CS 457/557: Assignment 4

Due: December 14, 2022, by 11:00 PM (Central)

Overview

In this assignment, you will be implementing the SARSA and Q-learning algorithms for a navigation task in a simple grid world environment.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "Q-learning in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

The general outline of the program is given below:

Algorithm 1 Program Flow

Process command-line arguments
Load environment from the specified file
Perform learning episodes, with periodic parameter updates and/or evaluation
Perform final evaluation of pure greedy policy based on learned Q values
Print final learned policy and additional details as appropriate

The program should be runnable from the command line, and it should be able to process command-line arguments to update various program parameters as needed. The command-line arguments are listed below:

- `-f <FILENAME>`: Reads the environment from the file named `<FILENAME>` (specified as a `String`); see the **File Format** section for more details.
- `-a <DOUBLE>`: Specifies the (initial) learning rate (step size) $\alpha \in [0, 1]$; default is `0.9`.
- `-e <DOUBLE>`: Specifies the (initial) policy randomness value $\epsilon \in [0, 1]$; default is `0.9`.
- `-g <DOUBLE>`: Specifies the discount rate $\gamma \in [0, 1]$ to use during learning; default is `0.9`.
- `-na <INTEGER>`: Specifies the value N_α which controls the decay of the learning rate (step size) α; default is `1000`.
- `-ne <INTEGER>`: Specifies the value N_ϵ which controls the decay of the random action threshold ϵ; default is `200`.
- `-p <DOUBLE>`: Specifies the action success probability $p \in [0, 1]$; default is `0.8`.
- `-q`: Toggles the use of Q-Learning with off-policy updates (instead of SARSA with on-policy updates, which is the default).
- `-T <INTEGER>`: Specifies the number of learning episodes (trials) to perform; default is `10000`.
- `-u`: Toggles the use of Unicode characters in printing; disabled by default (see the **Output** section for details).
- `-v <INTEGER>`: Specifies a verbosity level, indicating how much output the program should produce; default is `1` (See the **Output** section for details)

The `-f <FILENAME>` option is required; all others are optional. You can assume that your program will only be run with valid arguments (so you do not need to include error checking, though it may be helpful for your own testing). Your program must be able to handle command-line arguments in **any** order (e.g., do **not** assume that the first argument will be `-f`). Several example runs of the program are shown at the end of this document.

Extra credit: Any program that includes support for one or both of the following options will be eligible for a small amount of extra credit (see page 10 for details):

- `-l <DOUBLE>`: Specifies the λ parameter for **eligibility trace** decay; default is `0.0` (meaning that eligibility traces should not be used by default).
- `-w`: Specifies that the agent should use a weighted sum of features to estimate Q values for each state-action pair instead of maintaining these values in a table; disabled by default.

Environment Details

The environment in which the agent operates consists of a rectangular grid of cells, with each cell belonging to one of the following types:

- `S` indicates the start cell for an agent
- `G` indicates a goal cell
- `_` indicates an empty cell
- `B` indicates a block cell
- `M` indicates an explosive mine cell
- `C` indicates a cliff cell

In an environment, the agent begins in the start cell (`S`), and can choose to move in one of four directions (up, down, left, and right). Goal (`G`) and mine (`M`) cells are terminal states, which means no further action is possible upon reaching those states. Cliff (`C`) cells are "restart" cells: any action taken in a cliff cell causes the agent to return to the start cell for the next step. Block (`B`) cells are obstacles that the agent must circumvent (i.e., the agent cannot enter a block cell).

In all other types of cells, any attempt at movement either succeeds as intended with some probability $p \in [0, 1]$ or results in movement plus drift with probability $1 - p$. The drift is perpendicular to the intended direction of movement, and each drift direction is equally likely. For example, in Figure 1 below, the agent is in the cell marked `X` and is attempting to move up to the cell marked `T`. The following outcomes are possible:

- the agent moves up to enter cell `T` as intended with probability p;
- the agent moves up and drifts left to enter cell `L` with probability $(1 - p)/2$;
- the agent moves up and drifts right to enter cell `R` with probability $(1 - p)/2$.

Any movement (including movement plus drift) that would cause the agent to leave the bounds of the grid or enter a block cell (`B`) will fail and result in the agent staying in its current location. For example, in Figure 2 below, the agent is at the right edge of the environment and is attempting to move up to the cell marked `T`. The following outcomes are possible:

- the agent moves up to enter cell `T` as intended with probability p;
- the agent moves up and drifts left to enter cell `L` with probability $(1 - p)/2$;
- the agent agent remains in place with probability $(1 - p)/2$ because moving up and drifting right would take the agent out of bounds.

Figure 3 below shows how block cells influence outcomes. The agent is again in the cell marked `X` and attempting to move up. The only movement that can occur is moving up and drifting left (with probability $(1-p)/2$); moving up without drift and moving up with rightward drift both lead to block cells which the agent cannot enter, so these outcomes result in no movement. (Note that the possibility of drift allows the agent to move on the diagonal in an **indirect** way; this can lead to some interesting "wall-hack" behavior that allows the agent to squeeze through gaps between two block cells that are adjacent to the agent's current location.)

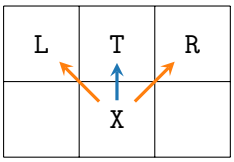

Figure 1

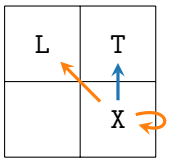

Figure 2

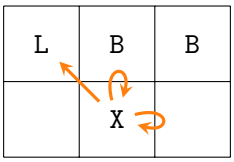

Figure 3

As the agent moves, it receives rewards as follows:

- Entering a goal cell yields a reward of $+10$.
- Entering a mine cell yields a reward of -100.
- Entering a cliff cell yields a reward of -20.
- Acting in a cliff cell and returning to the start yields a reward of -10.
- Any other result yields a reward of -1.

File Format

The environment for the agent will be specified in a text file containing cell markers (`S`, `G`, `_`, `B`, `M`, `C`) as described earlier. Empty lines and lines that begin with `#` should be ignored. A simple 2×5 grid world is shown below:

```
M_BG_
CS__C
```

Several example input files are provided in the `a04-data.zip` archive on Canvas.

Basic Reinforcement Learning

The basic learning agent will use either the SARSA or Q-Learning algorithms to learn Q values for all state-action pairs via temporal difference updates. Your basic learning agent will need to maintain $Q(s, a)$ values for all possible state-action pairs (s, a), where s is one of the cells in the grid world and a is one of the four movement directions (up, down, left, right).

To learn, the agent should perform T **learning episodes** (trials), where the value of T is either specified via the command-line flag `-T` or the default of $10,000$. In each learning episode (trial), the agent begins in the start cell of the environment. The agent should then select and execute actions according to an **ϵ-greedy policy** and update its Q values using either **on-policy** (SARSA) or **off-policy** (Q-Learning) updates as appropriate. The agent should continue acting until it reaches a terminal state (goal or mine cell) or it takes a number of actions equal to the entire size (width $\times$ height) of the environment, at which point the episode ends.

Parameters and Decay

Initially, the agent should use the control parameters α, γ, and ϵ with values set to their defaults or specified via corresponding command-line arguments. As learning proceeds, the agent should **periodically update** its α and ϵ parameters to allow the Q values to converge and to direct the agent towards a pure greedy policy. Let $\alpha^{(0)}$ and $\epsilon^{(0)}$ denote the initial values of α and ϵ, and assume that the learning episodes performed by the agent are labeled $1, 2, \ldots, T$. In episode t, the agent should use $\alpha^{(t)}$ and $\epsilon^{(t)}$ values determined as

$$\alpha^{(t)} = \frac{\alpha^{(0)}}{\lceil t/N_\alpha \rceil} \quad \text{and} \quad \epsilon^{(t)} = \frac{\epsilon^{(0)}}{\lceil t/N_\epsilon \rceil},$$

where N_α and N_ϵ are (integer-valued) hyperparameters that control the decay of the learning rate and policy randomness, respectively, and $\lceil \cdot \rceil$ is the **ceiling** function.

By design, the $\alpha^{(t)}$ values will **only change after** N_α episodes have completed (similarly for the

$\epsilon^{(t)}$ values). For example, we have

$$\alpha^{(t)} = \begin{cases} \alpha^{(0)} & \text{if } t \in \{1, 2, \ldots, N_\alpha\}, \\ \alpha^{(0)}/2 & \text{if } t \in \{N_\alpha + 1, N_\alpha + 2, \ldots, 2N_\alpha\}, \\ \alpha^{(0)}/3 & \text{if } t \in \{2N_\alpha + 1, \ldots, 3N_\alpha\} \end{cases}$$

This means that the agent only needs to **update** its α value **after completing** an episode that is a multiple of N_α (similarly for ϵ). With a little algebra, we can derive the following **update rules**, where t is the number of the episode that was just completed:

- **After** every N_α episodes, **update** α to $\dfrac{\alpha^{(0)}}{1 + \lfloor t/N_\alpha \rfloor}$.
- **After** every N_ϵ episodes, **update** ϵ to $\dfrac{\epsilon^{(0)}}{1 + \lfloor t/N_\epsilon \rfloor}$.

The periodic update rules above avoid having to recalculate $\alpha^{(t)}$ and $\epsilon^{(t)}$ at the start of each episode, and they also simplify the calculations somewhat (*Hint:* you can use integer division in Java to compute the floor function).

Evaluation of Final Learned Policy

After all learning episodes have completed, the agent should **evaluate** the performance of a **pure greedy policy** based on its learned Q values. To do this, the agent should perform 50 **evaluation episodes**; in each evaluation episode, the agent should always act greedily based on its Q values, and it **should not** update its Q values during evaluation. At the end of each evaluation episode, the agent should record the total reward (without discounting) that it earned by following the greedy policy. The performance of the greedy policy is then estimated as the average reward earned across the 50 evaluation episodes.

(*Hint:* There are only a few differences between **learning episodes** and **evaluation episodes** – you should be able to write **common code** for performing an episode of either type, and customize behavior based on one or two conditional flags.)

Output

The level of output produced by the program is controlled by the `-v` flag (for *verbosity*). The levels range from 1 to 4, with 4 containing the most output; the output produced is additive, meaning that the output for level i should also be produced for any level j with $j \geq i$. The desired output for each level is discussed in more detail below.

Output Level 1

Level 1 prints a brief summary of the program flow followed by details of the learned policy:

```
shell$ java Driver -f data/simple-example.txt -v 1
* Reading data/simple-example.txt...
* Beginning 10000 learning episodes with SARSA...
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.500
* Learned greedy policy:
MvBG<
<>>^<
```

Printing the policy involves displaying the best action (or actions in the case of ties) in each non-terminal and non-block state, where best is determined by the Q values, while displaying the cell marker for `M`, `G`, and `B` states. If you have a console that supports unicode output, then you can display the policy using unicode characters with the `-u` flag. The various combinations of best action and corresponding printed character (ASCII or Unicode) are shown below:

		Unicode	
Best Action Set	ASCII	Symbol	Java character
{ Left }	`<`	←	\u2190
{ Up }	`^`	↑	\u2191
{ Right }	`>`	→	\u2192
{ Down }	`v`	↓	\u2193
{ Left, Right }	`-`	↔	\u2194
{ Up, Down }	`|`	↕	\u2195
{ Up, Left }	`\`	↖	\u2196
{ Up, Right }	`/`	↗	\u2197
{ Down, Right }	`\`	↘	\u2198
{ Down, Left }	`/`	↙	\u2199
{ Up, Down, Right }	`>`	⊢	\u22a2
{ Up, Down, Left }	`<`	⊣	\u22a3
{ Down, Left, Right }	`v`	⊤	\u22a4
{ Up, Left, Right }	`^`	⊥	\u22a5
{ Up, Down, Left, Right }	`+`	(just use ASCII)	

You can print a unicode character in Java using `System.out.print("\u2190");` or similar. An image of unicode output on the console, produced with the command-line arguments

```
-f data/simple-minefield.txt -q -u -p 1
```

is shown below:

```
* Reading data/simple-minefield.txt...
* Beginning 10000 learning episodes with Q-Learning...
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.000
* Learned greedy policy:
M→→G+M
M↗↗↑MM
→↗↑M↓←
↗↗↑←←M
M↗↑↖M+
```

Embedding unicode output in this pdf document directly is non-trivial, so all remaining examples here will use ASCII output.

Output Level 2

Level 2 includes the final learned Q values in the output:

```
shell$ java Driver -f data/simple-example.txt -v 2 -q
* Reading data/simple-example.txt...
* Beginning 10000 learning episodes with Q-Learning...
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.620
* Learned greedy policy:
MvBG<
^>>^v
* Learned Q values:
-----------------------------------------------------------
|     0.0  |    -4.4  |     0.0  |     0.0  |     7.3  |
|0.0       |-92.5     |0.0       |0.0       |9.8       |
|       0.0|      -1.8|       0.0|       0.0|       7.4|
|     0.0  |     0.4  |     0.0  |     0.0  |   -23.5  |
-----------------------------------------------------------
|    -5.1  |   -18.8  |     5.5  |     9.4  |    -5.0  |
|-5.3      |-33.9     |4.1       |6.1       |-5.1      |
|      -5.4|       4.9|       7.3|     -20.2|      -5.2|
|    -5.6  |     4.0  |     5.8  |     7.6  |    -4.9  |
-----------------------------------------------------------
```

Each cell (state) in the environment is displayed in the above output, along with the four Q values associated with the different actions in that state. Note that terminal cells and blocks should have all of their Q values set to 0, because the agent never acts in such cells. Each cell has an internal width of 10 characters (which excludes the borders). Each Q value is displayed with a width of 6 characters (leading zeroes are suppressed) and one decimal point (format specifier `%6.1f`) and the Q values are arranged as follows:

- $Q(s, \text{Up})$ and $Q(s, \text{Down})$ are centered at the top and bottom of the cell, respectively, each with a minimum of two characters of blank space on either side;
- $Q(s, \text{Left})$ is flushed left on the second line in the cell (use format specifier `%-6.1f` here);
- $Q(s, \text{Right})$ is flushed right on the third line in the cell.

Your output doesn't have to match this exactly, but it should be displayed in a clean and readable format; in particular, strive to ensure that the cells are displayed with a fixed width.

Output Level 3

Level 3 will **periodically evaluate** a pure greedy policy based on the current Q values at various stages during the learning process. Specifically, after every $T/10$ **learning episodes**, the agent should perform 50 **evaluation episodes** and record the average total reward that is earned across these evaluation episodes. These evaluation episodes should operate as described earlier in the **Evaluation of Final Learned Policy** section. (Note that at this output level, it is fine to have your program evaluate the last greedy policy twice, once after the last learning episode and once at the end for final evaluation; depending on uncertainty within the environment, you might get different average total reward estimates for these separate evaluations.)

```
shell$ java Driver -f data/simpler-example.txt -v 3 -T 1200
* Reading data/simpler-example.txt...
```

```
* Beginning 1200 learning episodes with SARSA...
  * After     Avg. Total Reward for
  * Episode   Current Greedy Policy
        120     -8.000
        240      3.580
        360     -8.000
        480     -8.000
        600      3.020
        720     -8.000
        840      4.780
        960     -6.860
       1080      6.760
       1200      6.960
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.060
* Learned greedy policy:
v>>G
>>>^
* Learned Q values:
-------------------------------------------
|    -3.1  |     0.3  |     5.7  |     0.0  |
|-3.9      |-2.2      |1.4       |0.0       |
|      -4.8|       7.1|      10.0|       0.0|
|     1.2  |     0.3  |     5.1  |     0.0  |
-------------------------------------------
|    -0.4  |     3.4  |     3.4  |     9.7  |
|1.2       |-0.2      |0.2       |4.1       |
|       4.7|       7.1|       7.8|       7.5|
|     0.7  |     3.3  |     4.1  |     6.9  |
-------------------------------------------
```

Output Level 4

Level 4 displays the changes made to the α and ϵ parameters during the learning process:

```
shell$ java Driver -f data/simple-example.txt -v 4 -q -T 200 -na 100 -ne 50
* Reading data/simple-example.txt...
* Beginning 200 learning episodes with Q-Learning...
  * After     Avg. Total Reward for
  * Episode   Current Greedy Policy
         20     -4.720
         40      7.340
    (after episode 50, epsilon to 0.45000)
         60      7.620
         80    -10.000
        100      0.340
    (after episode 100, alpha to 0.45000)
    (after episode 100, epsilon to 0.30000)
        120      7.440
        140      7.400
    (after episode 150, epsilon to 0.22500)
        160      7.440
        180      7.260
```

```
        200      7.700
    (after episode 200, alpha to 0.30000)
    (after episode 200, epsilon to 0.18000)
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.560
* Learned greedy policy:
MvBG<
^>>^>
* Learned Q values:
-------------------------------------------------------
|     0.0  |    -1.9  |     0.0  |     0.0  |     2.6  |
|0.0       |-100.0    |0.0       |0.0       |9.8       |
|       0.0|      -1.4|       0.0|       0.0|       6.5|
|     0.0  |     4.3  |     0.0  |     0.0  |   -22.2  |
-------------------------------------------------------
|    -3.7  |   -58.8  |     5.9  |     9.4  |    -5.0  |
|-7.1      |-52.6     |4.1       |6.3       |-4.7      |
|      -4.2|       6.0|       7.4|     -18.9|      -4.6|
|    -5.1  |     4.5  |     6.5  |     7.5  |    -5.1  |
-------------------------------------------------------
```

Extra Credit Options

Eligibility Traces

The optional `-l` command-line argument allows for specification of the λ parameter to be used in conjunction with **eligibility traces**. For $\lambda = 0$, the learning agent should perform temporal difference updates as before (i.e., only one $Q(s, a)$ value should be updated at each step in a learning episode).

For $\lambda > 0$, the agent will need to store an eligibility value $e(s, a)$ for each $(s, a) \in S \times A$. Whenever the agent takes action a in state s, $e(s, a)$ should go up by 1. After each step in a learning episode, the agent should update the Q and e values for each state-action pair $(s'', a'') \in S \times A$ using:

$$Q(s'', a'') \leftarrow Q(s'', a'') + \alpha\delta e(s'', a''),$$
$$e(s'', a'') \leftarrow \gamma\lambda e(s'', a''),$$

where δ is the **TD error** at the current step. Some additional notes:

- The eligibility values should be **reset** back to 0 at the start of each episode.
- Using eligibility traces in conjunction with SARSA is straightforward, but using traces in Q-Learning requires more care. Specifically, in Q-Learning, the eligibility values $e(s, a)$ need to be reset back to 0 whenever the next action is sub-optimal.

See Lecture 09-3 for further details.

The example below uses SARSA with 100 learning episodes and deterministic movement (i.e., every action succeeds as intended), **without** eligibility traces:

```
shell$ java Driver -f data/simple-cliff.txt -p 1 -T 100 -l 0.0
* Reading data/simple-cliff.txt...
* Beginning 100 learning episodes with SARSA...
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: -8.000
* Learned greedy policy:
<v>G
v>>^
```

When eligibility traces **are included**, learning improves:

```
shell$ java Driver -f data/simple-cliff.txt -p 1 -T 100 -l 0.3
* Reading data/simple-cliff.txt...
* Beginning 100 learning episodes with SARSA...
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.000
* Learned greedy policy:
v<<G
>>>^
```

Feature-Based Q-Learning

The optional `-w` command-line argument should toggle the use of feature-based Q-learning. Instead of maintaining Q values for each state-action pair in a table, the agent should maintain a set of weights associated with features that characterize the state-action pairs.

For any **state** s, we will define the following features:

$$f_0(s) = 1$$
$$f_1(s) = \frac{x \text{ coordinate of } s}{x_{\max}}$$
$$f_2(s) = \frac{y \text{ coordinate of } s}{y_{\max}}$$
$$f_3(s) = 1 \text{ if } s \text{ is a mine cell, 0 otherwise}$$
$$f_4(s) = 1 \text{ if } s \text{ is a cliff cell, 0 otherwise}$$
$$f_5(s) = 1 \text{ if } s \text{ is a goal cell, 0 otherwise}$$
$$f_6(s) = \frac{L^1 \text{ distance from } s \text{ to nearest goal}}{x_{\max} + y_{\max}}$$

where we assume that the x and y coordinates of states are positive integers, $(1, 1)$ is the state in the lower left corner of the environment, and $(x_{\max}, y_{\max})$ is the state in the upper right corner of the environment; i.e., state coordinates look like:

$(1, y_{\max})$			$(x_{\max}, y_{\max})$
$\vdots$			
$(1, 1)$	$(2, 1)$	$\cdots$	$(x_{\max}, 1)$

For any **state-action pair** (s, a), we will define the following features, most of which are based on the features of the **likely** next state s' that would result from taking action a in state s (i.e., s' is the state that the agent would reach if it did not drift):

$$f_j(s, a) = f_j(s') \quad \text{for } j \in \{0, 1, \ldots, 6\}$$
$$f_7(s, a) = 1 \text{ if } s = s', \text{ 0 otherwise}$$

The agent should maintain a weight w_j for each state-action feature f_j, and it should estimate the Q value for any (s, a) pair as

$$Q(s, a) = \sum_{j=0}^{7} w_j \cdot f_j(s, a)$$

These weights should be initialized to 1 and then updated during learning episodes as described in Lecture 09-4, where δ is the TD error:

$$w_j \leftarrow w_j + \alpha \delta f_j(s, a) \quad \forall j \in \{0, 1, 2, \ldots, 7\}$$

For verbosity level 2 and up, your program should print out the final learned weights after the Q values (note that these Q values should be the estimated values based on the weights and state-action features). An example is shown on the next page.

```
shell$ java Driver -f data/simple-cliff.txt -q -p 1 -T 50 -v 2 -w
* Reading data/simple-cliff.txt...
* Beginning 50 learning episodes with Q-Learning...
  Done with learning!
* Beginning 50 evaluation episodes...
  Avg. Total Reward of Learned Policy: 7.000
* Learned greedy policy:
>v+G
>>>^
* Learned Q values:
-------------------------------------------
|    50.3  |    58.5  |    57.2  |    95.3  |
|50.3      |50.6      |57.2      |32.3      |
|      58.9|      32.3|      57.2|      95.3|
|    57.2  |    65.4  |    57.2  |    81.9  |
-------------------------------------------
|    50.6  |    58.9  |    32.3  |    95.7  |
|56.9      |57.2      |65.4      |73.7      |
|      65.4|      73.7|      81.9|      81.6|
|    56.9  |    65.1  |    73.3  |    81.6  |
-------------------------------------------
* Learned weights
  Feature 0 (constant):  35.526
  Feature 1 (scaled-X):  33.358
  Feature 2 (scaled-Y):  12.902
  Feature 3 (   ind-M):   1.000
  Feature 4 (   ind-C): -34.795
  Feature 5 (   ind-G):  20.321
  Feature 6 (  dist-G):   0.658
  Feature 7 (ind-move):  -0.339
```

Some final notes:

- In my limited experimentation with feature-based learning, the learning process seems a bit more temperamental with regards to the various control parameters (e.g., α, ϵ, decay rates). Q-Learning with features tends to do better than SARSA with features, but you can probably get SARSA to behave well by adjusting the control parameters appropriately (e.g., extending the learning process, adjusting ϵ decay).
- If you're feeling adventurous, you can try to devise some additional features to use and see if they work better than what is given here. You could also try mixing and matching the features, e.g., perhaps by removing the coordinate-based features and replacing them with something else. Let me know if you find any features that seem to work well!
- You can also add eligibility traces to feature-based learning (i.e., support both the `-l` and `-w` flags at the same time). This requires a few modifications to the eligibility values and the weight update process, though. You can read up on this in the RL textbook and/or stop by office hours if you're curious to know how this might be done.