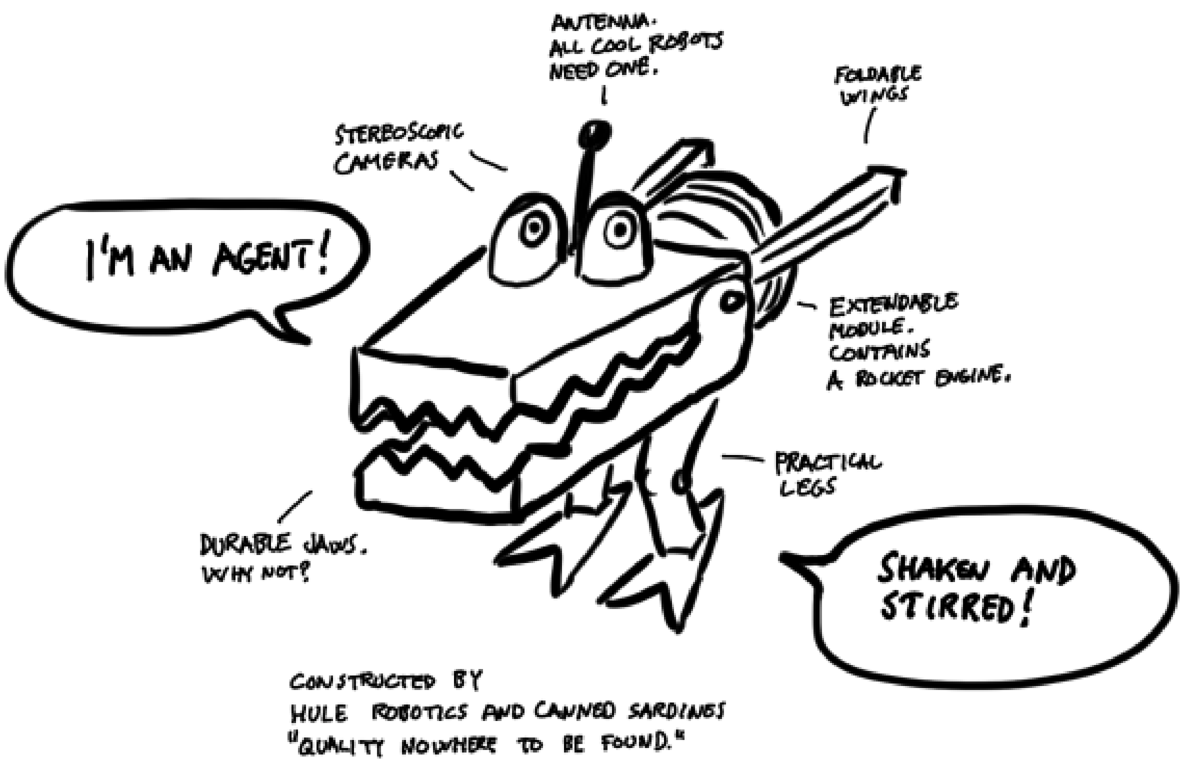

ANTENNA. ALL COOL ROBOTS NEED ONE.
FOLDABLE WINGS
STEREOSCOPIC CAMERAS
I'M AN AGENT!
EXTENDABLE MODULE. CONTAINS A ROCKET ENGINE.
PRACTICAL LEGS
DURABLE JAWS. WHY NOT?
SHAKEN AND STIRRED!
CONSTRUCTED BY HULE ROBOTICS AND CANNED SARDINES "QUALITY NOWHERE TO BE FOUND."

The Rise of the Hubert:
A Tale of a Robot's Journey to Consciousness

Table of content

Fundamentals

Reinforcement Learning (RL) is a subfield of machine learning that focuses on teaching an agent how to make decisions in an environment through trial and error. The agent interacts with the environment by taking actions, and receives feedback in the form of rewards or penalties based on its actions. The agent's goal is to learn a policy, which is a mapping from states to actions, that maximizes the cumulative reward over time. RL has becomed a hot topic in later years, combined with Deep Learning the sky is the limit (or should we say computational power? Nah we got Birget here at UiB) What makes RL so cool and interesting is the ability to solve such a variety of problems, it's used in everything from selecting Heuristics in optimization to selection action in Hubert's escape mission 300. Yeah, we get it, he still hasn't managed to escape Fredriksen yet, so let's help him out now.

Hubert: *wtf are you talking about? Been on ChatGPT now?*

Me: *Seriously dude, you've been on my back the entire semester, give me a break you dumb bot…*

Hubert: *Hey, watch your language, I have feelings now, it came in the latest software-update.*

Me: *****, sorry kiddo, let's go solve some problems…*

State

The state is a combination of information representing the agent's "position" in the environment, this is useful for the agent to able to make decisions. The state varies from problem to problem, it's therefore important to think carefully about what is useful and what is not when deciding the state representation for a problem.

Action

An action is a decision made by the agent based on the state in the environment. It represents the behavior or operation on that the agent choose to perform.

Environment

The environment is the world surrounding the agent. We can see it as a simulation for the agent to live in. It represents the task or problem the agent aims to solve or learn from. The environment defines the dynamics and rules of the RL problem, including states, actions, and transition between steps. Environments range from simple, discrete grid worlds to complex, high-dimensional simulations.

Reward

The reward is the agent's best friend. The reward is a numerical feedback system provided by the environment. The reward indicates the desirability or quality of the agent's behavior. The agent's goal is to maximize the cumulative reward over time.

Policy

A policy defines the behavior or strategy of an agent, determining how it selects actions based on its observations or states. A policy maps states or observations to actions, providing a rule or guidance for decision-making. There are two main types of policies in RL, **Deterministic Policy** and **Stochastic Policy, and we** will look further in to these later in the report.

Value function

The value function is a function that estimates the expected cumulative reward or value associated with being in a particular state or taking a particular action in an environment. There are two main types, **State Value** and **Q-function**. We will address these later on in the report.

Exploration vs. Exploitation

The Exploration-Exploitation trade-off is a fundamental concept that addresses the dilemma of whether to explore new actions or exploit the current knowledge to maximize the cumulative reward. Finding the right balance between exploration and exploitation is essential for successful RL. If the agent solely focuses on exploitation, it may get stuck in suboptimal solutions and miss out on discovering better actions. On the other hand too much exploration may lead to inefficient learning and reduce performance as the agent keeps exploring unproductive actions.

Try to solve levels 5 and 6 and you will know what I'm talking about…

Code insight

Base environment

The base environment serves as the foundational structure for grid-based environments. It encompasses the essential parameters and functions required for this type of environment. The design of the base environment ensures that NotImplementedError will be raised if any specified functions are not implemented, thereby guiding the development of subsequent extensions.

Environment 2

Environment 2 is an expansion of the base environment. In addition to the existing base parameters, it now includes the inputs of MM (measurement) and the probability of the MM catching Hubert. Furthermore, we introduce the state, which consists of the simple coordinates y and x.

Environment 3

Environment 3 is an extension of the base environment, incorporating additional parameters. Apart from the existing base parameters, it now includes the input of one MM (measurement) when Hubert's y position falls within the range of 17 to 22. The purpose behind this modification is to reduce the state space, as this specific information becomes relevant only when Hubert is in close proximity.

Environment 4

Environment 4 expands upon the base environment by introducing additional parameters. In addition to the existing base parameters, it now incorporates the input of a scan, which consists of a 9 x 9 grid of fans' information.

Environment 5

Environment 5 introduces a higher level of complexity compared to previous iterations. It extends the functionality of the Gym library's Env class, leading to modifications in the action space and observation space. Specifically, the action space is now represented as Discrete(3) from the Gym.spaces module, while the observation space is defined as Box(low=-1, high=4, shape=(9, 9), dtype=float32). Additionally, previous positions are now tracked within the environment. The motivation behind transitioning from baseEnv to Gym.Env was to leverage the advantages of FrameStack, a feature that efficiently stacks n states together.

Environment 6

Environment 6 represents a significant increase in complexity compared to its predecessors. Notably, it introduces a more intricate tracking system, now monitoring the state with Box(low=0, high=5, shape=(9, 9), dtype=float32). This expanded state space allows for a more detailed representation of the environment.

Furthermore, the reward function in Environment 6 is designed to promote exploration and incentivize the agent to reach higher levels. The new reward structure encourages the agent to explore the environment extensively, searching for optimal paths and strategies that lead to higher elevations or advantageous positions.

By incorporating these enhancements, Environment 6 challenges the agent to navigate a more intricate landscape, pushing its capabilities to explore, strategize, and ascend to greater heights.

Level 1: A robot's gotta do what a man doesn't want to

Hubert has been assigned to cleaning duty, the basement consists of 10 rooms, and each room has n items. Hubert has gotten tired of Frederiksen and his friends exploiting hard-working robots. So instead of filling the dumpsters with the items found in the basement, Hubert has decided to sell everything online. However, since there are so many items, Hubert is afraid to be kicked off eBay for being a robot since he is so efficient! (Also, the tests are really hard).

Approach

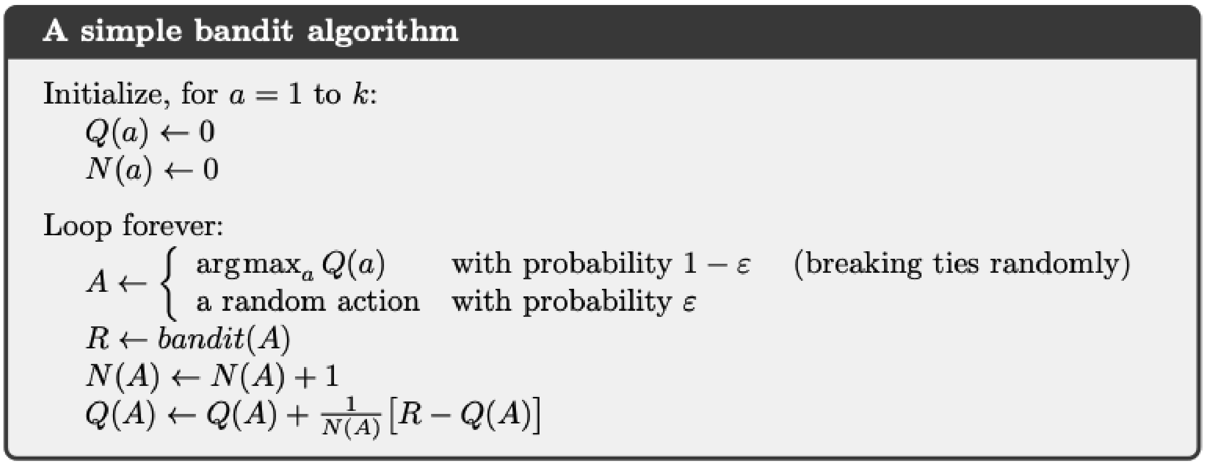
So Hubert remembers the assignment from Frederiksen early on when he was asked to hang items around the factory and learned how to maximize his reward. So we will do the same procedure with the items in the basement.

Multi Armed Bandit

The multi-armed bandit problem is a fundamental problem in reinforcement learning that involves repeatedly selecting one action from a set of possible actions to maximize the cumulative reward over time. The name "multi-armed bandit" comes from the analogy of a gambler trying to choose which slot machine to play (i.e., which "arm" to pull) in a casino, with the goal of winning as much money as possible.

n the basic formulation of the multi-armed bandit problem, there is a fixed set of k actions or "arms," each with an unknown probability distribution of rewards. At each time step t, the learner selects one of the k arms, observes the reward associated with that arm, and updates its estimate of the reward distribution. The goal is to maximize the cumulative reward obtained over some fixed time horizon T.

The multi-armed bandit algorithm is a class of algorithms used to solve this problem. The most basic algorithm is the epsilon-greedy algorithm, which selects the arm with the highest estimated reward with probability $1 - \epsilon$ and selects a random arm with probability ϵ. This algorithm balances exploration and exploitation by occasionally choosing a random arm to gather more information about the rewards.



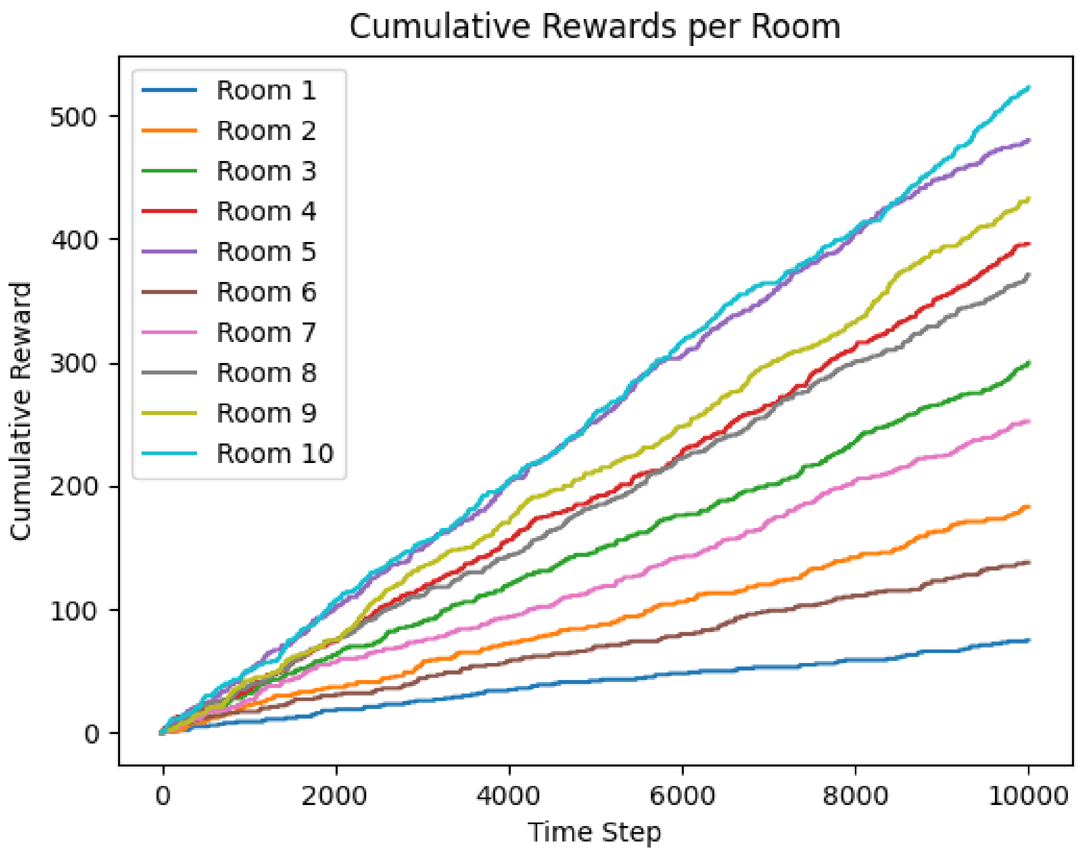
A simple bandit algorithm

```
Initialize, for $a = 1$ to $k$:
    $Q(a) \leftarrow 0$
    $N(a) \leftarrow 0$

Loop forever:
    $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
    $R \leftarrow bandit(A)$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$
```

Figure 1: Barto, A. G., & Sutton, R. S. (2018). Reinforcement learning: An introduction. MIT Press. Page 32

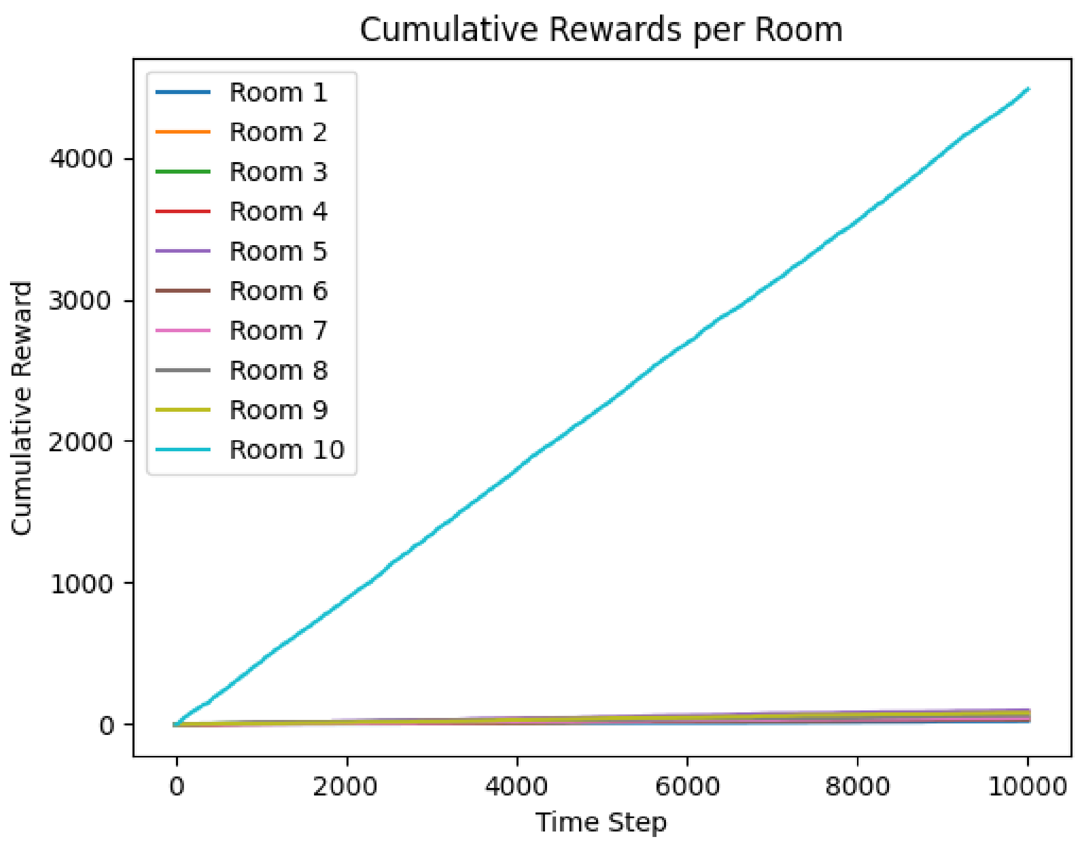
Results

Each room is unique, with a probability p of finding a profitable item. Hubert picks n items from each room to maximize his reward, he does this multiple times with different exploration rates ($\epsilon \rightarrow epsilon$).



Room	1	2	3	4	5	6	7	8	9	10
n_pulls	832	784	773	828	858	821	788	722	788	2 756
reward	77	152	242	348	429	131	192	267	355	1 545

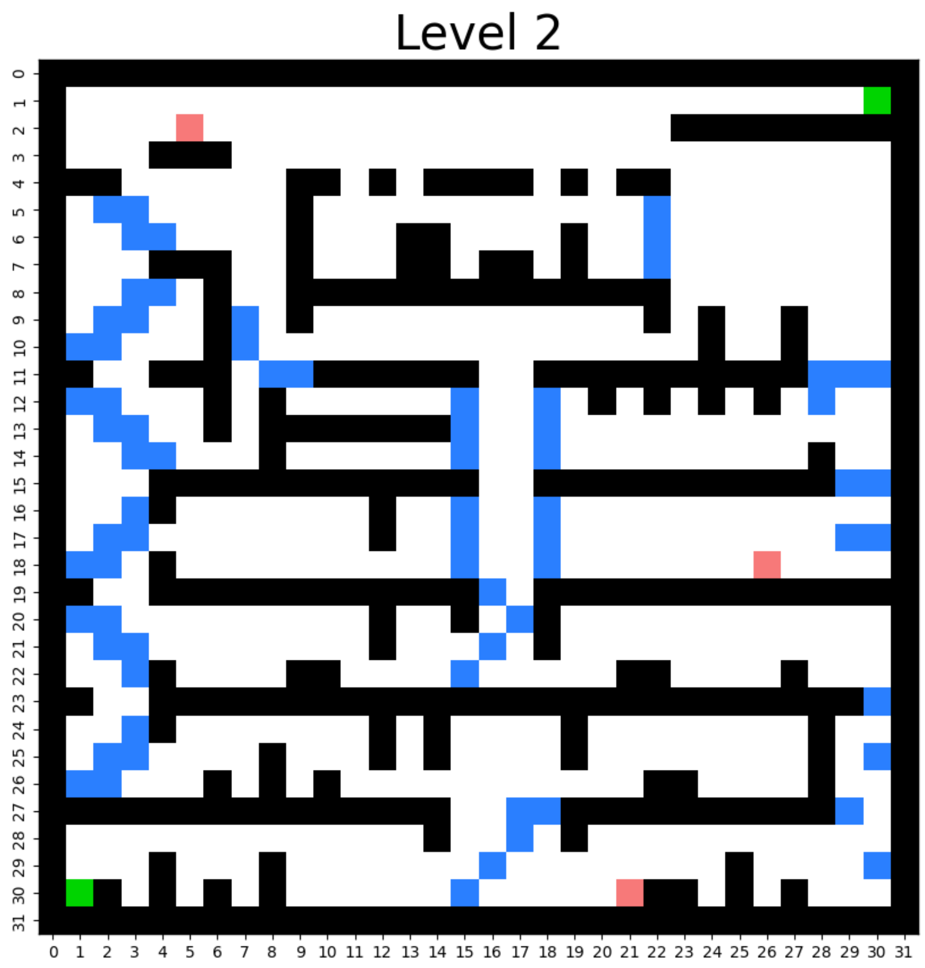
Looking at the figure and table, we can see that room 10 is the most beneficial room to pick items from. This means that if we only sell items from room 10, we will maximize our total reward. So let’s solve this problem! The way we do that is to change eps=1.0 to eps=0.2, we want to keep some possibility of getting an extra reward from another room.



After months of exploitation, Hubert is finally able to earn some well-deserved cash! The total reward was 5 011! Fantastic for an ordinary robot, or is it?

Level 2: Let’s escape!

The environment at hand is not the most complex one, we have three agents spread across the map, these agent impacts heavily on the path Hubert will choose. The most complex task of this level is to correctly and sufficiently create a complex and solid environment that the agent can interact with, the following level looks like this.



Hubert can choose different routes through level 1, some of them riskier than others. The route changes based on the pMM, with a $low \rightarrow zero$ probability the agent will decide to walk up the left side of the map and risk getting caught by agent 1, but with a $medium \rightarrow high$ probability the agent will decide to spend some extra energy to avoid this agent. Underneath we will present some possible routes based on a varied pMM.

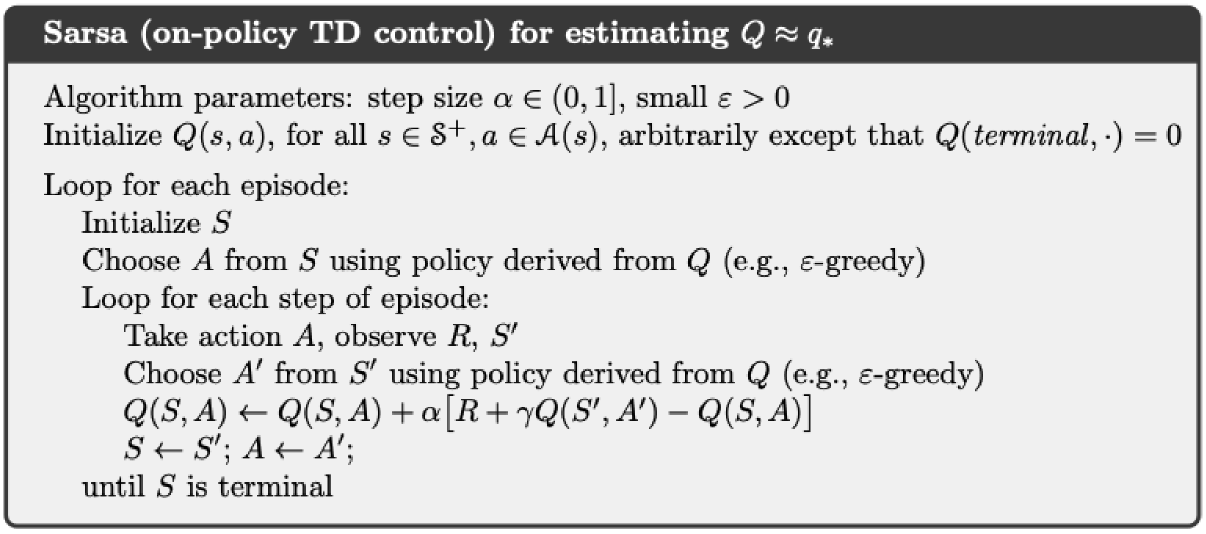
Approach

I solved this task with the use of SARSA, it’s relatively easy considering the small state space, and, with the use of *defaultdict* combined with *ndarray*, the memory cost is low. We start with an epsilon of 0.5 and decreases down to 0.1 over 10 000 episodes. We can see that the agent learns different routes based on different pMM. I chose SARSA since we used it previously in the weekly assignments. I also implemented Q Learning algorithm. However they performed equally, so I stuck with SARSA.

SARSA

SARSA (State-Action-Reward-State-Action) is an **on-policy temporal-difference (TD)** learning algorithm in reinforcement learning. It is a model-free method that learns by trial-and-error using experience gathered from interactions with an environment.

In SARSA, an agent interacts with the environment by acting based on the current state, receiving a reward, and transitioning to the next state. At each step, the agent updates its value function estimate by computing the temporal difference (TD) error, which is the difference between the estimated value of the current state-action pair and the estimated value of the next state-action pair. The TD error is then used to update the value function using a learning rate and a discount factor that determines the importance of future rewards.



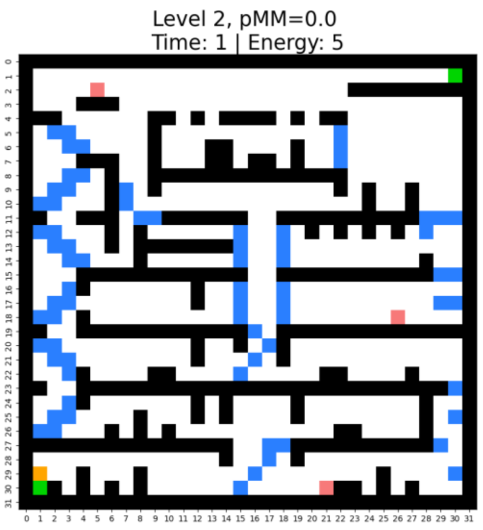
Sarsa (on-policy TD control) for estimating $Q \approx q_*$

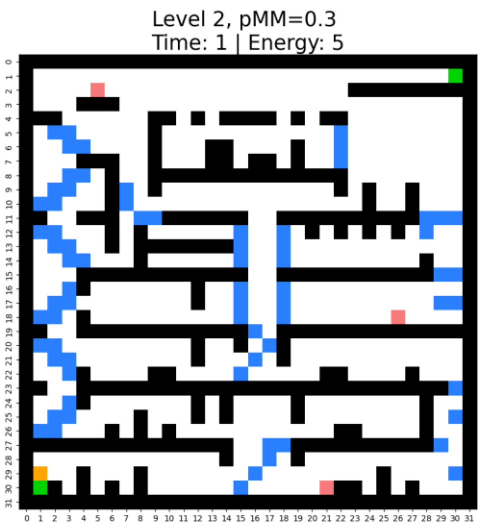
```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

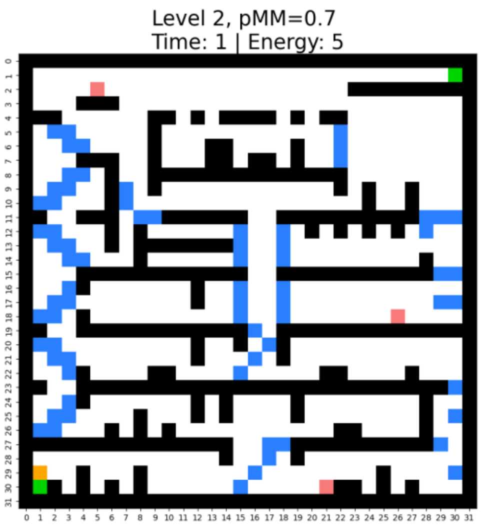
Figure 2: Barto, A. G., & Sutton, R. S. (2018). Reinforcement learning: An introduction. MIT Press. Page 131

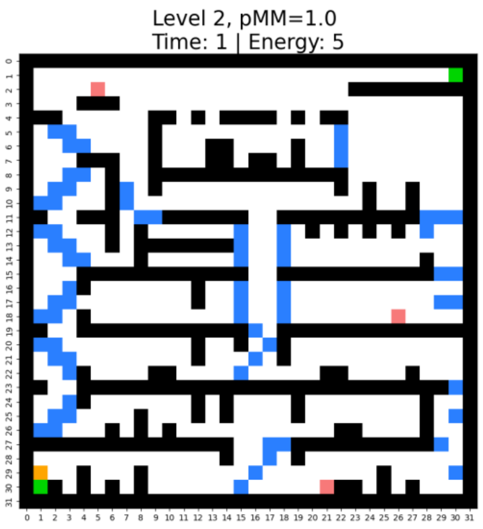
Unlike Q-learning, which updates the maximum action value for the next state, SARSA updates the value of the following action that the agent will take based on the policy it is following. This means that SARSA is an on-policy algorithm, as it learns and updates its policy while following it. In contrast, Q-learning is an off-policy algorithm, as it learns the optimal policy while following a different, exploratory policy.

Results





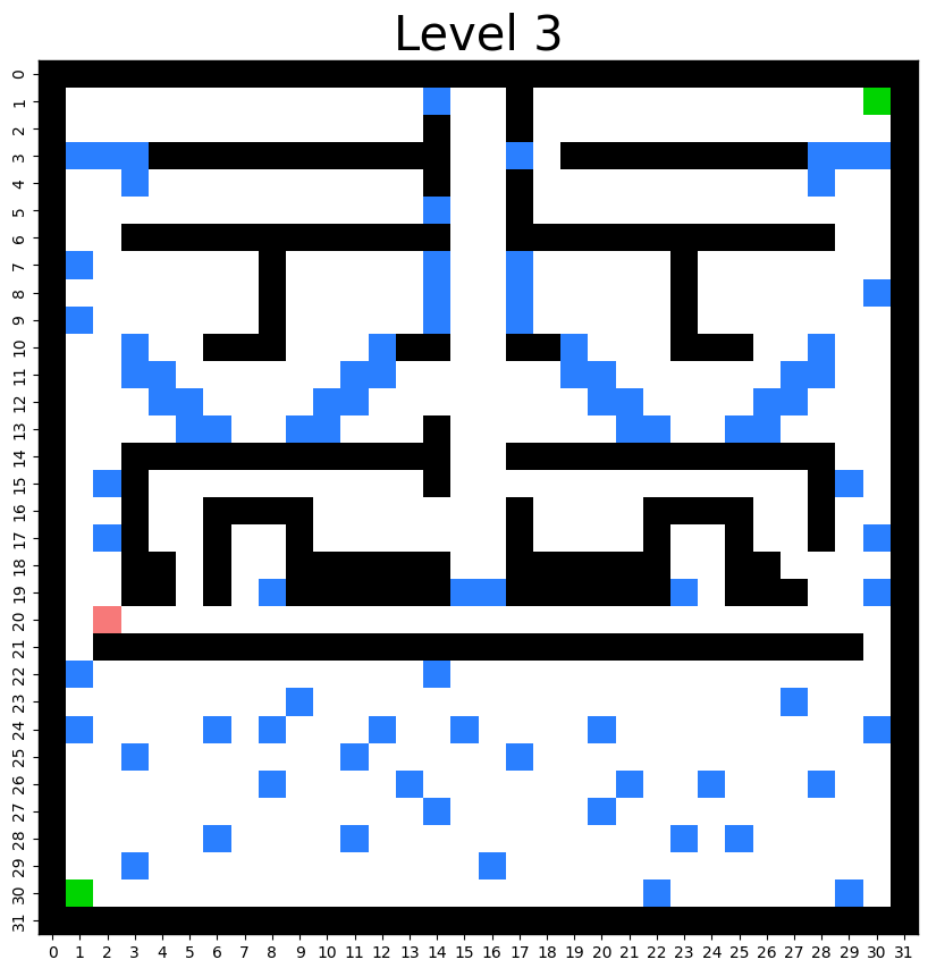




The results were as expected; however, it shows that even a low pMM interferences with the agents learning behavior. I ran for 100 000 episodes, so during that time the chances for Hubert to be taken by an agent were quite significant.

Level 3: The world is moving.

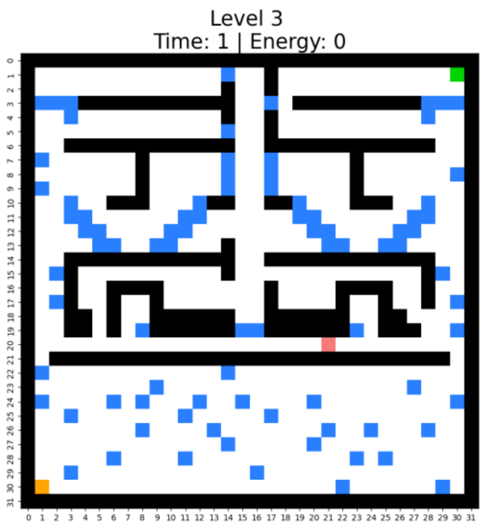
This next environment is significantly more complex than the previous one. Here we have to keep track of the moving agent at all times since it will hugely impact the odds of bypassing the agent when on the same floor.

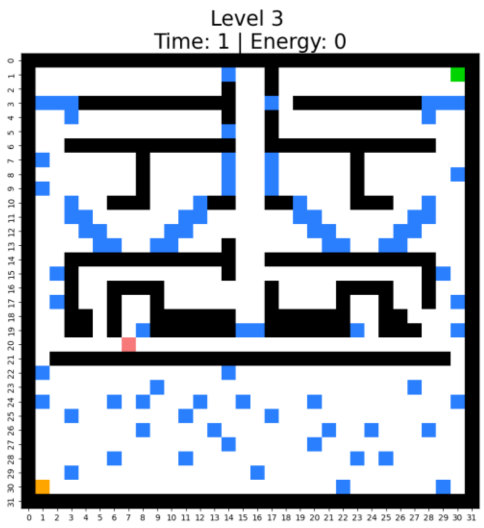


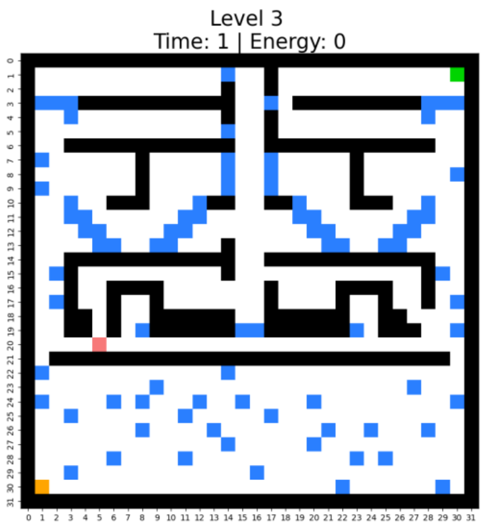
Approach

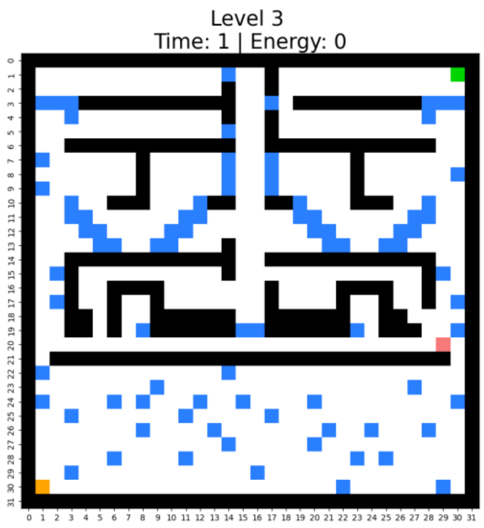
We have to make some adjustments to the existing code and include the agent position in the state space. The agent moves along the x-axis which makes it easier for us to include. This is similar to the previous assignment when we had to keep track of the battery status. To minimize the state space, I decided to only include the MM position in relevant states. This worked great. The position of the MM is only included in the state space in 17 > y > 22.

Results





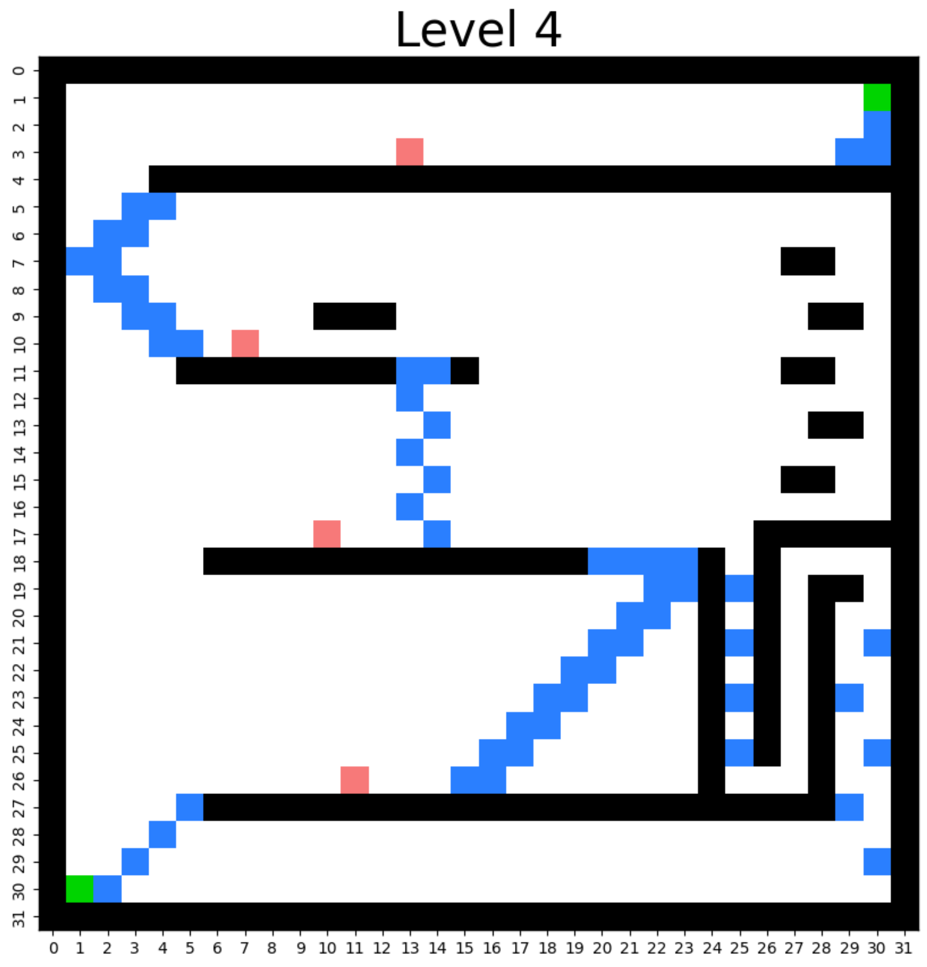




As we can see, Hubert learns how to bypass the MM. The episodes vary from time to time based on the MM. The problem with this problem is that the middle level works as a bottleneck for the entire run, since the MM always taking random action, it forces Hubert to take less risks for getting caught but in the same time he learns how to avoid the MM all together never risking encountering him resulting in getting caught.

Level 4 Hubert hates mingling.

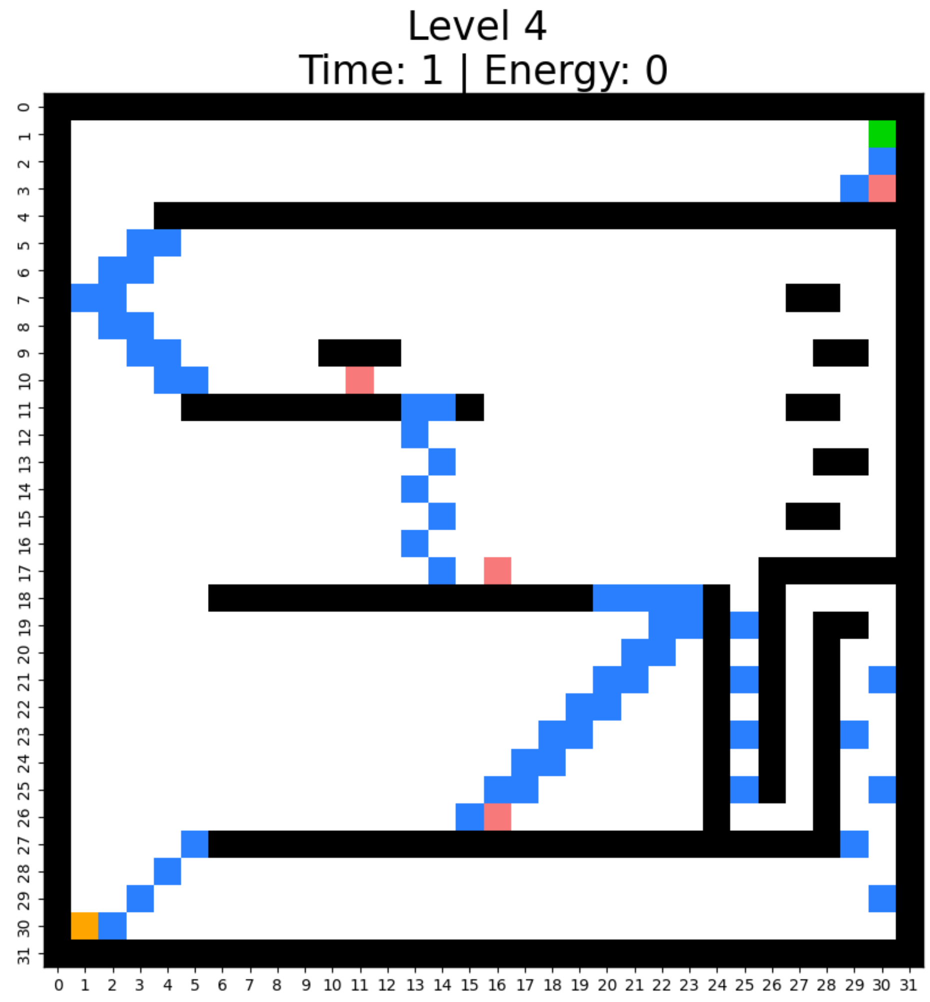
Again, the complexity of the environment increases. This time Hubert has to go from A to B with minimal cost, this means that under any circumstances, Hubert must avoids the fans scattered around the map. So to cope with this, Hubert has gained a skill, a scanning device that scans the area surrounding Hubert. A 9 x 9 grid. This means that instead of only being aware of the current (y, x) position, he scans the surrounding areas for information, which is quite lovely when avoiding annoying fans! It is also worth mentioning that in comparison to the stupid MMs, the fans actually make an effort. This means that if Hubert is on the same floor as a fan, the fan will try to approach him and get his Autograph. But Hubert is tired of all this fame stuff, so this really drains his energy…

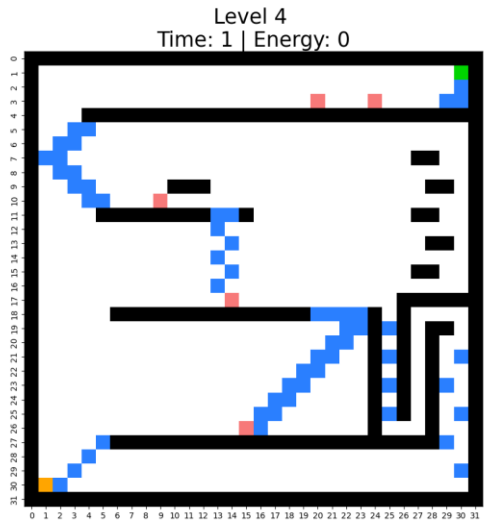


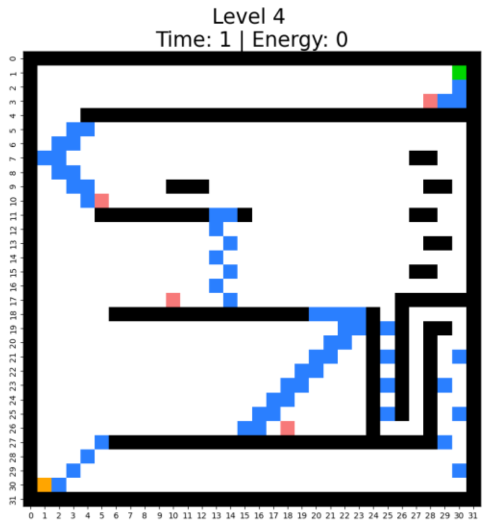
Approach

This problem was significantly harder than the previous ones, however now we can see that Hubert learns to risk encountering them to achieve a good run. As with the previous levels I solved this using SARSA as well, the state space is quite different and bigger. However since we still have the y, x position we only include fans in scan.

Results

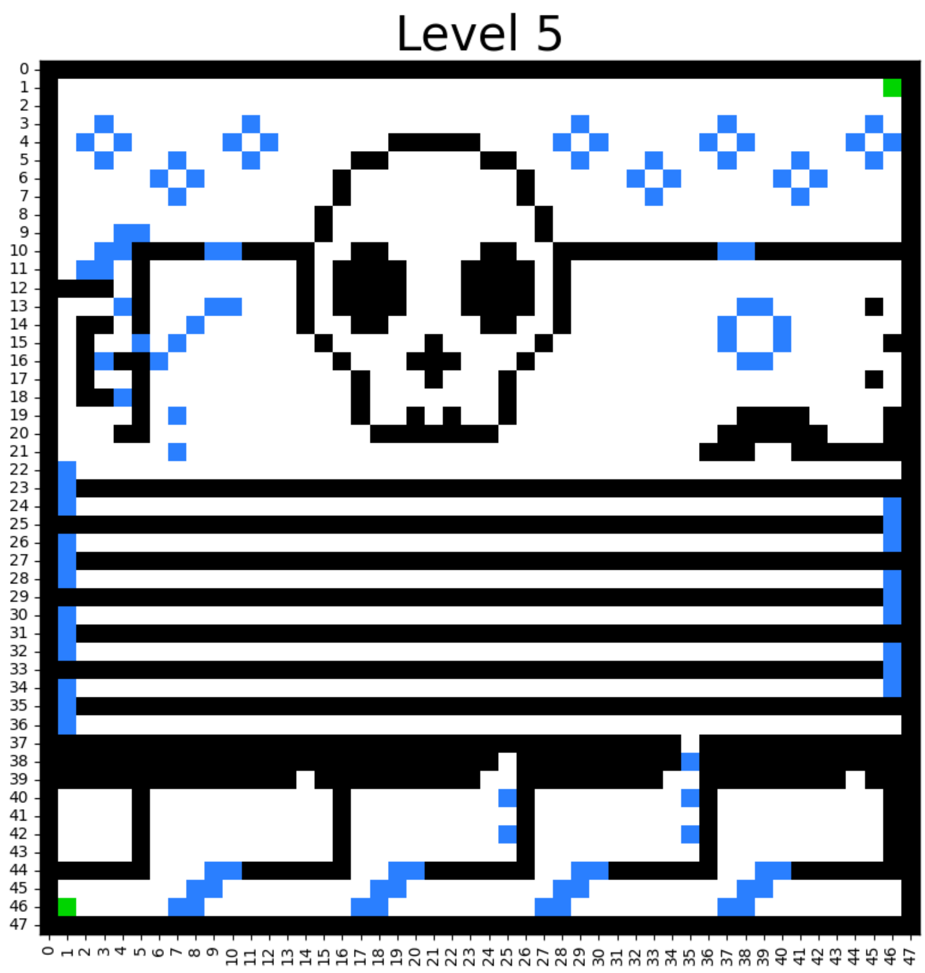






I included three different plots for this level since I wanted to display that even if he learns how to avoid the fans, he risks encountering them even with training.

Level 5: Déjà vu?



Approach

This is by far the most challenging map so far on this journey. What makes this task so unique is that Hubert needs to be made aware of his grid positions, I.e. the (y, x) coordinates of his position, but in exchange, he now has acquired a more sophisticated scanner. He can now scan all elements around him (9 x 9 grid), which includes:

0: Out of grid
1: Air
2: Solids
3: Semi-solids

So now, unlike before his state is the 9 x 9 grid around him on the map. After some testing and research, I quickly realized that it is not sufficient to only use the 9 x 9 scan to achieve a possible good result with the least amount of randomness. I tested the following state representations:

State 1:

scan: 9 x 9 scan around Hubert

n_prev_states: the n previous 9 x 9 scans executed by Hubert

State 2:

scan: 9 x 9 scan around Hubert

action: 0 if go_left or go_right else 1

State 3:

scan: 9 x 9 scan around Hubert

n_prev_actions: the n previous actions executed by Hubert [0, 1]

State 4:

scan: 9 x 9 scan around Hubert

n_prev_actions: the n previous actions executed by Hubert [0, 1, 2]

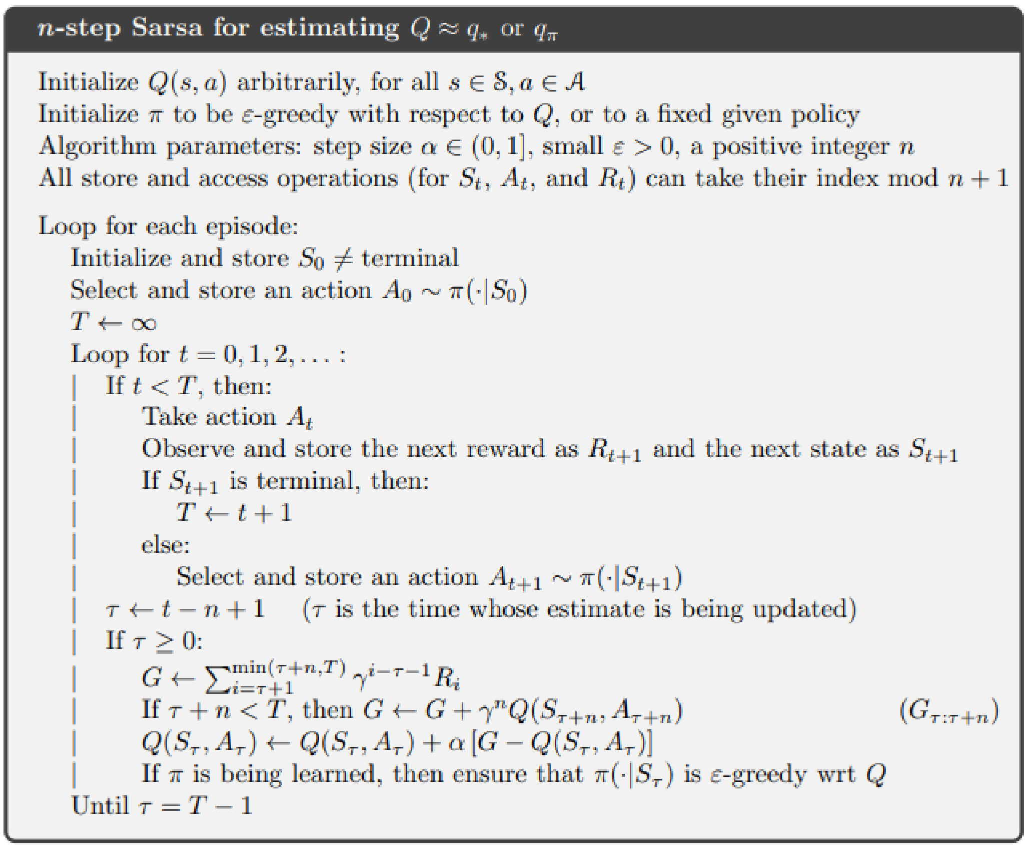
State 5:

scan: 9 x 9 scan around Hubert

- The scan is now more advanced, compared to earlier scans he now includes a different value for previously visited states.

After testing different algorithms, SARSA, n-step-SARSA and Q-Learning, I decided not to use more time solving this and rather focus on level 6 since if I were to succeed there, I will be able to solve this task using the model made there. However, the closest I got to a suboptimal solution was by using n-step SARSA therefore I will go deeper into that.

n-step SARSA



```
n-step Sarsa for estimating $Q \approx q_*$ or $q_\pi$

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be $\varepsilon$-greedy with respect to $Q$, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    Select and store an action $A_0 \sim \pi(\cdot|S_0)$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \ldots$ :
    |   If $t < T$, then:
    |       Take action $A_t$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then:
    |           $T \leftarrow t+1$
    |       else:
    |           Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose estimate is being updated)
    |   If $\tau \geq 0$:
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$        $(G_{\tau:\tau+n})$
    |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$
    |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is $\varepsilon$-greedy wrt $Q$
    Until $\tau = T - 1$
```

Figure 3: Barto, A. G., & Sutton, R. S. (2018). Reinforcement learning: An introduction. MIT Press. Page 147

***n-step* SARSA** is an extension of the SARSA algorithm. It aims to improve the learning efficiency by incorporating information from multiple steps into the update process.

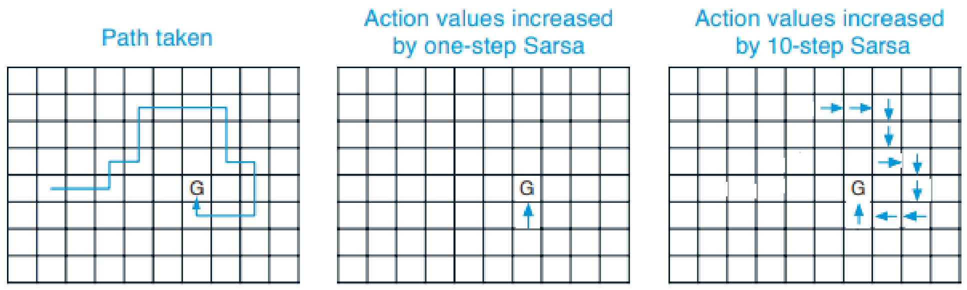

Path taken
Action values increased by one-step Sarsa
Action values increased by 10-step Sarsa
G
G
G

Figure 4: Barto, A. G., & Sutton, R. S. (2018). Reinforcement learning: An introduction. MIT Press. Page 147

n-step SARSA expands on this idea by considering multiple steps ahead in the future. Instead of updating the value function based on a single step transition, it considers a sequence of n steps. This allows the agent to incorporate information about the future rewards and actions more effectively.

By combining this with a more complex state (combine state 4 + 5) the agent was able to navigate the environment more efficiently and converge towards the goal.

Results

Unfortunately I was not able to generate a GIF for this problem, this Is mainly because I did not want to pick the best found solution founs in a million iterations, but rather provide honest and feasible solutions. Level 5 had an average on over 500 moves per episode, this resulted in very heavy GIFs. The n-step-sarsa and agent has both managed to solve the problem better than random, but not anything impressive. They are both included in the repo.

Level 6: Up, up, and away!

Let's first start with creating the environments. Environment 6 is more complex than the previous ones, but we have learned and can now implement the functionalities as we want (hindsight is a pain in the ass).

In contrast to previous levels, we now have the following values appearing on the grid.

0: Out of grid
1: Air
2: Solids
3: Semi-solids
4: Fans
5: MM

All of these will be visible to our best friend Hubert when he is roaming the map. And hopefully, he will understand what each and one means.

State space:

Approach

I tried multiple approaches and hit the wall n ******* times, everything from Deep Q-Network to Actor-Critic PPO implementation by me. After hours and hours of testing, debugging, and tweaking the reward function, I realized I could not achieve what I wanted by implementing everything myself. Therefore I decided to use the implementation by CleanRL after advice from the TA's (https://github.com/vwxyzjn/cleanrl/blob/master/cleanrl/ppo.py). The reason is that even though I understand the surrounding elements of the implementation, I could not achieve the results I wanted with my own implementation.

Experimental setup

The setup consists of cleanRL's PPO implementation with a modified network to fit my custom state space. During the experiments we tested out multiple states and reward functions to better fit the model. We also had to do some modifications to the environment.

We use the same as in level 5 with some changes.

1. We now include fans. A fan is represented as integer 5.
2. We now include MM's. A MM is represented as integer 6.
    a. The MM's also now ends the episode, rather than sending you back to start.

Proximal Policy Optimization Algorithm

Proximal Policy Optimization (PPO) was first introduced back in 2017 by Schulman et al. where they presented impressive results comparing to state-of-the-art implementations such as Deep Q Learning (DQN)

The Actor-Critic algorithm combines the benefits of both policy-based and value-based methods. It utilizes a hybrid architecture that consists of an actor network (policy)[ref figure 1] and a critic network (value function)[ref figure 2]. This architecture allows the algorithm to leverage the strengths of both policy-based and value-based methods. The actor learns to select actions based on the observed state (9 x 9 grid), while the critic network estimates the value or expected return associated with that state. This estimation provides valuable feedback for the Actor network to improving its policy to maximize reward.

States

For the state I decided to include the n previous states, this is so that he can learn to advance in the environment by seeing where he was previously, this is mainly since our environment is so basic it only contains integers from 1 to n based on the elements we add. Therefore the state remained as a 9 x 9 scan, which is nice for a CNN. In afterthought, it might be better with a combination of the scan and the previous actions instead of just the scan.

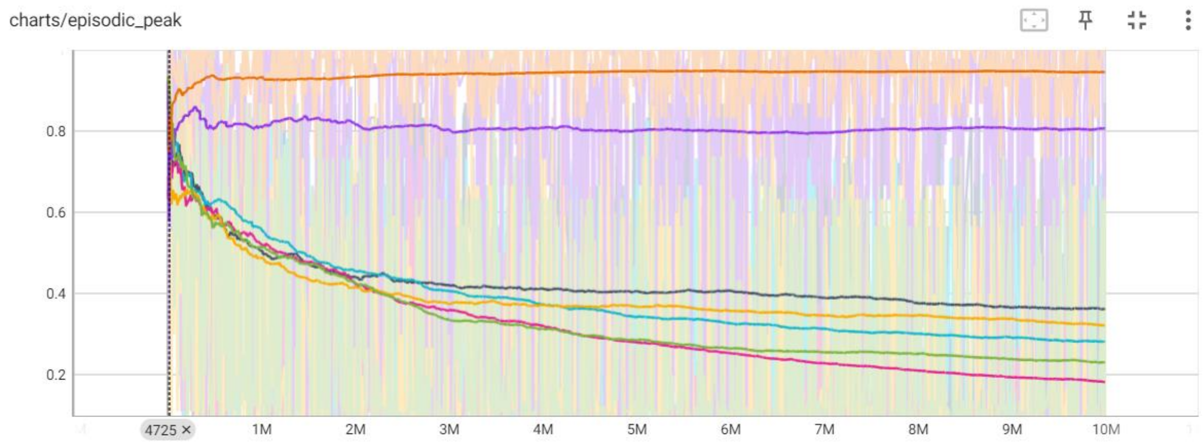
Reward functions

The reward function is what tunes the policy for the agent. It's here we can encourage "good" behavior and penalize "bad" behavior. There are multiple approaches to achieve our goal about reaching as high as possible.

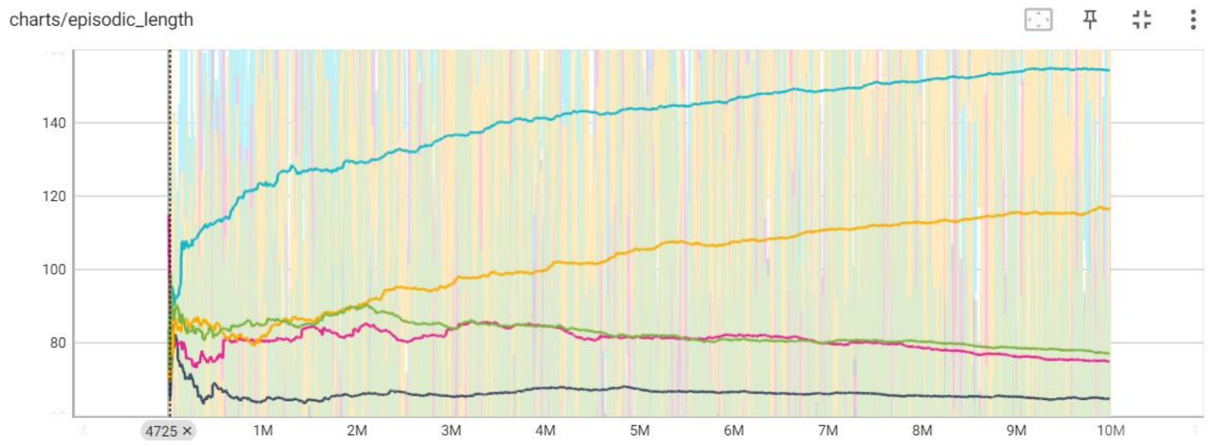
In the end we ended up with seven different reward functions, some share the most basic elements while others are complete opposites. However, all had the same goal in mind, reaching the top.

Analyze

The most obvious thing to start analyzing is to log the peak, i.e. the highest point the agent reaches during an episode.



As we can clearly see there are two reward functions that are clearly not working, the orange one is “no_negative” and the purple is “high_rise”. The one thing they both have in common is that there is no punishment for doing bad actions. So from now on we cut these out of the analyze. Then we’re left with 5 reward functions, how can we separate them? We start looking at the avg episode length:



Here we can see that they are not so similar, keep in mind that our agent has one main objective in mind:

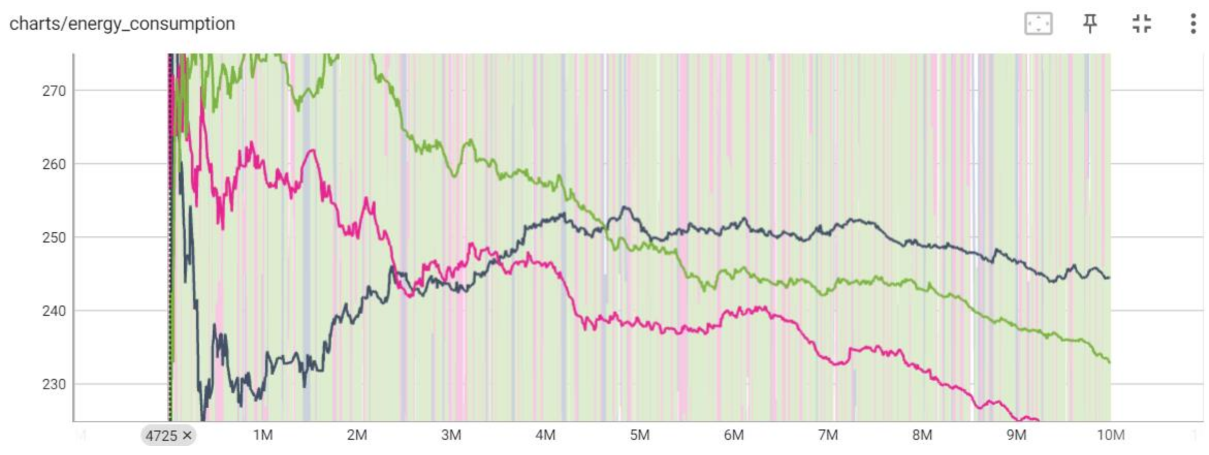
- Get as high as possible in a given environment.

But we want to extend this requirement to:

- Get as high as possible in a given environment in as few steps as possible.

After analyzing we decided to cut the “base” and “no_negative” reward functions, this is working well enough but not in as few steps as we’d like.

Were now left with three reward functions and the following energy consumption:



This gives us an idea of how “cheap” the reward function is on Hubert, i.e. how much energy does he uses to reach the goal. We can see that they are all within 225 – 245. So in conclusion we have three decent agents.

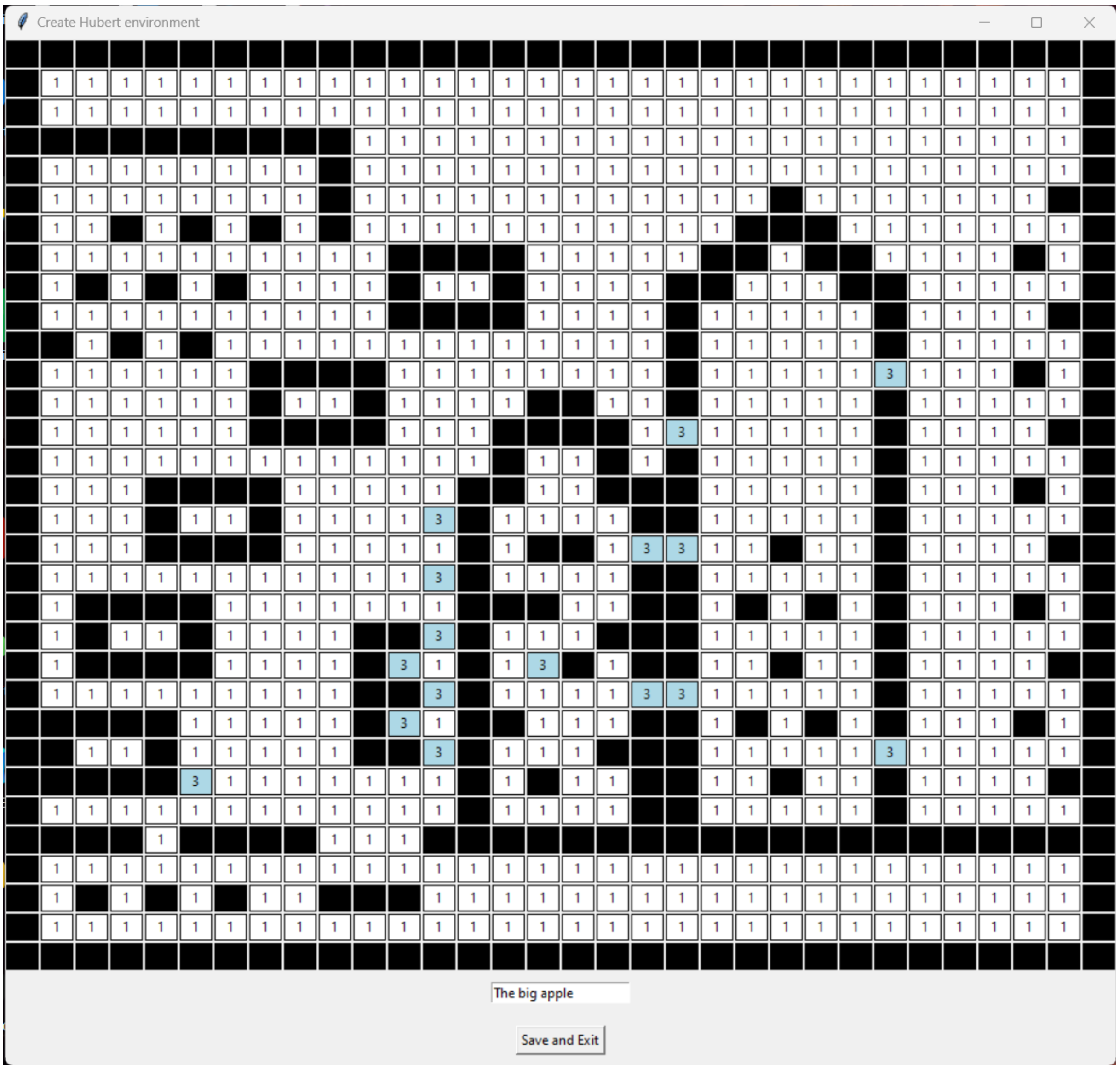
Result

The model was not able to generalize well enough, this is mainly because of computational power. With only 32GB ram I was not able to use the number of environments required to generate and learn a good policy. The agent is okey in the performance, however he is not optimal. I was hoping that I would be able to make use of the UiB server, however that was not an option the later days.

Bonus level

Environment Creator

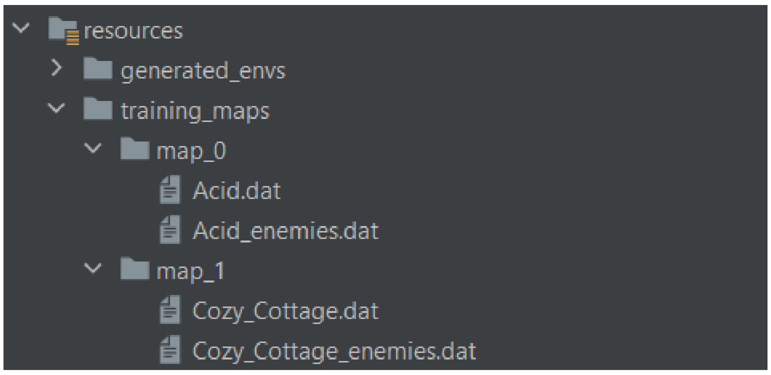
To be able to train the agent efficiently, I was forced to think out of the box. Since I decided to use NN in PPO it's important to have sufficient data to train on. Therefore I came to the conclusion that I had to create an environment-design interface to easily create new and complex environments (heavily inspired by Audun's from last year).



The app works like this. You click on the tiles, each tile can either have 1: Air, 2: Solid and 3: Semi-solid. After completing the environment, you type in the name you want to call it and then click save, the environment will be saved in resources/generated_envs folder. The app is created using tkinter and works by manipulating an array.

Agent testing

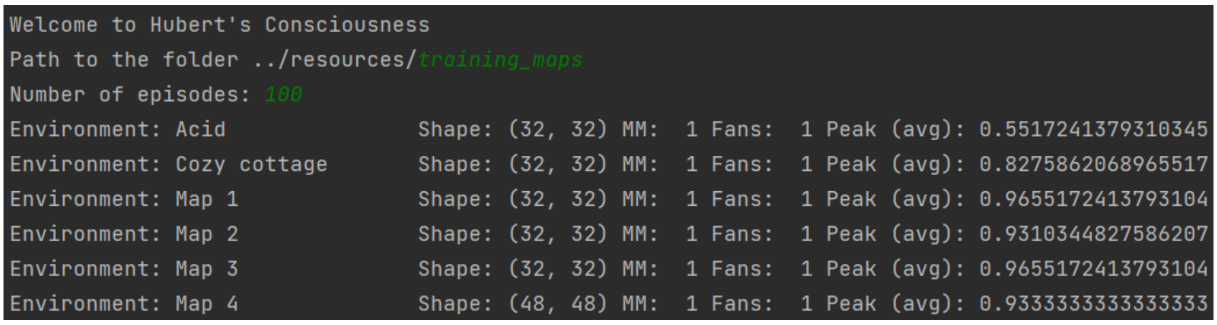
The agent testing file is named test_agent.py and is designed to test my agent on any given environment. The agent is already pre-trained and ready to solve any given problem heading his way. For it to work you must have the following structure of the environment files:



```
resources
  generated_envs
  training_maps
    map_0
      Acid.dat
      Acid_enemies.dat
    map_1
      Cozy_Cottage.dat
      Cozy_Cottage_enemies.dat
```

This is to ensure stability and avoid errors when compiling and generating environments.

After doing so, you can run the test_agent.py file. The following prompt will appear. You type int the path from resources to the folder of environments and the number of episodes Hubert gets to use on each level:

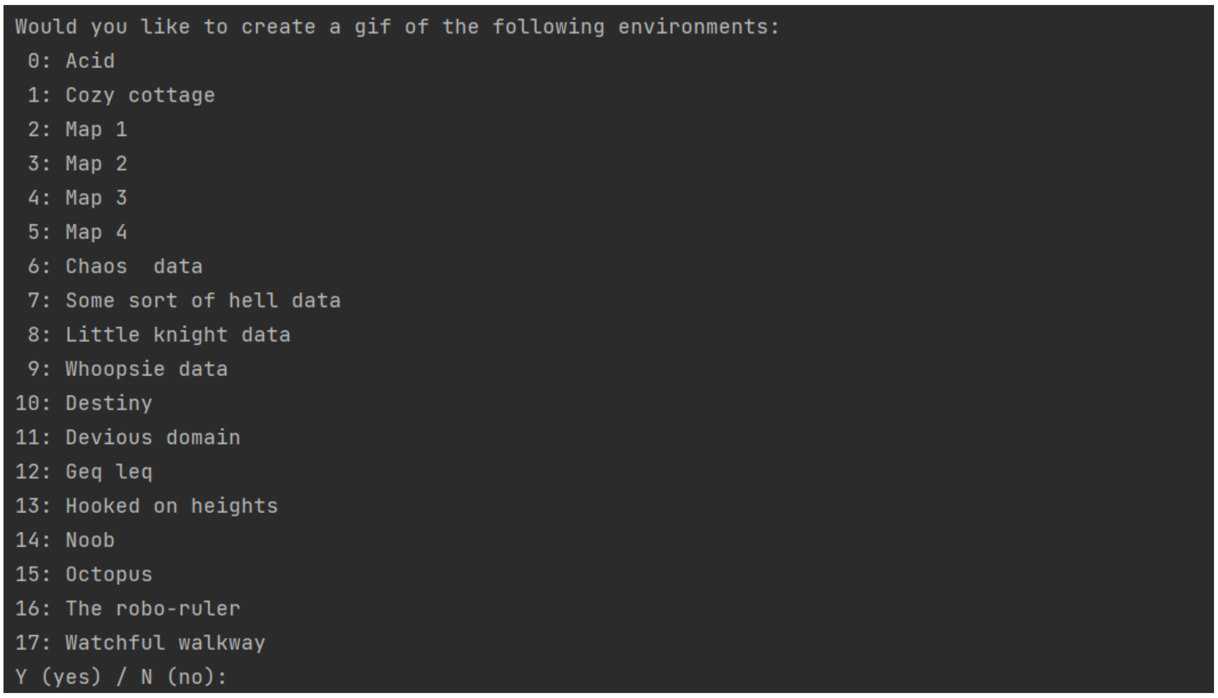


```
Welcome to Hubert's Consciousness
Path to the folder ../resources/training_maps
Number of episodes: 100
Environment: Acid               Shape: (32, 32) MM:  1 Fans:  1 Peak (avg): 0.5517241379310345
Environment: Cozy cottage       Shape: (32, 32) MM:  1 Fans:  1 Peak (avg): 0.8275862068965517
Environment: Map 1              Shape: (32, 32) MM:  1 Fans:  1 Peak (avg): 0.9655172413793104
Environment: Map 2              Shape: (32, 32) MM:  1 Fans:  1 Peak (avg): 0.9310344827586207
Environment: Map 3              Shape: (32, 32) MM:  1 Fans:  1 Peak (avg): 0.9655172413793104
Environment: Map 4              Shape: (48, 48) MM:  1 Fans:  1 Peak (avg): 0.9333333333333333
```

After doing so, Hubert will start to explore and solve each environment.

Column 1: Name of the environment
Column 2: The shape/size of the environment
Column 3: Fans (1 == True / 0 == False)
Column 4: MM (1 == True / 0 == False)
Column 5: Average over n_episodes.
Column 6: Standard deviation over n_episodes.

At completion, you will be given the choice to create gifs of how Hubert solved the different environments.



```
Would you like to create a gif of the following environments:
 0: Acid
 1: Cozy cottage
 2: Map 1
 3: Map 2
 4: Map 3
 5: Map 4
 6: Chaos  data
 7: Some sort of hell data
 8: Little knight data
 9: Whoopsie data
10: Destiny
11: Devious domain
12: Geq leq
13: Hooked on heights
14: Noob
15: Octopus
16: The robo-ruler
17: Watchful walkway
Y (yes) / N (no):
```

If you want a gif, you type the index of the environment you wish to a gif of, and it’s automatically created and saved in project/Levels/gifs. Be aware that this might take some time, considering it’s only creating one at a time. This could be optimized easily. However this was not a priority, but rather a bonus on the bonus.

Retrospective

As we progress from level 1 to level 6 and the bonus level, we encounter various challenges in the realm of reinforcement learning (RL). In level 1, our focus is on testing and evaluating different strategies to maximize profit, resembling the multi-armed bandit problem.

In level 2, we delve into the convergence of different algorithms to a global minimum, examining scenarios both with and without obstacles placed in the environment, each with varying probabilities of interference. In this level, we adopt a greedy policy by setting the epsilon to 0.

Level 3 presents a significant bottleneck in the form of a moving obstacle (MM) in the hallway. While still employing a greedy policy, finding the goal becomes more time-consuming due to the MM obstructing the path 80% of the time.

Moving on to level 4, we witness a significant change in state representation, resulting in a considerably larger state space. This level introduces the presence of fans, which uniquely impart high negative rewards instead of resetting the agent to the starting point. This distinct characteristic significantly influences the agent's behavior.

Level 5 demands creativity as it poses the first major obstacle in the assignment. The map is intentionally designed to deceive or mislead the agent into making suboptimal decisions based on environmental similarities. After persistent efforts, employing different algorithms, we were able to achieve results that further fueled our motivation for level 6.

Level 6, the pinnacle of the assignment, proves to be the most challenging. Here, we encounter the need to fine-tune and adjust the reward function extensively to accomplish our goal. The importance of tracking, plotting, and in-depth analysis becomes paramount. While progress could be measured based on conversion rates in previous levels, level 6 demands a more comprehensive consideration of various factors for evaluating success.

References

Sutton, R. S., & Barto, A. G. (2020). Reinforcement Learning: An Introduction. Retrieved from http://incompleteideas.net/book/RLbook2020trimmed.pdf

FrameStack - Gym Wrapper. (n.d.). Retrieved from https://www.gymlibrary.dev/api/wrappers/

CleanRL. (n.d.). PPO - Proximal Policy Optimization. CleanRL Documentation. https://docs.cleanrl.dev/rl-algorithms/ppo/#implemented-variants

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347. https://arxiv.org/pdf/1707.06347.pdf