the factor for choosing the next node for a node where all the following nodes have at least one recorded play. The higher this factor is, the more likely it is that this node is chosen.

The factor is calculated as

$$\frac{w}{n} + c\sqrt{\frac{2\ln(n_{\text{parent}})}{n}}$$

where

- w = the recorded wins in all the playouts from this node
- n = the number of recorded playouts from this node
- c = the exploration factor
- n_{parent} = the number of recorded playouts from the previous node

The first term w/n is the win ratio for the playouts from this node. If this node is likely to lead to victory, this ratio is close to 1 and gets chosen often.

The second term is large, if there are only a few plays from this node and many plays from the *other nodes* that branch off the same parent node (the competing play options). If this factor was not included, any node whose initial play lead to a loss would never be considered again by the algorithm. Since this may happen simply due to bad luck, it is important to give these nodes some additional plays.

Returns: the bias factor for this node
Return type: float

`create_branches()` [source]

Creates new MC tree nodes corresponding to all the play options available at the game state represented by this node. The new nodes are created as MC_node objects and saved in the list self.branches of this node.

`explore_branches()` [source]

Travels through the MC tree, starting from this node, looking for a leaf node.

At each node, the algorithm checks if the current node is a leaf node using MC_node.is_leaf(). If it is, the current node is returned.

If the currently examined node is not a leaf, the algorithm calculates bias factors for all the nodes following the current node and then picks the node with the highest factor using MC_node.pick_played_branch(). This node then becomes the current node and the process is repeated.

Eventually, the process will end up at a leaf node At that point, the exploration stops and the found leaf is returned.

Note:

This function is incomplete!

Returns: the found leaf node
Return type: MC_node

`has_branches()` [source]

Checks if more gameplay branches follow from this node.

Returns: True if more nodes follow this node in the MC tree.
Return type: bool

`is_leaf()` [source]

Checks if this is a leaf node.

A node is said to be a leaf node if

a. the game ends at this node (there are no following nodes) **or**
b. at least one of the following nodes has 0 recorded plays

I.e., a leaf is a node at the outer edge of the tree.

If this node is not terminal but the following nodes have not yet been created, the function calls MC_node.create_branches() to create nodes for all possible plays.

Returns: True if this is a leaf node.
Return type: bool

`pick_best_play()` [source]

Chooses the play action (i.e., move) recorded in the node that

a. follows this node **and**
b. has the most recorded plays

Returns: the most tried out play
Return type: object

`pick_played_branch()` [source]

Chooses the node that

a. follows this node in the MC tree **and**
b. has the highest play factor as given by MC_node.bias_factor().

Returns: the chosen node
Return type: MC_node

`pick_unplayed_branch()` [source]

Randomly chooses one node out of the nodes that

a. follow this node in the MC tree **and**
b. have no recorded plays

Returns: the chosen node
Return type: MC_node

`simulate()` [source]

Simulates the outcome of the game starting from the current node.

The algorithm alternates between players making each player randomly choose one of their available play options until the game ends.

Returns: the player number of the winner or 0 if the game ended in a draw
Return type: int

`game_ai.play_game(player_types, thinking_time=5000)` [source]

Plays a full game.

Parameters:
- **player_types** (list) - a list of players that may be either HUMAN or AI
- **thinking_time** (int) - the number of iterations allowed for the AI

Returns

int: the player number of the winner or 0 for a draw

tictac.py

`tictac.all_plays(grid, player=None)` [source]

Lists the coordinates of all empty slots.

returns: list of possible play actions

`tictac.all_reasonable_plays(grid, player=None)` [source]

Same as all_plays().

returns: list of possible play actions

`tictac.all_reasonable_plays_alternative(grid, player=None)` [source]

Lists the coordinates of all empty slots that have an occupied neighbor.

returns: list of possible play actions

`tictac.ask_for_move(player, grid)` [source]

Asks a human player for a play action (move).

This is done by visualizing the game and then asking for the index of the column and row where the player wants to play.

player: number of the player whose turn it is grid: the play area (current game state)

returns: the coordinates of the chosen slot as tuple (row, column)

`tictac.can_continue(grid)` [source]

Checks if there are available spaces in the play area.

Does not check if the game is done due to someone winning.

returns: True if there are empty spaces, False if not.

`tictac.can_take_slot(i, j, grid)` [source]

Checks if one can play in the given slot.

i: the row to check j: the column to check grid: the play area (current game state)

returns: True if there are empty slots in column i, False otherwise

`tictac.check_winner(grid)` [source]

Checks if someone has won the game.

returns: the number of the winning player if there is one, 0 otherwise

`tictac.copy_game(grid)` [source]

Returns a copy of the given game state.

grid: the play area (current game state)

returns: a copy of grid

`tictac.count_consecutives(i, j, grid)` [source]

Starting from the slot at grid[i,j], counts the highest number of consecutive squares with tokens of the same player.

If the starting position has no token, returns 0.

For ay given square, there are 8 possible directions for forming a line: up, down, left, right and the 4 diagonals. This function only checks half of them: down, right, down-right and down-left.

i: vertical position (y coordinate) of the initial slot j: horizontal position (x coordinate) of the initial slot grid: the play area (current game state)

returns: the highest number of similar consecutive tokens

`tictac.declare_winner(winner)` [source]

Celebrates the victory of a player or declares a draw.

winner: number of the winning player or 0 for a draw

`tictac.draw(grid)` [source]

Draw the play area with text graphics.

grid: the play area (current game state)

`tictac.make_move(play, player, grid)` [source]

Let a player have a turn.

The function modifies the given game state (grid) to match the situation after the move has been carried out.

play: the play action (move) to take player: number of the player who makes the move grid: the play area (current game state)

`tictac.new_game()` [source]

Creates an empty play area.

returns: the empty play area (the initial game state)

`tictac.next_player(player)` [source]

Number of the player whose turn comes after the given player.

Note: Player numbering starts from 1 and ends at n_players.

player: the player to check

returns: the number of the next player

`tictac.previous_player(player)` [source]

Number of the player whose turn comes before the given player.

Note: Player numbering starts from 1 and ends at n_players.

player: the player to check

returns: the number of the previous player

n_in_a_row.py

`n_in_a_row.all_plays(grid, player=None)` [source]

Lists the indices of all columns that have empty slots.

returns: list of possible play actions

`n_in_a_row.all_reasonable_plays(grid, player=None)` [source]

Same as all_plays().

returns: list of possible play actions

`n_in_a_row.ask_for_move(player, grid)` [source]

Asks a human player for a play action (move).

This is done by visualizing the game and then asking for the index of the column where the player wants to play.

player: number of the player whose turn it is grid: the play area (current game state)

returns: the index of the column where the player decides to play

`n_in_a_row.can_continue(grid)` [source]

Checks if there are available spaces in the play area.

Does not check if the game is done due to someone winning.

returns: True if there are empty spaces, False if not.

`n_in_a_row.can_take_slot(i, grid)` [source]

Checks if one can play in the given column.

i: the column to check grid: the play area (current game state)

returns: True if there are empty slots in column i, False otherwise

`n_in_a_row.check_winner(grid)` [source]

Checks if someone has won the game.

returns: the number of the winning player if there is one, 0 otherwise

`n_in_a_row.copy_game(grid)` [source]

Returns a copy of the given game state.

grid: the play area (current game state)

returns: a copy of grid

`n_in_a_row.count_consecutives(i, j, grid)` [source]

Starting from the slot at grid[i,j], counts the highest number of consecutive squares with tokens of the same player.

If the starting position has no token, returns 0.

For ay given square, there are 8 possible directions for forming a line: up, down, left, right and the 4 diagonals. This function only checks half of them: down, right, down-right and down-left.

i: vertical position (y coordinate) of the initial slot j: horizontal position (x coordinate) of the initial slot grid: the play area (current game state)

returns: the highest number of similar consecutive tokens

`n_in_a_row.declare_winner(winner)` [source]

Celebrates the victory of a player or declares a draw.

winner: number of the winning player or 0 for a draw

`n_in_a_row.draw(grid)` [source]

Draw the play area with text graphics.

grid: the play area (current game state)

`n_in_a_row.make_move(play, player, grid)` [source]

Let a player have a turn.

The function modifies the given game state (grid) to match the situation after the move has been carried out.

play: the play action (move) to take player: number of the player who makes the move grid: the play area (current game state)

`n_in_a_row.new_game()` [source]

Creates an empty play area.

returns: the empty play area (the initial game state)

`n_in_a_row.next_player(player)` [source]

Number of the player whose turn comes after the given player.

Note: Player numbering starts from 1 and ends at n_players.

player: the player to check

returns: the number of the next player

`n_in_a_row.previous_player(player)` [source]

Number of the player whose turn comes before the given player.

Note: Player numbering starts from 1 and ends at n_players.

player: the player to check

returns: the number of the previous player