Report for Assessment 1
of CS3616

Joana Welti, Philipp Lucas

14th April 2011

Contents

1 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 use potions to regain health or energy.

2 Reinforcement Learning

The learning techniques we decided to employ is reinforcement learning.

2.1 States

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

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

2.2 Actions

Tom 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 absolut distance and "not distance to walk".

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

2.3 Rewards

Tm gets the following rewards or penalties

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

"used potion": rewards depends on how much energy/health Tom regained doing this. TODO: explain in detail "changed state": if Tom changed state but didn't achieve anything of the above.

2.4 Implementation

We implemented the reinforcement learning in a rather generic way.

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 (exept 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.)

2.5 Graphs

3 How to run the scenario

To run the scenario just run the game and load *findExit.xml*.