Software framework for learning algorithms

This is my part of a machine learning project at UNSW (Sydney) in 2011. The aim was to use reinforcement learning for a traffic simulator. I have extracted my code and am providing it under the **GNU General Public License (GPL)**.

The following documentation is extracted from the final report at UNSW.

The source code provides a framework for algorithms solving Markov Decision Processes[1] and Reinforcement Learning[2]. The key for successfully learning a usable policy is an appropriate definition of the state base and the reward function.

The software framework allows for the following:

1. Easy testing of new state bases and reward functions without having to adapt the learning code, and with minimal changes to the rest of the code. For this, a modular software structure allowing for easy and flexible adaptation/extension of the code to include new state base descriptions and reward functions is important.
2. Implementations for Q-Learning, value and policy iteration [2]. The software framework allows for easy exchange of the learning algorithm on the same domain, with minimal adaptations.

Software design

The most important aspects to capture in the software design are

1. Modular separation of the *domain description* (e.g. traffic system) and the *learning algorithms* (q-learning, value and policy iteration)
2. Modular independence of the *learning algorithms* to specific *states*, *actions*, *reward* and *transition* functions (where transition functions are only needed for value and policy iteration)

To achieve this, I designed the following software architecture. Only the most important base classes are depicted in Figure 1, and an example usage of the class structure is illustrated for a example domain instantiation, simply called *ExampleDomain1 / LightAction / State1 / Reward1 / Transition1* in Figure 1.

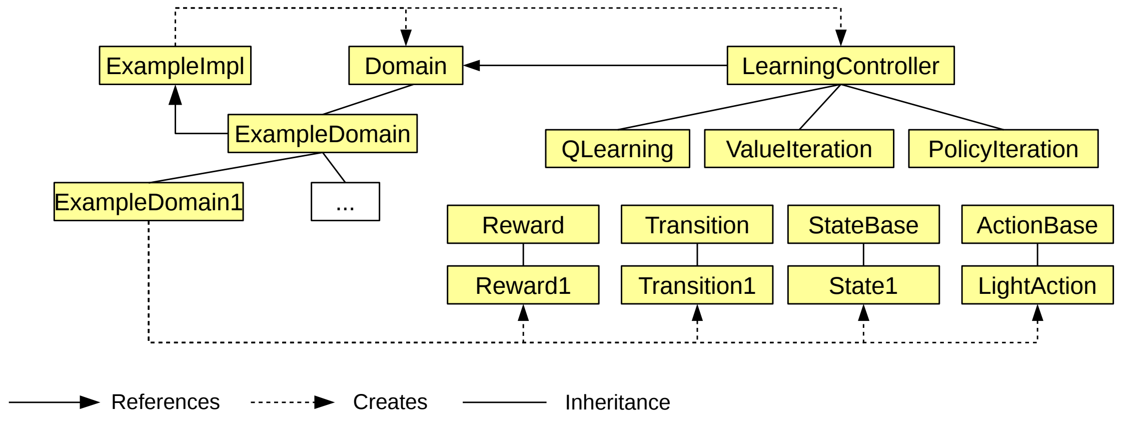

Figure 1: Basic class structures and relations. For the sake of clarity, only the most important classes and the most relevant relations are depicted here.

To provide a new state description for the example domain, and assign a suitable reward (and transition) function for it, one only has to implement a class derived from *ExampleDomain*, which already provides a reference to *ExampleImpl*[3]. New *StateBase* implementations can contain any information about the domain (e.g. distance of cars to the traffic light). A *StateBase* derived class only has to provide a comparison operator which allows this object to be used as a key for a map.

1 For example using algorithms value iteration and policy iteration to solve the problem
2 For example using Q-Learning
3 *ExampleImpl* is only an example class containing specific implementations for an example scenario. It provides all relevant values to instantiate a state, e.g. *State1*.

The *Reward* (e.g. *Reward1*) implementation has to “know” which specific *StateBase* (e.g. *State1*) implementation is used. The *ExampleDomain* implementation, e.g. *ExampleDomain1*, ensures that a suitable combination of *StateBase* and *Reward* implementations is instantiated.
Similar constraints apply for *Transition*, which provides a transition function for value and policy iteration. It is to be instantiated with a suitable combination of *ActionBase* and *StateBase* implementations.

The particular state/reward implementation to be used is loaded at initialisation time, when the appropriate *TrafficDomain* object is instantiated (e.g. *ExampleDomain1*).
The binding of the world code (contained in classes like *ExampleImpl*) and the particular learning algorithm is achieved in the following way:

- For **offline learning** processes as **value and policy iteration**, the learning is performed at initialisation time of *ValueIteration* and *PolicyIteration*. Following calls to *LearningController::updateAndGetAction()* will return the optimal action to be taken, given the policy learned. This can be used to update the world.
- For **online learning** processes as **Q-Learning**, *LearningController::updateAndGetAction()* will update the q-table and then return the action which the learner wants to be tried in the world.

The *LearningController* receives proper instantiations of *StateBase*, *ActionBase*, *Reward* and *Transition* from its reference to the *Domain* implementation and is therefore loosely coupled to the particular domain used.

More details about the software structure are out of scope of this report. The code is well documented, and a reference manual is provided (see doc/html/index.html), where more details can be looked up.

Implementation of learning algorithms and testing

To validate a correct implementation of the learning methods, I tested on the well-known **2D Grid World** [2]. This test scenario is good because results for utility values and optimal policies are provided in [2].

			G
	B		P
S			

The example grid world. G=Goal; Block=1; P=Pit; S=Start.

First, value and policy iteration were implemented as described in [2], and the correct results were validated on the grid world. Then, Q-Learning was implemented, and it was to make sure that it converges to the same policy as with value and policy iteration, given the same discount factor.
Convergence of q-learning is only ensured given a decaying learning rate (see [3]). Testing on the grid world, I saw that the simple learning rate decay as suggested in [3] (Formula 1 below) did not converge as well as another implementation (Formula 2 below).

$\frac{1}{f(s,a)}$	$\frac{1}{1.0+d*f(s,a)}$
Formula 1	Formula 2
where *f(s,a)* is the frequency that action a was tried from state s, and d is a constant which we found to be best at d=0.1	

I validated the Q-Learning convergence by picking “borderline” states where the policy would most often vary between different trials. In the grid world, these are for example the coordinates (2,1) and (3,0), which are the fields below and to the left of the pit. Because convergence is only ensured if the action is tried an infinite number of times form a state (see [3]), I chose initially a high number of iterations (300.000), to find the values which must be near to the correct ones. I ran the tests with different learn rate decay factors, and found decay factor *d=0.1* (see Formula 2) to perform best. Using this decay factor, I ran the experiment several times with decreasing number of iterations, to see when the learned policy differs.
I found that the policy starts to diverge from the optimal one at approximately 10.000 iterations and less (see experimental results in Appendix A). The effects of the probabilistic environment, and the randomness in the order the state-action pairs are tried, can be observed in the diverging q-values. Still, with 10.000 iterations, mostly the q-values are still quite similar to the near correct ones.
In comparison, it is important to note that **value iteration** needs only **33 iterations** through all states[4] to converge, and **policy iteration** only requires **6 trials[5]** before no changes are made to the policy.

4 with each iteration visiting 4*3-3=9 states (3 substracted due to pit, goal and block), in total this makes an equivalent of 33*9=297 state visits. This is using maximum error of 0.01 (as stated in [2])

5 A “Trial” is a movement form a random field to the goal

Appendix A

Experiments on Q-Learning convergence of the grid-world.
Learn rate decay factor d=0.1, discount factor =1.0 , side action probability p=0.1.

Correct policy to be learned:	*Run3, with 10.000 iterations, yielded in a different policy:*
0/0 -> UP 0/1 -> UP 0/2 -> RIGHT 1/0 -> LEFT 1/2 -> RIGHT 2/0 -> LEFT 2/1 -> UP 2/2 -> RIGHT 3/0 -> LEFT	0/0 -> UP 0/1 -> UP 0/2 -> RIGHT 1/0 -> LEFT 1/2 -> RIGHT 2/0 -> LEFT 2/1 -> LEFT 2/2 -> RIGHT 3/0 -> LEFT

Run 1	*Run 2*	*Run 3*
Test 1: Number of iterations: 300.000		
2/1 / Action=RIGHT, value=-0.61691 2/1 / Action=UP, value=0.700428 2/1 / Action=DOWN, value=0.44332 2/1 / Action=LEFT, value=0.67854 3/0 / Action=RIGHT, value=0.242582 3/0 / Action=UP, value=-0.666171 3/0 / Action=DOWN, value=0.4114 3/0 / Action=LEFT, value=0.431039	2/1 / Action=RIGHT, value=-0.6031 2/1 / Action=UP, value=0.696475 2/1 / Action=DOWN, value=0.43763 2/1 / Action=LEFT, value=0.675103 3/0 / Action=RIGHT, value=0.259966 3/0 / Action=UP, value=-0.671923 3/0 / Action=DOWN, value=0.409652 3/0 / Action=LEFT, value=0.429401	2/1 / Action=RIGHT, value=-0.596767 2/1 / Action=UP, value=0.701408 2/1 / Action=DOWN, value=0.459116 2/1 / Action=LEFT, value=0.684414 3/0 / Action=RIGHT, value=0.24542 3/0 / Action=UP, value=-0.710637 3/0 / Action=DOWN, value=0.407272 3/0 / Action=LEFT, value=0.42936
Test 2: Number of iterations: 200.000		
2/1 / Action=RIGHT, value=-0.603603 2/1 / Action=UP, value=0.700898 2/1 / Action=DOWN, value=0.452154 2/1 / Action=LEFT, value=0.683523 3/0 / Action=RIGHT, value=0.242803 3/0 / Action=UP, value=-0.644478 3/0 / Action=DOWN, value=0.41244 3/0 / Action=LEFT, value=0.441054	2/1 / Action=RIGHT, value=-0.606982 2/1 / Action=UP, value=0.690878 2/1 / Action=DOWN, value=0.509092 2/1 / Action=LEFT, value=0.6796 3/0 / Action=RIGHT, value=0.277236 3/0 / Action=UP, value=-0.659021 3/0 / Action=DOWN, value=0.42002 3/0 / Action=LEFT, value=0.423536	2/1 / Action=RIGHT, value=-0.621651 2/1 / Action=UP, value=0.691403 2/1 / Action=DOWN, value=0.443008 2/1 / Action=LEFT, value=0.673918 3/0 / Action=RIGHT, value=0.289255 3/0 / Action=UP, value=-0.760101 3/0 / Action=DOWN, value=0.410105 3/0 / Action=LEFT, value=0.431157
Test 3: umber of iterations: 100.000		
2/1 / Action=RIGHT, value=-0.55787 2/1 / Action=UP, value=0.701916 2/1 / Action=DOWN, value=0.437469 2/1 / Action=LEFT, value=0.669188 3/0 / Action=RIGHT, value=0.266779 3/0 / Action=UP, value=-0.757405 3/0 / Action=DOWN, value=0.415866 3/0 / Action=LEFT, value=0.42815	2/1 / Action=RIGHT, value=-0.672489 2/1 / Action=UP, value=0.713795 2/1 / Action=DOWN, value=0.40463 2/1 / Action=LEFT, value=0.64069 3/0 / Action=RIGHT, value=0.215733 3/0 / Action=UP, value=-0.718577 3/0 / Action=DOWN, value=0.400838 3/0 / Action=LEFT, value=0.432146	2/1 / Action=RIGHT, value=-0.596315 2/1 / Action=UP, value=0.685427 2/1 / Action=DOWN, value=0.46593 2/1 / Action=LEFT, value=0.655376 3/0 / Action=RIGHT, value=0.261904 3/0 / Action=UP, value=-0.707368 3/0 / Action=DOWN, value=0.395986 3/0 / Action=LEFT, value=0.423012
Test 4: number of iterations: 50.000		
2/1 / Action=RIGHT, value=-0.728668 2/1 / Action=UP, value=0.694098 2/1 / Action=DOWN, value=0.410144 2/1 / Action=LEFT, value=0.672189 3/0 / Action=RIGHT, value=0.20414 3/0 / Action=UP, value=-0.742749 3/0 / Action=DOWN, value=0.414026 3/0 / Action=LEFT, value=0.428987	2/1 / Action=RIGHT, value=-0.667897 2/1 / Action=UP, value=0.700751 2/1 / Action=DOWN, value=0.424069 2/1 / Action=LEFT, value=0.633169 3/0 / Action=RIGHT, value=0.20577 3/0 / Action=UP, value=-0.689676 3/0 / Action=DOWN, value=0.393846 3/0 / Action=LEFT, value=0.452481	2/1 / Action=RIGHT, value=-0.732175 2/1 / Action=UP, value=0.721272 2/1 / Action=DOWN, value=0.348726 2/1 / Action=LEFT, value=0.632896 3/0 / Action=RIGHT, value=0.264696 3/0 / Action=UP, value=-0.650568 3/0 / Action=DOWN, value=0.404225 3/0 / Action=LEFT, value=0.432783

Test 5: number of iterations: 10.000		
2/1 / Action=RIGHT, value=-0.729659 2/1 / Action=UP, value=0.705003 2/1 / Action=DOWN, value=0.327501 2/1 / Action=LEFT, value=0.582847 3/0 / Action=RIGHT, value=0.359729 3/0 / Action=UP, value=-0.605662 3/0 / Action=DOWN, value=0.328291 3/0 / Action=LEFT, value=0.436908	2/1 / Action=RIGHT, value=-0.685865 2/1 / Action=UP, value=0.691416 2/1 / Action=DOWN, value=0.44885 2/1 / Action=LEFT, value=0.607354 3/0 / Action=RIGHT, value=0.249617 3/0 / Action=UP, value=-0.569884 3/0 / Action=DOWN, value=0.32236 3/0 / Action=LEFT, value=0.419178	*2/1 / Action=RIGHT, value=-0.694072* *2/1 / Action=UP, value=0.595223* *2/1 / Action=DOWN, value=0.372494* *2/1 / Action=LEFT, value=0.603586* ***3/0 / Action=RIGHT, value=0.0690025*** ***3/0 / Action=UP, value=-0.823245*** *3/0 / Action=DOWN, value=0.358045* *3/0 / Action=LEFT, value=0.427999*

References

[1] Mitchell T.: *Machine Learning*, McGraw Hill (http://www.cs.cmu.edu/~tom/mlbook.html), 1997

[2] Russel/Norwig: *Artificial Intelligence: A Modern Approach*, Prentice Hall, Second edition, 2003 (http://www.cs.berkeley.edu/~russell/aima.html)

[3] Website provided to the PhD thesis of Mark Humphrys, Chapter on Reinforcement Learning, (http://www.compapp.dcu.ie/~humphrys/PhD/ch2.html), 1997