Project Whitepaper: Reinforcement Learning Language

Michael Groble (michael.groble@motorola.com)

September 24, 2006

1 Problem Description

The language I want to develop for this project is one to specify and simulate various reinforcement learning algorithms. Reinforcement learning is described in [SB98]. The language will initially focus on the application of Reinforcement Learning to finite Markov Decision Processes (MDPs). Briefly, a finite MDP is specified by the tuple $< \mathcal{S}, \mathcal{A}_s, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'} >$ where $\mathcal{S}$ is the finite state space, $\mathcal{A}_s$ is the finite action space (in general, the allowable actions depends on the current state $s \in \mathcal{S}$, $a \in \mathcal{A}_s$), $\mathcal{P}^a_{ss'}$ is the environment transition probabilities

$$\mathcal{P}^a_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$$

in other words, the probability that a certain state will be reached (s') as a result of taking a specific action (a) in a specific starting state (s) and finally $\mathcal{R}^a_{ss'}$ are the expected action rewards

$$\mathcal{R}^a_{ss'} = E[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s']$$

in other words, the expectation of the reward (r_{t+1}) as a result of taking action (a) in a specific starting state (s) which results in a specific subsequent state (s').

The reinforcement learning problem is that of finding a policy π which in some sense optimizes our reward. Problems can be categorized into ones with finite ends (termed *episodic*) and those without end (*continuous*). The language will support the typical reward formulation, one of maximizing the expected discounted future rewards. In other words, the policy attempts to maximize the *return* R_t where

$$R_t = E\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1}\right]$$

The term $0 < \gamma \leq 1$ is called a discount factor (it discounts future rewards). For continuous tasks, $T = \infty$ and the discount factor must be less than 1 to

ensure the return is bounded. For episodic tasks, the termination of the episode corresponds to reaching a particular subset of the state space. These terminal states are not considered part of $\mathcal{S}$. The set $\mathcal{S}^+$is used to denote the union of $\mathcal{S}$ and the terminal states. Technically in the definition of $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ above, $s \in \mathcal{S}$ and $s' \in \mathcal{S}^+$.

The purpose of the language is to specify finite MDP problems and the solution and simulation of both episodic and continuous problems using standard reinforcement learning algorithms.

2 Sample Input

This section contains some examples of language inputs. The first two MDP examples come from [SB98] and the last one comes from [TBF05]. Note the examples show a Python-like use of whitespace in a seemingly semantic way. I don't bother to try to define the semantics since this is just candidate notation. I haven't thought much about how to really identify different scoping areas.

2.1 Gambler's Problem

The first case is a simple gambling game where the probability of winning any particular turn is 0.4. The objective is to acquire a total amount of exactly 100. On each turn, the gambler must bet at least 1 and at most their total holdings (up to the amount that would give them 100 if they were to win). The learning problem is to determine the optimal gambling policy, in other words what amount should be bet for each amount the gambler is holding.

```
1   mdp Gamblers
2       states capital
3       actions stake
4       reward  r
5
6       valid states { 0 < capital < 100 }
7
8       episode
9           set loseStates { capital == 0 }
10          set winStates  { capital == 100 }
11          terminals { winStates, loseStates }
12
13      valid actions {0 < stake <= min(capital, 100 - capital)}
14
15      environment
16          random win = binomial(0.4)
17
18          capital' = capital + 2 * (win - 0.5) * stake
19          r = 1 if capital' in winStates else 0
```

Lines 2 through 4 define the names used to represent states, actions and reward. Line 6 defines the valid states space $\mathcal{S}$. In this case, the problem is episodic. Lines 8 through 11 define variables related to the episodic behavior. Lines 9 and 10 define named sets that are used in the scope of an episode. Elements of the set can be enumerated or defined by constraints. For example, the following are equivalent

```
set winStates  { capital == 100 }
set winStates  { capital in {100} }
```

where `{100}` is a set literal consisting of the single element 100. The following are also equivalent

```
valid states { 0 < capital < 100 }
valid states { capital in {1:99} }
```

In this case, the set literal $\{1:99\}$ uses the range operator to represent the 99 integers between 1 and 99 inclusive.

Line 11 denotes the states which are terminal states. Line 13 defines the valid actions $\mathcal{A}_s$. This example shows a case where the set of valid actions does depend on the current state. Finally, lines 15 through 19 define the environment behavior. Logically, the system can be divided between an agent and the environment. The agent observes the current state and reward and chooses an action. The environment propagates the old state to a new state based on the chosen action and computes the reward. These lines describe how the environment updates for this problem.

Line 16 defines a random variable within the scope of the environment. In this particular case, the random variable is drawn from a binomial distribution. Since it is defined in the scope of the environment, a new value is drawn at each turn. Line 18 shows how capital is updated and line 19 shows how the reward is computed. The last line also shows a Python-like conditional statement of the form $<$ `true` $-$ `value` $>$ `if` $<$ `condition` $>$ `else` $<$ `false` $-$ `value` $>$. It also shows the syntax for set mebership predicate $<$ `value` $>$ `in` $<$ `set` $>$.

2.2 Jack's Car Rental

This next example is a car rental company with two sites. People stop at each site to request rental cars and return cars. At night, the owner has the discretion of transferring up to 5 cars from one site to another. In this case the policy we want to learn is how many cars to transfer given the number of cars remaining at each location. For simplicity, cars at each location are limited to 20. Any transferred or returned over 20 effectively disappear. Each rental provides a reward of +10 and each car transfer incurs a cost of 2 (provides a reward of -2).

```
1   mdp CarRental
2       states cars[2]
3       actions netTransfer
```

```
4       reward  r
5       valid states  { 0 <= cars <= 20 }
6       valid actions { -min(5,cars[1]) < netTransfer < min(5,cars[0]) }
7       environment
8           random returns  = [poisson(3);poisson(2)]
9           random requests = [poisson(3);poisson(4)]
10          nextMorn = min( cars + [-netTransfer;netTransfer], 20)
11          rentals = min(requests, nextMorn)
12          cars' = min(nextMorn + returns - rentals, 20)
13          r = rentals * 10 - 2 * abs(netTransfer)
```

The first thing to notice about this case is that the state is no longer a scalar, but is a vector. The notation for vector quantities and operations match that of Matlab in spirit. Line 2 declares the state element cars as a vector with two elements (one for each rental location). Line 3 defines the action as a net transfer, the number of cars transferred from location 1 to location 2 (a negative value indicates transfer from 2 to 1).

The next thing to notice about this case is that it is continuous, not episodic, so there is no episodic section.

The environment section shows the use of temporary variables (`nextMorn` and `rentals`) to simplify the calculation of state updates and rewards.

2.3 Heaven or Hell

This is an example of a "gridworld", an environment where the agent's actions allow it to move about a 2-dimensional grid. In this case, there is one start position, two terminal positions (Heaven and Hell) and one Map position. The reward for Heaven is +100 while the reward for Hell is -100. The act of moving has a reward of -1. Two locations in the world are fixed for Heaven and Hell, but the agent cannot tell which is which. Each episode has Heaven and Hell randomly assigned to the two locations. If the agent enters the Map location, it is told which location is Heaven.

```
1   mdp HH
2       states position[2]
3       states map
4       action direction
5       diagram
6           A     B
7           .......
8              .
9              .
10             S
11             .
12             .
13             ...M
```

```
14
15      valid states {position in {.,S,M} &&
16                    map in {unknown, a, b}}
17
18      valid actions {n,s,e,w}
19
20      episode
21          starts {position = S && map = unknown}
22          terminals {position in {A,B}}
23          random aIsHeaven = binomial(0.5)
24          set heaven {position in {A} and aIsHeaven ||
25                      position in {B} and !aIsHeaven}
26          set hell {terminals - heaven}
27
28      environment
29          next = [position[0] + (1 if action == w else
30                                 (-1 if action == e else 0));
31                  position[1] + (1 if action == n else
32                                 (-1 if action == s else 0))]
33          position' = next if next in {valid states, terminals}
34                          else position
35          map' = map if position not in {M}
36                     else (a if aIsHeaven else b)
37          reward = 99 if position in heaven
38                      else (-101 if position in hell else -1)
```

This example shows quite a few new aspects of the language. First is that states can be structured. In this case states are represented by a vector component (position) and a scalar component (map). Second is that states and actions can consist of sets of nominal named values rather than integer values. The map state can have the values "unknown" (meaning the location of Heaven is unknown), "a" or "b". Similarly, the action in line 18 is movement in one of the compass directions "n", "s", "e" or "w".

Third, shown in lines 5 through 13, is a potential shorthand notation to ease creating gridworlds by depicting them with ASCII diagrams. Lines 15 and 16 map the ASCII notations onto the "normal" constraint notation. Logically, each character in the ASCII diagram represents the set of positions in the diagram where that character is found. So on line 15, the term {.,S,M} means the set of 16 locations matching those three character annotations.

Lines 20 through 26 show an episode section. In this case, a random variable is declared in the episode section, it is therefore drawn at the beginning of each episode and remains constant during the turns within an episode. Line 21 also shows a new special "starts" set which defines the set of states which are initial states in the episode. A missing "starts" declaration means the start states are unconstrained and consist of all "valid states".

Lines 29 through 32 translate the motion names to potential movement while lines 33 and 34 ensure the agent cannot move out of the valid positions. Lines 35 and 36 update the map state if agent is in the Map location.

2.4 Solution Language

The previous three sections described sample inputs for problem descriptions. We also need a language for solving the problems. For now, I plan on having a limited number of hard-coded algorithms available rather than specifying a language for defining algorithms. The invocation of those algorithms will be fairly Matlab-like:

```
// determines policy for the HH mdp using policy iteration
policy = policyIteration(mdp = HH, threshold = 0.1)
// determines policy for Gamblers using value iteration
policy = valueIteration(mdp = Gamblers, threshold = 0.1)
// determines policy for Racetrack using Sarsa(lambda)
[policy,returns] = sarsa(mdp = Racetrack, lambda = 0.6,
                         epsilon = 0.01, episodes = 5000)
```

Once policies are learned, they can be simulated for evaluation purposes

```
[states,actions,rewards] = simulateSteps(policy = p, steps = 100)
[states,actions,rewards] = simulateEpisode(policy = p)
```

My end-goal is to provide the ability to specify different algorithms in the language itself and to specify concepts of Programmable Reinforcement Learning [AR00] in the language, but those will likely not be done in the scope of this course.

3 Issues

My main language design issue I have is the avoidance of procedural statements in the environment specification. Some algorithms require that I be able to "compile" the environment descriptions into explicit values for the probabilities $\mathcal{P}^a_{ss'}$ and expected values $\mathcal{R}^a_{ss'}$ across the entire state-action space while others rely on Monte Carlo simulation of the environment. My current belief is that the language therefore needs to be as declarative as possible and as devoid of procedural "control flow" statements as I can make it. The conditional operator is my only concession and enforces my belief that I need a value for every "branch".

My other big issue is related to the interaction of the structured state representation and the set notation. Imagine a gridworld with dynamics where states are `position`[2], `velocity`[2] and actions are `acceleration`[2]. Also imagine some sets that define special positions, for example

```
set locations = {...}
```

On the one hand, I'd like to provide error checking to ensure sets are being compared correctly. It should make sense to say

```
position in locations
```

but not

```
velocity in locations
```

or even worse, comparing an action with a set defined on states

```
acceleration in locations
```

On the other hand, you could argue something like

```
velocity in locations
```

is valid and should evaluate to "true" (since `velocity` is unconstrained by `locations`).

I wonder if sets need some (implied or explicit) type system to minimally distinguish between states and actions and possibly distinguish between structured states.

4 Mechanics

I have prior experience building parsers using high-level tools (I am taking this course because I need to meet the core requirement and haven't formally studied languages before). I implemented a proprietary language used in Motorola for defining binary message formats. I extended the language to support comments similar to Javadoc and created a parser that would create FrameMaker interface documentation from the commented message definition files. The complete system was about 6000 lines of C++ using the Spirit parser library from Boost [dG]. I also created parsers for two proprietary languages used to specify dialog management systems. These parsers were part of Eclipse editors for those languages (with interactive syntax and semantic error annotation, hyperlinking and refactoring similar to the capabilities of JDT). The parser plugins were about 3000 lines of Java using Antlr.

I'd like to start with an architecture that matches what I did in my plugin work. For good or bad, I picked up on the convention used by the Antlr Eclipse plugin[Jue]. I define a ParserBehavior interface in Java and call that behavior from the Antlr code rather than put explicit logic in the .g files. For example

```
object:
    fo:OPAREN "form" {behavior.beginForm();}
         form_object_guts
```

```
        fe:EPAREN {behavior.endForm(fo,fe);}
    | mo:OPAREN "menu" {behavior.beginMenu();}
        menu_object_guts
        me:EPAREN {behavior.endMenu(mo,me);}
    ;
```

I constructed the abstract syntax representation within the ParserBehavior implementation. The abstract syntax was represented by EMF-generated code (this abstract syntax will be shared between the concrete textual syntax from the Antlr parsers and a graphical concrete syntax implemented by a separate diagram editor in Eclipse).

I like the separation of interface and implementation. I in fact wrote a few different ParserBehavior implementations for different functionality in the development environment only one of which constructed a full abstract syntax representation. But I found error handling a little tedious, for example

```
public void beginMenu() {
    Menu menu = FrameFactory.eINSTANCE.createMenu();
    currentObjectScope.push(menu);
    stack.push(menu);
}
public void endMenu(Token hoverStart,Token hoverEnd) {
    if (stack.peek() instanceof Menu) {
        Menu m = (Menu)stack.pop();
        currentObjectScope.pop();
        if (stack.peek() instanceof Frame) {
            ((Frame)stack.peek()).getObjects().put(m.getName(),m);
            updateHover(m,hoverStart,hoverEnd);
        }
        else {
          throw new UnexpectedParseBehaviorException(
                        stack.peek().toString());
        }
    }
    else {
        throw new UnexpectedParseBehaviorException(
                        stack.peek().toString());
    }
}
```

I'd appreciate any suggestions on how to maintain this separation without the burden of seemingly implementing the language twice, once in Antlr and again in the ParserBehavior.

I have already manually implemented a number of these MDPs and reinforcement learning algorithms in C++. I experimented with porting them to Java and they run at least ten times slower. As a consequence, I want to either

implement the parser in C++ or implement it in Java as something that generates C++ code. I am leaning towards the Java implementation just because I am more productive in Java and have quite a bit of Java Antlr experience.

References

[AR00] D. Andre and S. J. Russell, *Programmable Reinforcement Learning Agents*, Proceedings of Neural Information Processing Systems, 2000, pp. 1019–1025.

[dG] Joel de Guzman, *The Spirit Parser Library*, http://spirit.sourceforge.net/.

[Jue] Torsten Juergeleit, *ANTLR plugin for Eclipse*, http://antlreclipse.sourceforge.net/.

[SB98] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, MA, 1998.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, The MIT Press, Cambridge, MA, 2005.