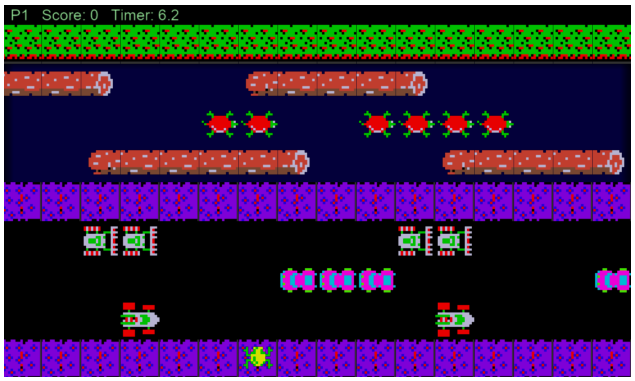
CS 380: Artificial Intelligence

Assignment 3: Reinforcement Learning

Frogger (20 pts)

Frogger [https://en.wikipedia.org/wiki/Frogger] is an arcade game from the 1980s that has been ported to many different home gaming systems and is still available to play online (e.g., https://froggerclassic.appspot.com). In the game, you control a frog that is trying to cross a busy road and a busy river to arrive at a home row at the top of the screen. On the road, the frog must avoid the moving cars; at the river, the frog must jump between the floating logs and turtles to avoid falling into the water. There is a 30-second timer during which the frog must complete its journey; otherwise, it dies and the frog is regenerated at the bottom of the screen. Each time the frog reaches the home row, it is awarded a number of points proportional to the time left on the timer.

In this assignment, you will develop a reinforcement-learning (RL) agent that plays Frogger. The agent will play within a simplified version of the game that runs in Python and includes many aspects of the original game. Here is a sample screen shot:



This assignment, like others for this class, is an individual assignment that will be tested and graded in the usual way. As a bonus, we also plan to conduct a mini-tournament between all the submitted Frogger agents; the details will be announced later, but please keep in mind that your code will be embedded in this tournament and thus all students must carefully follow the guidelines below to ensure that their agent can compete successfully in the tournament.

Setup

To start the assignment, please download the given code, which includes two main components: the **frogger** module that implements the game itself, and the **agent** module which houses the agent code. At the top level, there is a **main.py** file that accepts command-line arguments and runs the simulation.

The game module uses the Python Arcade library [https://arcade.academy] which provides a base engine on which to build the game itself. For this reason, you will first need to install this library on your machine (or a virtual machine if you prefer). This should be straightforward by entering:

```
> pip3 install arcade
```

You can test out the game by entering:

```
> python3 main.py --player=human
```

You should be able to control the frog with the arrow keys, and eventually quit the game by pressing 'q' or the escape key.

While we are hopeful that everyone will be able to run the full system with graphics, Python Arcade can be dependent on your specific setup. If you have issues, you might check their web site [https://arcade.academy] for further installation instructions specific to your machine. Nevertheless, if you do not get the graphics working in the end, there is a backup plan: our game provides a way to print the state to the terminal with no separate window or graphics. To use text-only output, simply include the **--output=text** flag in the command line when running an agent, for example:

```
> python3 main.py --player=agent --steps=10 --output=text
```

(Unfortunately, the text-only output version of the game runs only with agent control, and does not allow for human control via the keyboard.) The full suite of command-line options will be detailed later in this document. Note that, even if you can run the graphics version, you may choose to use text-only output for faster training, since the text output is noticeably faster than the graphics when running at full speed.

For this assignment, all your own code and files must be enclosed within your **agent** module; you should not modify **main.py** or the **frogger** module (except if you need text output as noted above). As before, except for the Python Arcade library (and its dependencies), you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT directly use any external libraries or packages, including those installed with Arcade (e.g., numpy).

Frogger Module

The **frogger** module includes all the code and other files (e.g., sprite images) needed to run the game. In general, you will not need to understand the details of this module; in fact, in many applications of AI/ML, the details of the underlying environment are unknown, opaque, or hidden behind a black-box gaming engine. However, it is important to understand the details of the API: the state information sent from the game to your agent, and the action information your agent needs to send back to the game.

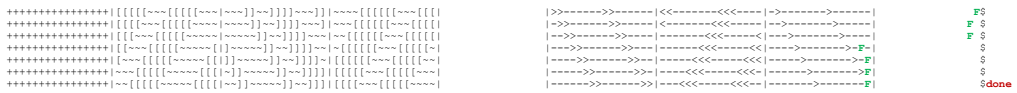
When communicating with the agent, the API sends the game state to the agent with all the necessary information encoded in a string. The screen itself is encoded as a grid of characters where each character denotes a different texture. For example, the following is an example of a state string passed via the API:

```
++++++++++++++++|~~~[[[[[~~~[[[[[[|]]~~]]]]~~~]]~~~|[[[~~~~[[[[[[
~~~|                  |-----<<---------<<|--<<<-------<<<-|------<<--
-------|         F        $
```

Here, the various characters represent different objects:

- 'F' : frog
- ' ' : safe position
- '<' or '>' : car going left or right
- '–' : road
- '[' or ']' : log going left or right
- '~' : water
- '+' : home goal row (at the top of the screen)
- '$' : end of screen string;
  if the string "goal=<score>" appears after the '$', the frog has reached the goal;
  if the string "done" appears after '$', the frog has failed to reach the goal

For example, here is a sequence of state strings for a sample game; you can see the frog (‘**F**’) move side-to-side, then into the road, before a car (‘**>**’) hits the frog and the game ends (‘**done**’):



You will not need to code the details of parsing this API string, however; this is provided for you in the agent base code, as described below.

Agent Module

The primary task in this assignment is to build out your **agent** module to implement an RL agent that plays Frogger. As a starting point, the given code includes two files, **state.py** and **agent.py**. The given code includes a sample agent that simply makes random moves. You can test out this agent by entering:

```
> python3 main.py --player=agent --steps=10
```

This command will run 10 steps of the simulation with the random agent and then terminate.

The **State** class in **state.py** parses the state API string and provides a number of useful variables: **frog_x** and **frog_y** for the position of the frog, **at_goal** which indicates whether the frog has reached the home row, **score** which gives the score achieved if at the goal, and **is_done** which indicates whether the frog is done for this episode (i.e., either reached the goal or failed to reach the home row before time expired). There are additional methods to check for legal positions on the screen and to get the cell character at a position on the screen. You should not need to modify **state.py**, but most importantly, since your agent will eventually be tied into the tournament code, you must ensure that the API string parsing remains the same so that your agent will work within the tournament.

The file **agent.py** is where you will need to build your RL agent, specifically an agent that implements Q-learning. The details are provided in the sections below.

State Representation

The **Q_State** class provided extends the basic **State** class to add additional functionality needed for Q-learning. First, the **reward()** method shows how we might calculate a reward for a given state, giving **state.score** as a reward if the frog has reached the goal, and a negative reward if the state is “done” and the timer has run out.

The **_compute_key()** method in the **Q_State** class is critical to learning: it reduces the complex game state to something manageable that can learn and generalize from the state. In essence, if the Q-table is implemented as a Python dictionary, this key is the string that will serve to index the state into this dictionary. There are many possible options for this key. On the one hand, if the key represents the entire screen, Q-learning would need to learn actions for every possible screen! – clearly not a feasible option. On the other hand, if the key represents only (for example) the cell in front of the frog, it would be missing a great deal of context. The sample method provided uses the three cells in front of the frog. You should experiment with better options after you have implemented the basic Q-learning algorithm below.

Q-Learning

The **Agent** class in **agent.py** is where you will implement the actual learning algorithm. First, you should consider how to implement the Q-table itself, likely as a dictionary over

state keys and secondarily over possible actions. Then, you will need to implement Q-learning within the **choose_action()** method. This method (which must keep this name for consistency with other agents) is the central method that receives the current state string and returns the action to be taken. It should construct an internal **Q_State** using the given state string, and then either train the Q-table (if training is on) or simply choose an action (if training is off). The possible actions, as included in **state.py**, are simply one of ['u', 'd', 'l', 'r', '_'] (for up, down, left, right, and none, respectively).

The critical piece of Q-learning to be implemented is the equation seen in lecture:

$$Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha[R + \gamma \max_A Q(S',A')]$$

The **choose_action()** method will be called for each update of the state, and thus you will need to keep track of the previous state and action in order to properly implement this equation. Note that when using the equation above, you would normally update the $Q(S,A)$ value for the *previous* state and action when you see the reward that resulted from it; thus, another way to think of the learning rule is

$$Q(S^{prev},A^{prev}) \leftarrow (1-\alpha)Q(S^{prev},A^{prev}) + \alpha[R + \gamma \max_A Q(S,A)]$$

updating the Q value for the previous state S^{prev} and action A^{prev} given the reward that resulted from them, R.

Also, please give some thought to the "exploration vs. exploitation" aspect of this algorithm discussed in class: while the agent may want to take the best action most of the time, it may be a good idea to allow for some probability of performing some other action (e.g., a random action) to allow the agent to explore unseen states.

The **Agent** class also provides methods for saving and loading the Q-table. For simplicity, we recommend saving the Q-table on every step of the simulation, ensuring that all training is saved incrementally. You can also back up your Q-tables in these files, and choose the best trained agent as your final submitted agent.

Command-Line Interface

The code provided includes a command-line interface with many options. The general command is:

```
> python3 main.py [options]
```

The possible options are as follows:

- --player=<*player*> : if *player* is 'human', the game runs with human input; otherwise, *player* should be 'agent' (default) and will run using the agent
- --screen=<*screen*> : *screen* should be one of 'medium' (default), 'hard', or 'easy' to denote environments of different difficulty
- --steps=<*steps*> : *steps* indicates the number of simulation steps to take; if not provided, the simulation will run until manually terminated (default)
- --train=<*train*> : *train* provides the filename of the training file that contains the Q-table, to be saved in the **train/** subdirectory; if not provided, the simulation runs without training (default)
- --speed=<*speed*> : *speed* can be either 'slow' for real-time simulation, or 'fast' for training purposes
- --restart=<*restart*> : *restart* is a number between 1 and 8 indicating the row at which the simulation should restart the frog for each episode; this is useful for training to keep the frog near the home row for initial training

- --output=<*output*> : *output* is either 'graphics' for a full graphics window using Python Arcade (default), or 'text' for text-only output to the terminal

Like other aspects of this code, you should not modify the command-line interface, since we will rely on a similar structure for testing and in the tournament.

Training and Testing

Once your state representation and learning algorithm are implemented, you can begin trying to train your Q-table. There are many ways to accomplish this, and you should feel free to experiment with various state representations and training regimens to see what works best.

You might start by training it to simply make the final move from the top-most river row to the home row. For example, this command

```
> python3 main.py --player=agent --train=q --screen=medium \
--steps=500 --restart=1
```

uses the medium-difficulty screen to train the agent with 500 simulation steps. At each episode, the frog restarts at row 1 (the top-most river row below the home row). If you have used the **save()** method, the resulting Q-table will be saved in **train/q.json** as a JSON file; you should be able to manually inspect this JSON file to see the various entries of the Q-table. It may help to watch the training in real time, using **--speed=slow**, and once it's clear that things are working properly, speed it up using **--speed=fast**.

You might then continue by training your agent incrementally so that, eventually, it is able to navigate to the home row from the starting position. If your agent is having trouble finding the home row, you might train it first on higher rows (like above) to get it working for those rows before training lower rows. Also, you might consider creating a training script that runs through a training regimen automatically (e.g., from higher to lower rows, and using all three boards for generalizability).

When your agent has been sufficiently trained, this command would test your agent on the easy screen, viewable in real time:

```
> python3 main.py --player=agent --screen=easy --steps=100
```

If this is working, you could try running your agent on the 'medium' and 'hard' screens, perhaps using **--speed=fast** to rapidly test it and see whether it reaches the home row.

In general, you should be able to train an agent that does the 'easy' screen without too much difficulty. The 'hard' screen is indeed hard, but it should occasionally succeed with this screen as well (though perhaps not very often).

Tournament

We are planning to run a tournament at the end of the course where all the students' agents will compete against one another. The details of the tournament are still to be determined, but you can assume that we will import your **agent** module and play this agent against other agents — so you should make sure that the agent loads, by default, your best Q-table and any other settings needed to run. Also, especially if you have successfully completed the main part of this assignment, we encourage you to experiment with different state representations, training regimens, etc. to make your agent play as well as possible; the only hard requirement is that your agent use Q-learning as its central component (with no imported libraries including those installed with Arcade).

Submission

For this assignment, you should submit all your files in the entire directory. As mentioned, your Python code will be located in the **agent** module, but please also include the trained Q-table (JSON) files and any training script(s) that you may have written. Please use a compression utility to compress your files into a single ZIP file. The final ZIP file must be submitted electronically using Blackboard—please do not email your assignment to a TA or instructor. If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment it due. If you have any doubts, complete your work early so that someone can help you.

Academic Honesty

Please remember that all material submitted for this assignment (and all assignments for this course) must be created by you, on your own, without help from anyone except the course TA(s) or instructor(s). Any material taken from outside sources must be appropriately cited.