HIGH SCHOOL EDITION
Journal of Student Research
Volume 12 Issue 4 (2023)

Reducing Time Steps Needed of Multi-Agent Reinforcement Methods in a Dynamic Navigation Environment

Ziduo Yi[1], Fateme Golivand[#] and Brian Wescott[#]

[1]Stephen F. Austin High School, USA
[#]Advisor

ABSTRACT

In a simpler version of the delivery variation of the multi-agent pathfinding problem, we use a reinforcement learning approach instead of other common traditional algorithmic heuristics to tackle this NP-hard problem. The paper proposes using either fully decentralized or decentralized data and a centralized model to alleviate the problem of requiring large training time steps in multi-agent reinforcement learning by reducing the number of time steps needed, especially in an environment with cumulative rewards. These approaches don't require the concatenation of local data among agents, and, as such, are relieved of the computational burden of increased data complexity. The proposed approaches were tested in an environment where multiple agents try to work together and clean the maximum number of total nonregenerative unclean tiles in a grid filled with obstacles in a set amount of time. In a final test on a randomized dynamic environment with varying grid sizes, one of our approaches proved very promising in application and scalability. This paper's most commonly used algorithm was proximal policy optimization, with brief mentions of deep q-network. Different implementations of these algorithms had their training tracked and recorded, including Stablebaseline3 and self-implementation using TensorFlow.

Introduction

Since the computer program Deep Blue made by IBM was able to defeat the chess world champion at the time, Garry Kasparov, in a seven-game set, the idea that computing advances could be used to minimize and optimize human performance in a specific area has been constantly tested [1]. With the introduction of Artificial Intelligence (AI), however, computer technologies were enabled to mimic the complex human decision-making process using an oversimplified version of the brain. In many cases, AI has been shown to not only optimize human performance but also create strong heuristic solutions in traditionally NP-hard problems, a subset of problems where the time complexity cannot be expressed as a polynomial, which makes it too time-intensive to scale. In particular, reinforcement learning and its multi-agent counterpart have made a breakthrough in the last decade in their feasibility in the real world, being used to dominate any human player in complex one-on-one games like Go and beat top professional players in dynamic cooperative games like Dota along with many other achievements without preexisting game knowledge [2] [3].

The impacts of the implementation of AIs in software have enhanced the user experience, which is especially apparent in the presence of social media platforms. In the area of autonomous driving especially, many AI approaches, including reinforcement learning, have been extensively researched and tested. In a similar sense to what we are trying to solve in this paper, by analyzing the area around it using cameras, an AI program can follow patterns in the road, identify its surroundings, and then safely follow a determined path by GPS and pathfinding algorithms to reach its target. While controlling a single car's actions is a single-agent

problem, many interactions between cars, like allowing one another to merge and creating optimal spacing between cars in lanes, require cooperation between multiple cars to make autonomous driving safer and more efficient. When expanding on this idea, a similar approach can be applied to the pathfinding aspect of the self-driving problem. Traditionally, GPS applications like Google Maps utilize shortest-path problem algorithms like A*, which can find perfect or near-perfect solutions with the given data in reasonable times [4]. The Multi-agent path finding (MAPF) problem, a variation where various agents could be matched with any target, is known to be NP-hard with only polynomic solutions that produce good estimates.

This paper aims to investigate a variation of the MAPF problem, namely one similar to the more complex delivery variation where the amount of destinations greatly exceeds the number of agents, but pickup locations are removed [5]. Traditionally, this type of problem is solved with either genetic or search-related algorithms, but we aim to tackle and further investigate this task using reinforcement learning [6]. Similar to the approach described in autonomous driving, many agents will receive image data, but in the form of a bird's eye view of environment buildup as well as each agent's local data. However, a major roadblock is that with different sizes and buildups of grids, retraining new setups with new agents will take unreasonable amounts of time (reinforcement learning is notorious for long training times to learn useful information).

The problems caused by the expensive and unstable nature of reinforcement learning must be overcome to apply reinforcement learning to real-world tasks. We aim to create a program that can create near-perfect solutions while minimizing the time steps needed.

Understanding the Processes Behind Reinforcement Learning

The basic idea of reinforcement machine learning can be split into three main parts: trial by error, optimal control, and temporal difference.

Trial by error is the process of an agent trying a variety of actions and then receiving a result immediately or later. It relies on the tradeoff between trying new actions (exploration) and choosing actions based on the agent's current knowledge (exploitation). Helpful actions that lead to good rewards will be remembered, while other actions that don't achieve a desired outcome may be less likely to be selected.

Optimal control focuses on finding the best control policy for an agent to achieve its objectives efficiently in a given environment. While traditional optimal control problems are well-studied in control theory and typically involve finding a control policy that optimizes a predefined performance criterion while adhering to system dynamics and constraints, in many real-world scenarios, the system dynamics are unknown beforehand, making it challenging to design a controller analytically. Optimistic control reinforcement learning aims to find an optimal policy that maximizes the expected return or minimizes the expected cost over a given time horizon. With various RL algorithms, such as Q-learning, it can be adapted and extended to solve optimal control problems. This is especially effective when system dynamics are unknown, hard to model, or nonlinear. RL algorithms can learn from interactions and discover effective control strategies combined with trial and error.

The temporal difference is how an agent learns from the observed experiences, updating its value function based on an error function between successive estimates. These are calculated as the discrepancies between the predictions made at different time steps during the agent's interaction with the environment. The selected temporal difference algorithm updates the value function to optimize control using these differences, and in this paper we use the deep Q-network approach, which we will refer to in this paper as DQN. This algorithm learns through the equation:

$$loss = (R + \gamma \star Q(S', A') - Q(S, A))^2 \qquad (1)$$

Here, the variable R refers to the reward an agent receives by taking action A at state S, while the variable A' corresponds to the highest value action at state S', which is obtained from taking action A at state S. γ is the discount rate by which rewards decay over time so that agents prefer reaching rewards earlier and punishments later. By minimizing the value of loss over many states and promoting the program to take actions that yield positive rewards, the neural network that estimates the value of state action pairs can eventually converge onto having higher values and thus learn to solve the environment.

Similar to DQN, the proximal policy optimization algorithm, which we refer to as PPO, optimizes an agent's policy to maximize the expected cumulative reward. Instead of estimating value functions, as done in Temporal Difference algorithms, policy gradient methods update the parameters of the policy to improve the agent's decision-making. Both DQN and PPO use a learning rate a to control the speed at which their neural network changes/learns. The biggest difference with DQN is that instead of only using one model and picking the action that the model predicts will receive the highest reward, policy gradient methods use two neural network models; one provides the probability of picking each action and another the value of a given state. Throughout the rest of this paper, many of PPO's hyperparameters will be tuned, especially since the loss function is now made up of three separate components instead of DQN's one, and all of them include a constant that needs to be manually tuned, as shown in the equation [7]:

$$L^{CLIP+VF+S}(\theta) = E\left[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s)\right] \quad (2)$$

Here, the total loss is equal to $L^{CLIP}(\theta)$, the actor-network loss with a manually changeable clipping parameter that prevents overly large updates, $c_1 L^{VF}(\theta)$, the value network loss that is calculated similar to DQN but having an extra tunable constant, and $c_2 S[\pi_\theta](s)$, which increases in value as the probability of a certain action decreases to the degree of a tunable constant.

Introduction to Multi-Agent Reinforcement Learning (MARL)

In the real world, many people can reach many jobs and goals much faster through cooperation. The first step in solving any cooperative task with reinforcement learning is deciding whether to use a decentralized or centralized learning approach. A decentralized approach involves multiple agents interacting with the environment independently. Although an agent doesn't directly interact with the others, it predicts how other agents may change the environment and takes action accordingly. On the other hand, a centralized approach involves an overseeing program, taking control of all agents and assigning them actions to maximize environmental rewards. While centralized learning theoretically yields better results as agent cooperation is required, decentralized learning is practically superior in training agents. This result can be attributed to the exponential increase in action space with the number of agents added, while the growth is only linear in decentralized learning.

To test the limits of these two approaches, substantial research has been done either designing methods to solve certain tasks (especially autonomous driving/drones) or understanding the complex math behind processes such as the Markov Decision Process.

Regardless of which methodology is used to train the program, reinforcement learning remains computation and data intensive. The problem is further amplified after adding more agents or players that can cooperate, increasing the action and state spaces. As a result, agents spend too much time or outright fail to learn the fundamentals of an environment, bottlenecking learning. A potential solution to improve training efficiency would be to add easily learnable human-engineered features to speed up early training times. By providing the program with basic features, one or more of an agent's neural network is essentially skipped, greatly speeding up the necessary training. However, this comes at the cost of hampering future learning, as the number of human-shaped features is often less than the number of neurons in a layer and can be easily subject to human bias. Similar to finding a balance between exploration and exploitation in TD algorithms, there must also be a

balance between the number of human-fed features and the amount of program-learned features to train a model efficiently.

Environment Setup

We created and compared multiple models with varying amounts of agents, learning through either DQN or PPO, using a centralized or decentralized learning approach and a magnitude of human-generated features in a navigational environment to find the most effective training variables. While many environments are difficult due to their sparse reward nature, my environment here has difficulty having too many potential rewards. The environment in question is a 12 by 12 grid where an agent, represented by a red pixel, tries to clean all the dirty pixels, represented by the color yellow. The goal of an agent is to achieve the highest possible average of tiles cleaned per run. Reinforcement learning is commonly known for having bad data efficiencies and long training times. Hence, using a small and simpler grid allowed me to test various hyperparameters, network constructions, and environment variables in reasonable amounts of time. A clean cell is denoted by the color white, and a wall the agent cannot pass through is denoted by black. As shown in Figure 1, an agent will be given a small reward for every cell it cleans that increases as the fraction of uncleaned cells left decreases. Since more than a third of the environment will start as an uncleaned tile, an agent can easily rack up rewards in the beginning by randomly moving around. As such, the model will learn from the beginning steps of an episode to move optimistically and greedily as if unclean tiles will always be within three steps. This learned strategy from the agent would perform poorly; however later into an episode where the sharp reduction in the number of cleanable tiles would require the agent to learn how to traverse long distances and around obstacles. The reward for each tile clean ranges from a minimum of 5 and a maximum of 20 in order to provide more reward for cleaning harder tiles and is represented by the equation:

$$Reward\ Given\ =\ 5 \times 4^{(\#\ of\ tiles\ cleaned)\ /\ (\#\ of\ total\ dirty\ tiles)} \qquad (3)$$

A huge bonus given to an agent that completed the environment was considered to promote completion of the environment, but in the end, it was removed to stabilize training and because it provided no learning benefits. To encourage thorough exploration, making a move that doesn't result in a cleaned tile gives a negative reward of -0.5. By providing a slight punishment, the agent will be more likely to explore all options in a state before pursuing a dedicated path. In the context of illegal moves, where an agent either runs into a wall or out of bounds, testing yielded that giving a small negative reward like -0.5 was more beneficial than making a very punishing reward of -100 and more efficient than removing illegal moves from the action space of a certain state. By giving a very negative reward, the agent's learning, especially using DQN, would be heavily flatlined later into an episode, causing it to play extremely cautiously and interestingly learn to become stuck in a fidgeting state, finding open space and then repeatedly moving back and forth. The unintended observed interaction can be attributed to randomness in reinforcement learning methods, as an agent is non-deterministic and has a notable probability of choosing an illegal action when the e-greedy value, the chance of picking an illegal action for exploration purposes, is high in DQN, or the ent_coef value, which motivates increasing probabilities for actions with low p-values, is too large in PPO. Having environment variables that were too specific created unintended results. Although both these problems could be fixed by restricting actions to only ones that are legal, we decided to allow the program to make illegal moves as another part of its learning. Furthermore, we add a gamma value of 0.995 to stimulate optimization in pathing.

Another factor that must be considered is how long each episode lasts. Giving a flat generous time step limit throughout harms the agent's early episode learning by not sufficiently punishing fidgeting while limiting late episode learning by wasting too much time initially and not reserving enough for later on. Increasing the amount of max time steps per episode only exemplifies this problem. A simple workaround to this problem

would be ending an episode when all tiles are clean or reaching a terminal state. This approach, however, introduces a new problem where episodes may not be able to terminate. Ideally, episodes, where an agent performs numerous poor actions, are reset quickly to prevent detrimental updates to the policy.

In contrast, more successful episodes are given more time steps so that the agent can learn how to perform better in the future. Following this idea and some motivation from video games, the construction for time steps follows: a total of three halves of the area of the grid is given to the agent as "lives," which will be consumed one at a time if the agent doesn't move onto an uncleaned tile. Each additional agent increases the total amount of lives by half of the area of the grid. An episode terminates when the agent cleans the entire grid or runs out of life.

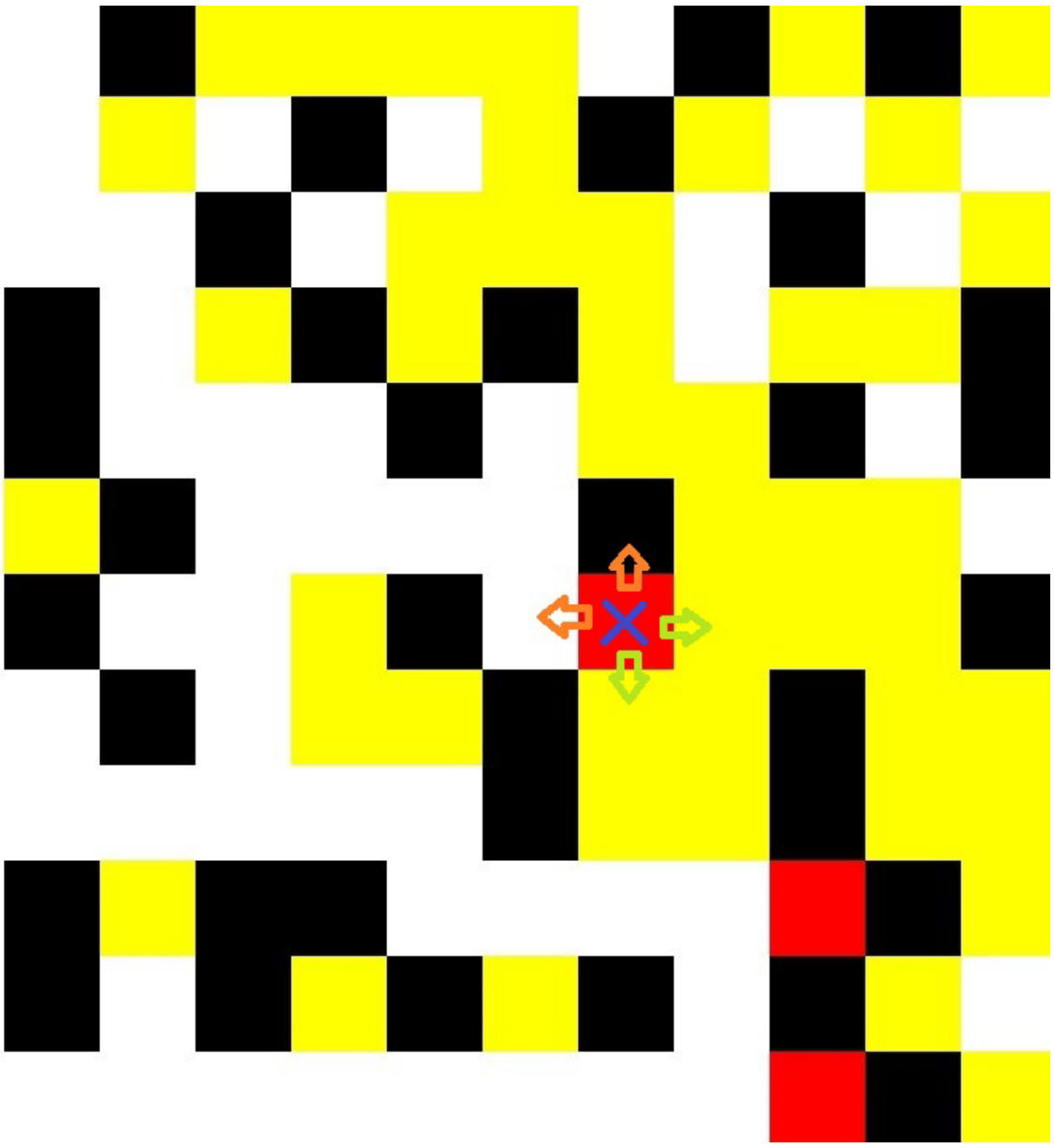

Figure 1. An example of the type of reward an agent will receive depending on its moves in relation to the environment. The agent in question has a blue X on it, and orange arrows result in negative rewards, while

green arrows result in positive rewards. Yellow tiles indicate an unclean tile or positive reward, red tiles an agent, white tiles already clean spaces, and black squares an impassable wall.

Solving and Understanding the Custom Environment

We created a base single agent to test the basic characteristics of the environment and the model. The agent consists of a convoluted neural network to process images and then a deep neural network to process characteristics using PPO or DQN to improve itself. Since the action space is discrete and the state is simple, this initially pushes for a DQN implementation. PPO will also be used as it has its advantages since a bad move chosen randomly by DQN may require the agent to waste another action to return to its original state; the PPO algorithm can become more deterministic, so using it has its merits. We initially proceeded with a PPO approach since PPO has a more general application than its DQN counterpart.

Through testing, a baseline learning rate of 0.00015 was found to be the striking balance between speedy learning and convergence ability. To gauge the feasibility of using reinforcement learning, a single agent was trained over 1.2 million time steps (around 10,000 episodes) using the StableBaseline3 PPO implementation with a batch size of 64 and n_steps, the number of time steps before an update, of 2048.

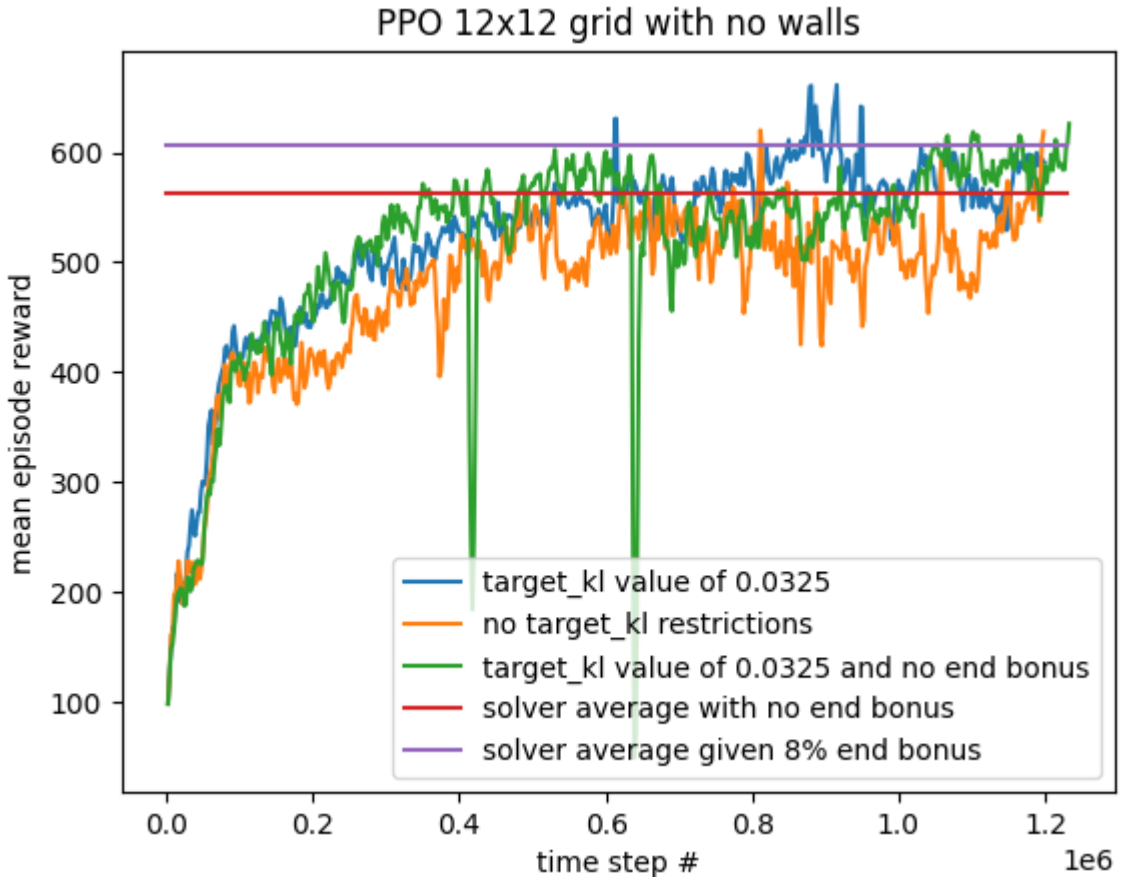

Figure 2. A graph compares PPO agents' training results with and without certain variables. The training process over 1.2 million time steps of agents with differing setups like target_kl value and having an environment end bonus is put into context with environment solvers that use heuristic approaches.

The graph in Figure 2 was not continued past 1.2 million time steps as the agents all shakily converged around these values. The agents tested in the graph processed the image input with two-layer pairs of 3x3 kernels with one padding and two layers of max pooling, as well as agent location for runtime speedup reasons shown in Figure 3.

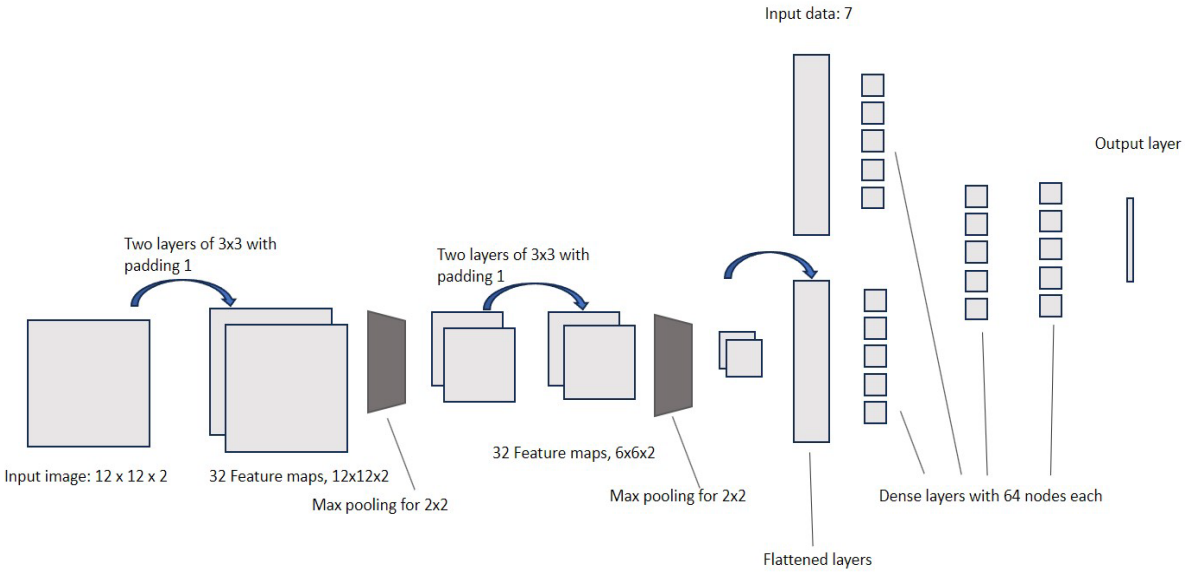

Figure 3. The neural network architecture that was used to process states for PPO. Input data is split into two types, an image input and some stats regarding the agents, and is then sent through the network construction.

Other layers, like dropout and batch normalization, were tested with various amounts and locations. Still, these only boosted learning early on and significantly hindered mid to late agent learning potential. Restricting gradient updates further using a gradient clip of 0.2 and a target_kl clip of 0.0325, which terminates updates when the current kl value is over 0.0325. The formula for kl_divergence is as follows:

$$D_{KL} = \sum_{i} \quad P(i) * ln(\frac{P(i)}{Q(i)})$$

$P(i)$ the probability of action i with old policy P and $Q(i)$ the probability of action i with new policy Q. When the kl_divergence value is large while still in the gradient clip range, the high probability values are being pushed even higher, making the agent prone to overfitting. By ending updates on a dataset when this kl_value exceeds a threshold called target_kl, we can prevent more potentially detrimental updates from happening. Through testing, it was found that 0.0325 was the best value for the clip, which will change depending on the environment and agent setups. A target_kl value shows notable improvements in learning speeds and reliability, especially when improving the agent from a mean episode reward of 400 to consistently above 500. On top of stabilizing performance, having a target_kl value also decreased training times by around 25%, as from mid to late learning and with large updates, the agent constantly crosses the target_kl threshold, ending learning early a lot of the time. My theory was that the end reward for completing the state, 20 * (# of cleaned tiles), may be causing the mode to sacrifice average tiles cleaned with a slightly higher environment completion rate. To test this, we created an environment solver using the breadth-first search algorithm to find which moves will lead the agent to the nearest uncleaned tile and randomly pick between equally good moves. In every episode it was run on, the solver completed the environment. Despite this, it is still possible to outperform, as the paths it takes may not be optimal to reach each unclean tile in the quickest time. Since the window size for

averages we used was 25, in testing, it was found that, on average, the agent with target_kl value and was given end bonuses had around two completions inside the queue and, hence, used 8% completion as a generous benchmark. By comparing when the blue line outperformed the purple line and when the green line outperformed the red line, it is clear that the end reward is a detriment to convergence and hurts overall agent performance. At face value, the environment is straightforward and, if able to be solved with PPO, should be solved through DQN in less time.

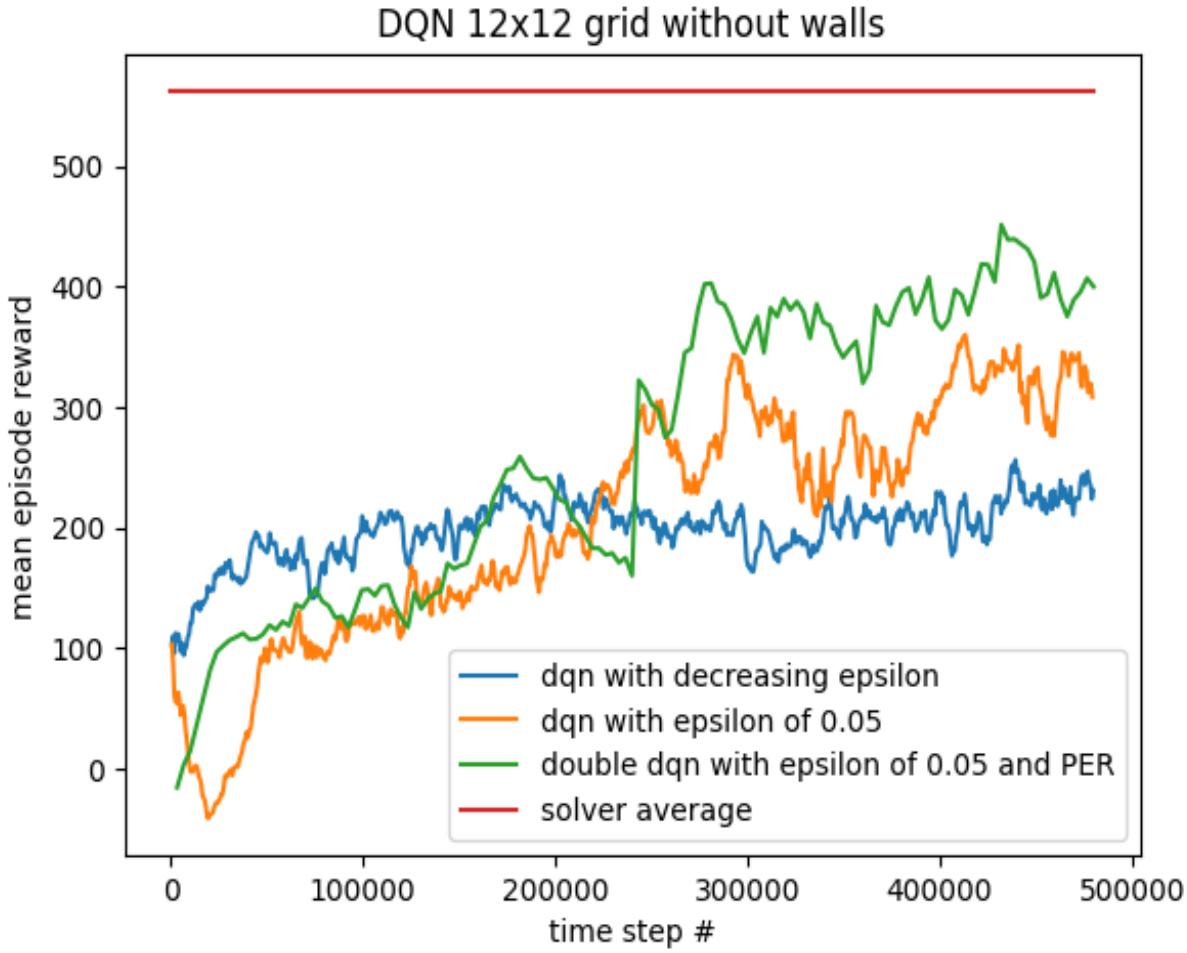

Figure 4. A graph comparing different DQN architectures with the solver average. The learning curves of each type of learning construction over 500,000 timesteps.

With the chart above in Figure 4, however, there are indications that the environment is more complex than it initially seems. All DQN methods, including normal, decreasing epsilon to stimulate exploration, and a self-implemented double DQN (using two value networks to stabilize learning) with a prioritized experienced relay (prioritize sampling states from the buffer that decreases loss more), were insufficient in maximizing rewards in the environment compared to even basic PPO methods despite hyperparameter tuning. All this indicates that increasing the number of reward objectives significantly increases an environment's complexity.

Despite the agent's ability to converge and eventually produce a policy that solves almost the entirety of the environment, it took longer than expected for the agent to solve the environment. Although with an average episode length of 200, the agent seemingly takes a very long time to solve this simple environment as

the first time the blue line solved the environment was around 3000 episodes and the green line around 1750 episodes, each update was performed every 2048 time steps, the number of updates needed were at much more reasonable amounts of 293 and 183 respectively. However, the amount of training required was still unusually high, so we tested lower numbers of uncleaned tiles to see if the results were also attributed to poor reward values or agent setups.

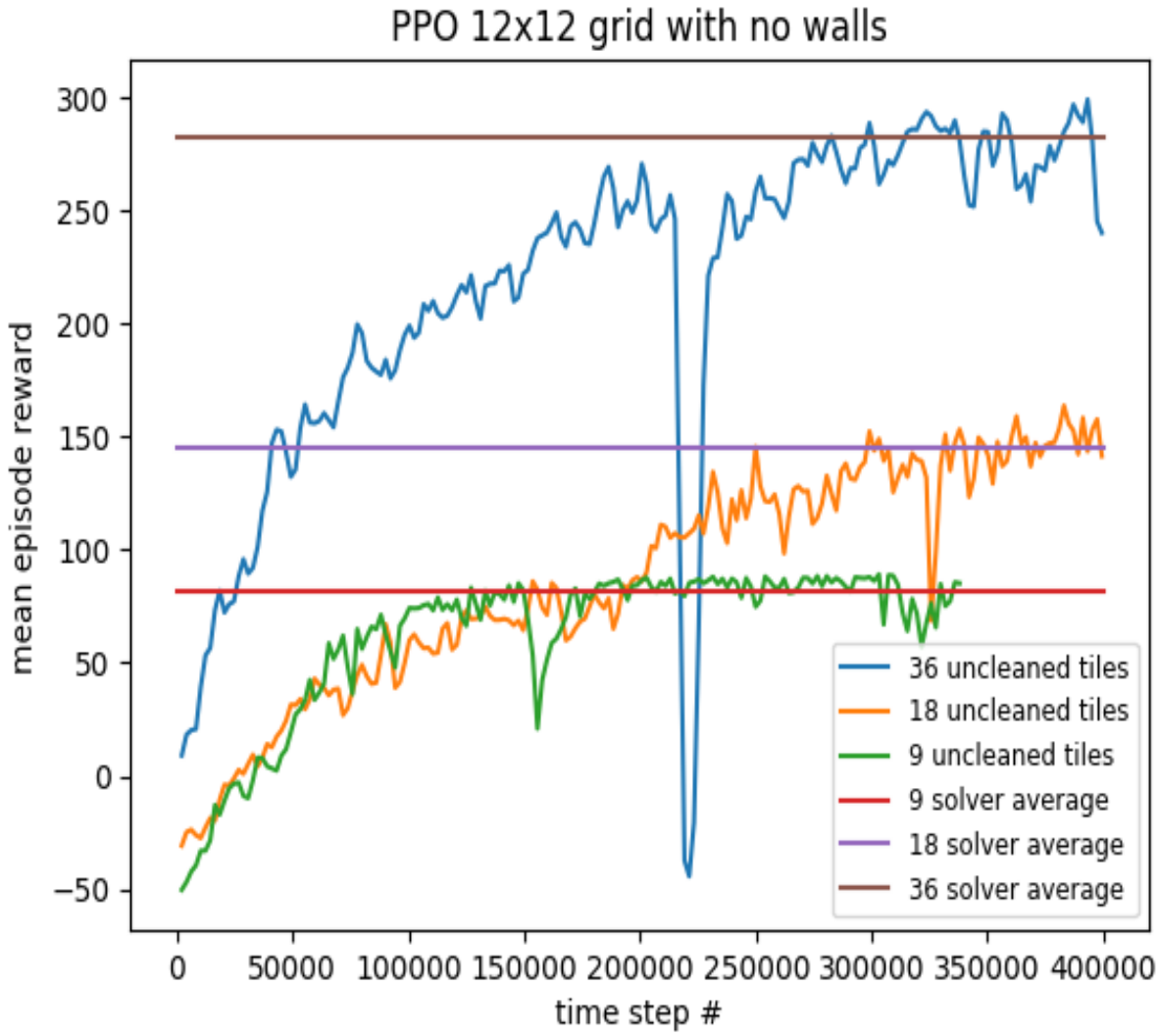

Figure 5. A graph showcasing the rewards trajectory of a PPO solver given a certain amount of uncleaned starting tiles with solver benchmarks. When the line produced by a solver and its corresponding line begin overlapping, the agent is nearing optimization.

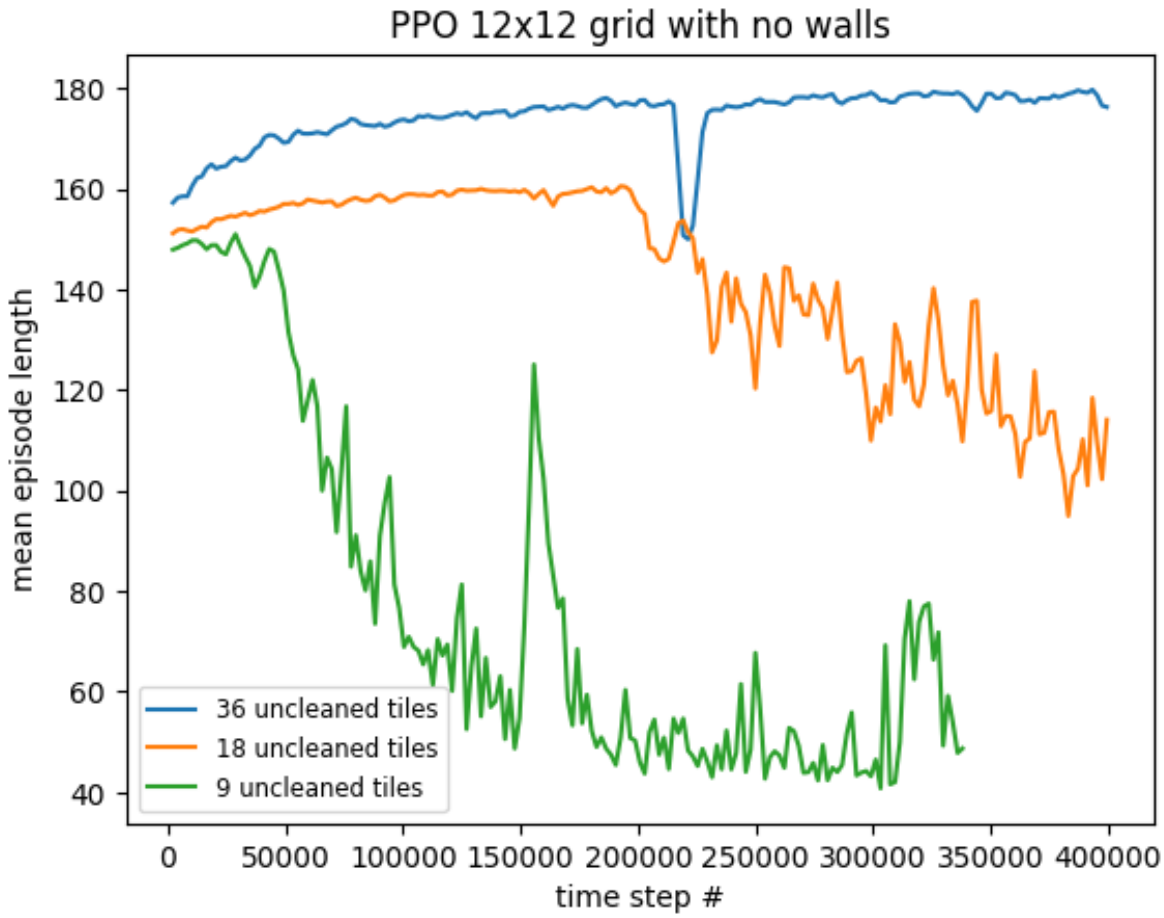

Figure 6. A graph showcasing the episode length trajectory of a PPO solver given a certain amount of uncleaned starting tiles. When the line produced by a solver begins decreasing, it means that the agent can consistently complete the environment and is starting to optimize its route.

As shown in Figure 5, the lower the amount of total unclean tiles in the grid, the faster the agent can converge. Having 36 uncleaned tiles allowed the agent to achieve its solver average much quicker at around 275,000-time steps, while having 18 and 9 uncleaned tiles boosted first-solve performance to 250,000 and 125,000 time steps, respectively. The agents all follow a similar pattern for solving the environment due to its architecture. First, during the learning/exploration phase, each agent is rewarded with more "lives" and can make more moves for each unclean tile it successfully reaches. This behavior is demonstrated in the average episode length graph in Figure 6, which initially increases as the average episode reward increases until completion. Once the agent can consistently complete the environment, the agent enters a state of optimization, trying to complete the environment as quickly as possible. This state is clearly shown in the orange and green lines in the figure above when the agent decreases its mean episode length while increasing its reward. A problem occurs, however, when the agent cannot learn to consistently complete the environment, meaning that episode lengths will stay the same. Convergence, however, can still be observed through stagnant average reward increases. With fewer unclean starting tiles in the grid, it is apparent that the training time needed for mastery is significantly reduced. A notable feature is how the agent can outperform solver capabilities by lowering starting uncleaned tiles.

Since the solver has a highly greedy nature, it finds perfect solutions to more straightforward states while only finding "good solutions" to complex ones with more starting uncleaned tiles.

However, since these complex environment setups are also significantly harder to train, agents will struggle to match the solver performance on complex models but have a much higher potential to outmatch solvers than simple environment setups, further exemplifying the mastery the agents had on the simple environment setups.

Now that the agents could solve the simple environment with no walls, we moved on to a more complex environment that included walls. Using a PPO implementation through stablebaselines3, a single agent began first completing a 16x16 version of the environment in 3 million time steps using the previously mentioned hyperparameters and an ent_coef value of 0.0001 obtained through rigorous trials.

Throughout the project's progression, the grid's dimensions were changed multiple times from 40 to 20 to 16 to 12 to shorten runtimes in achieving convergence.

Finding the Best Multi-Agent Reinforcement Learning (MARL) Method

Finally, we tested multiple multi-agent reinforcement learning approaches to see which methods would most effectively solve an environment with many reward goals. The two most basic approaches are fully centralized and fully decentralized models. An important factor in evaluating a model's overall performance is the correlation between training performance and speed.

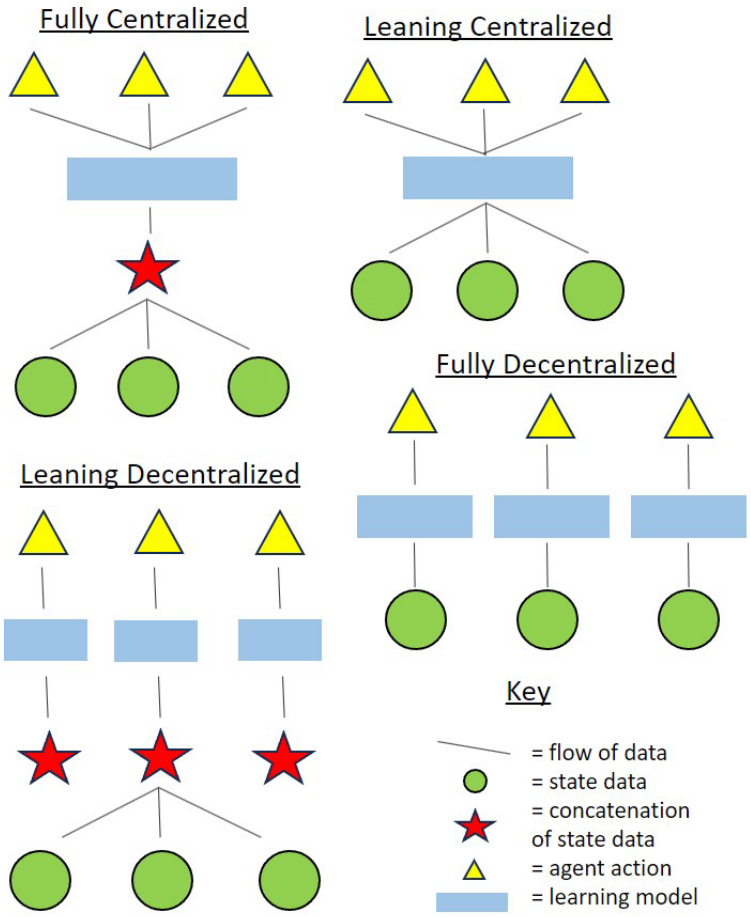

Figure 7. Illustrations showing the general process of each of the four training procedures. The sequence of events occurs from bottom to top and left to right, with input being fed to the bottom and resulting actions coming out of the top.

In Figure 7, the basic structure of each multi-agent reinforcement learning method is illustrated. The fully centralized model involves a central control room that takes in the state of all the agents and gives each action accordingly. At the same time, a fully decentralized approach offers no form of communication between agents and has them learn independently of one another. While the centralized approach is theoretically better, its action space scales exponentially as the number of agents increases. The decentralized approach can also learn cooperation without explicitly doing so by considering how the environment is, on average, changing due to other agents and taking actions accordingly without a huge action space increase. Between these two approaches, there are also other methods like parameter sharing, striking a balance between runtime increase and cooperative ability. Two of these in-between approaches we tested were a centralized learning approach, where each agent would independently feed their state information to a central model that would give them a move, and a decentralized learning approach, where each agent was trained independently of one another but would still receive the state information from other agents to factor into its calculations.

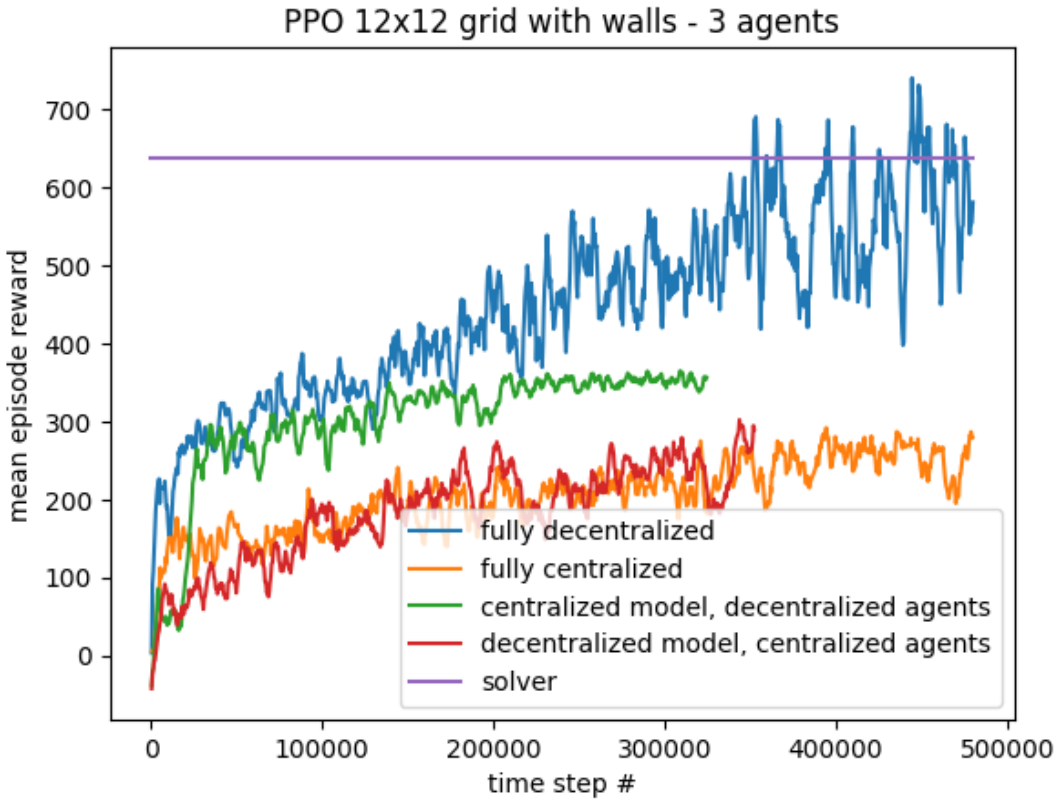

Figure 8. A graph comparing the reward progress of different multi-agent reinforcement learning constructions with each other. The performance of each agent architecture type was tested over a maximum of 500,000 timesteps.

Due to Stablebaseline3 not supporting multi-agent environments and wanting more customization, we self-implemented PPO and the variants of its usage in multi-agent learning with the help of online implementations [8]. From the tests utilizing these methods in a walled 12x12 environment consisting of 36 walls with 72 uncleaned tiles, the completely decentralized approach proved to be most efficient in solving this highly reward-abundant problem. As depicted in Figure 8, The rate of learning for the fully centralized model is far too slow to be an effective method once state space and agents increase, and the leaning centralized approach plateaus when the reward is around 325. Some agents were stopped early since it became clear midway that they wouldn't be near the performance of the fully decentralized approach. Although the learning decentralized model shows great promise in its progression, it is still slow compared to a fully decentralized method, taking too long for each agent to decide which set of parameters would be most useful for it and which set could be used for support, but could work better in an environment where the bottleneck isn't so much the bottleneck.

Since all these methods, even the fully decentralized model that seemed to perform quite well, are still far too slow to be efficient when the environment complexity and agent numbers are scaled up, we decided to add "cheating" parameters that originated from the solver to the agents. On top of normal state information, four additional information variables as parameters produced by the algorithm the solver used would receive would be given to each agent. In theory, this would drastically improve the training times of all agents and allow them to achieve high average rewards faster.

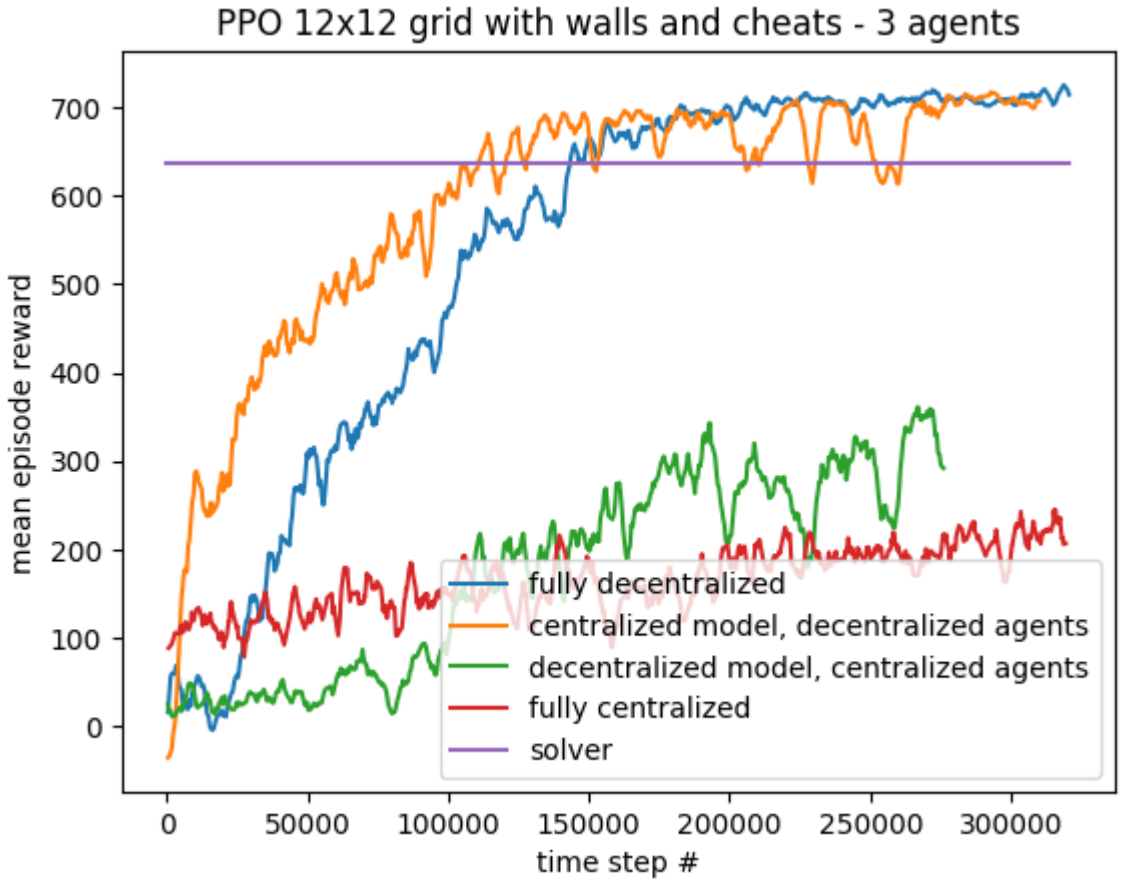

Figure 9. A graph comparing the reward progress of different multi-agent reinforcement learning constructions with each other given additional solver parameters. Over a maximum period of 300,000 timesteps, the agent's average rewards were documented and compared with each other as well as the solver.

With this additional parameter, the learning speed for each agent setup is increased compared to Figure 8 to Figure 9 despite having three-fiftens the total time steps. The approaches of a fully decentralized method and a leaning centralized method had their performances skyrocket, with the leaning centralized approach even outdoing the fully decentralized method in learning speed and theoretically scaling even better but suffering from stability issues, although it eventually converges.

The fully centralized method had greatly improved learning performance, but its action space was still heavily hindering the speed at which it could learn and would take an unreasonably long time to train. Interestingly, however, the learning decentralized method's learning was hindered initially by the new parameter, possibly because it needed to digest even more parameters and confused itself more than when it was run with fewer parameters. In 320,000 time steps, both the fully decentralized and leaning centralized methods were able to solve the environment consistently, averaging 99.5% full competition in the last 400 episodes before training termination.

Final Results

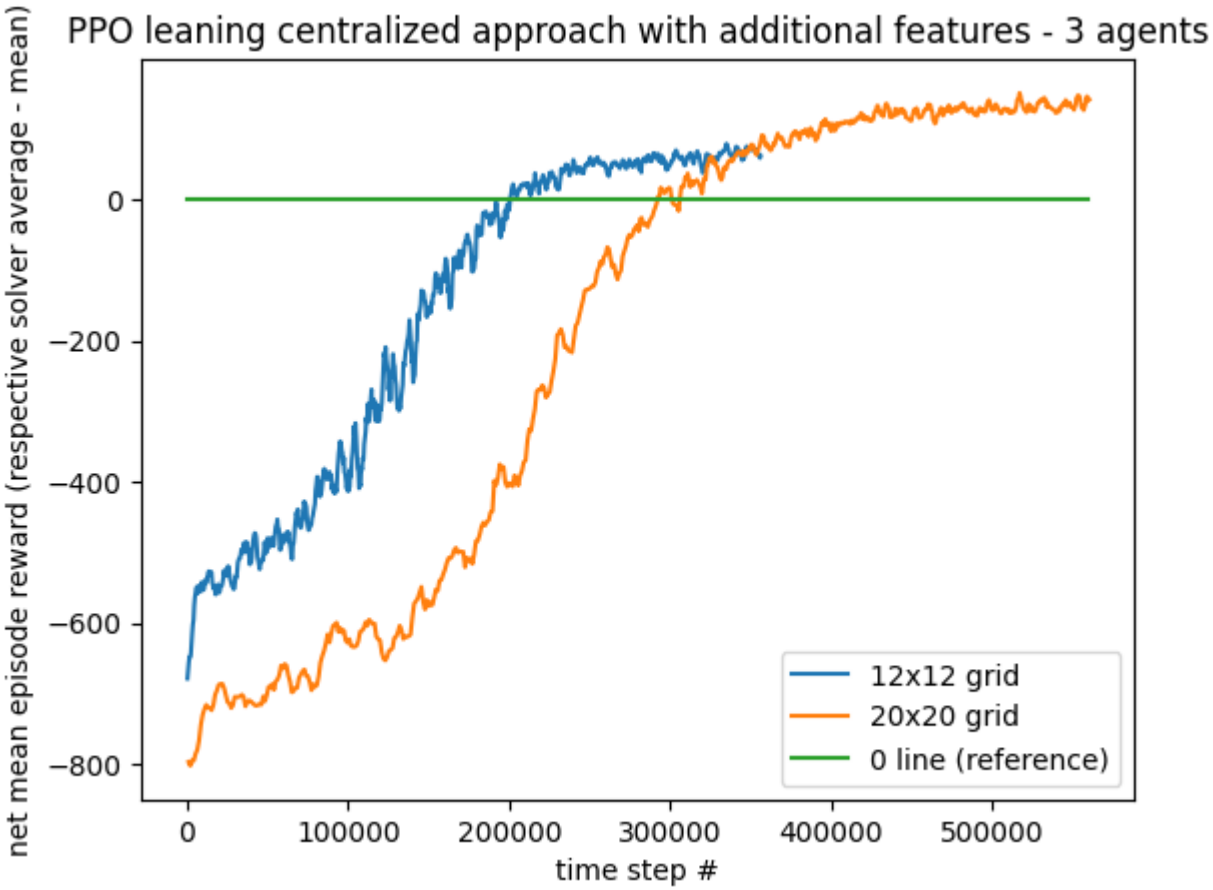

Figure 10. A graph displaying how the rewards of our approach minus each solver's average compare from a 12x12 to a 20x20. Over around 350000 time steps and 650000 time steps for a 12x12 and a 20x20, we visualize (agent average over last 25 episodes) - (solver average over 10000 runs). The average rewards for the solver were 637.2 and 496.0.

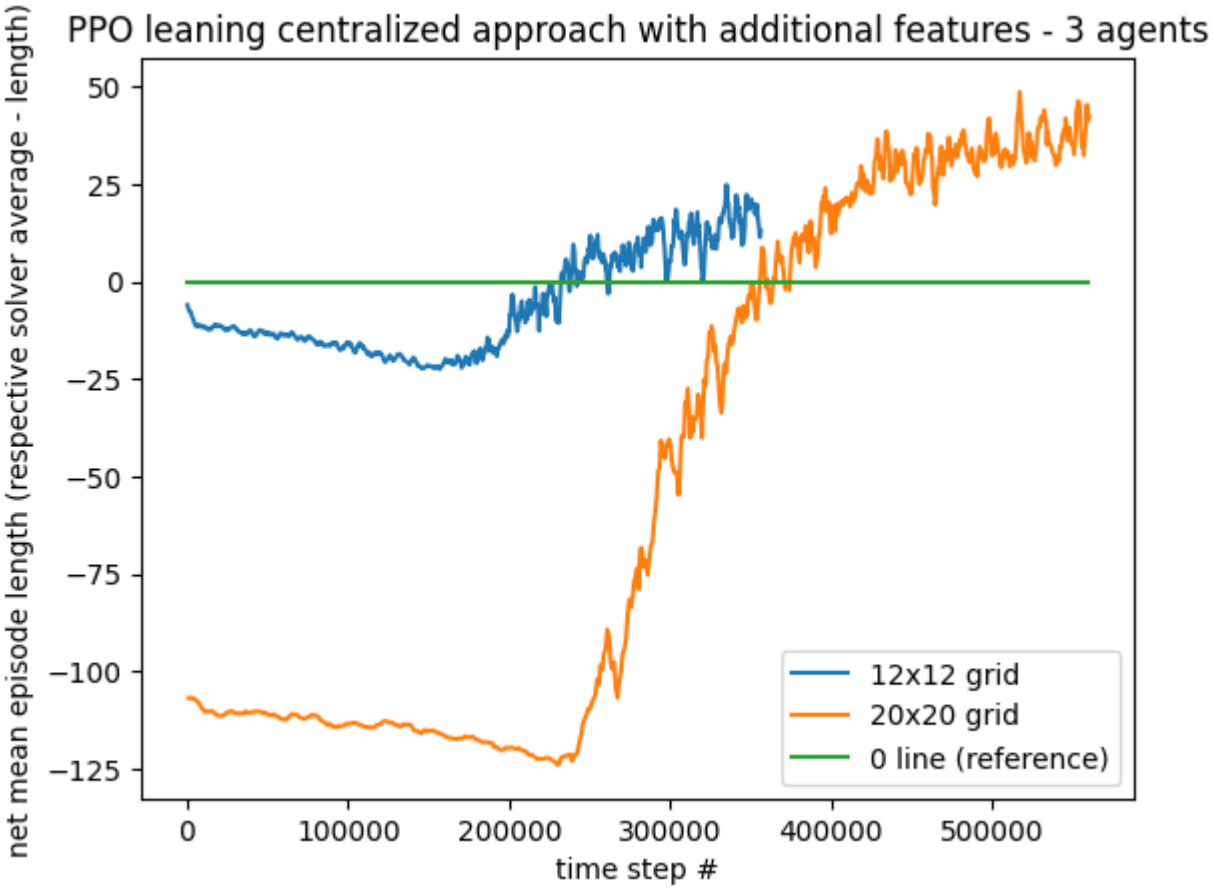

Figure 11. The episode advantages of our approach on a 12x12 and a 20x20 grid are visualized against each other. Over the same time as the previous graph, the values (agent average length over 25 episodes) - (solver length average over 10000) are plotted. The average episode lengths of the solver were 96 and 166.

By injecting in helpful features from our constructed imperfect algorithmic approach, the time steps it took for the agents were observed to be vastly decreased, so we attempt to apply the approach to a variation where all wall and uncleaned tiles are now randomized for greater real-world applications. Since target destinations are unlikely to be packed in an area to the point where they populate half the grid tiles, we also test our leaning centralized approach on a 20x20 grid that contains the same amount of unclean tiles but has around a three times greater grid size and wall amount. While the shortest episode lengths for the solvers over 10000 simulated episodes were 56 and 71, the shortest episode lengths for the reinforcement learning approach were 48 and 57, yielding advantages of 8 and 14 in their best episodes, respectively. In Figure 10 and Figure 11, each approach was terminated when the improvement percentage over the last 10,000 time steps was below a certain threshold. An interesting observation is that the point at which the net average reward breaks even for both the 12x12 and the 20x20 at around 180000 and 280000 time steps is earlier than when the agents can outperform the solvers at around 225000 and 350000 time steps. This is likely due to the nature of the environment due to a decay rate of 0.995, which pushes the agents to be more efficient early on and significantly outperform the solvers even if they collect 1 or more tiles. However, since the agents haven't mastered the path properly later on and the episode doesn't end until either a limit is reached or all tiles are cleaned, the average episode lengths are inflated while not being reflected in episodic rewards. If the agents could terminate an episode early, it would be more efficient than the solver over the course of an entire episode. Even though adjusting from a 12x12 to a 20x20

grid increases the area by around 2.78 times or 1.67 times the side length, all the metrics suggest a linear relationship between the time steps needed and the side length. The ratios between the 20x20 grid and 12x12 grid are 1.56, 1.56, 1.6, and 1.75 for the reward break-even point, the episode length break-even point, the time step when the learning is terminated due to potential convergence, and the advantage in minimum episode length. This suggests that the agent has good linear scalability based on side length, and can be used to make predictions that 100x100 or even 1000x1000 grids are feasible with this paper's proposal. While the 12x12 grid finalizes with around a net reward average of 61 and a net length average of 19, the 20x20 grid finishes with more than double these benefits, netting a reward average of 141 and a net length average of 41. The pattern reveals the huge performance increases the model adds to the solver algorithm, especially exemplified in larger and more complex states.

Approach Limitations

While both final tests solved the dynamic environment in seemingly reasonable times, the vast increase in time steps needed for acceptable performance from a 12x12 grid to a 20x20 grid shows that its scalability is concerning. Without increasing the number of unclean tiles needed in this transition, which was proven earlier in the paper to be positively correlated to the time steps needed, the ratio is already linear, meaning that combined factors could lead to a quadratic relationship. Although the change from a fix to a randomized grid allows the program to be adapted to any grid configuration with smaller dimensions, the trend of requiring massively more time steps to train will only be exemplified when moving onto larger and more generalized grids like 100x100 or 1000x1000 with more unclean tiles. To tackle being feasible in larger grid states, parallelization, n-step, and various other optimizations need to be applied to speed up episode run speeds. Furthermore, more widely used multi-agent algorithms like MADDPG could also be tested against the approach proposed, as well as more engineered features being provided to the models. Most importantly, applications of this approach in the real world won't be as simple as being free tiles or walls, and many external variables would need to be accounted for. It is important to note that while not including these extra features could allow an agent to attain a better final result, commonly used algorithms can already attain near-perfect results, and it would be more worthwhile to expand further on what already exists rather than start from scratch. Many algorithms that strive for greedy solutions and are applicable could be greatly improved through increased complexity without much runtime sacrifices by having an AI model take in potential action candidates and output what it thinks is the best one.

Key Takeaways

Overall, with a large scale and an overabundance of non-replaceable rewards and many players, multiple-agent reinforcement learning through complete decentralization is best for environments without a clear heuristic approach. If such an approach does exist, using a central model where each agent feeds their information would be the most efficient and best scaling, although the learning is a bit unstable. In the case where agents are relatively few, and the environment requires complex cooperation to maximize, having a completely decentralized approach along with each agent also getting the state of others would possibly yield the best results over a long period of time.

Although this environment is fairly simple, as a simulation of a complex navigational problem, it still has many real-world far-reaching implications. In particular, this project focuses on the pathfinding aspect for multiple agents in navigating to numerous rewards. The many cleaning tiles in the simulation can be replaced with multiple target destinations, while the many agents represent multiple cooperating vehicles. Many real-world problems like using multiple vans to efficiently pick up expired food at many locations in an area to

provide for a homeless kitchen or to optimize the delivery of packages to scattered people, could be approached with this application. Using the proposed solutions to determine pathing together with preexisting self-driving technologies could allow for the efficient pickup and speedy delivery of items or people in a dense area. It would also help with efficiency in warehouse drop-offs by robots. With this, the cost of operation and maintenance decreases drastically, potentially allowing funds to be used for more area coverage and increasing the total number of people that can be positively affected.

The learning rate was the most important hyperparameter that had to be tuned throughout the learning process. Coming into the project, we thought the initial learning rate for a model, especially since Adam optimizer was being used, was irrelevant to overall model performance. However, after experimenting with different values, we realized that having a large learning rate can make or break long-term agent learning performance while having one too small would potentially more than double the amount of training time needed on top of the hours it took for me to run my more optimized models.

Self-implementation was also incredibly important to the overall understanding of an algorithm, as we learned the importance of various hyperparameters we once thought were redundant, like kl value, only looking at stablebaseline3 PPO documentation. While earlier tests in the paper, like the basic environment solving with Stablebaseline3 methods, were run on Google Colab to utilize its GPU over three hours, the multi-agent tests, which were self-implemented, were not adapted to parallel and GPU usage and were run on my local machine without GPU. This, however, proved to be very slow, and every training period took around eight hours each while also running millions of fewer time steps.

A problem's first glance doesn't accurately affect its difficulty. While we initially thought my custom environment was straightforward, it turned out to be much harder to train compared to more seemingly complex environments like Gymnasium's Lunar Lander and Car Racing environments. Furthermore, an environment's complexity significantly increases when objectives become more abstract or numerous. Although training a single agent is faster and more straightforward, the inclusion of multiple agents working together produces much better potential results. More research in this direction is worth the hassle and can revolutionize reinforcement learning by broadening its reach and improving preexisting implications.

Acknowledgments

I would like to thank my advisor for the valuable insight provided to me on this topic.

References

1. Mahoney, Chris. "Reinforcement Learning." *Medium*, 17 June 2021, towardsdatascience.com/reinforcement-learning-fda8ff535bb6#4b10.
2. Silver, David, et al. "A general reinforcement learning algorithm that Masters Chess, Shogi, and go through self-play." *Science*, vol. 362, no. 6419, 2018, pp. 1140–1144, https://doi.org/10.1126/science.aar6404.
3. Ben Dickson, et al. "Ai Defeated Human Champions at Dota 2. Here's What We Learned." *TechTalks*, 23 Nov. 2019, bdtechtalks.com/2019/04/17/openai-five-neural-networks-dota-2/.
4. Foead, Daniel, et al. "A systematic literature review of a* pathfinding." *Procedia Computer Science*, vol. 179, 2021, pp. 507–514, https://doi.org/10.1016/j.procs.2021.01.034.
5. Stern, Roni, et al. "Multi-agent pathfinding: Definitions, variants, and benchmarks." *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, no. 1, 2021, pp. 151–158, https://doi.org/10.1609/socs.v10i1.18510.

6. Queiroz, Ana Carolina, et al. “Solving multi-agent pickup and delivery problems using a genetic algorithm.” *Intelligent Systems*, 2020, pp. 140–153, https://doi.org/10.1007/978-3-030-61380-8_10.
7. Simonini, Thomas. “Proximal Policy Optimization (PPO).” *Hugging Face – The AI Community Building the Future.*, huggingface.co/blog/deep-rl-ppo. Accessed 12 Aug. 2023.
8. Suran, Abhishek. “Proximal Policy Optimization (PPO) with Tensorflow 2.X.” *Medium*, 21 Sept. 2020, towardsdatascience.com/proximal-policy-optimization-ppo-with-tensorflow-2-x-89c9430ecc26.