rl_peg_solitaire

Reinforcement learning in Python for solving the peg solitaire problem

Activate virtual environment

Create a new environment by running `python3 -m venv ./venv`. Then activate it by running `venv\Scripts\activate.bat`. Lastly, install all dependencies by running `pip install -r requirements.txt`. Update dependencies by running `pip freeze > requirements.txt`.

Components

The project consists of two distinct components, namely the Reinforcement Learner and the Simulated World (or environment, i.e., the game itself). This seperation is built upon the concept of the Critical Divide, where the learner does not make any assumptions about the game and the game itself does not provide any deduced (learned) information about its states. They are in seperate folders for this project.

Reinforcement Learner (learner)

The reinforcement learner consists of an Actor, a Critic and an encapsulating class called ReinforcementLearner. This class employs a general learning algorithm that uses the actor and critic when necessary. It does not concern itself with exactly *how* these classes operate internally. For instance, whether the critic is table-based or ANN-based does not matter. Additionally, it never directly interacts with the game board. It asks for the game state, legal actions, rewards, etc. through functions built into the simulated world. That way, the reinforcement learner does not know much about the world other than the relevant states for it.

Simulated World (game)

The simulated world is based on the PegGame class. This class is the one that interacts with the other classes in the world, and acts as an intermediate to the reinforcement learner. It contains the board rules, decides upon rewards and legal actions, and returns board states in readable manners. The Board class contains the board structure and Peghole objects. Additionally, the BoardGraph class is used for display if the current PegGame is asked for it.

Documentation

The `main.py`-file can be modified to train a new model, display the performance graph, and run a game after training for display purposes (does not affect the learner's values in any way). In order to create settings for the learner and game, use the `learner_settings.py`-file. Create a dictionary for the settings, and import it from `main.py`. The dictionary consists of sub-dictionaries, with the keys of `game_settings` (settings for the game to be learned), `critic_settings` (settings for the learner's critic), and `actor_settings` (settings for the learner's actor). Additionally, the `episodes` field specifies how many episodes are to be used during training.

The variables are documented below.

game_settings

- `board_type`: Type of board to be used. Must be either `triangle` or `diamond`.
- `board_size`: Specifies the board size. Must be integer.
- `initial_empty`: Specifies the initial holes to be empty, by using tuples (x, y) in the set. If the set is empty, one hole in the center is default.
- `live_update_frequency`: The amount of seconds between each frame during game display. Must be float or integer.
- `display_game`: A tuple of episodes to display during training, 1-indexed.

critic_settings

- `c_type`: The type of critic being used. Table (`table`) or artificial neural network (`ann`).
- `c_learning_rate`: Critic learning rate. Must be float or integer.
- `c_eligibility_decay`: Decay rate of critic eligibility. Must be float or integer.
- `c_discount_factor`: Critic discount factor. Must be float or integer.
- `c_nn_layers`: Hidden layers for the artificial neural network. Each index specifies a layer, with the given amount of nodes. Must be a list of integers.

actor_settings

- `a_learning_rate`: Actor learning rate. Must be float or integer.
- `a_eligibility_decay`: Decay rate of actor eligibility. Must be float or integer.
- `a_discount_factor`: Actor discount factor. Must be float or integer.
- `a_e_greediness`: Specifies actor greediness factor. 1 is fully greedy, 0 is fully random. Must be float or integer.