4. Reinforcement Learning

Your task is to implement the *Q-learning* algorithm to find the best strategy in an environment about which you have only incomplete information. You can only perform available actions and observe their effects (reinforcement learning).

Specifications

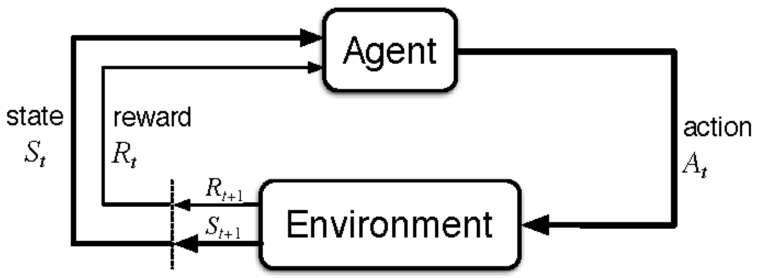
In the `rl_agent.py` module, implement the `RLAgent` class. The class must implement the following interface:

method	input parameters	output parameters	explanation
`__init__`	`env: RLProblem` , `gamma: float` , `alpha: float`	*none*	Agent initialization.
`learn_policy`	*none*	`Policy`	Returns the best strategy, i.e., a dictionary of pairs (state, action).

- The class will be initialized with the following **parameters**:
  - `env` is the environment, i.e., an object of type `kuimaze2.RLProblem` ,
  - `gamma` is the so-called "discount factor" from the range `(0,1)` ,
  - `alpha` is the so-called "learning rate"; the passed value is just a recommendation: if you want to use a different one, or change/adapt it during learning, feel free to do so.
- The **output** of the `learn_policy()` method must be a **policy** represented as a **dictionary**, where the key is always a state (instance of the `kuimaze2.State` class) and the value is the optimal action for the given state (instance of the `kuimaze2.Action` class). The strategy must contain an action for all free states, including terminal ones. For terminal states, the specific chosen action does not matter.
- **Timeout:** you have a time limit of 20s to learn a strategy for a given problem instance.
- We expect that method `learn_policy()` of your `RLAgent` will implement the *Q-learning* algorithm. But of course, you can/should decompose it to shorter functions/methods.
- In the implementation of the algorithms, you can only use the public interface of the ''RLProblem'' class [/wiki/courses/be5b33kui/semtasks/kuimaze/30_rlproblem]. If you feel that you

need to use methods of other classes than `RLProblem` or that you need to use non-public variables and methods (whose name starts with `_`), discuss it with your instructor.

Note: The environment (`env`) is an instance of the `RLProblem` class. The initialization of the environment and visualization methods are the same as for `MDPProblem` , but working with the environment is different. Executing an action is necessary to learn anything about the environment. We do not have a map and the environment can only be explored using the main method `env.step(action)` . The environment-simulator knows what the current state is.



[/wiki/_detail/courses/b3b33kui/cviceni/sekvencni_rozhodovani/agent-environment-reward.png?id=courses%3Abe5b33kui%3Asemtasks%3A04_rl%3Astart]

How to

- We recommend creating a new working directory for the task, separate from the previous sem. tasks with the `kuimaze2` environment. Set up [/wiki/courses/be5b33kui/semtasks/kuimaze/00_install] an updated version of the `kuimaze2` package in it.
- Familiarize yourself with the RLProblem [/wiki/courses/be5b33kui/semtasks/kuimaze/30_rlproblem] environment.
- In the `kuimaze2` package, you will also find the script `example_rl.py` . The script contains a skeleton of the RLAgent class, demonstrates a random walk through the environment, initialization of the q-value table, and visualization. It can be used as a starting code for the implementation of the `RLAgent` class.

How will the evaluation script call your code?

Your code will be called by the evaluation script approximately like this:

```
import kuimaze2
import rl_agent

env = kuimaze2.RLProblem(...)  # here the script creates the environment

# Calls of your code
agent = rl_agent.RLAgent(env, gamma, alpha)
policy = agent.learn_policy()  # 20 second limit
```

```
# Evaluation of one episode using your policy
state = env.reset()
episode_finished = False
while not episode_finished:
  action = policy[state]
  next_state, reward, episode_finished = env.step(action)
  total_reward += reward
  state = next_state
```

Q-function Representation

During the implementation, you will probably need to work with the Q-function. In our discrete world, it will take the form of a table. This table can be represented in various ways, for example:

- as a dictionary of dictionaries `q_table` (as shown in the `example_rl.py` module), where you can access individual elements as `q_table[state][action]` ;
- as a 3D `numpy` array, which is indexed by three “coordinates”: *r*, *c*, *action*;
- …

Q-function Initialization

In real RL tasks, we do not have a “map” of the environment and often do not even know the set of all states. RL then never ends because we can never be sure that we have already reached all reachable states (and that we have learned all q-values well enough). But in our task, RL must end and you must return a complete strategy, i.e., the best action for each state. Therefore, there must be a way to find out the set of all states. The list of all valid states can be obtained using the `get_states()` method.

The `example_rl.py` script already contains the initialization of `q_table` in the form of a dictionary of dictionaries. If you choose a different representation of the q-value table, the initialization needs to be appropriately adjusted.

Submission

- You can find the deadline for submitting the task in BRUTE [https://cw.felk.cvut.cz/brute], task `11-RL` .
- Submit the `rl_agent.py` module, or a ZIP archive with the `rl_agent.py` module and other modules you created that your agent needs/imports. **These files must be in the root of the archive, the archive must not contain any directories!** Do not submit any modules that you received from us!

Evaluation

Familiarize yourself with the evaluation and scoring [/wiki/courses/be5b33kui/semtasks/04_rl/scoring] of the task.

courses/be5b33kui/semtasks/04_rl/start.txt · Last modified: 2024/04/16 13:44 by xposik