A C++ Program for Markov Decision Process and Q-learning

Xinyan Wang

# Introduction

To understand this specification and the program, readers are assumed to have the fundamental knowledge about MDP and Q-learning as well as the basic concept like V value, Q value, 𝛾 (discount rate), exploiting, exploring, etc. If not, it is highly recommended to take a look at Berkeley’s CS 188 course materials [1].

This document serves as a technical specification and user guidance for the C++ program. The complete script contains the files shown in Figure 1. In a word, the program has an *environment* (section 2.1) and a *Learningstrategy* (section 2.2). The *Learningstrategy*, which can be markov decision process (MDP) agent or Q-learning agent in the current version,will look for the optimal scenarios in the *environment* under given circumstances. Details of these files and the overall architecture of the program will be covered in section 2.

Graphical user interface, text, application

Description automatically generated

Figure 1 Files of the Decision-Making Program

# Software Architecture

As stated in Section 1, in the program, the *Learningstrategy* will look for the optimal operation scenarios in the *environment*.

## Environment

All the *environments* will be subclasses of the abstract base class defined in “env\_base.h/cpp”. The base class of environments, *env\_base*, is defined in env\_base.h and env\_base.cpp. *env\_base* only defines the interface of the environment class, but the functions are not implemented (i.e. it is an abstract class). In principle, the *Learningstrategies* will be compatible with all *environments* following the interface defined in env\_base (I.e. subclasses of env\_base). Thus, users are free to implement new environments as subclasses of env\_base upon their requests. Again, these user-implemented environments should also be compatible with the MDP and Q-learning agent as long as the appropriate interface is followed. As an example, *env\_gridworld* implemented in “env\_gridworld.h/cpp” is a subclass of *env\_base*’s. The architecture is shown in Figure 2. Note that only the important functions are shown in the diagram for simplicity. Helper functions and debugging functions (e.g. functions to print Q values) are not shown. Check the corresponding .h and .cpp files for more details.

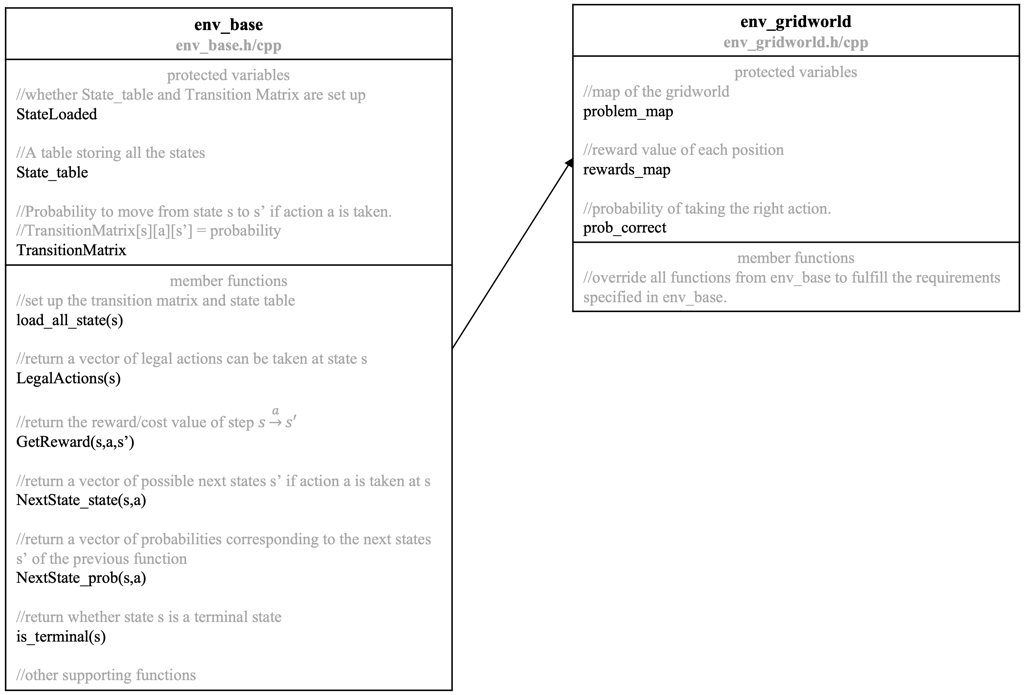


Figure 2 Environment Classes. Notations: s=current state. a=action. s’=next state.

## Action and State

A *state* and an *action* should be defined for each environment. For example, a state in gridworld describes the coordinate and an action in gridworld describes the direction to go. For class function parameter compatibility, two base classes, state\_base and action\_base, are defined in env\_state.h/cpp. The state and action of each environment should be defined as subclasses of state\_base and action\_base as shown in Figure 2. Pointers of the base classes (action\_base\* and state\_base\*) are used as parameters and return types of any functions so as to be compatible with the program’s interface.

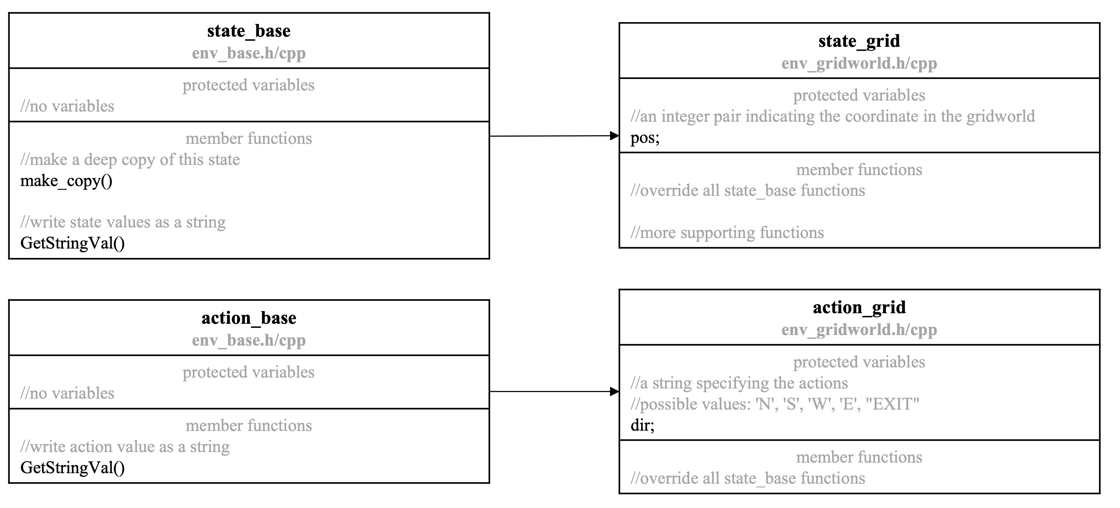


Figure 2 State and Action Classes

## Learningstrategy classes and subclasses

Just like the *environments*, the *Learningstrategies*, so far the MDP agent implemented in “learningstrategy\_mdp.h/cpp” and Q-learning agent in “learningstrategy\_qlearning.h/cpp”, are defined as subclasses of a base class *learningstrategy\_mqbase* implemented in “learningstrategy\_mqbase.h/cpp”, which ensures that the interfaces of all these *Learningstrategies* are identical. Note that, unlike *env\_base*, *learningstrategy\_mqbase* is not a pure abstract class. Most of its functions are IMPLEMENTED because they are used in both the MDP agent and the Q-learning agent. I.e. *learningstrategy\_mqbase* is not a standalone usable class, but it provides functions for the MDP agent and the Q-learning agent.

Their relationship diagram is shown in Figure 3. Note that only the important functions are shown in the diagram for simplicity. Helper functions and debugging functions. Check the corresponding .h/cpp files for more details.

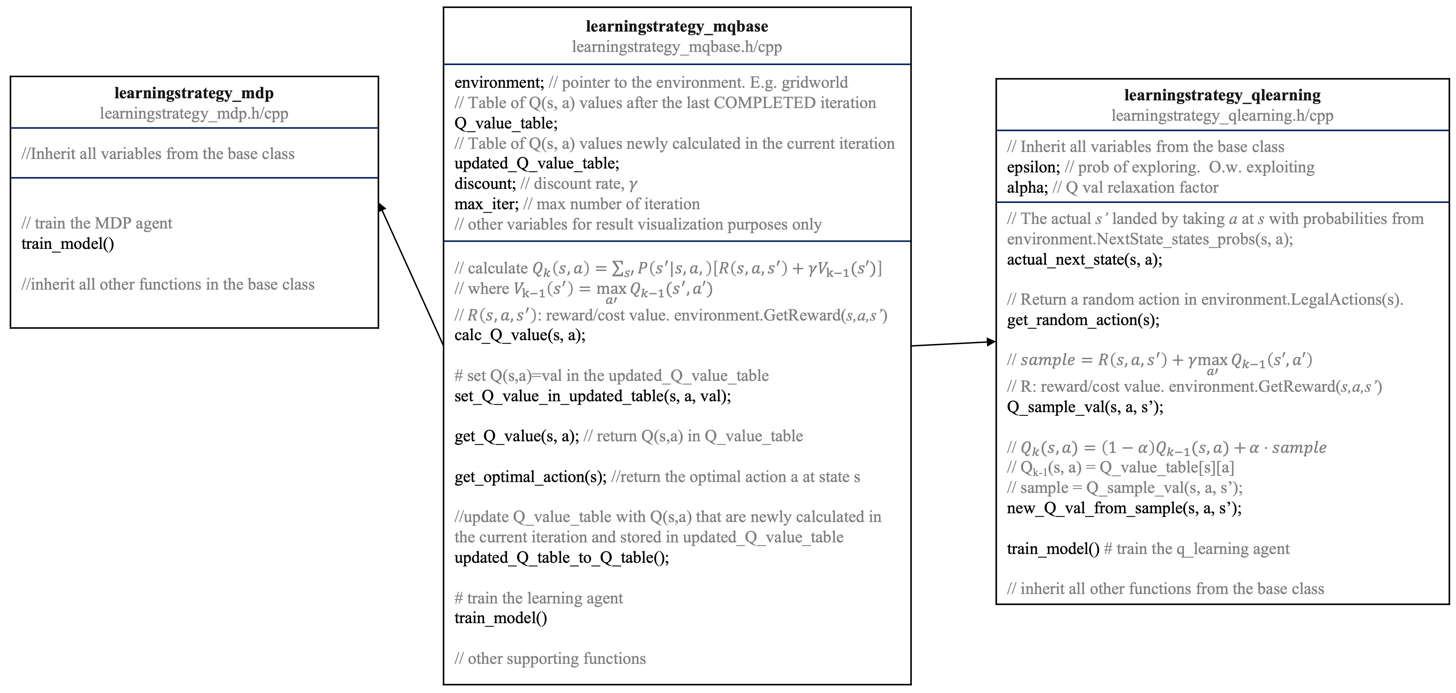


Figure 3 Learningstrategy Classes. Notations: s=current state. a=action. s’=next state.

Also note that *environment* is a variable in *learningstrategy*. The optimal operation scenario will be found using the member function train\_model(), which will be introduced in section 2.3 and 2.4.

## MDP

Detailed mathematics and principles of MDP can be found in Berkeley’s CS 188 course materials [1]. This chapter will focus on its implementation in the program. The pseudo code is shown in Figure 4. The Q value for each state and action is updated iteratively until converge.

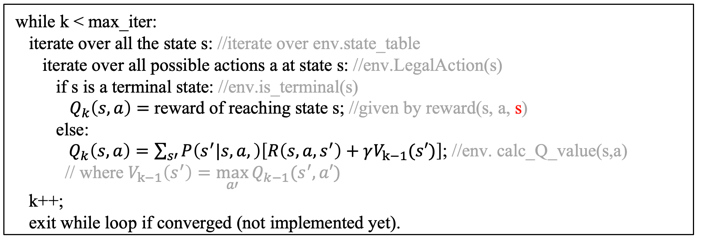


Figure 4 Pseudo code of MDP’s train\_model()

In the Q value equation

|  |  |
| --- | --- |
|  | ( 1 ) |

*k* is the iteration episode; *s* is the current state, *a* is an available action at state *s*, is the probability of moving to state *s’* when taking action *a* at state *s* (i.e. ), is the reward/cost value of taking action *a* at state *s* and moving to state *s’*. is the discount factor. is the V value, which is the maximum Q value among all possible actions (), at *s’*.

## Q-learning

Detailed mathematics and principles of Q-learning can also be found in Berkeley’s CS 188 course materials [1]. Logically, the main difference between MDP and Q-learning is that, unlike MDP, which requires knowing all the states in advance to iterate over them, Q-learning can start the model training without knowing the entire picture of the problem. Q-learning agent will learn the environment online while performing the model learning.

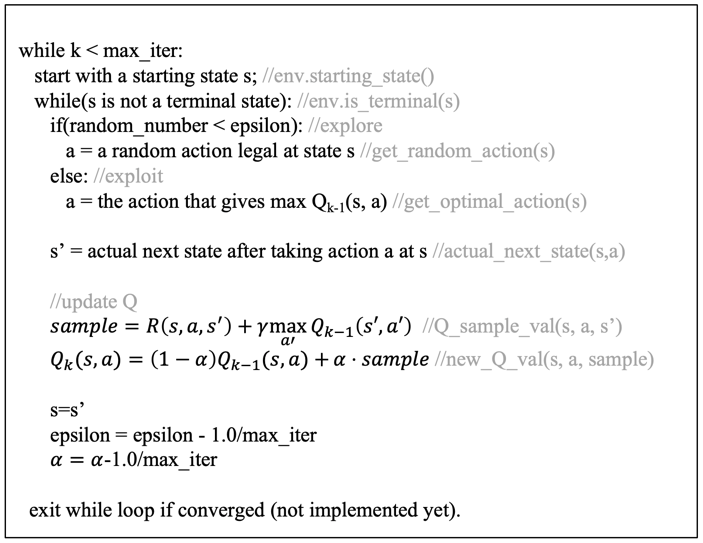


Figure 5 Pseudo code of Q-Learning’s train\_model()

Pseudo code of the Q-learning agent in the program is shown in Figure 5. The program starts at a starting state *s0*, takes an action *a0*, updates the Q value Qk(s0, a0), then goes to the next state *s1*, and repeats the process for (*s1*, *a1*), (*s2*, *a2*), … until reaches a terminal state.

When choosing the action *a* to take, there are two approaches: 1) randomly choose an action in the candidate action pool (exploration) and 2) pick the action with the highest Q value in the current Q\_value\_table (exploitation). As shown in is shown in Figure 5, the approach to adopt is determined stochastically, and the probability is determined by the parameter *epsilon*. Usually, *epsilon* starts to be 1, gradually decreases and finally reaches 0 at the end of the iteration. The net effect is that, at the beginning of the learning process when the Q-learning agent knows very little about the environment, the agent tends to take random actions to explore the environment. As iterations go, the agent will tend to converge the Q values along the optimal scenarios. More details can be found in [1].

As for Q-value update, equation ( 1 ) can be used in Q-learning, but there is another way as shown in equation ( 2 ).

|  |  |
| --- | --- |
|  | ( 2 ) |

# Demonstration Using Gridworld

Let’s use the gridworld example to demonstrate the performance of the two algorithms. As shown in Figure 6, the agent starts at the bottom left corner and ends at either +1 or -1 with the corresponding reward. At each step, the agent has 4 possible actions: north (N), south (S), west (W) and east (E). To make the rule more general, each action only has a probability of 80% to be executed correctly and 20% slip to the sides. For example, if the action is E, the agent has 80% going to the east, 10% to the north and 10% to the south. Also, if the agent hit the wall or the solid block, it will stay at where it was. Discount rate is 0.95 and costs to take an action is 0.01.

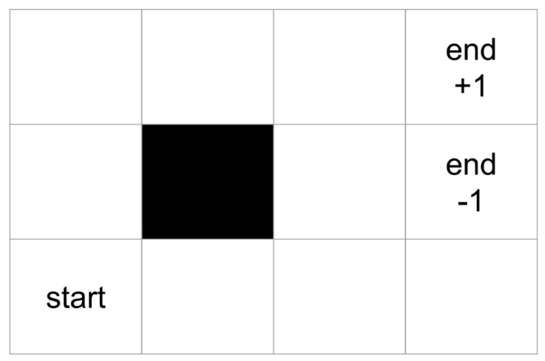


Figure 6 Gridworld Map

Results are shown in Table I and

Table *II*. There are some interesting conclusions

1. Both the agents give the correct optimal scenario: N->N->E->E->1 even with the lowest max\_iter setting.
2. MDP even suggested the correct actions to take for states not on the optimal scenarios. Q-learning failed on this task. The reason could be that the epsilon decay approach makes the Q-learning agent concentrate more on the optimal scenario less likely to explore states off the optimal scenario.
3. 2) is more obvious when looking at the Q-value map.
4. For the same number of iterations, Q-learning takes less than half the time taken by MDP.

Table I Gridworld Performance: MDP

|  |  |  |  |
| --- | --- | --- | --- |
| Max\_iter | Optimal Action Map | Q value map | Time consuming |
| 500 |  |  | 0.10 s |
| 1000 |  |  | 0.19 s |
| 2000 |  |  | 0.41 s |
| 3000 |  |  | 0.54 s |
| 5000 |  |  | 0.92 s |

Table II Gridworld Performance: Q-learning

|  |  |  |  |
| --- | --- | --- | --- |
| Max\_iter | Optimal Action Map | Q value map | Time consuming |
| 500 |  |  | 0.04 s |
| 1000 |  |  | 0.07 s |
| 2000 |  |  | 0.15 s |
| 3000 |  |  | 0.20 s |
| 5000 |  |  | 0.31 s |

# Bibliography

|  |  |
| --- | --- |
| [1] | UC Berkeley, "UC Berkeley CS188 Intro to AI -- Course Materials," [Online]. Available: http://ai.berkeley.edu/lecture\_videos.html. |
| [2] | Google Inc., "What is Colaboratory," [Online]. Available: https://colab.research.google.com/?utm\_source=scs-index#scrollTo=5fCEDCU\_qrC0. |
| [3] | FMPy Group, "FMPy," [Online]. Available: https://fmpy.readthedocs.io/en/latest/. |
| [4] | FMPy Group, "FMPy Tutorial," [Online]. Available: https://fmpy.readthedocs.io/en/latest/tutorial/. |