687 Final: MDP for Survival Game Five Nights at Freddy's

John Ryu, Nathaniel Wai, Ryan Zaid

Problem Description:

We chose to create a custom MDP and apply three existing RL methods to it. Our environment was the survival horror video-game, Five Nights at Freddy's (FNaF). In the game, the player watches over animatronics inhabiting a family pizzeria, and must survive for the full duration of the night shift without letting the animatronics attack them. Each of the 4 animatronics has a different stochastic movement behaviour to move between rooms, before eventually arriving at the player's location, where they attack. The player, being unable to leave the office, has several tools to defend themselves. First, they may look through office cameras into 11 different rooms to locate the animatronics, which freezes them momentarily. Second, they control two office doors to block animatronics from entering and attacking. Third, they may use office lights to check for animatronics just outside of their office. All tools deplete battery, which is a limited resource that cannot be regained. If the player runs out of battery, all tools become inoperable, leaving them defenseless to the animatronics, and in a losing state. Succeeding at FNaF requires learning and strategizing against the distinct animatronic behaviours, while making crucial decisions to save power, which we believe translates to an interesting MDP problem.

Similar Problems and RL Techniques:

From our research, we did not find implementations of reinforcement learning agents learning to play FNaF. We did find Non-RL methods including pre-programmed state-machines [a] [b] that react to the presence of animatronics on the screen with predefined code. However, they are limited in their static implementation, and so are not adaptable to changes in the environment.

We instead focus our attention towards games that have been solved by RL which we believe revolve around similar gameplay mechanics as our MDP implementation, in which an agent must survive as long as possible with finite resources.

Lunar Lander

The first of these environments which we identified was Lunar Lander, a popular reinforcement learning environment which involves an agent learning to safely land a lunar lander while being careful to not use too much fuel. Our main takeaway from researching this environment was learning about popular techniques used to solve these types of MDP's. This implementation by Julian Kappler appears to achieve high performance with the actor-critic algorithm and also with deep-Q learning. Actor-critic appears to take more time to train, however both algorithms appear to have similar probability distributions of return. We believe this environment was similar enough to our own, since the design of the MDP also requires the agent to manage its resources carefully in order to maximize reward.

Snake

We also researched implementations of the game, Snake. In this implementation by DragonWarrior15, the agent must control a snake within a grid, which grows in length as it eats food which spawns randomly in the grid, in a space not currently occupied by the snake's tail. The game terminates when the snake either runs into one of the walls of the grid or runs into its own body. This implementation included a simple reward structure (i.e. +1 for eating food and -1 for terminating) and a complex reward structure, however, it appears that both reward structures result in similar trends in the mean length of the snake at termination. This motivated us to also implement a simple reward structure in our own implementation in the interest of efficiency.

MDP Definition:

Prior attempt at POMDP

In our initial attempts to emulate FNaF, we made the action of checking cameras reveal the location of animatronics to the agent. However, this implied that the agent did not know parts of the state, making the formulation a "Partially Observable Markov Decision Process" (POMDP). In a POMDP, the agent's observations when in a single state are a probability distribution over all the states. The agent must build a policy that maps beliefs about its state to actions (rather than solely states to actions). Given that the POMDP requires more training time and more complex algorithms than an MDP, we instead assumed that the agent would always be aware of the locations of all the animatronics, which eliminated much of the challenge of the game.

Simpler, Non-POMDP Model:

In order to model the game as a MDP, we made a considerable amount of changes. The core gameplay remains the same. The player is confined to their office space, and they must use tools to keep animatronics away from their office for as long as possible. First, since the agent always knows animatronic locations, the camera solely works to freeze a selected animatronic, preventing their movement. Second, door lights have also been removed, since they only provide utility for locating animatronics. Third, we represent time on discrete intervals, based on the approximate time a player would take in-game to complete each of the actions (i.e. checking cameras, closing doors, etc.). A night in the actual games takes 535 seconds (8 minutes and 55 seconds), so we made each timestep represent one second of in-game time. Fourth, to ensure we decrease the number of animatronics to the two most distinct: Chica, who resets back the beginning when blocked by a door, and Freddy, who approaches the player, and does not return to the beginning. We find that this results in an interesting challenge between freezing Freddy

with the cameras and blocking the door. Finally, we adjust the animatronic movement behaviour, eliminating timers, as detailed in the “Transition Function” section.

Implementation:

We utilized gymnasium for the implementation of our MDP, stable_baselines3 for the implementation of our RL algorithms and optuna for hyperparameter tuning.

States:

While the variables below are defined in a discrete manner, we utilized gymnasium’s Box class, which implements all of these variables as continuous values. We believe this made the most sense since continuous variables work best with algorithms which utilize neural networks.

- Door Left: {0, 1} [Open / Closed]
- Door Right: {0, 1} [Open / Closed]
- Timestep: $\{x \in \mathbb{Z} \mid 0 \leq x \leq 535\}$
- Battery: $\{x \in \mathbb{Z} \mid 0 \leq x \leq 100\}$
    - The battery always decreases by 1 every timestep by default. The amount decremented increases by 1 for every tool currently in use (left door closed, right door closed, camera focus). For example, if the user has toggled both doors closed and is currently focusing Freddy, they will drain 4 battery points in that timestep.

States For Every Animatronic (Chica/Freddy):

- Location: $\{x \in \mathbb{Z} \mid 0 \leq x \leq 3\}$
    - Defines where the animatronic is within the map, the higher the index, the closer they are to the ATTACK state. The indices of each room is also noted in the diagram in the Transition Function section.
- in_office: {0, 1} [False / True]

    - Defines whether or not the animatronic has successfully attacked the player.
- focused: {0, 1} [False / True]
    - Defines whether or not the player is currently focusing their camera on the animatronic. A focused animatronic cannot move.

Initial State:

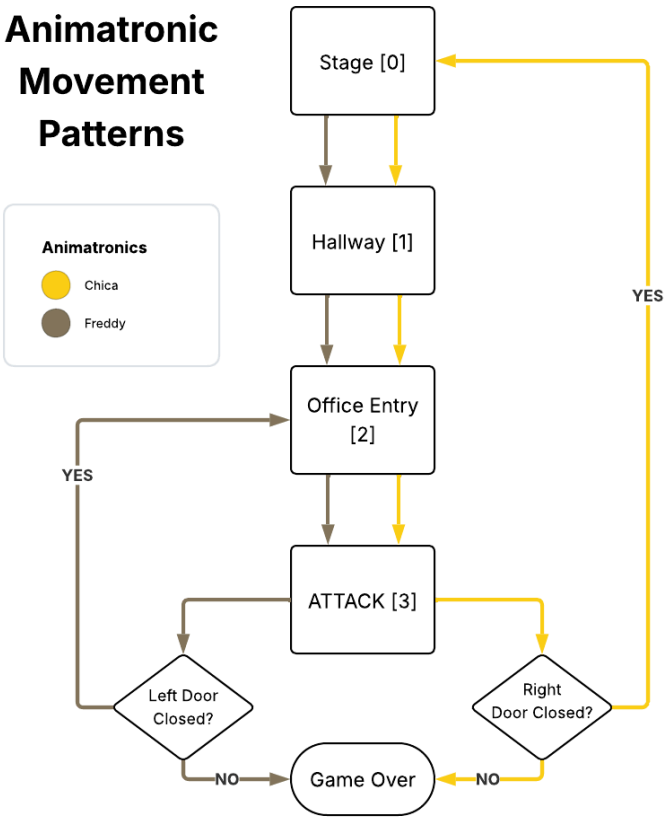
Every initial state for this MDP is the same. All animatronics begin at location = 0 (the stage), are not focused and are not in the office. Doors are initialized to open, timestep is equal to zero and battery is equal to 100.

Action:

The agent may toggle either door, which will switch it from open to closed and vice versa. They may also check either animatronic, which turns focus off of the other animatronic if they were being previously focused. They may also opt to turn the camera off, which will unfocus all animatronics. The agent may also perform no actions. The action space was implemented as a gymnasium discrete space, which totaled 6 possible actions: (toggle_left_door, toggle_right_door, check_camera_chica, check_camera_freddy, camera_off and no_action)

Transition Function:

The only stochastic element of our MDP is the movement of the animatronics. Every timestep, Chica and Freddy have a 0.5 * (1/5) = 0.1 probability of advancing to the next room. This is a simplification of the in-game movement system, where the animatronics move around a larger more complex map and also check to see if they can move approximately every 5 seconds instead of every second. This roughly mirrors animatronic movement frequency in the real game, when moving with a probability of 0.5, which would only occur on a rather difficult setting.



The above diagram portrays each animatronic's movement patterns. Every timestep, the animatronic has a 0.1 probability of advancing to the next room in the movement pattern defined in the diagram. If the animatronic enters the ATTACK room, they are guaranteed to attempt to attack the player. If the door is open, they will successfully attack and end the game (and the episode terminates). If it is closed, they will be sent backwards according to the diagram.

Reward:

The agent is simply given +1 for every timestep it is alive and receives -1 if they are attacked by an animatronic (when the episode terminates).

Hyperparameter Tuning:

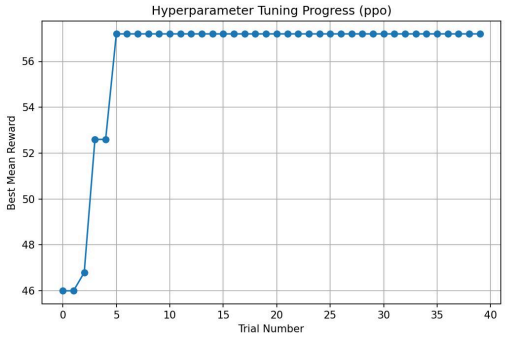
Hyperparameter tuning was performed using a pre-existing automated framework called Optuna. Optuna structures its tuning into studies. One “study” consists of multiple “trials”, or runs of a given algorithm on the MDP. Mean reward is measured at each trial and compared to the previous trials. The output of our hyperparameter tuning code is a JSON object containing the best found hyperparameters for each model, of which we then use to train and evaluate our final models. We ran 40 trials before termination, and tried a range of standard hyperparameters per algorithm, listed below. Note that some hyperparameters, though the same across algorithms, had different testing values due to the nature of these algorithms. For example, all values of n_steps in ppo are far larger than a2c because ppo uses long steps updating batches, while a2c uses shorter steps. Also note that some hyperparameters selected within a range are selected in a random, continuous uniform fashion while others are selected in a random, logarithmic uniform fashion. This is done for certain parameters such as ent_coef to sample smaller values more densely than larger values, as larger entropy values encourage far too much exploration. It still is worth testing a few of these however, hence the wide range.

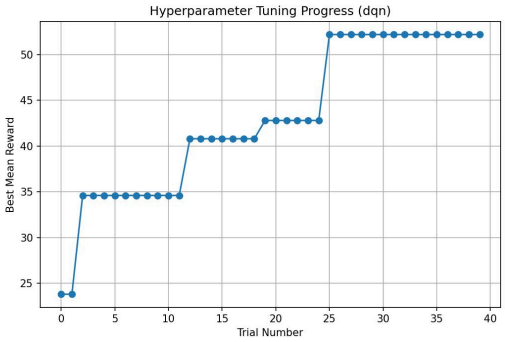
TESTED PARAMETERS

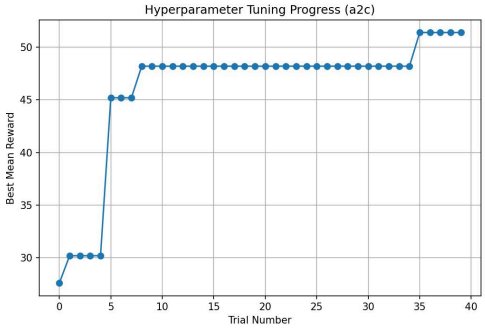
Hyperparameter	DQN	PPO	A2C
batch_size	32, 64, 128, 256	32, 64, 128, 256	—
buffer_size	10000, 50000, 100000	—	—
target_update_interval	100–2000	—	—
exploration_fraction	0.1–0.5	—	—
exploration_final_eps	0.01–0.2	—	—
n_steps	—	256, 512, 1024, 2048	5, 10, 20, 50
n_epochs	—	3–20	—
clip_range	—	0.1–0.3	—
ent_coef	—	1e-5–1e-1	1e-5–1e-1
vf_coef	—	—	0.1–1.0
max_grad_norm	—	—	0.3–1.0

Results:

Hyperparameter Tuning



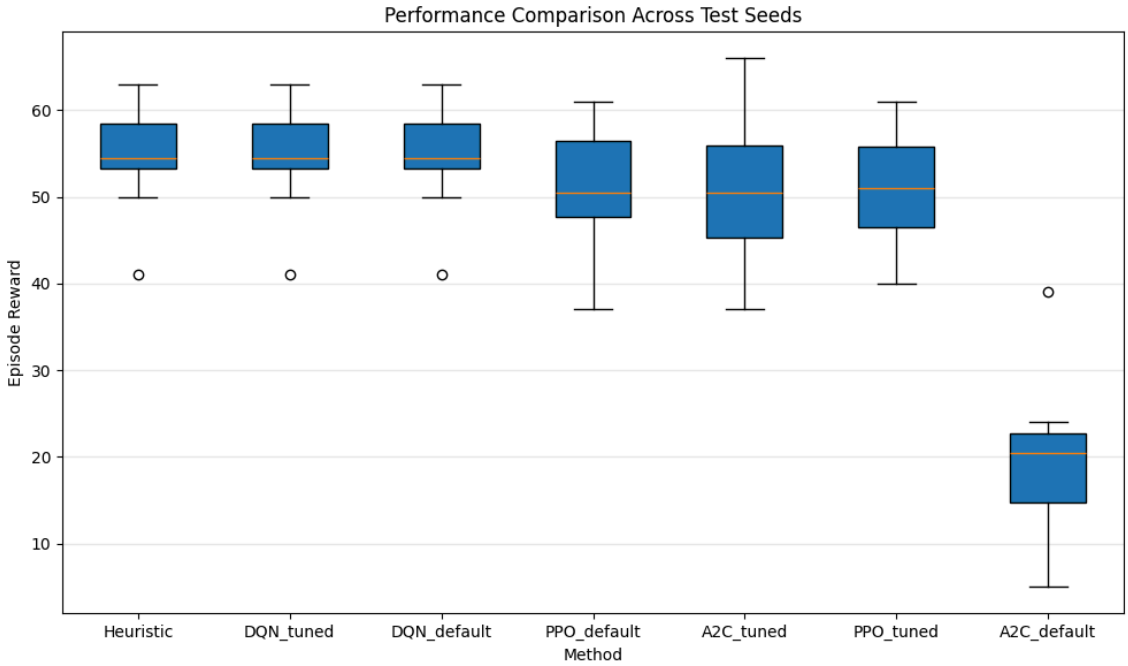




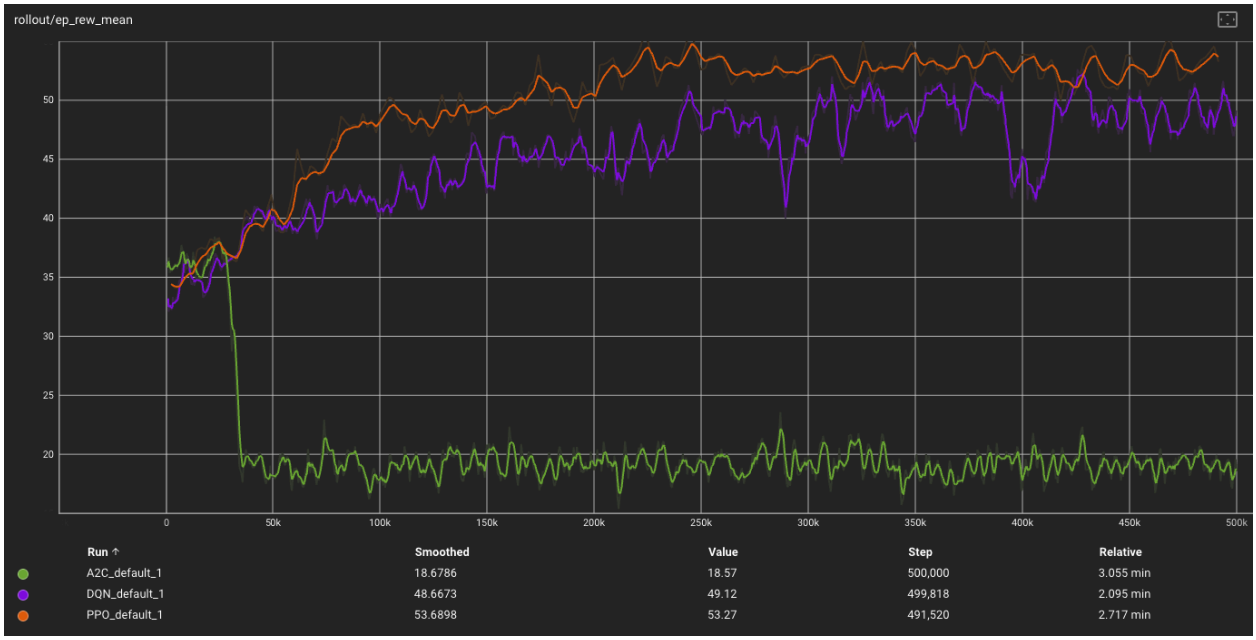
Above: Mean Reward over Trial number for each algorithm

Algorithm Performance:

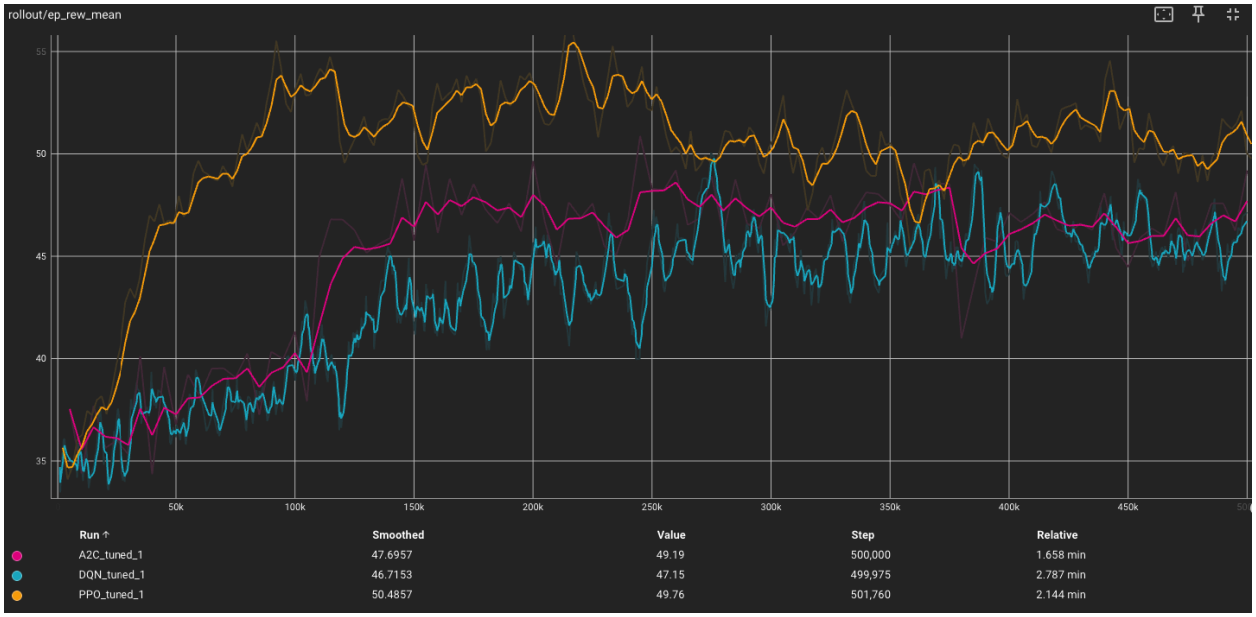
Every algorithm, except for A2C Default, learned the simplified MDP version of FNaF approximately just as well as our heuristic. This resulted in a score of surviving for >50 seconds. In the actual game, power decreases at a rate 10-times slower, so we believe this is a near-optimal score given the decreased battery capacity. DQN and DQN-tuned were able to learn the best with less variance than PPO and A2C. These figures also show that hyperparameter tuning was effective, as it contributed to a large improvement in performance for the A2C agent.



Above: Boxplots of reward of each algorithm on an evaluation set of 10 runs.



Above: Episode rolling average mean over 500,000 training episodes on algorithms with default parameters



Above: Episode rolling average mean over 500,000 training episodes on algorithms with hyperparameter-tuned parameters

Above: Episode rolling average mean over 500,000 training episodes on algorithms with hyperparameter-tuned parameters and also default parameters

Member Contributions:

- **John**: Contributed to designing and implementing MDP in python using gymnasium, implemented code for heuristic and code for training and evaluating stable baselines 3 models. Contributed to writing the report, namely researching similar problems, problem description and MDP description.
- **Nathaniel**: Researched FNaF game code, contributed a large portion towards designing MDP, contributed to designing simple baseline heuristic. Contributed to writing the report, namely problem description, MDP description and results analysis.
- **Ryan**: Contributed to designing MDP and implementing MDP in python using gymnasium and implemented code for running hyperparameter tuning. Contributed to writing the report, namely MDP description and hyperparameter tuning analysis.