Report for Assessment 1
of CS3616

Joana Welti, Philipp Lucas

17th April 2011

Contents

1 Introduction

This paper describes the project implemented for the first assessment of the Artificial Intelligence For Computer Games course.

Our project is based on reinforced learning, a machine learning technique, which we used to implement the artificial intelligence for the dungeon game used in the lecture.

Our main achievements are the implementation of a generic reinforcement learner which allows to separate the learning process from the specific scenario and the design of a flexible configuration functionality, which enables users to pass parameters to the dungeon.

The project includes a new scenario, which is described in section 2. Sections 4 and 3 deal with the implementation and section5 explains how to run our scenario.

2 Scenario

In our scenario there is one fast ogre (let's call him John) in fraction "blue" and a variable number of slower creatures in fraction "yellow". The primary goal of the fast ogre is to escape from the dungeon by reaching the exit. For this it is not necessary to kill all the enemies however it might become necessary to fight against some of them as they block the way out.

The normal weapon used by John is the longbow, as this allows him to attack without being attacked himself. In addition he can decide to pickup and use potions to regain health or energy.

3 Reinforcement Learning

The learning techniques we decided to employ is reinforcement learning. Reinforced learning allows actors to learn what actions to take when being in a certain state based on rewards or penalties he got earlier on.

3.1 States

We use the following state variables, all of which are divided into 5 equidistant intervals:

- health points of John
- energy points of John
- distance between John and closest enemy

3.2 Actions

John can take the following actions:

- attack closest enemy

- evade close enemies
- escape to exit
- get and use closest health potion
- get and use closest energy potion

"closest" in this context means absolute distance and "not distance to walk".

"evade close enemies" lets John move directly away from the geometric center of all enemies that are within a certain radius from John.

3.3 Rewards

Tm gets the following rewards or penalties

Action	Reward
reached exit	+10
died	-10
hit enemy	+3
got hit by enemy	-3
kill enemy	+5
used potion	variable
changed state	0

"reached exit" and "died" Rewards
Reaching the exit and dying are the two ends of the spectrum, so John gets the biggest reward/penalty respectively.

"hit enemy" and "got hit by enemy" Rewards
Killing all the monsters is not the main objective, so John only gets a small reward for hitting any enemies.

"used potion" Rewards
The reward John gets for picking up an health potion depends on his current health. He gets the most points when his health level is low. As getting to the exit is the goal of the game, he gets a slight penalty when he picks up a potion although he's health level is good and he wouldn't need one. For health levels in between he gets a gradual amount of reward. With this reward system, we want to train John to pick up health potions when he needs them in order to win the game.

The same rules apply also for energy potions.

"changed state" Rewards
When John changes state but doesn't achieve anything listed above, he doesn't get any reward or penalty.

4 Implementation

4.1 Reinforcement Learner

Our implementation allows to separate the Q-Learner completely from the specific scenario. It is up to the client to define what attribute a state constitutes and what actions an actor can take, which makes our Q-Learner reusable and generic.

Qtable.java This class holds a Q-table to be used in reinforcement learning. As long as `Action.java` and `State.java` provide all necessary methods Q-Table is entirely independent on the actual actions or states. It provides a number of methods to update the qtable and retrieve actions:

- `static void updateTable (double reward, State oldState, State newState, Action oldAction)` updates the Q-table using the given parameters, i.e. in our case the previous state John was in, the current state John is in, the action that led to this state and the reward that is to be assigned for this.
- `static Action getRandomAction()` returns a random action, i.e. on of those defined in Action.java.
- `static Action getGreedyAction(State state)` returns to greediness % the best, otherwise a random action.

Action.java `Action.java` is a enumeration of all actions that can be taken. See the source code for a description of the methods.

To add a new action, add the name of that action to the enumeration and assign a reward for this action in `double getReward()`. Also add the actual action in your `doAction(Action action)` method in your behaviour class.

State.java `State.java` handles the different states needed for reinforcement learning. It provides a number of public methods:

- `boolean hasNotChanged(State state)` Returns true of this state and state share the same index.
- `int getMaxIndex()` Returns the number of possible different indices.
- `int getIndex()` Returns the index of this state, which is a number between (including) 0 and (excluding) `getMaxIndex()`.

To change the states, you only need to adapt the following:

- update `getMaxIndex()` accordingly.
- update `setIndex()` accordingly.

(currently the class has much more private variables. However, in fact there is no need (except for debugging) to store them as only the index matters: reading them from the game, then calculating the index and forgetting about all state variables after this again would be perfectly fine.)

4.1.1 Graphs

4.2 Configuration functionality

Our configuration functionality makes it possible for a scenario to have configuration options. The GUI provides a settings panel where the adjustable parameters are displayed and can be changed by the user during run time.

4.2.1 Usage

A scenario XML can have configuration options for at most one behaviour. In order to specify which settings should be loaded for a particular scenario, XMLs can include a `Configurations` element. The `type` attribute refers to the name of the configuration class that is to be used:

```
<Configurations Type="ReinforcementLearnerConfigurations"/>
```

In this example, the dungeon GUI would display configurations as specified in the `ReinforcementLearnerConfigurations` class. These configuration classes have to be located in the `dungeon.configurations` package.

4.2.2 Implementation

dungeon.configurations
Every behaviour that is customizable needs to have a class in this package that implements the abstract `Configurations` class.

- `Configurations`: Each parameter in a configuration has a key. The parameters of a configuration can be retrieved as a `HashMap`, where the key is used to access parameter values such as the default value, current value, min/max value possible etc.
- `ConfigurationHandler`: Loads configuration class as specified in the XML (type attribute of `Configurations` element) with the help of Java reflection.
- `ReinforcementLearnerConfigurations`: specific configurations for the reinforcement learning problem (initialization values for Q-Table, values to be used for Alpha/Discountfactor/Greediness)

dungeon.ui
- `ParameterPanel`: GUI element for the settings menu. Lists the configuration options of the specific `Configuration` class which is used in the game. Independent of the kind of settings used.

- `DungeonForm`: passes the configuration entered by the user to the game class every time the game is restarted. Displays the configuration option as spinners if the current scenario has any.

5 How to run the scenario

To run the scenario, start the game and load the *findExit.xml* file. Change any configuration values if needed and click on start.