Coursework 2 – Tic-Tac-To: Markov Decision Processes & Reinforcement Learning (worth 25% of your final mark)

Deadline: Thursday, 28th November 2024

How to Submit: To be submitted to **GitLab** *(via git commit* & *push*) – Commits are timestamped: **all *commits* after the deadline will be considered late.**

Introduction

Coursework 2 is an individual assignment, where you will each implement *Value Iteration*, *Policy Iteration* that plan/learn to play 3x3 Tic-Tac-Toe game. You will test your agents against other rule-based agents that are provided. You can also play against all the agents including your own agents to test them.

The Starter Code for this project is commented extensively to guide you, and includes Javadoc under `src/main/javadoc/` folder in the main project folder - you should read these carefully to learn to use the classes. This is comprised of the files below.

You should get the **Starter Code from GitLab:** Follow the step by step instructions in the document I have put together for you:

Open Canvas->F29AI -> Modules -> GitLab (and Git) Learning Materials (Videos and Crib Sheets) -> **Introduction to Eclipse, Git & GitLab**.

If you are **unfamiliar** with ***git*** *and/or* ***GitLab*** I **strongly** suggest watching Rob Stewart's **instructive videos on Canvas** under the same module

Files you will edit & submit

`ValueIterationAgent.java`	A Value Iteration agent for solving the Tic-Tac-Toe game with an assumed MDP model.
`PolicyIterationAgent.java`	A Policy Iteration agent for solving the Tic-Tac-Toe game with an assumed MDP model.
`QLearningAgent.java`	A q-learner, Reinforcement Learning agent for the Tic-Tac-Toe game.

Files you should read & use but shouldn't need to edit

`Game.java`	The 3x3 Tic-Tac-Toe game implementation.
`TTTMDP.java`	Defines the Tic-Tac-Toe MDP model

`TTTEnvironment.java`	Defines the Tic-Tac-Toe Reinforcement Learning environment
`Agent.java`	Abstract class defining a general agent, which other agents subclass.
`HumanAgent.java`	Defines a human agent that uses the command line to ask the user for the next move
`RandomAgent.java`	Tic-Tac-Toe agent that plays randomly according to a `RandomPolicy`
`Move.java`	Defines a Tic-Tac-Toe game move
`Outcome.java`	A transition outcome tuple (s,a,r,s')
`Policy.java`	An abstract class defining a policy – you should subclass this to define your own policies
`TransitionProb.java`	A tuple containing an `Outcome` object and a probability of the Outcome occurring.
`RandomPolicy.java`	A subclass of policy – it's a random policy used by a `RandomAgent` instance.

What to submit: You will fill in portions of `ValueIterationAgent.java`, `PolicyIterationAgent.java` and `QLearningAgent.java` during the assignment. **Commit & push your changes to your fork of the repository**. Do this frequently so nothing is lost. There will soon be **automatic unit tests** written for this project, which means that you'll be able to see whether your code passes the tests, both locally, and on GitLab. I will send an announcement once I've uploaded the tests.

PLEASE DO NOT UPLOAD YOUR SOLUTIONS TO A PUBLIC REPOSITORY. We have spent a great deal of time writing the code & designing the coursework and want to be able to reuse this coursework in the coming years.

Evaluation: Your code will be tested on GitLab for correctness using Maven & the Java Unit Test framework. Please *do not* change the names of any provided functions or classes within the code, or you will wreck the tests.

Mistakes in the code: If you are sure you have found a mistake in the current code **let me or the lab helpers know and we will fix it.**

Plagiarism: While you are welcome to discuss the problem together in the labs, we will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. **We trust you all to submit**

your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences with the school that are available to us.

Getting Help: You are not alone! If you find yourself stuck on something, **ask in the labs**. You can ask for help on GitLab too – but it means you will need to commit & push your code first: don't worry, you won't be judged until the deadline. It's good practice to commit & push your code frequently to the repository, even if it doesn't work.

We want this coursework to be intellectually rewarding and fun.

MDPs & Reinforcement Learning

To get started, run Game.java without any parameters and you'll be able to play the RandomAgent using the command line. From within the top level, main project folder:

```
java –cp target/classes/ ticTacToe.Game
```

You should be able to win or draw easily against this agent. Not a very good agent!

You can control many aspects of the Game, but mainly which agents will play each other. A full list of options is available by running:

```
java –cp target/classes/ ticTacToe.Game -h
```

Use the –x & -o options to specify the agents that you want to play the game. Your own agents, namely, Value Iteration, Policy Iteration, and Q-Learning agents are denoted as vi, pi & ql respectively, and **can only play X in the game.** This ignores the problem of dealing with isomorphic state spaces (mapping x's to o's and o's to x's in this case). For example if you want two RandomAgents to play out the game, you do it like this:

```
java target/classes/ ticTacToe.Game –x random –o random
```

Look at the console output that accompanies playing the game. You will be told about the rewards that the 'X' agent receives. The `O' agent is always assumed to be part of the environment.

Question 1 (6 points) Write a value iteration agent in `ValueIterationAgent.java` which has been partially specified for you. Here you need to implement the `iterate() & extractPolicy()` methods. The former should perform value iteration for a number of steps (k steps – this is one of the fields of the class) and the latter should extract the policy from the computed values.

Your value iteration agent is an offline planner, not a reinforcement agent, and so the relevant training option is the number of iterations of value iteration it should run in its initial planning phase – you can change this in `ValueIterationAgent.java`.

`ValueIterationAgent` constructs a `TTTMDP` object on construction – you do not need to change this class, but use it in your value iteration implementation to generate the set of next game states (the sPrimes), their associated probabilities & rewards when executing a move from a particular game state (a `Game` object). You can do this using the provided

`generateTransitions` method in the `TTTMDP` class, which effectively gives you a probability distribution over `Outcome`s.

Value iteration computes k-step estimates of the optimal values, V_k. You will see that the the Value Function, V_k is stored as a java HashMap, from `Game` objects (states) to a `double` value. The corresponding `hashCode` function for `Game` objects has been implemented so you can safely use whole Game objects as keys in the HashMap.

***Note**:* You may assume that 50 iterations is enough for convergence in this question.

Note: Unlike the MDPs in the class, in the CW2 implementation, your agent receives a reward when *entering* a state – the reward simply depends on the target state, rather than on source state, action, and target state. This means that there is no imagined terminal state outside the game like in the lectures. Don't worry – all the methods you have learned are compatible with this setting.

***Note*:** The O agent is modelled ***as part of the environment**,* so that once your agent (X) takes an action, any next observed state would include O's move. The agents need NOT care about the intermediate game/state where only they have played and not yet the opponent.

The following command loads your `ValueIterationAgent`, which will compute a policy and executes it 10 times against the other agent which you specify, e.g. random, or aggressive. The –s option specifies which agent goes first (X or O). By default, the X agent goes first.

```
java target/classes/ ticTacToe.Game -x vi -o
random –s x
```

Question 2 (1 point): Test your Value Iteration Agent against each of the provided agents 50 times and report on the results – how many games they won, lost & drew against each of the other rule based agents. The rule based agents are: *random*, *aggressive*, *defensive*.

This should take the form of a very short .pdf report named: vi-agent-report.pdf. Commit this together with your code, and push to your fork.

Question 3 (6 point) Write a Policy Iteration agent in `PolicyIterationAgent.java` by implementing the `initRandomPolicy()`, `evaluatePolicy()`, `improvePolicy()` & `train()` methods. The `evaluatePolicy()` method should evaluate the current policy (see your lecture notes), specified in the `curPolicy` field (which your `initRandomPolicy()` initialized). The current values for the current policy should be stored in the provided `policyValues` map. The `improvePolicy()` method performs the Policy improvement step, and updates `curPolicy`.

Question 4 (1 point): As in Question 2, this time test your Policy Iteration Agent against each of the provided agents 50 times and report on the results – how many games they won, lost & drew. The other agents are: *random*, *aggressive*, *defensive*.

This should take the form of a very short .pdf report named: `pi-agent-report.pdf`. Commit this together with your code, and push to your fork.

Questions 5 & 6 are on Reinforcement Learning:

Question 5 (5 points): *Write a* Q-Learning agent in `QLearningAgent.java` by implementing the `train() & extractPolicy()` methods. Your agent should follow an ε-greedy policy during training (and only during training – during testing it should follow the extracted policy). Your agent will need to train for many episodes before the q-values converge. Although default values have been set/given in the code, you are strongly encouraged to play round with the hyperparameters of q-learning: the learning rate (α), number of episodes to train, as well as the epsilon in the ε-greedy policy followed during training.

Question 6 (1 point): Like the previous questions, test your Q-Learning Agent against each of the provided agents 50 times and report on the results - how many games they won, lost & drew. The other agents are: *random*, *aggressive*, *defensive.*

This should take the form of a very short .pdf report named: `ql-agent-report.pdf`. Commit this together with your code, and push to your fork.

Javadoc: There is extensive comments in the code, Javadoc (under the folder doc/ in the project folder) and inline. You should read these carefully to understand what is going on, and what methods to call/use. They might also contain hints in the right direction.

Value of Terminal States: you need to be careful about the values of terminal states - terminal states are states where X has won, states where O has won, and states where the game is a draw**. The value of these game states - *V(g)* - should under all circumstances and in all iterations be set to 0**. Here's why: to find the optimal value of a state you will be looping over all possible actions from that state. For terminal states this is empty, and **might,** depending on your implementation of finding the maximum, lead to a result where you would be setting the value of the terminal state to a very low negative value (e.g. *Double.MIN_VALUE*). **To avoid this, for every game state g that you are considering and calculating its optimal value, CHECK IF IT IS A TERMINAL STATE (using g.isTerminal()); if it is, set its value to 0, and move to the next game state (you can use the 'continue;' statement inside your loop).** Note that your agent would have already received its reward when transitioning **INTO that state, not out of it.**

Testing your agent: If everything is working well, and you have the right parameters (e.g. reward function) ***your agents should never lose***.

You can play around with the reward values in the *TTTMDP* class – especially try increasing or decreasing the negative losing reward. Increasing this negative reward (to more negative numbers) would **encourage your agent to prefer defensive moves to attacking moves.** This will change their behavior (both for Policy & Value iteration) and should encourage your agent to never lose the game**.** Machine Learning isn't like Mathematics with complete certainty - you almost always have to experiment to get the parameters of your model right!