Project: **Reward Shaping Analysis**

In this project, you will investigate how different *reward shaping* strategies affect learning efficiency in reinforcement learning. You will implement two RL algorithms (Monte Carlo control and SARSA) and evaluate how various shaped reward functions influence learning speed, throughput, and eventual convergence.

What is reward shaping?

Reward shaping is the process of *modifying the reward signal* in a reinforcement learning environment to guide the agent toward better or faster learning, without changing the overall task.

In the original MDP, the environment provides a reward $R(s, a, s')$. With reward shaping, we define a new reward:

$$R'(s, a, s') \stackrel{\text{def}}{=} R(s, a, s') + F(s, a, s')$$

where F is designed by the practitioner (you, the student).

Why do we use reward shaping?

- To make learning faster when the original reward is sparse or uninformative.
- To encourage desirable behaviours such as:
    - shorter paths, safety, exploration
- To reduce training time and improve early performance

Special Case: Potential-Based Shaping

A special class of shaping functions (called *potential-based shaping*) guarantees that the *optimal policy does not change*.
These have the form:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s)$$

where $\Phi(s)$ is a potential function – it can be anything.

This project will let you compare **different shaping strategies** and analyse how they affect learning speed, behaviour, and throughput in a grid-world environment.

Read more about this special case and its proof here: Scientific Paper: Policy Invariance under reward transformations: Theory and application to reward shaping

Exponentially decaying reward shaping

Let

- $F(s, a, s')$ be any shaping term.
- k be the **current episode number**, starting from $k = 1$,
- $h \in (0,1]$ be a decay constant (e.g., $h = 0.99$).

We define the **decayed shaping term**:

$$F_k(s,a,s') = h^{k-1}\ F(s,a,s').$$

Then the shaped reward for episode k is:

$$R'_k(s,a,s') = R(s,a,s') + h^{k-1}F(s,a,s').$$

This definition is included as an additional tool for you to use in the project.

Implementation Details

Environment

You are required to use the Frozen Lake environment from Gymnasium. You should also use wrappers if needed to track custom metrics such as per-episode throughput. You must set $success\_rate = 0.7$ with $is\_slippery = True$. You can set $grid$ to any map you would like to train on: you may use any map larger than 6x6 in each dimension provided the map is not trivial, includes exactly ONE goal, and has hole density of 10% to 20%. (In other words in a 6x6 grid you have 6*6=36 tiles, at least a range of 10%-20% of holes which means [36*10%,36*20%]=[3.6,7.2] so anywhere between [4,7]). For all algorithms use $\gamma = 1.0$. Keep the initial reward function as the default in Frozen Lake.
If you are feeling adventurous and want to try having more holes in your map – I encourage you to try it, but it will likely cause learning to be more difficult.

Gymnasium as a tool: Gymnasium Basics, Gymnasium simple tutorial
Project's Environment: Frozen Lake
Wrappers in Gymnasium: Gymnasium Wrappers, Implementing Custom Wrappers
Note about Wrappers: They are a good place to add code for tracking metrics and shaping reward. For example, it might make separation of concepts easier if you choose to create a wrapper to track metrics and a wrapper for the reward shaping. In the end, this is a design choice. It is only important that it is easy to track and verify how your environment works.

Algorithms to Implement

1. Every-Visit Monte Carlo Control (with ϵ –greedy policies, and constant α learning rate)

2. SARSA(0) (with ϵ-greedy action selection)

Both algorithms must be implemented from scratch.

You may use the following libraries in Python:

1. NumPy, Pytorch
2. Gymnasium
3. Seaborn, matplotlib, pandas, tqdm.

No other libraries may be used without permission and justification. If you wish to use other libraries you must justify their use. Additional Library use will likely be approved IF I think that it does not make aspects of the project trivial.

Reward Functions to use

Hint: If the reward shaping is perfect it should not change the optimal policy $\pi_\star$ - but even if it does affect it we may still end up in pretty good shape. If reward shaping encourages a good degree of exploration we may expect to generate good policies more quickly.

0. Baseline Reward (no shaping)

This is the original or default reward structure of your environment. For FrozenLake, this typically means:

- +1 for reaching the goal
- 0 otherwise (meaning: 0 for safe ice, 0 for holes)

1. Step-Cost Shaping

A small penalty is added for every step to encourage faster paths.

$$R'(s,a,s') = R(s,a,s') + c,$$

where $c < 0$ is a constant (e.g., -0.01 or -0.02).

This shaping encourages efficient navigation and increases throughput.

Try playing with different values for c.

2. Potential-Based Distance Shaping

Define the potential function using the distance to the goal:

$$\Phi(s) = -d(s),$$

where $d(s)$ is the Manhattan distance (or grid distance) from state s to the goal.

The shaped reward is:

$$F(s,a,s') = \gamma\Phi(s') - \Phi(s),$$
$$R'(s,a,s') = R(s,a,s') + \beta F(s,a,s'),$$

where $\beta > 0$ controls the strength of the shaping term. You should try setting β with several values, and showcase at least when $\beta = 1$ and $\beta = success\_rate$.

This form of shaping is *potential-based* and preserves the optimal policy in theory.

3. Custom Reward Shaping (designed by you)

Design your own shaping function. You must design **two non-trivial** methods. Your shaping must:

- Be clearly defined mathematically
- Have an intuitive explanation

- Encourage some behaviour (e.g., exploration, safety, directness)
- Be compared against the baseline and other shaping methods

This custom shaping is your opportunity to be creative and demonstrate understanding.

Experiment Reports

You will need to use **matplotlib** or **seaborn** or **other plotting libraries** to plot visualizations. Seaborn is built on top of matplotlib and is easier to use. Matplotlib should give you more options for customization but may be harder to use. You also have examples in the bandit code snippet we did in Lecture 2.

Matplotlib: Tutorial, W3Schools
Seaborn: Tutorial.

1. Throughput over episodes

When we shape reward it can be harder to know if the agent is still behaving optimally. For this reason, we introduce a new metric: throughput over episodes. Throughput is the ratio between the number of times the agent has reached its goal and the number of episodes that have passed:

$$Throughput \stackrel{\text{def}}{=} \frac{\#\ times\ reached\ goal}{\#\ number\ of\ episodes}$$

x-axis: Episode number (e.g., episodes 1 to 500)
y-axis: Throughput.

2. Per-Episode Returns

The raw reward obtained per episode.

x-axis: Episode number

y-axis: Episode return (sum of rewards in that episode). (Not **shaped return** – only the real return).

3. Policy Distribution Distance (Optional)

Total distance between the learned policy and the optimal policy:

$$D(t) = \frac{1}{|\mathrm{A}| \mid S \mid} \sum_{s \in S} \sum_{a} \mid \pi_t(\, a \mid s\,) - \pi_\star(a \mid s) \mid$$

- **x-axis**: Episode number
- **y-axis**: TV distance (0 to 1)

This is **Optional** – if time is tight you may skip this metric.

Note that this metric requires you to take the actual $\pi_\star$ and compute against it. To actually obtain $\pi_\star$ we typically use DP. Instead, for the purposes of this project, you may use the strongest policy you obtained in your experiments as an approximate $\pi_\star$, and compare ***all other methods*** against this reference policy. Do not compare your approximate $\pi_\star$ against itself.

To maintain accuracy of all plotting schemes 1-3:

a) Use at least 20 independent runs (Higher = Better Average Behaviour estimation - Preferably 1000+). You will need to plot the mean at parallel timesteps. (e.g. say you are plotting episodes on the x-axis and rewards on the y-axis, in 20 dependent runs you get 20 independent episodes running in parallel runs. You must take the mean of the rewards across the runs at that specific episode number).

b) Add a 95% confidence interval. (seaborn does this automatically)

It can be confusing to choose which algorithms to plot and how to compare two algorithms, so know that you must only compare the “best” ones – you will find which ones are the best through experimentation and performing sweeps of the hyperparameter space. To find the best ones you will need to do sweeps on your own.
What do we mean by that? For MC methods: compare all the best-performing variants of reward shaping methods

- There will be 3 plots: Throughput over episodes, Per-episode returns, Policy Distribution Distance.
- For each plot you have several shaping strategies to compare (Baseline with no reward shaping, step-cost reward shaping, potential reward shaping, your custom reward shaping...).
- And for each strategy several hyperparameters to select from ($\epsilon, \alpha, c, \beta, \ldots$)

As you can see the “pool” of algorithms to select from is huge, and yet you are tasked to compare between them. You will perform “sweeps” of the hyperparameter values across the different reward shaping strategies and select the ones that performed best for you, and compare those against each other. You will be asked to justify based on what metric you decided an algorithm is “best”.

Additionally to the plots above, you must also perform a small hyperparameter sensitivity study outlined below.

Hyperparameter Sensitivity Study

For one method of your choice, vary one or two of the key hyperparameters:

Hyperparameters to Sweep

- **ε** ∈ {0.05, 0.1, 0.2}
- **α** ∈ {0.05, 0.1, 0.2}
- **Shaping strength (c or β)** ∈ {0.01, 0.05, 0.1}
- $\boldsymbol{\gamma} \in \{0.95, 0.98, 0.99, 0.995\ ,1\}$

Then showcase the returns on per-episode basis by showing all the curves for each value of the hyperparameter sweep. (Again, do this only for one or two of the hyperparameters).

Tips

1. You should start in a smaller grid world (2x2, 2x3, 3x3) with no holes, then add a hole, then graduate to a bigger grid. This will greatly speed up your coding, and checking for logical errors.
2. You may start with $is\_slippery = False$ to keep things easier to understand when you are working on your implementation. Don't forget to set it back though.
3. Don't try to implement everything at once – start with the baselines you know, and plot those. Generalize later.
4. Make it as easy as possible to switch the size of the map. Perhaps as quick as changing a parameter number from '2x2' to '2x3', using hard-coded grids etc – this is only while you are still in the implementation phase. It will improve your productivity.
5. Consider generating a plot for the map showing the value function estimate per state as well as the S,G,H,F status (see Frozen Lake docs). This will greatly help with debugging. In some gymnasium environments it is possible to also add render() see "render" in env docs.
6. First don't do any parallel run (to obtain the confidence interval) – only add runs when the basic environment is working correctly.
7. If you are certain your programming is correct but you are having trouble reaching optimal behaviour in MC – it may just be that the map is too hard! Try different grids, try setting is_slippery=False, decreasing the number of holes, making the goal closer. All of these may genuinely impact learning, which is why it is better to start with a super easy map! Like 2x2 and then try to test your algorithm on the full settings.

What you will learn in this project

- Introduced to Gymnasium, and understanding how gymnasium works.
- How to use Gymnasium wrappers to modify rewards, observations, or track custom metrics (e.g., throughput).
- How to visualize results using Python plotting libraries (e.g., Matplotlib, Seaborn).
- Implementation of two core RL algorithms: SARSA and MC.
- Work with stochastic policies and environments in RL.
- Introduced to the concept of reward shaping: When and why it is used, how it affects learning, how to design shaping terms.
- The basics of how to scientifically compare two algorithms: running baselines, testing hyperparameter sensitivity, plotting learning curves.

Report (2-6 Pages, PDF format)

Prepare a PDF document, and in it Explain:

- Your grid world used in training.
- Your custom shaping function
- Intuition behind each shaping strategy
- Which shaping helped most in early learning
- Which shaping preserved final optimal behaviour
- Why potential-based shaping behaves differently
- Any surprising results or failure cases.

- Discuss where you encountered the Exploration vs. Exploitation problem, and some ideas you had to solve it. (Whether it was a bug which made your agent do 0 exploration, or a theoretical pattern, or an empirical issue, or something else entirely).
- The plots you were required to output, and any additional plots you think are interesting or help to back up your claims.
- A summary of your findings.
- Back up any of the earlier claims theoretically and empirically whenever possible.

Your Report Must also include a “Quick Start” section which gives a brief explanation of your project structure, where to find important components, how to navigate them, how to run it, and how to obtain the results you display in your PDF. Very importantly, you must say which libraries and versions you used, or include the correct “pip install” command so your results can be easily checked. They may also be submitted in the form of requirements.txt (if you are familiar). Without the correct library versions, I won’t be able to compile your code. To this end, I highly recommend you use a virtual environment to make things easy for you. See virtual environment (venv) command, W3Schools, Youtube Explanation. To keep things as simple as possible use the latest available versions of every library.
Before you submit your project: try downloading it from scratch on a clean (new) environment, install all the packages you specified in your report with “pip install” (or requirements.txt) and check if you can run it after installing the libraries.

Note: The “Quick Start” is NOT considered part of the 2-6 pages.

Grading Criteria

Code and Report

Your project will be evaluated on:

- Correct implementation of MC and SARSA
- Correct mathematical definitions of shaping functions
- Quality of analysis and plots
- Depth of explanation in the report
- Insightful interpretation of shaping effects
- **Crucial:** Clarity and cleanliness of code, In particular:
    1. You should define classes and functions to keep things clear.
    2. When classes have similar functions, don’t forget you can use inheritance! Don’t forget that classes can hold variables which are classes themselves. Both of these, help so much in writing fast and bug-free code.
    3. You must add documentation to all functions and classes and explain them in an easy manner. Their roles must be clear, and they must not be doing anything obscure.
    4. You must use assign informative names to every variable in your code.

5. If a function is longer than 70 lines it must benefit from further abstraction: it can be separated into multiple functions.
6. Similarly, if a class is growing beyond your intended scope – it is time to refactor it into multiple classes.
7. All imports must be included at the top of the file.
8. Files and Folders must be named and grouped in a logical manner. Example for good file name: MonteCarloAlgorithm.py, MCAlgorithm.py, algorithms.py, visualization_functions.py.
9. At the top of each .py file – you must include an explanation of what can be found in this file, and which components use it in the project.
10. I should be able to look at your code and easily understand every component, and the flow of code.

Presentation

Your presentation should include:

1. A brief explanation of FrozenLake, and which grid you chose to learn on. You must present the grid so we can see how it looks like. Explain the reward function you initialized the environment with.
2. A brief explanation of SARSA and MC (just the update rules, and explaining them). Should include $constant - \alpha$ and theoretical implementation.
3. The definition and explanation of your custom reward shaping strategy.
4. An explanation of every plot you were asked to showcase. You should be able to explain everything about the plot (e.g. x-axis, y-axis, which algorithms or policies are you comparing, hyperparameters, etc).
5. Show us one or some of your better-performing policies in action using render(). You can also show us your policies "through time". For example: after training for one episode, after training for 100 episodes, and the optimal policy.
6. Anything else you feel is worth noting, or found interesting.
7. **Note:** You may be asked to run your code to generate any of the plots you showcased from scratch. Therefore, it should be easy for you to run your code easily and with the right configuration.

Note: Your presentation backs up your report – so your presentation must not contradict your report.

Final Note

1. You may be asked to explain any part of your code and your plots before receiving your final grade. I expect both students to be able to answer independently of each other.
2. During the presentation, I may ask questions about your findings and I expect you to think about them and try to find logical explanations for the findings even if you are not sure yourself of the answer.
3. During the presentation, we may relate your project or results to other topics from the course and brainstorm together (e.g., extensions of your project, challenges you encountered, or how alternative algorithms might change your results). These discussions

are meant to deepen your understanding of RL, clarify common assumptions, and help you see how different design choices affect performance.

Good luck! Learning is iterative, enjoy the process.