A Proximal Policy Optimization Algorithm for Solving Logistical Optimization Problems

Master Thesis Business Analytics and Operations Research

by

Author: Joost Verbakel (SNR: 2012657, ANR: 320062)

Supervisor Tilburg University: Maxence Delorme
Supervisor ICT Group: Jonathan Bogerd

April, 2024

Tilburg University

Contents

Chapter 1

Abstract

This thesis investigates the application of the Proximal Policy Optimization (PPO) algorithm for logistical optimization problems. The higher goal is that the PPO algorithm could be used to train a model that can be applied to logistical optimization problems, such as problems that arise in a container terminal or a warehouse. Before the model can be applied to the problems for the container terminal and warehouse, the model should first be able to solve more simple problems such as a maze, Traveling Salesman Problem (TSP), Vehicle Routing Problem (VRP) or Container Stacking Problem (CSP). The algorithm will be developed by solving these simpler problems. This research aims to improve the PPO algorithm so that the model can solve more complex problems than initially possible. This research develops the model by solving Maze and TSP instances. This thesis aims to explore various improvement techniques that can augment the quality of the PPO model. One significant finding of this research is identifying the necessity for entropy regularization to overcome local optima. Furthermore, this thesis examines deploying a second model to address unsolvable mazes, leading to a higher solvability. Additionally, this thesis introduces a hyper-heuristic approach, trained with the PPO algorithm, that effectively learns to combine the initial heuristics to reach near-optimal solutions and outperforms every single heuristic.

Chapter 2

Introduction

This research is done on behalf of ICT Group. ICT Group is a company that delivers industrial technology solutions for a wide range of customers. Some sectors where ICT Group is active are automotive, healthcare, and logistics. Currently, ICT Group is researching a new optimization tool that should be applicable to different logistical problems.

The motivation for this research comes from the logistics division within ICT Group, which is active in sectors such as container terminals and warehousing. Example of warehousing problems are batchting or the routes for order picking. In a container terminal, all kinds of logistical challenges arise. Examples are choosing which cranes to use for loading and unloading a ship, equipment assignment, and route planning for transporting containers over the yard. Another example is choosing the stacking location for containers at the terminal, also known as the container stacking problem (CSP). ICT Group is active in this sector and provides software solutions for container terminals. Their software determines, among other things, where each container should be stacked. However, ICT Group believes that the quality of the current solution can be improved by using Reinforcement Learning (RL). For that reason, ICT Group is researching the development of a Reinforcement Learning based optimization tool that could be used for solving optimization problems corresponding to container terminal and warehousing problems.

The CSP has been a widely studied problem for the past decades. Exact solutions for the CSP using solvers are only possible for smaller instances. For real-life instances, these solvers will not work due to the problem's size and the uncertainty in the model. In practice, information about containers and the terminal is uncertain until the ship's unloading is started. Starting the solver and waiting until a solution is found when a ship arrives is impossible. One way to tackle the CSP is by using metaheuristics. Euchi, Moussi, Ndiaye and Yassine (2016) used an Ant Colony optimization model for the port of Le Havre.

Metaheuristics are able to find a local optimum. Domain knowledge could steer the model in a good direction when constructing the solution. One disadvantage of using metaheuristics is that it takes time to construct a solution for a single instance. This disadvantage makes a metaheuristic not suitable for the real-life CSP. When using a metaheuristic for the CSP, the solution should be constructed before a ship

arrives. This is undesirable because some of the relevant information necessary to construct the solution is unknown before the ship arrives. For example, the order in which the containers are taken off the ship is unknown in advance and the order of unloading itself could be seen as an optimization problem. It is also unknown in advance which cranes are available to unload the ships. Besides this lack of information on the forehand, there is also a chance for unforeseen events such as delays or containers not being in the exact location on the ship as they are on paper. For these reasons, it is desirable to have a model that is able to quickly generate a new solution based on new available information about the environment.

Besides metaheuristics, for example, rule-based decision algorithms could be used (Gunawardhana, Perera and Thibbotuwawa, 2021). Rule-based decision algorithms are currently seen in practice in container terminals for the CSP. The advantage of rule-based decision algorithms is that expert knowledge is used to make the algorithm. It is a common practice to measure the goodness of these decision algorithms based on simulations. These decision algorithms can only take into account a limited number of variables, whereas a Reinforcement Learning model could theoretically take into account more variables. For this reason, ICT Group has the idea that rule-based decision algorithms could be improved by using Reinforcement Learning.

Taking these considerations into account, a RL algorithm will be used in this research. The major advantage of using an RL model is that the model can be trained beforehand. After training, the model is able to generate a solution from a given input state quickly. In the case of the CSP, a solution is generated the moment a container is picked up. This solution is based on the most recent information about the environment.

One of the algorithms being researched is Proximal Policy Optimization (PPO). ICT Group chose to research PPO because PPO is proven to be useful for complex problems and deals well with large state and action sizes, which will be further discussed in the literature review.

The real-life CSP is a complex problem that involves many specific business rules. Furthermore, PPO has different aspects that can be tuned, impacting the model's performance. The combination of a complex problem with a complex model may make it difficult to optimize this at once. It could be hard to detect where the model could be improved when it does not work. In this research, the model will be trained and tested in several environments. These environments are a Maze and the Traveling Salesman Problem (TSP). One purpose of developing the model on these test environments is to monitor the performance of the PPO model. This will help to get insights into how different aspects of the PPO model work. Another reason for using these test environments is that if ICT Group has a future project that could be solved with RL, these test environments could be used as starting points for building new algorithms. Elements of the Maze and the TSP can be found in many logistical decision problems. Therefore, building a good algorithm for solving them will help solve more complicated real-world problems.

ICT Group already implemented a basic framework of a PPO algorithm. Differ-

ent test environments were built in order to work towards the container terminal. However, the algorithm fails to learn in these test environments. This research aims to extend and improve the existing PPO algorithm to make it applicable to larger and more complex problem instances. This is done by developing, training, and testing with the help of the Maze and the TSP environments.

The remainder of this thesis is structured as follows: Chapter 2 provides a literature review. In Chapter 3, the PPO algorithm will be explained in detail. This is done by addressing relevant background information about RL and explaining the specifications of PPO. Chapter 4 provides details of the implementation of the model used in this research. This explains the motivation for the choice of hyperparameter values, network architecture, and other specifications not discussed in the PPO chapter. Chapters 5 and 6 present the research about the Maze and the TSP environment, respectively. These chapters cover the changes to the environments and the model used to improve the model's performance. In chapter 7, the results of the improvements are summarized. Chapter 8 discusses the modeling choices made in this research, the model's limitations are also given in this chapter. Chapter 9 covers recommendations for future research.

Chapter 3

Literature review

Reinforcement Learning is successful in various fields. Some fields where RL made breakthroughs are the development of self-driving cars (Aiswarya, Mariah, Katragadda and Makam, 2023), playing Chess (Schrittwieser et al., 2020), and cancer treatment (Hou, Lee and Keidar, 2022). A more recent example of RL being used in practice is ChatGPT. RL also made progress on combinatorial optimization problems such as the TSP (Bresson and Laurent, 2021) and the Vehicle Routing Problem (VRP) (Nazari, Oroojlooy, Takáč and Snyder, 2018). Through the years, different types of RL algorithms have been developed. Common methods are Proximal Policy Optimization (PPO), Deep Q-learning, or Deep Deterministic Policy Gradient methods. In this research, PPO will be used.

PPO was developed by researchers of OpenAI (Schulman, Wolski, Dhariwal, Radford and Klimov, 2017). The researchers have adjusted existing policy gradient methods such as Vanilla Policy Optimization and Trust Region Policy Optimization. These adjustments were made in order to make these policy gradient methods more practical, easier to implement, and faster. PPO has been successful in different fields, one example being esports. PPO has been used to train an agent which is able to defeat the world champions of the game Dota 2 (OpenAI, 2019). This PPO-based model was the first AI system to defeat world champions at an esport. Other fields where PPO has been successful are telecommunication (Liu, Quan, Cheng, Xu, Deng and Gao, 2022) and stock trading (Chen, Shih, Lai, Chang and Huang, 2023). Jin, Duan, Song and Li (2023) did research on the use of a PPO algorithm for the CSP. In their research, their algorithm outperformed heuristics and rule-based approaches. Furthermore, Kozlica, Wegenkittl and Hirländer (2023) compared PPO with Q-learning for a sorting problem. In their research, PPO outperformed the Q-learning model.

The first environment of this research is the Maze environment. Solving a maze using RL is studied by Osmankovic and Konjicija (2011) and Sharma, Kaur and Prashar (2023). They used a Q-learning model to solve a single instance of a maze. This research aims to develop a solver that can solve any random maze.

The second problem in this research is the TSP. One method used for the TSP is a hyper-heuristic. A hyper-heuristic automates the process of selecting lower-level heuristics to solve the problem. This approach is used for a competitive TSP

(Kendall and Li, 2012). Müller and Bonilha (2021) used a hyper-heuristic to improve the TSP solution found by ant colony optimization.

Another method used for the TSP is Reinforcement Learning. Wang, Xiao, Wang and Ruan (2023) compared the use of Q learning, SARSA, and Double Q learning for the TSP. A difference with their research is that they trained their model on single instances of the TSP. Other researchers built models that are able to solve random TSP instances after they are trained successfully (Bello, Pham, Le, Norouzi and Bengio, 2016), (Kool, Van Hoof and Welling, 2019) and (Bresson and Laurent, 2021). These models are composed of different RL techniques in order to improve the model's quality. These techniques include using more complex network structures, encoding and decoding algorithms, and optimizing the found solution after the model is used. RL could also be combined with a hyper-heuristic. In that case, the policy for choosing a heuristic is trained with RL. Kallestad, Hasibi, Hemmati and Sörensen (2023) successfully implemented a hyper-heuristic for the TSP combined with RL, their model is trained using PPO.

The algorithms with more complex structures are able to achieve higher performance compared to the hyper-heuristic approaches. Bello et al. (2016) argue that because a hyper-heuristic operates on the search space of heuristics instead of the search space of solutions, a hyper-heuristic may not be able to achieve as good of a performance as their model. The scope of this research is to investigate the use of PPO, the combination of different complex Reinforcement Learning techniques is outside the scope of this research. A RL-based hyper-heuristic has been proven to be successful and will therefore be used in this research for the TSP.

Chapter 4

PPO

In this section, the PPO algorithm will be explained. First, relevant background information about Reinforcement Learning will be treated. Then, Actor-Critic style Reinforcement Learning will be explained because this is the basis for PPO. Lastly, the model characteristics specifically used in PPO will be covered.

4.1 Background

4.1.1 Reinforcement Learning

Reinforcement Learning is a subclass of Artificial Intelligence where an agent is trained to make decisions in order to maximize total reward. RL has been successful in various different fields, one example is the use of RL for training a model that achieved superhuman level in playing Atari 2600 games, Chess, Shogi or Go (Schrittwieser et al., 2020). The agent learns by interacting with the environment. Eventually, after sufficient training, the agent is able to determine which actions to choose at a given state that maximizes the expected return.

4.1.2 Markov Decision Process

Reinforcement learning models are usually defined by a Markov Decision Processes (MDP). A MDP is a mathematical foundation for RL algorithms. Components of a MDP are:

$\mathcal{S}$ State space is the set containing all possible states of the environment.

$\mathcal{A}$ Action space is the set containing all possible actions in the environment.

$P(s_t, a_t, s_{t+1})$ Transition function P is the interaction of the agent with the environment. This transition can either be deterministic or stochastic. In case of a deterministic transition, the function $P(s_t, a_t) : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ returns the next state based on a given state-action pair. In case of a stochastic transition, the function $P(s_{t+1}|s_t, a_t) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ returns transition probabilities, that is the probability that a state is the next state for a given state-action

pair. Note that the deterministic transition function can be seen as a special case of the stochastic transition function where $p(s_{t+1}) = 1$ for the next state and 0 for all other states. All environments used in this research have a deterministic transition function. In context of the container terminal, the transition function is usually stochastic.

$r(s_t, a_t, s_{t+1})$ $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ Reward function returns the reward after the agent interacts with the environment. Because all transitions are deterministic, reward is only defined at one future state s_{t+1}. Without lose of generality, reward will be written as $r(s_t, a_t)$ in this research.

The following components are derived from a MDP and are needed in for RL.

$\pi(s_t)$ is the policy at state s_t, this is the probability distribution for all actions at state s_t.

$R(\tau)$ as described in Equation 4.1 is the discounted return of one trajectory with discount factor γ. Maximizing the return of any trajectory is the main objective of Reinforcement Learning. The challenge is to determine which actions to take to maximize return. The return of any state within a trajectory can be calculated with the formula described in Equation 4.2. Note that the infinity range for the sum does not imply that there are infinite time steps. The infinity implies that the summation is taken using the remaining steps in the trajectory.

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \text{ where } \gamma \in [0, 1] \tag{4.1}$$

$$R(s_t) = \sum_{l=0}^{\infty} \gamma^l r_{t+l} \text{ where } \gamma \in [0, 1] \tag{4.2}$$

$V^{\pi}(s_t)$ as described in Equation 4.3 is the value function. This is the expected discounted return in state s_t where actions are taken according to policy π.

$$V^{\pi}(s_t) = \mathrm{E}\left[R(s_t)\right] \tag{4.3}$$

$Q^{\pi}(s_t, a_t)$ as described in Equation 4.4 describes the expected discounted return, given action a_t will be taken in the next step.

$$Q^{\pi}(s_t, a_t) = \gamma \mathrm{E}\left[R(s_{t+1})\right] + r(s_t, a_t) \tag{4.4}$$

$A^{\pi,\gamma}(s_t, a_t)$ as described in Equation 4.5 is the discounted advantage function, which is an expectation how good an action is in comparison to other actions at this state under policy π and discount factor γ. If the advantage is negative, action a_t is expected to be a worse action than average under policy π. If the advantage is positive, action a_t is expected to be a better action than average under policy π.

$$A^{\pi,\gamma}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{4.5}$$

4.1.3 Policy gradient methods

In reinforcement learning, two primary categories of learning methods exist: value-based methods and policy-based methods. In value-based methods, the model's principal objective is to learn the value function. This is achieved by iteratively updating its value function. During training, actions are typically chosen using an ϵ-greedy policy. After training, the optimal policy is derived by selecting the actions that lead to the states with the highest value. In policy-based methods, the principal objective of the model is to learn the optimal policy directly. The model accomplishes this by updating its policy during training, with actions typically chosen according to the current policy. After training, the optimal policy can be extracted by selecting actions with the highest probability.

A notable distinction between value-based and policy-based methods lies in their exploration process. In value-based methods, the model distinguishes only between optimal and non-optimal actions. With ϵ-greedy action selection, the model has a high probability of visiting the current optimal state, while other states are visited with uniformly lower probabilities. The exploration process is independent of the agent's confidence in the optimality of a specific action. In contrast, policy-based methods not only differentiate between optimal and non-optimal actions but also consider the probability distribution of actions. The exploration process is directly tied to the agent's confidence in the optimality of a particular action.

PPO is an Actor-Critic Reinforcement Learning algorithm, this is a hybrid method of value-based and policy-based Reinforcement Learning. PPO uses two distinct networks to determine and evaluate its actions. The actor network, which is policy-based, determines the policy $\pi(s)$. The critic network, which is value-based, determines the value function $V(s)$. The key of Actor-Critic Reinforcement Learning lies in the interaction between the actor and the critic network, where the value network is used to help the policy updates of the actor network by providing an estimate of the true value of a state. A more detailed explanation about Actor-Critic will be explained in the next section.

4.2 Actor-Critic

As PPO uses an Actor-Critic reinforcement learning model as its foundation, this section provides a detailed explanation of Actor-Critic models. As outlined previously, AC uses an actor network to determine its policy and a critic network to evaluate that policy. The schematic diagram in Figure 4.1 illustrates the structure of an AC model, which undergoes two main steps after initialization: experience generation and learning from these experiences.

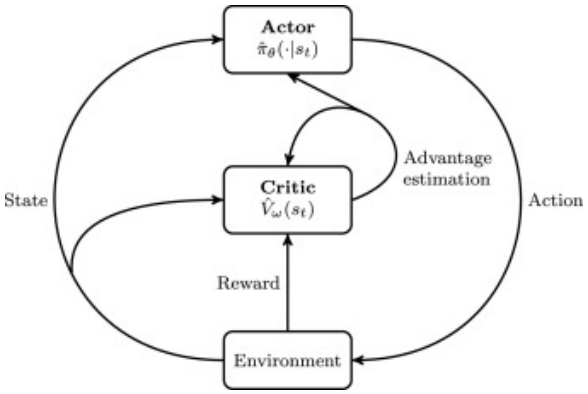

Figure 4.1: *Schematic diagram of the Actor-Critic model. Figure from Monaci, Agasucci and Grani (2021).*

Generate experiences: Experiences have to be generated for the model to learn from, this is done by generating multiple trajectories. Relevant information about these trajectories, such as state, action, policy, value, and reward, is stored in memory. Each trajectory starts with an initial state drawn from the environment, which is put into both networks. The actor network outputs the current policy as a probability distribution of all actions from which an action is selected. Simultaneously, the critic network predicts the expected value for the given state.

Following the selection of an action, the next state and corresponding reward are determined using the transition function. The reward is crucial for calculating the advantage of preceding states. This process repeats until the stopping criteria for a trajectory are met, such as task completion or reaching a maximum number of steps.

Trajectories are constructed until the memory reaches a predefined size. At this point, the model transitions to the learning phase. During this phase, the stored information is used to update the models.

Learning phase: Advantages are calculated by Equation 4.5. These advantages are derived solely from the information within a singular trajectory, and they do not use information from other trajectories. The advantage is a measurement of how well an action performed compared to alternative actions at that step. A positive advantage implies that the action yielded a better return than initially expected at that given state. In that case, the action in that state should be encouraged. Additionally, such positive advantages suggest that the model is able to to outperform the initial estimate provided by the critic network. Consequently, the value estimation in that state should be increased. This logic forms the basis for the policy updates, making the advantage a crucial component of the loss functions for both the actor and the critic network.

The objective of the actor network is to maximize the expected return,which is a function of the policy. The advantage is a measurement to estimate the expected return, therefore the network is updated in the direction that the advantage is maximized. A positive advantage implies that the state action should be encouraged and a negative advantage implies that the state action should be discouraged.

In AC, the purpose of the critic network is to provide an estimate of the expected return in a state. Therefore, the error of the critic network is the difference between expected return and the true return. Where the expected return is the value at the state and the true return is estimated by the return of the the state following the trajectory. A natural choice is to use Mean Squared Error loss.

It is important to keep in mind that the value is an estimation. At the start of training, when the critic network is just initialized, these estimations are poor. As the model learns, estimations for the value function become more accurate, allowing the model to calculate the advantage more accurately, resulting in better network updates. Furthermore, when the actor network is just initialized, the policy assigns approximately equal probabilities to all actions, leading to exploration at the start of training. When the model learns and the same state action pairs result in approximately the same advantage estimation, the model will converge. This idea is the power of Actor-Critic models, where both networks use each other's information to improve the updates.

One final note is that the model described above is a specific case of an Actor-Critic model namely Advantage Actor-Critic (A2C), because the advantage is used as measurement. Original Actor-Critic model uses the Temporal Difference (TD) residual as measurement.

4.3 PPO specific characteristics

In this section the elements specifically used in PPO which need more clarification are explained. Policy clipping explained in Section 4.3.4. This concept is the part of the algorihtm that distinghuishes PPO from other policy gradient methods.

4.3.1 Generalized Advantage Estimator

As described earlier in this chapter, the advantage is an important measurement in the loss functions. The value and the expected return of a state-action pair are needed to calculate the advantage. The value is estimated by the critic network. However, determining the expected return is impossible because not all trajectories are calculated. In the memory, only one trajectory is available. Therefore, $Q^{\pi}(s_t, a_t)$ should be estimated by the discounted sum of future rewards. This results in an advantage estimator given in Equation 4.6.

$$\hat{A}(s_t, a_t) = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t) \tag{4.6}$$

This formula uses only one trajectory to calculate the expected trajectory reward. In this formula, the discounted sum of the future trajectory rewards is taken. In the summation, l is the number of steps in the future. As l increases, rewards will be less accurate in estimating the true l-step reward because it is an accumulation of uncertain events. While steps close to t give a more accurate estimate for the expected l-step reward. So as l increases, the variance of the advantage estimator

increases and the estimation becomes more inaccurate. In the Generalized Advantage Estimator (GAE), a new parameter is introduced that controls the weight of future rewards. Doing this will introduce a bias, but it will lower the variance.

In Schulman, Moritz, Levine, Jordan and Abbeel (2015) it is shown that Equation 4.6 can be rewritten as Equation 4.7 with δ the TD residual.

$$\hat{A}(s_t, a_t) = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t) \quad = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} \tag{4.7}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4.8}$$

New parameter $\lambda \in [0, 1]$ is introduced in the GAE. This gives the following advantage estimator.

$$\hat{A}^{GAE}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{4.9}$$

For $\lambda = 1$, the GAE is the same as the standard advantage estimator. For $\lambda = 0$ it holds that $\hat{A}^{GAE}(s_t, a_t) = \delta_t$, which is just a single step advantage. This single step advantage has a low variance but is has a high bias, because the true discounted advantage included all future rewards. For all $0 < \lambda < 1$, the trade-off between bias and variance can be controlled. This trade-off is generally considered positive because it stabilizes learning and makes updates more robust.

4.3.2 Pseudo code

The PPO algorithm can be divided into two main parts: generating experiences and the learning phase. When generating experiences, trajectories are generated based on the current policy. Relevant information about each step is stored in the memory. The algorithm will go to the learning phase when the memory size exceeds M. Rewards are normalized and advantages are calculated. In the learning phase, batches are randomly drawn from the memory, and subsequently, the advantages of the batch are normalized. Both the actor and the critic losses are calculated and used for updating both networks using stochastic gradient descent. Details about the loss function will be explained in Section 4.3.4 and Section 4.3.5.

Algorithm 1 PPO

$S \leftarrow S_0$
for episode **do**
    reset environment
    while not completed **do**
        $v_t = critic(s_t)$
        a_t is drawn from $actor(s_t)$
        retrieve s_{t+1} and r_t
        $s_t \leftarrow s_{t+1}$
        save information of step t in memory
    if Memory size $\geq M$ **then** ▷ Learning phase
        calculate $\hat{A}$
        normalize rewards
        for K epochs **do**
            sample batch of size B
            normalize $\hat{A}_B$
            calculate Loss Functions
            update *actor* by optimizing L_{actor}
            update *critic* by optimizing L_{critic}

4.3.3 Learning phase

When the memory has reached a minimum of size M, the model will go to the learning phase. In the learning phase, batches of size B are drawn from the memory. Batches consists of randomly drawn samples from the memory, one sample is one action. A total of $K\frac{M}{B}$ batches are used. In literature, K is the number of epochs, since a total of KM steps are used to update the policy, which are devided over KB batches. It is common to use $K > 1$, in this cases both networks are updated multiple times with same samples in each learning phase. Using samples multiple times to update the networks makes the algorithm more sample efficient.

Using sample multiple times for updating the networks has a downside. Updating the policy too much may lead to unstable learning. To avoid this problem, the size of the total policy update in the learning phase should be limited. This idea is applied in Trust Region Policy Optimization (TRPO) methods (Schulman, Levine, Moritz, Jordan and Abbeel, 2015). In TRPO, a constraint is added in order to limit

the size of the policy update. In PPO, the idea of limiting the policy updates is the same as in TRPO. The difference is that in PPO the hard constraint is replaced by a change in the loss function. This change may lead to small violations of the hard constraint used in TRPO, but it is simpler to implement, more general, has a better sample complexity and does not require the computation of second order gradients (Schulman et al., 2017).

After updating, it is arguable whether to clear the memory. The advantage of using older samples is that vital information from previous samples could be used for future updates. A disadvantage of using older information is that these estimations are less accurate because PPO is on policy. Therefore, using too old data will result in learning wrong predictions. If a prioritized experienced replay buffer is correctly implemented, the learning could be improved by using helpful past experiences and removing inaccurate and less valuable experiences. In the models used in this research, the memory is cleared after each update phase. In future research, experiments could be done with a prioritized experienced replay buffer to increase performance (Liang, Ma, Feng and Liu, 2021).

4.3.4 Policy clip actor loss

As described earlier in this chapter, the advantage function is an important component of the loss function. More precicely, $-A_t$ is a good starting point for the loss function when using gradient descent. Specifically, if $A_t > 0$, action a_t is estimated to be better than average in s_t. So if $A_t > 0$, $p_\pi(a_t|s_t)$ will increase after applying gradient descent on the actor network. The actor loss function in PPO is constructed from the GAE as defined in Equation 4.9 with an additional minimum operator and clip function that limits the size of the policy update in each updating phase.

In TRPO, the policy change is expressed by the Kullback-Leibler divergence, in PPO, the policy change is expressed in the ratio between the current policy and the old policy of the state action pair that is considered, shown in Equation 4.10. Where $p_{\pi_{old}}(a_t|s_t)$ is the probability of action a_t in state s_t at the moment the action was taken. $p_\pi(a_t|s_t)$ is the probability of action a_t in state s_t under the current policy, which is extracted from the result of the current actor network at s_t. Note that for the first update in each update phase, these two policies are always the same and therefore $r_t(\pi) = 1$ in the first policy update in each update phase. The idea of limiting the policy update is that after k updates, the policy difference between these two policies should not be too large. PPO approximately limits $r_t(\pi)$ between $[1 - \epsilon, 1 + \epsilon]$. It approximately limits the policy update because the hard constraint is replaced by a clip function, which does not strictly limit the policy update. Furthermore, it approximates the policy update because the constraint does not take the entire distribution into account, it only takes the probability distribution at one specific state into account.

$$r_t(\pi) = \frac{p_\pi(a_t|s_t)}{p_{\pi_{old}}(a_t|s_t)} \tag{4.10}$$

The actor loss function is defined by Equation 4.11, where the clip function is defined by Equation 4.12.

$$L^{actor} = \hat{\mathbb{E}}_t \left[-\min \left(r_t(\pi)\hat{A}_t, \text{clip}\left(r_t(\pi), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{4.11}$$

$$\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) = \begin{cases} 1-\epsilon & \text{if } r_t(\theta) < 1-\epsilon \\ 1+\epsilon & \text{if } r_t(\theta) > 1+\epsilon \\ r_t(\theta) & \text{otherwise} \end{cases} \tag{4.12}$$

Combining the clip function and the minimum operator gives 6 scenarios which are described in Table 4.1 and are visualized in Figure 4.2.

	$r_t(\pi)$	$\hat{A}_t$	L^{actor}
1	$\in [1-\epsilon, 1+\epsilon]$	> 0	$-r_t(\pi)\hat{A}_t < 0$
2	$\in [1-\epsilon, 1+\epsilon]$	< 0	$-r_t(\pi)\hat{A}_t > 0$
3	$< 1-\epsilon$	> 0	$-r_t(\pi)\hat{A}_t < 0$
4	$< 1-\epsilon$	< 0	$-(1-\epsilon)\hat{A}_t > 0$
5	$> 1+\epsilon$	> 0	$-(1+\epsilon)\hat{A}_t < 0$
6	$> 1+\epsilon$	< 0	$-r_t(\pi)\hat{A}_t > 0$

Table 4.1: *Scenarios of actor loss.*

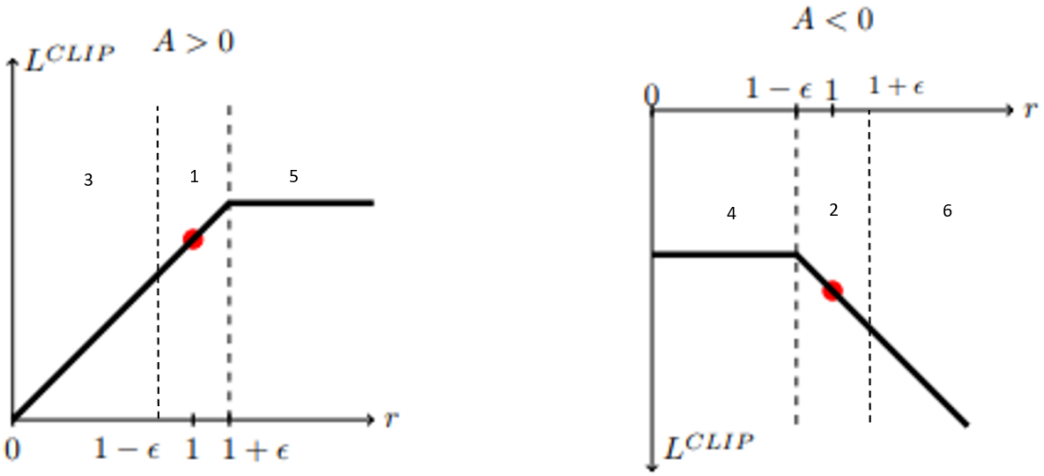

Figure 4.2: *Scenarios of actor loss visualized. Picture is a modified version of the one in Schulman et al. (2017).*

In scenarios 1 and 2, the policy ratio is within $[1-\epsilon, 1+\epsilon]$, which suggests that the policy is not updated too much when considering this sample, so a policy update is allowed. In scenario 3 and 4, $r_t(\theta) < 1-\epsilon$. This implies that $p_\pi(a_t|s_t)$ is much smaller than $p_{\pi_{old}}(a_t|s_t)$, represented on both figures' left side. In this case, a policy update is only allowed if it increases $p_\pi(a_t|s_t)$ and, therefore, shifts $r_t(\pi)$ towards $[1-\epsilon, 1+\epsilon]$. This increase in $p_\pi(a_t|s_t)$ corresponds with an $L_t^{actor}(\pi) < 0$, which

corresponds to an $\hat{A}_t > 0$, which is the case in scenario 3 in the figure. In scenario 4, where $\hat{A}_t < 0$, a policy update would decrease $r_t(\pi)$ even further than it already is, which is not allowed. In this case, the loss function is clipped. This results in a constant loss function, therefore, the gratient of the loss function is 0. When the gradient is 0, the actor network does not change with the gradient descent update. In scenarios 5 and 6, the same logic is applied but then the other way around.

4.3.5 Critic loss

A squared error loss is used for the critic network, defined by Equation 4.13. Here, $V_{old}(s_t)$ is the value estimated when the action was taken, this is also the value used for calculating the advantage. $V(s_t)$ is the value of the state according to the current network. Other loss functions, such as the Huber Loss, are also possible. However, the MSE is used since this is the loss function implemented in the original PPO paper (Schulman et al., 2017).

$$L^{actor} = (V_{old}(s_t) + \hat{A}(s_t, a_t) - V(s_t))^2 \tag{4.13}$$

4.3.6 Network weight sharing

When using Actor-Critic Reinforcement Learning, sharing layers of the actor and critic network is possible. Where the weights of the first layers are shared between the actor and the critic network, the last layer is different due to the different outputs of both networks. In this case, the actor and critic loss are added and used to update the networks. This layer sharing is, for example, done in the original PPO implementation (Schulman et al., 2017). One argument for sharing layers is that the learning process is more parameter efficient since fewer parameters are updated. It is also possible to detect underlying structures which affect both networks earlier. The choice of sharing layers can be motivated by the fact that both networks are based on the same underlying state space, and therefore, it is plausible that sharing layers will work. A downside of sharing layers between networks is that the loss functions must be balanced. If one of the two loss functions has a too high share in the total loss, the model will not properly learn. Another argument against layer sharing is that it is not necessarily true that both networks share the same underlying structure for a given network.

Chapter 5

Implementation details

This section discusses the hyperparameter values. Most of these hyperparameters remain untouched by the research. This is the starting point of the model.

5.1 Hyperparameters

5.1.1 PPO parameters

$\gamma = 0.99$

$\lambda^{GAE} = 0.95$

$\epsilon = 0.2$

γ and λ^{GAE} are the hyperparameter of the advantage estimator. These values are bases on the PPO paper (Schulman et al., 2017) and the GAE paper (Schulman et al., 2017). In the GAE paper, the researchers argue that for both γ and λ between 0.9 and 0.99, the model has the best overall perormance. They also found empirically that a λ lower than γ gives the best results. They argue that this is likely because λ introduces less bias than γ for a reasonable accurate value function. In the PPO paper, these values as stated above are used. In the PPO paper, based on their problems, $\epsilon = 0.2$ performed the best. Therefore, these values for $\gamma, \lambda^{GAE}, \epsilon$ are used.

5.1.2 Adam optimizer

$\lambda = 0.0003$

$\beta_1 = 0.9$

$\beta_2 = 0.999$

$\epsilon = 1e - 07$

The optimizer and its hyperparameters are based on the results of Andrychowicz et al. (2021). They performed gridsearch on hyperparameters of commonly used optimizers. They found that Adam opitimizers with learning rate $\lambda = 0.0003$ performed best. They tested $\beta_1 = 0$ against $\beta_1 = 0.9$, which is default by Tensorflow

and found that the optimizer with momentum perfomed best. They did not not find differences in performance for ϵ and they did not include β_2 in their research. Therefore, these values are set to the Tensorflow default values are are stated here.

5.2 Learning phase

$K = 4$ is the number of epochs

$M = 4092$ is the total memory size required before learning

$B = 64$ is the batch size

The number of epochs varies in different PPO papers. In the original paper, values between 3 and 15 are used. Andrychowicz et al. (2021) used 10 without further explanation. (Trott, Zheng, Xiong and Socher, 2019) used 4 epochs for their maze environment. In none of these papers, a clarification is given for the number of epochs. In this research, 4 epochs are used because this value is used in (Trott et al., 2019). In literature, common batch sizes implemented are between 16 and 256. (Trott et al., 2019) and (Li, Gama, Ribeiro and Prorok, 2019) have both batch size 64 in their research. Therefore the batch size is set to 64. The memory size M differs in literature, it ranges from a few hundred up to millions. In this research, M is determined by trial and error for small mazes.

Chapter 6

Maze

The first environment used to develop the model is the maze environment. In contrast with the reviewed literature about RL models used for solving mazes, this model aims to solve any random maze of a particular size instead of one maze. Therefore, this research starts with smaller mazes. If the algorithm is able to solve these smaller mazes, the maze sizes will be increased. The maze environment can be related to the transportation of containers over the yard. Transportation from the ship to the stacking location is done by Automated Guided Vehicles (AGV). On the route, there are obstacles, such as other AGVs.

6.1 Environment setup

An $n \times n$ grid describes the maze environment with a random start point and a random endpoint within the grid. The agent's objective is to move from the start to the goal in as few steps as possible. Obstacles are randomly generated. The probability of a square being an obstacle is fixed at 0.5 in this research. Before a generated grid is used, the grid is tested on feasibility. In case the grid is infeasible, a new grid is generated. One episode is finished if either the agent reaches the goal or a maximum number of steps is reached. In this research, the maximum number of steps is set at 50.

Feasibility is checked by applying a shortest-path algorithm. When this algorithm succeeds, this grid will be used. Furthermore, the shortest path algorithm finds an optimal solution, which will be used to check if the agent is able to solve the maze optimally. This chapter will discuss solving a maze and solving a maze optimally. If the agent finds the goal, the maze is solved. The maze is solved optimally if the agent finds the goal in the minimum steps possible.

The state is defined by a binary matrix of size $n \times n \times 3$. The first $n \times n$ layer represents the obstacles, the value equals 1 if there is an obstacle on the corresponding square. The second layer represents the agent's location and the last layer represents the goal's location. Action space $\mathcal{A}$ is defined by moving up, down, left, and right.

The following reward system is used: -3 if the agent tries to walk into an obstacle, 0 if his step reaches the goal, and -1 for any other step. The -1 is included in order to make sure that the agent will find the shortest path. In this research, a

deliberate choice is made to include the possibility of the agent walking into an obstacle. This choice makes the input simpler, but the agent must learn to avoid these actions. The motivation behind this choice is the goal in mind that the model should be applicable to other problems, such as the container terminal. In the container terminal, it is not always easy to determine the invalid actions. In the context of the AGV routing, it could be possible that some actions are blocked because another AGV occupies a place. Therefore, these invalid actions are included as possibilities, so the model must learn not to use them.

6.2 Network structure

Li et al. (2019) researched a path-finding algorithm for a robot in a grid environment with obstacles. In their model, convolutional layers were used. Convolutional layers are suitable for detecting patterns in the input. In the maze problem, there is a clear dependency between neighbour points in the state. The maze is a grid-like environment, and therefore, a Convolutional Neural Network will be used.

A model with two convolutional layers is used. Each layer has 32 filters and a 3×3 kernel. This model is trained for 25,000 training episodes. The model is tested after every 500 episodes. While testing, the model takes the action with the highest probability. A total of 200 test instances are performed. The average reward of the test instances is shown in Figure 6.1, and the average probability of the actions taken in the test mazes is shown in Figure 6.2. This is done for 200 mazes, and the average results are shown in the graphs. Figure 6.2 shows the average probability of the actions that are taken during testing. From these graphs, it can be observed that the model quickly converges to a solution that is not optimal.

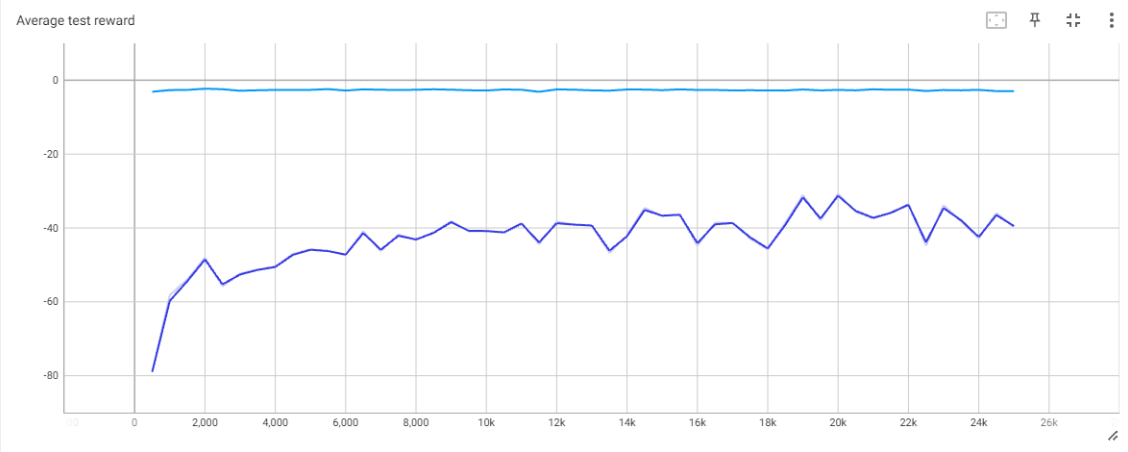

Figure 6.1: *Average test reward plotted over the nubmer of training episodes. Dark blue is the test reward of the model, light blue is the optimal reward.*

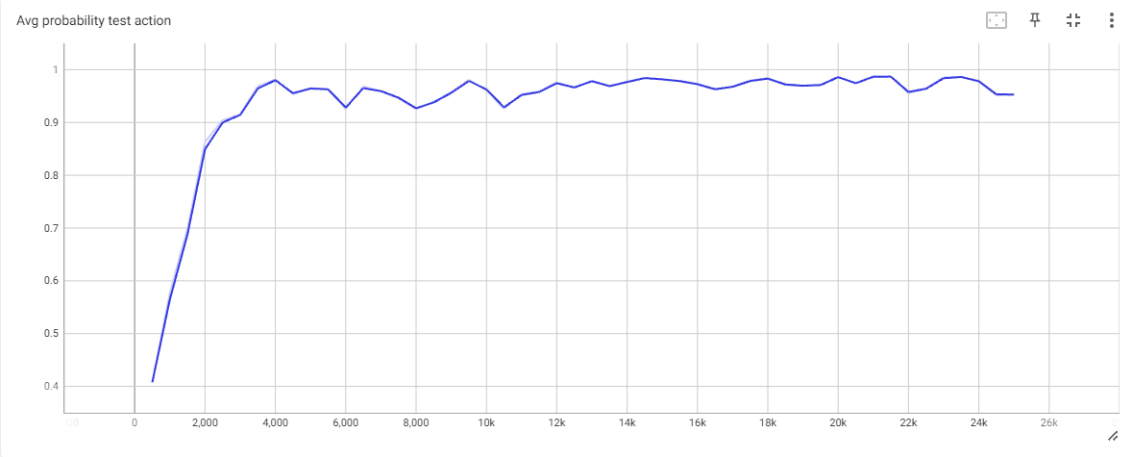

Figure 6.2: *Average probability of test actions taken plotted over the number of training episodes.*

6.3 Entropy regularization

The problem with the model above is that it quickly converges to a solution which is not optimal. This means that the exploration-exploitation process is not well balanced. The model exploits too quickly in this case. Entropy regularization can be used to control the exploration-exploitation process.

The entropy term measures randomness in the policy's action distribution and is described in Equation 6.1. A high entropy indicates that the policy is uncertain. In that case, the probabilities corresponding to all actions are close to each other. This is the case for an exploring policy, where the probabilities of all actions are close to each other. A low entropy indicates that the policy is confident, corresponding to probabilities far from each other. When the agent is exploiting, the probability is close to 1 for one action and close to 0 for other actions, corresponding to a low entropy. The entropy term is a measurement of randomness and can indicate how exploring a policy is, this will be used to steer the policy updates is such a way that exploration can be controlled.

$$H_t = -\sum_{a_t \in \mathcal{A}} p(a_t) \log(p(a_t)) = -\mathbb{E}[\log(p(a_t))] \tag{6.1}$$

The problem that the model converges too quickly is solved by favouring policy updates that result in a more exploring strategy. In order to do this, the entropy term will be used. This entropy term is used as a bonus in the loss function as described in Equation 6.2. The objective of the policy updates is to minimize the loss function. By subtracting the entropy term from the loss, the loss is lowered by a relatively large entropy term for an exploring policy. The loss is lowered by a relatively small entropy term for an exploiting policy. Because the objective of a policy update is to minimize the loss, policy updates in the direction of an exploring policy have an advantage over exploiting policy updates. Therefore, by including the entropy term in the actor loss function, the actor network not only maximizes

the expected advantage but also favours exploring policy.

$$L^{actor} = \hat{\mathbb{E}}_t \left[-\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) - \omega H_t \right] \tag{6.2}$$

Subtracting the entropy from the loss function does not necessarily imply that the model will not converge. This entropy term favours updates in an exploring direction, but it still minimises the loss function. Therefore, the size of the entropy term must be balanced. This is done by introducing the parameter ω, which is the entropy weight. In the loss function, ωH_t is used as an entropy term, where ω is the weight of the entropy. If ω is too small, the entropy does not have enough impact, and therefore, the model will still converge too quickly. If the entropy term is too large, the model will not converge since converging is discouraged too much. It is common practice to schedule ω from a positive number towards zero. This ensures that the model will converge.

In PPO literature, the entropy term is added to the loss function (Schulman et al., 2017). However, in actor-critic algorithms, researchers implemented the entropy term in the advantage function (Su and Lu, 2021), (Dong, Zhang, Shi and Li, 2022). Adding the entropy to the loss function function directly impacts the gradient of the loss function since it is directly included in the loss function (Schulman, Chen and Abbeel, 2017). When including the entropy term to the advantage, as shown in Equation 6.3, the entropy term does not directly influence the gradient because the advantage is a fixed number in the loss function. The advantage measures how good an action is. By incorporating the entropy term in calculating the advantage, actions taken under an exploring policy get a relatively high bonus on the advantage. Actions taken under an exploiting policy get a relatively low bonus. Because advantage is maximized, the entropy term indirectly encourages exploration by adjusting the advantage. Another way of interpreting the difference is that by including the entropy term in the loss function, policy updates are encouraged to go in an exploring direction. Therefore, the entropy of the new policy is used. Including the entropy term in the advantage encourages exploring strategies by assigning a higher advantage to steps taken under an exploring policy. Therefore, the entropy of the policy under which the action was taken is used. Although there is a difference between these two implementations, both methods encourage exploration. This research follows the original PPO literature (Schulman et al., 2017), and the entropy term is added to the loss function as described in Equation 6.2.

$$\hat{A}_t = \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t}(r_k + \gamma V(s_{k+1}) - V(s_k) + \omega H_k) \tag{6.3}$$

6.3.1 Results

The entropy weight is linearly decayed to zero over 5000 episodes. This encourages exploration at the start of training but allows the model to exploit after 5000 episodes. Test results for different entropy weights are given in the figures below. The following start values are used in these figures: Blue has no entropy term. Red starts at 0.01. Yellow starts at 0.1. Green starts at 1.

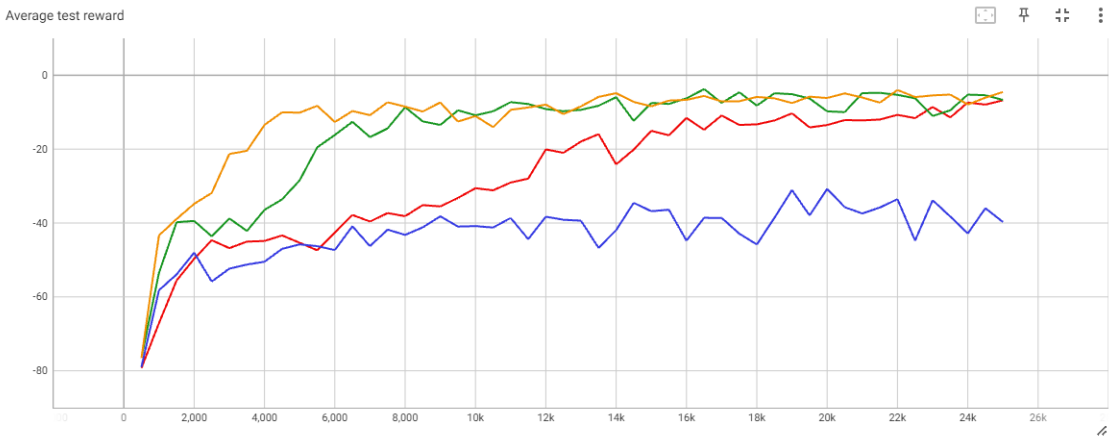

Figure 6.3: *Average test reward plotted over the number of training episodes for different entropy weights. Blue has no entropy weight, red starts at 0.01, yellow starts at 0.1, green starts at 1. All entropy weights decay to 0 over 5000 episodes.*

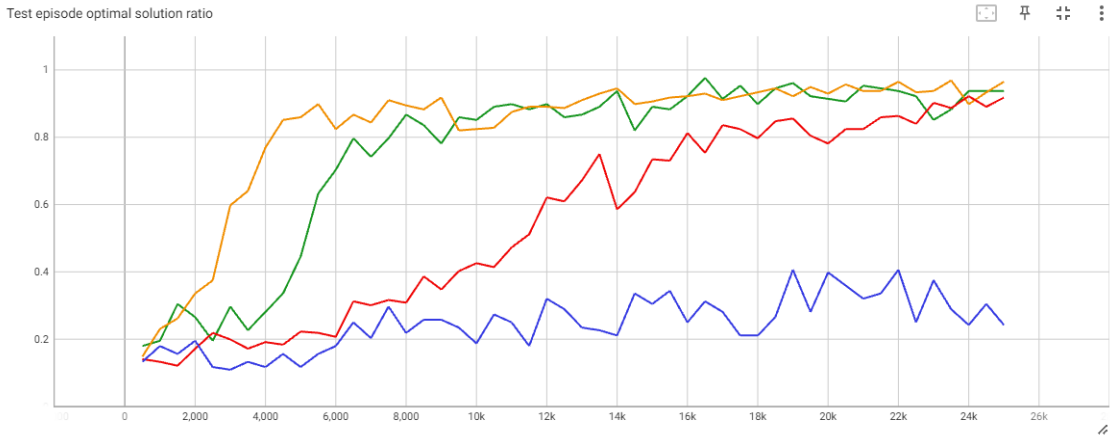

Figure 6.4: *Ratio of test instances that the model solves the maze plotted over the number of training episodes for different entropy weights. Blue has no entropy weight, red starts at 0.01, yellow starts at 0.1, green starts at 1. All entropy weights decay to 0 over 5000 episodes.*

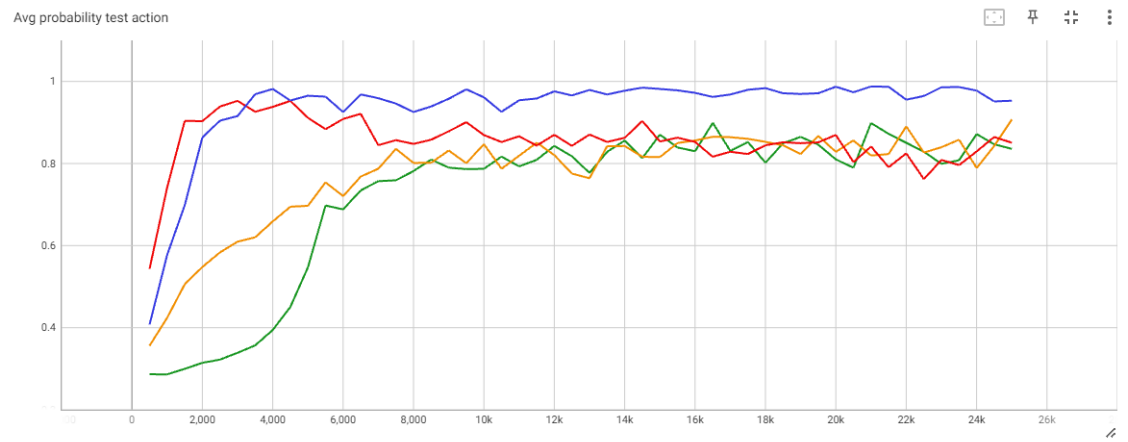

Figure 6.5: *Average probability of test action taken plotted over the number of training episodes for different entropy weights. Blue has no entropy weight, red starts at 0.01, yellow starts at 0.1, green starts at 1. All entropy weights decay to 0 over 5000 episodes.*

The model without entropy regularization converges quickly to a local optimum since the probability of test actions is near 1 after 2000 episodes. The test reward struggles to get higher than -40. An entropy weight that starts at 0.01 is too low since the probability of the test action quickly increases, but the test reward does not. However, compared to no entropy term, this red line is able to escape the local optimum within the 25,000 training episodes. The green and yellow lines do not have the problem of converging too quickly. These models benefit from the exploration at the start. In the green line's model, the entropy term is higher than necessary, which slows down learning. The yellow model has a better balance between exploration and exploitation. Therefore, an entropy weight that decays from 0.1 to 0 in 5000 episodes will be used.

The configuration with an entropy weight that starts at 0.1 and decays to 0 in 5000 episodes is trained for more episodes. Results after 50,000 training episodes are given in Figure 6.6. The model solves about 96 per cent of the mazes optimally. A noteworthy result is that the model learns to solve the first 90 per cent of the mazes relatively quickly, but learning to 96 per cent takes relatively many steps. This will be further covered in the next section. Another observation is that the difference between optimally solving and not optimally solving the mazes is neglectable. If the agent is able to solve the maze, it will solve it almost certainly optimal. In Figure 6.7, two mazes are shown. On the left is an example of a maze the agent cannot solve. On the right is a maze where one wall is moved one square, which the agent can solve. In the left maze, the agent's policy is to walk between squares 2 and 3. The problem here is that the agent walks into a dead-end and it does not know how to avoid this situation. The algorithm did not learn this structure. A possible explanation for why the model did not learn the structure from the left maze is that this structure does not occur much in the training data since the model is able to solve 96 per cent of the mazes. The model is converged to a policy in which the agent walks into this dead-end.

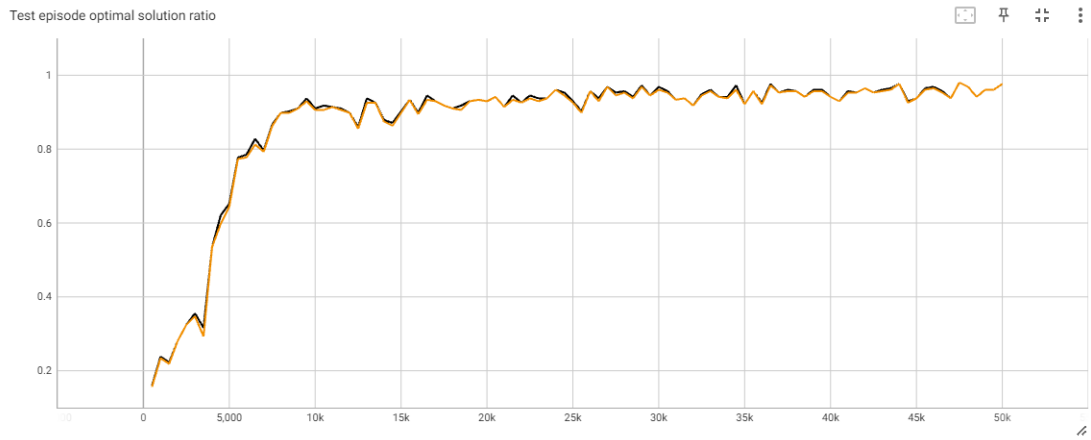

Figure 6.6: *Ratio that the agent solves* 5×5 *test instances plotted over the number of training episodes. Orange line is optimally solving the maze, black line behind it is solving the maze in more steps than necessary.*

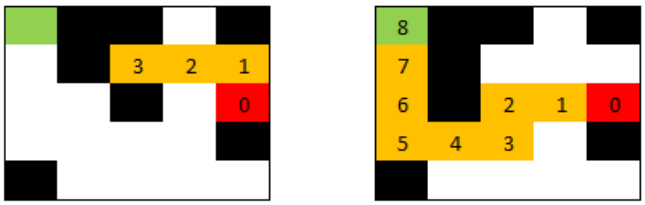

Figure 6.7: *Two examples of test instances after 50,000 training episodes. Red is the start, green is the goal and orange is the path the agent takes.*

6.4 Second model

As discussed in the section above, the model quickly learns to solve the first 90 per cent of the mazes, but learning to solve 96 per cent of the mazes takes relatively many episodes. The model is able to solve a large part of the mazes, but it encounters challenges in solving exceptional cases. This limitation arises from the model's inability to learn the characteristics of these exceptional cases. One problem is the sparse representation of these unsolvable mazes in the memory. Consequently, the model mainly learns from mazes, which can already be solved. This sparse representation could be handled by employing two data pools and randomly selecting one to extract a sample (Narasimhan, Kulkarni and Barzilay, 2015). This concept is used in combination with a second model. D'Eramo and Chalvatzaki (2022) used different agents to learn different tasks. This research introduces a second model, exclusively trained on mazes that are not solvable by the first model. The second model's task is to solve unsolvable mazes with the initial model. Combining the models should improve overall solvability.

After training the initial model, a testing phase follows. A total of 50,000 unsolvable mazes are saved. Subsequently, the second model is trained. Rather than randomly generating mazes, mazes are drawn from the unsolvable mazes, thereby

targeting the specific challenges encountered in these unsolvable mazes.

The second model is trained with the same hyperparameter as the initial model. Using the same configuration led to the same problem, which was solved by adding the entropy term. The model converged too quickly to a local optimum and did not explore enough. Results for different entropy weights are given in Appendix A.1.. An entropy weight starting at 0.1 that decays over 10,000 episodes yielded the best return. A model is trained with this configuration. Because the model was still learning at 50,000 training episodes, the learning continued to 100,000 training episodes. Because there are more episodes than the number of mazes in the dataset, mazes will be used more than once. Since the dataset is large enough, this will not lead to overfitting. Results are given in the plots below, together with the initial model.

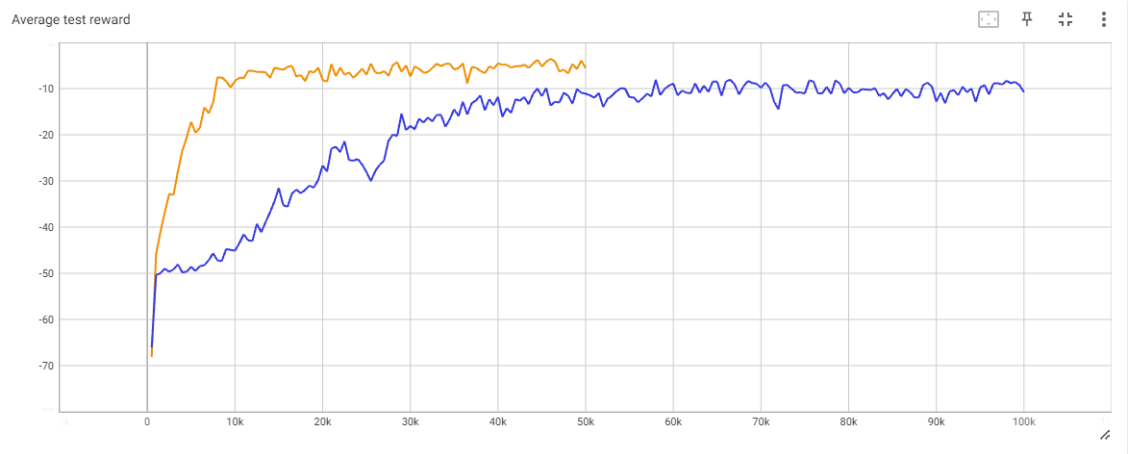

Figure 6.8: *Average test reward plotted over the number of training episodes. Orange is the initial model and blue is the model trained using the exceptional instances.*

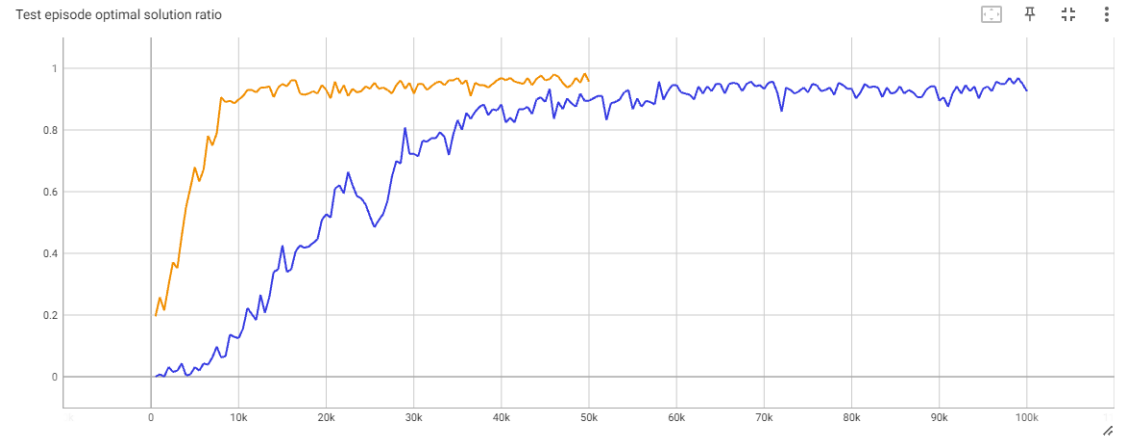

Figure 6.9: *Ratio of test instances that the model optimally solves the maze plotted over the number of training episodes. Orange is the initial model and blue is the model trained using the exceptional instances.*

These graphs illustrate that the second model effectively learns to solve instances the initial model could not. Learning these exceptional cases is more difficult. The

second model is trained and tested solely on exceptional mazes, but it is interesting how it performs on all kinds of mazes. To test the overall solvability of these trained models, both models are tested with 50,000 randomly generated mazes. Note that most of these test mazes are easier mazes. Both models are tested with the same instances to make a fair comparison.

	Solved by model 1	Not solved by model 1
Solved by model 2	0.948	0.035
Not solved by model 2	0.013	0.003

Table 6.1: *Division of solvability between the two models.*

Model	Solve ratio
Model 1	0.962
Model 2	0.983
Models combined	0.997

Table 6.2: *Solvability of the two models.*

Based on the results in Table 6.1, there is no clear superiority of one model over the other since both models are able to solve mazes which the other model could not solve. Therefore, the second model learns to solve different mazes than the initial one. Combining both models increases the overall solvability to 99.7 per cent, which is higher than the individual models.

One remark on these results is that the initial model is trained for 50,000 episodes and the second model is trained for 100,000 episodes. Comparing the combined model with the initial model is not reliable. A more interesting comparison would be the combined model's performance against the initial model's performance when it is trained for the same amount. Therefore, the first model is trained further to make the comparison more reliable. The model's performance is measured every 50,000 training episodes and the results are given in Table 6.3. The initial model is able to solve 98.4 per cent, which is lower than the 99.7 per cent solvability of the combined model. Therefore, combining both models increases solvability.

Training episodes	Solve ratio model 1
50,000	0.962
100,000	0.973
150,000	0.982
200,000	0.984
250,000	0.983

Table 6.3: *Solvability of the inital model. At these points, the model is tested for 10,000 epiosdes.*

6.5 Larger mazes

In this section, two new models are trained for larger mazes: one for 7×7 mazes and one for 10×10. The configuration used for the initial 5×5 model is used.

Furthermore, only one model is deployed for each maze size. After 250,000 episodes, the model is able to solve 95.7 per cent of the test instances for the 7×7. However, it fails to learn significantly for the 10×10 mazes. This will be further investigated in this chapter.

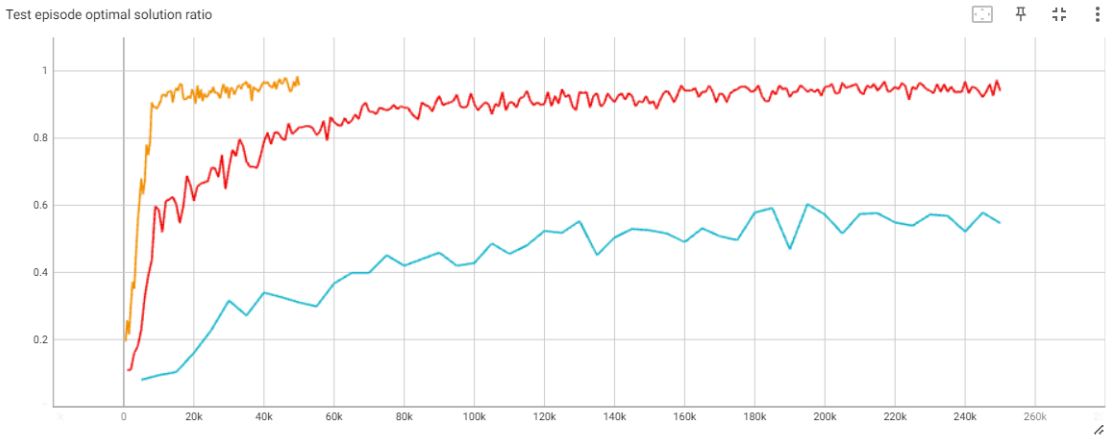

Figure 6.10: *Ratio of test instances that the model optimally solves the maze plotted over the number of training episodes for models used for different maze sizes. Orange* 5×5*, red* 7×7*, light blue* 10×10*.*

6.5.1 Entropy weight

Looking at the test probabilities in Figure 6.11 of the model deployed on 10×10 mazes, the model does not explore at the start. The probabilities quickly increase at the start without significant performance improvement. Subsequently, the probabilities decrease, which could be explained by the model learning a suboptimal policy. Therefore, different results with different entropy weights are given in Appendix A.2.

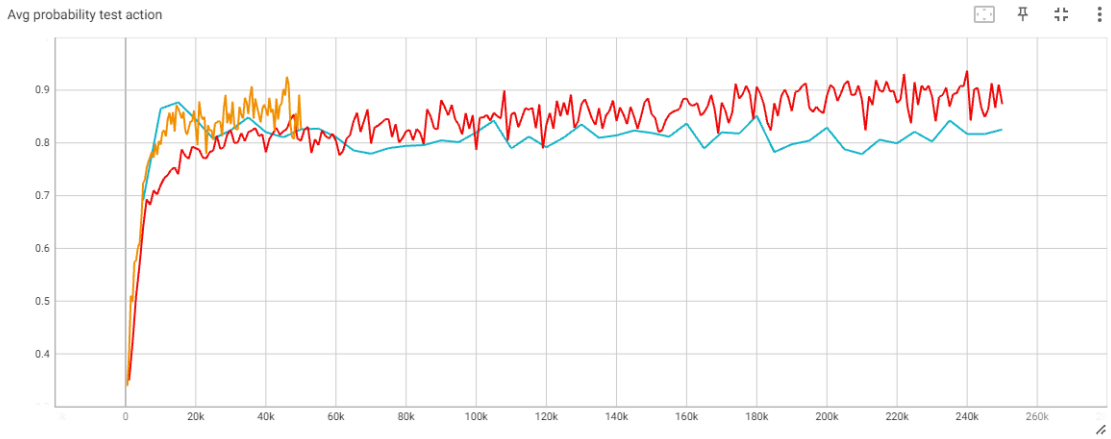

Figure 6.11: *Average probability of test actions taken plotted over the number of training episodes for models used for different maze sizes. Orange* 5×5*, red* 7×7*, light blue* 10×10*.*

The best-performing model has an entropy weight that starts at 0.1 and decays over 50,000 episodes. The solvability of a model trained with this entropy weight is

given in Figure 6.12. Athough the model's learning has not finished after the 250,000 training episodes, the model does not seem to learn as effectively as the the model used for 7×7 mazes. The change in entropy weight only has a small impact on the performance. Therefore, antoher strategy has to be used to increase solvability.

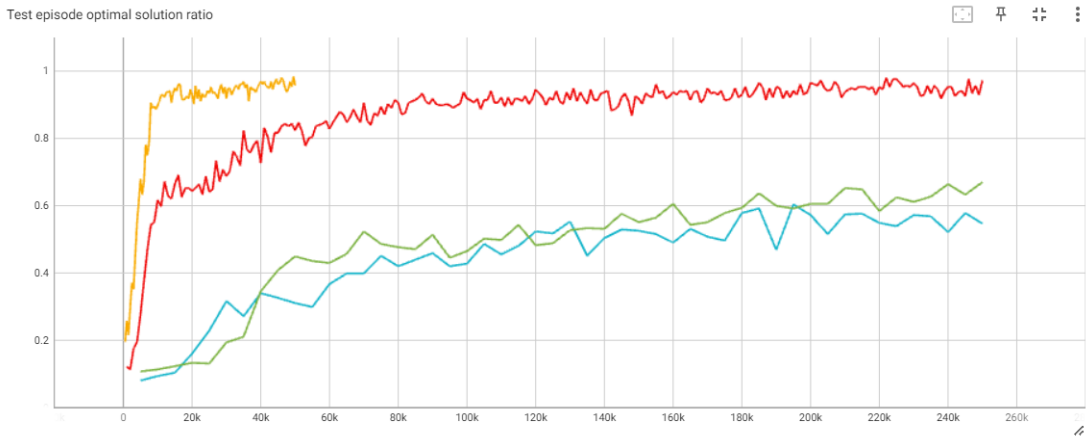

Figure 6.12: *Ratio of test instances that the model optimally solves the maze plotted over the number of training episodes for models used for different maze sizes. Orange* 5×5*, red* 7×7*, light blue* 10×10 *with entropy weight that decays over 5,000 episodes, light green* 10×10 *with entropy weight that decays over 50,000 episodes.*

6.5.2 Memory and batch size

The memory size and the batch size are increased in this section. The motivation for this increase is that an explanation for the model's performance is that the model cannot generalise. When increasing the memory size, more mazes will be used in each update phase, which helps generalise the model. Moreover, Hoffer, Hubara and Soudry (2017) changed the batch size to generalise their model. They argued that the batch size could have an impact on the convergence.

By increasing the memory size, more trajectories will be created before each update phase. In each update, more different mazes will be used and more updates will be performed. More diverse trajectories in the memory should generalise the policy updates. This increase does not affect the total number of updates performed. By increasing the batch size, more different steps will be used for each update. The number of updates in each update phase is defined by $K\frac{M}{B}$. Therefore, an increase in batch size B results in a decrease in the total total number of policy updates. Since one policy update uses more different steps, each policy update is more generalised. Results for different memory and batch sizes can be found in Appendix B. Based on these results, the memory size is increased from 4096 to 16,384 and the batch size from 64 to 256. Results of the model with these new memory sizes and batch sizes are given in Figure 6.13.

Figure 6.13 shows that the change in memory size and batch size increases the solvability of the algorithm. After 1 million training episodes, the algorithm is able to solve 0.938 of the mazes, and it solves 0.930 of the mazes optimally.

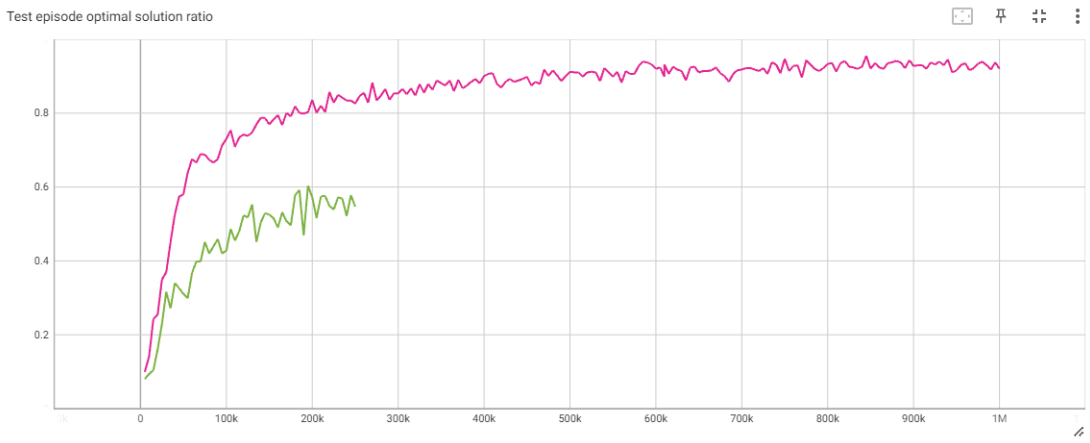

Figure 6.13: *Ratio of test instances that the model optimally solves the maze plotted over the number of training episodes. Green has* $M = 4096$ *and* $B = 64$*. Pink has* $M = 16,384$ *and* $B = 256$*.*

Another result that should be mentioned is that for 10×10 mazes, on average, 1 per cent are solved but not optimally. This value was either zero or close to zero for 5×5 mazes. There is a part of the structure of the state for which the agent does not learn to take the optimal actions. It is a logical result that this difference is notable for larger mazes because there are more possibilities to solve the maze. Chances are higher that there are solutions that the agent could take that are not optimal. The structures are more complex, and therefore, it is more likely that the model did not learn to recognize all structures. An example of 10×10 mazes is given in Figure 6.14. In the left figure, the agent solves the maze optimally. In the middle figure, the agent solves the maze but not optimally. It is unclear why the agent chooses to turn left at square 2. In the figure on the right, it is unclear why the model chooses to move from square 4 to 3 and not up. In conclusion, the model is able to solve a large share of the mazes, but it did not learn all the structures it could encounter.

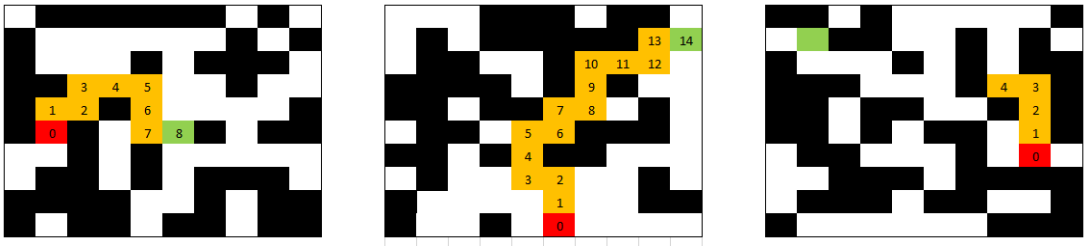

Figure 6.14: *Three test instances after training. Red is the start, green the finish and orange the path the agent takes. On the left, an instance that is solved optimally. In the middle, an instance that is solved but not optimally. On the right, an instance that cannot be solved.*

6.6 Summary results

Entropy regularization is necessary to the model to balance the exploration-exploitation process. With this addition, the algorithm is able to solve up to 98.4 per cent of

the 5×5 mazes. By introducing a second model, which learns solely based on the initially unsolvable mazes, the combined model is able to solve 99.7 per cent of the 5×5 mazes. Fine-tuning was needed for this second model. Interestingly, the second model is not strictly better than the initial model. Both models learn to solve different mazes. Therefore, combining both models increased solvability. The configuration used for the initial model of 5×5 mazes worked for 7×7 but not for 10×10 mazes. The solvability of the 10×10 mazes was increased by increasing the memory and batch sizes.

Chapter 7

TSP

The following environment is the Traveling Salesman Problem. As discussed in the literature review, a RL-based hyper-heuristic model will be used for this Problem.

The model's goal is to find the most efficient route that visits a set of cities exactly once and returns to the starting city. The objective is to minimize the total distance of the tour. The challenge is determining the optimal sequence in which the cities should be visited to achieve the shortest possible path.

One consideration is how to define the problem to be suitable for RL. A key challenge researchers are tackling is converting the TSP's graph coordinates into a state representation suitable for the RL models. One approach uses an encoder like struc2vec (Dai, Khalil, Zhang, Dilkina and Song, 2017). Other researchers implemented attention mechanisms to capture relationships between cities (Kool et al., 2019), (Bresson and Laurent, 2021). Before applying their model, Khachay and Neznakhina (2019) clustered the cities into sections.

The literature on the TSP is extensive, featuring diverse methodologies for the state definition. This research defines the TSP within the framework of a grid. The TSP is defined as a grid by drawing cities from a grid instead of drawing coordinates, where each grid square is a possible location for each city. The first reason for this choice is that the model developed for the maze environment successfully operates on a 10×10 grid. The model is able to extract information from the problem structured as a grid. Making the choice for a grid-like environment a logical foundation for further model development.

7.1 Environment setup

Cities are randomly drawn from a grid of size 10×10. Multiple cities may be drawn in the same square. State s is defined by a $10 \times 10 \times 3$ matrix. The 10×10 layers represent the current start of the tour, the current end of the tour and the number of cities yet to visit for each square in the grid. It is assumed that the agent walks in a straight line to the cities, so the Euclidean distance is used. A straightforward reward function in each step is $-d_{ij}$, where city j is added to the tour and connected to city i. Other reward functions could also be used for the 30 TSP. Wang et al., 2023 considered $-\frac{1}{d_{ij}}$ as reward in each step. They argue that this reward encourages the

exploration of cities further away. A downside of this reward is that the trajectory with the highest reward does not necessarily correspond to the optimal tour. In this research, the reward is $-d_{ij}$

7.2 Heuristics

Müller and Bonilha (2021) and Kallestad et al. (2023) used a hyper-heuristic to improve an initial solution. The goal of the model in this research is to generate a solution quickly after the model is trained. Therefore, only constructive heuristics will be used. The two heuristics which will be initially used are Nearest Neighbour and Maximum Regret.

Nearest Neighbour (NN) Connects the city that is closest to the end of the tour.

Maximum Regret (MR) Regret of a city is defined as the additional distance that a city must traverse if it is not selected as the next city. Assuming that the shortest edge is the one connecting to the current city, regret arises when this city is bypassed as the next destination. This is because not selecting the city precludes it from connecting to its shortest edge, necessitating a connection to a city that is farther away. The difference between the distance to the current city and the distance to the second-closest city characterizes this regret. Consequently, cities will exhibit regret only when their shortest edge corresponds to the one connecting to the current city. The Maximum Regret heuristic chooses the city that has the maximum regret. An example of MR is provided below.

Max Regret was inspired by Gutin, Goldengorin and Huang (2007), who researched the MR algorithm for the TSP. An example of MR is provided in figure 7.1. Blue is the only city with regret because the shortest edge from blue is to the green node with length 2. If the current city is excluded, the shortest edge is to [3,6], with length $\sqrt{5} = 2.236$. The regret of the blue node is 0.236, and the regret of the other nodes is 0. Therefore, MR chose the blue city.

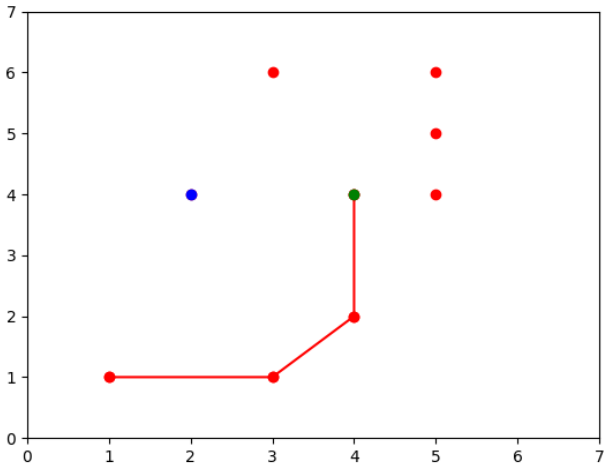

Figure 7.1: *Partial tour of a TSP example. Green is the current city and blue is the city which has positive regret.*

Because cities are drawn from a grid, it will occur that a heuristic has multiple optimal cities. In the context of NN, for example, it could occur that two cities are equally close to the current city. In these situations, the agent randomly chooses one of the options. For MR, it may occur that no city has positive regret, rather than selecting any city with zero regret, NN is applied.

7.3 Model architecture

The initial network architecture for the TSP is the same as the final model used for the 10×10 maze, which consists of 2 convolutional layers, each with a 3×3 kernel, 32 filters and same padding. The output is flattened and followed by a dense layer with 256 hidden nodes. All layers use the relu activation function. The dense layer is followed by a softmax output layer for the actor network and a linear output layer for the critic network. The entropy term is scheduled from 0.1 to 0 over 50,000 episodes.

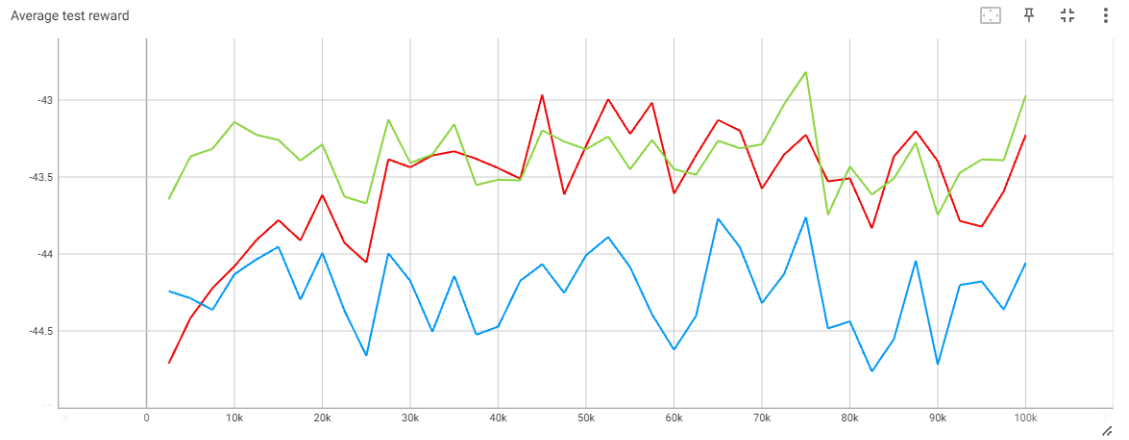

Figure 7.2: *Average test reward plotted over the number of training episodes. Red is the model's average test reward, blue the average NN reward and green the average MR reward.*

It can be observed from figure 7.2 that the agent is able to perform approximately as good as the MR heuristic. After 100,000 episodes, the agent chooses MR in 0.6 of the cases. Therefore, the agent does not converge to MR.

7.3.1 Pooling layer

A common practice for convolutional networks is to use pooling layers. A pooling layer divides the input layer into small sections and aggregates these sections. Pooling layers have two primary purposes. The first purpose is to reduce the number of trainable parameters, which lowers the number of computations and leads to a lower training time. The second purpose is to control overfitting by extracting only helpful information (Gholamalinezhad and Khosravi, 2020). Two main types of pooling layers are average pooling and max pooling. Average pooling takes the average value of each pool. Max pooling takes the maximum value of each pool.

LeCun et al. (1989) used average pooling in their network to read handwritten zip codes. They argued that the exact location of the features is irrelevant. Only their approximate position relative to other features is relevant. Li et al. (2019) used max pooling in their model, where robots need to find the shortest path while avoiding collisions. In this setup, the exact location of the features is important. It is important to use the exact positions of other robots and obstacles when the model should avoid those.

In the TSP environment, the exact location of the cities is important. The exact distances between cities must be used in the features. Therefore, max pooling will be used in the model. In the new model, a 2×2 pooling layer, no stride is added in between the two convolutional layers.

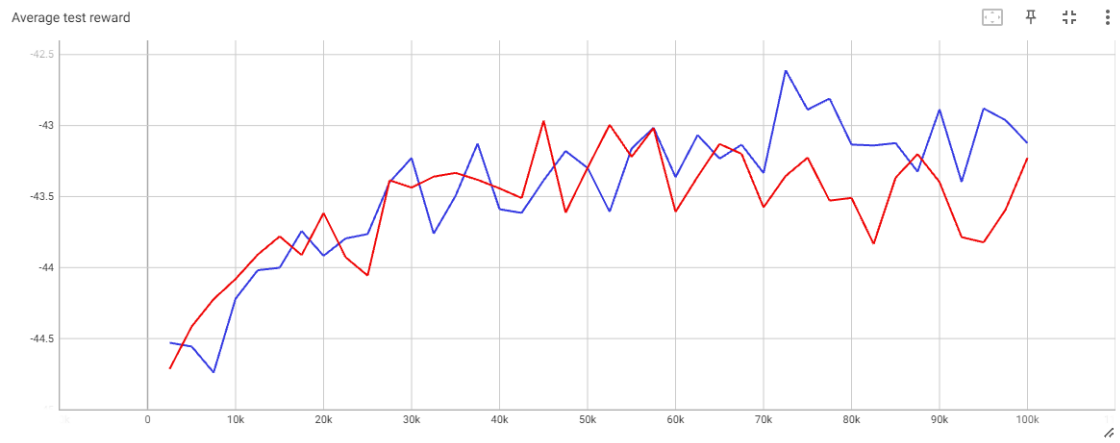

Figure 7.3: *Average test reward plotted over the number of training episodes. Red is the model without pooling, blue is the model with max pooling.*

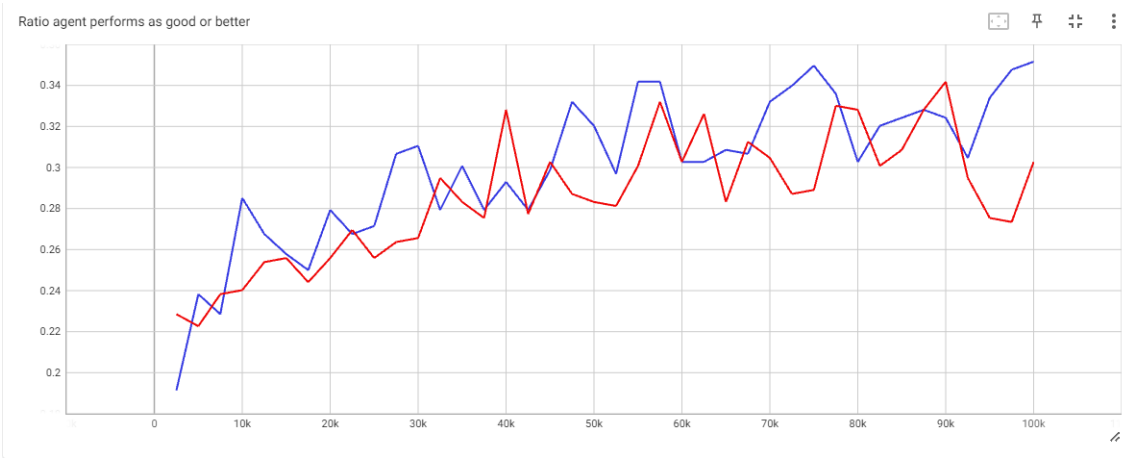

Figure 7.4: *Ratio that the model performs at least as good as all implemented heuristics plotted over the number of training episodes. Red is the model without pooling, blue is the model with max pooling.*

There is no clear difference in performance between the models. However, the training time of the model with a pooling layer is about 3 times faster than the model without pooling. Therefore, the pooling layer will be used in future TSP models.

7.4 Extended action space

The model does not learn to solve these instances because the action space is not well suited. The reason for believing this follows from Figure 7.5, where one instance is solved to the optimal reward achievable using NN and MR. In this figure, the model could potentially obtain a higher reward if the action space is extended. Therefore, two actions are added: NN2 and MR2, which are NN and MR applied to the current start of the tour instead of the end. The same instance is optimally solved with the extended action space, shown in Figure 7.6, to show that extending the action space

is beneficial. The optimal achievable reward is estimated by training the model on a single instance. This is an estimation since it cannot be guaranteed that the model learns to solve this instance to the potential optimum. Figure 7.5 shows the route for which the model achieved the highest reward by using NN and MR. Figure 7.6 shows the route for which the model achieved the highest reward using the extended action space. Both figures provide a step-by-step construction of the solution found. The title of the subplot is the heuristic chosen in that step. The reward is increased from -33.88 to -30.70 when including the new action. Therefore, adding NN2 and MR2 to the action space is a valuable improvement from these results.

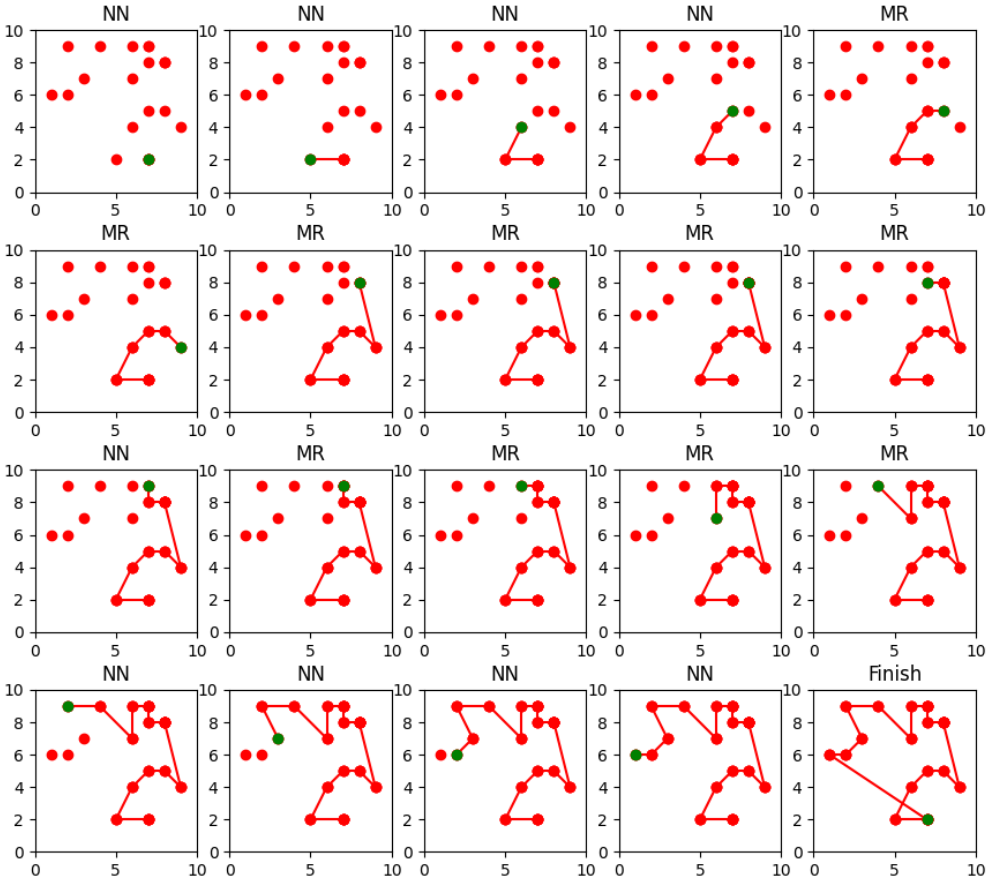

Figure 7.5: *Reward: -33.88. Step by step example plot from the model which is allowed to use NN and MR. The green dot is the current end of the tour.*

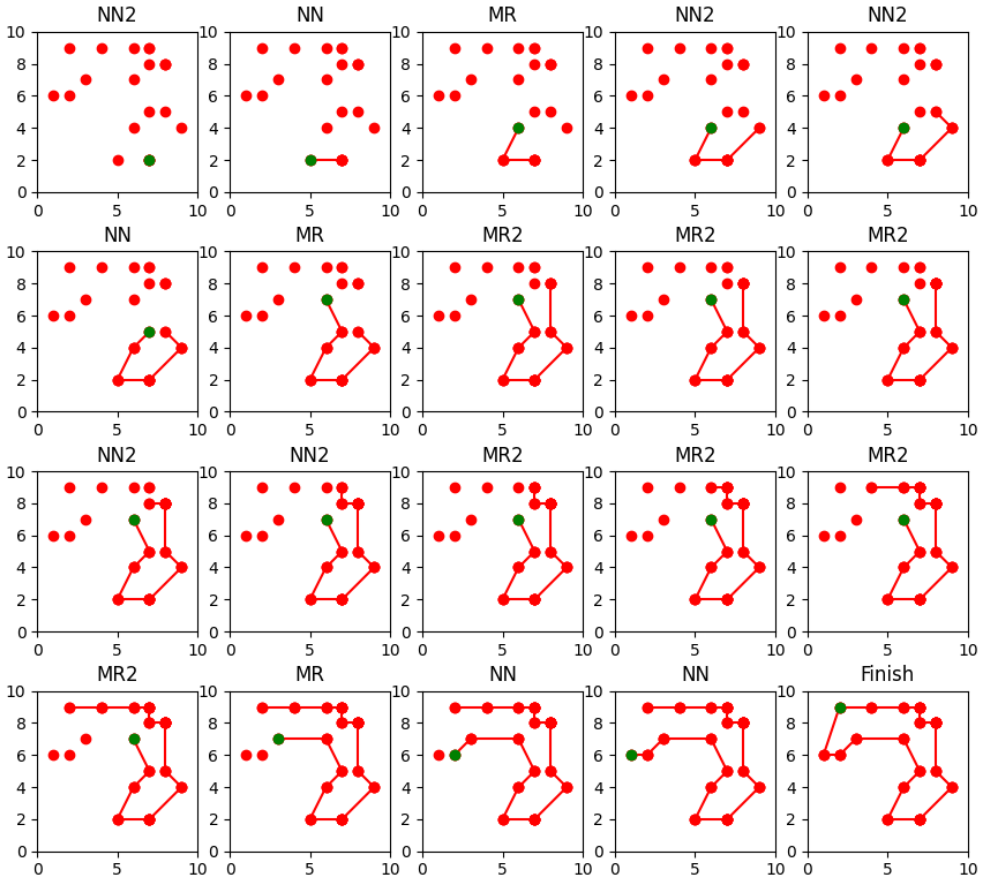

Figure 7.6: *Reward: -30.70. Step by step example plot from the model which is allowed additional actions NN2 and MR2. The green dot is the current end of the tour.*

It is shown that a model that uses the extended action space could achieve higher rewards than a model which only uses NN and MR. This extended action space is deployed to random TSP instances. Results are given in the plots below. Figure 7.7 shows the test reward, and Figure 7.8 shows the ratio that the model (strictly) outperforms the implemented heuristics.

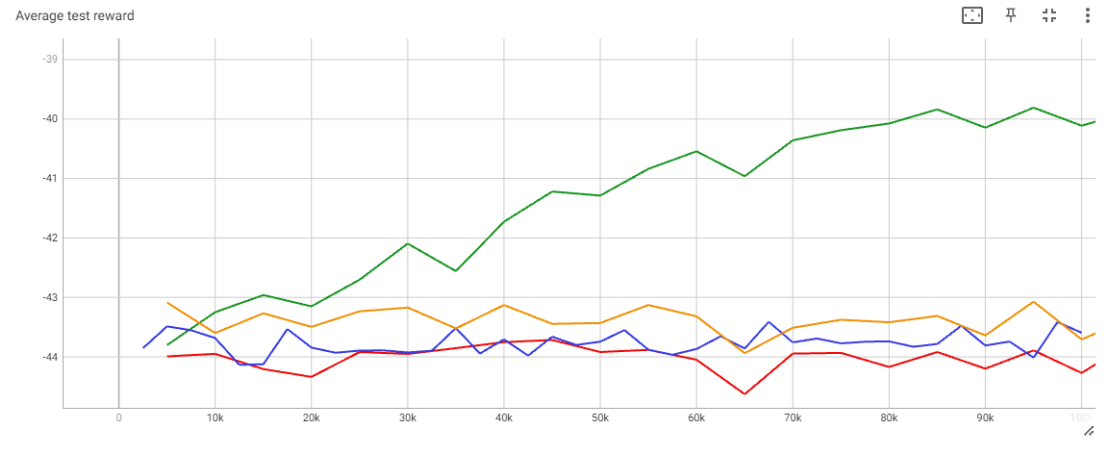

Figure 7.7: *Average test reward plotted over the number of training episodes. Green is the model with the additional actions. Blue is the model with two actions. Red and orange are average NN and MR test rewards respectively.*

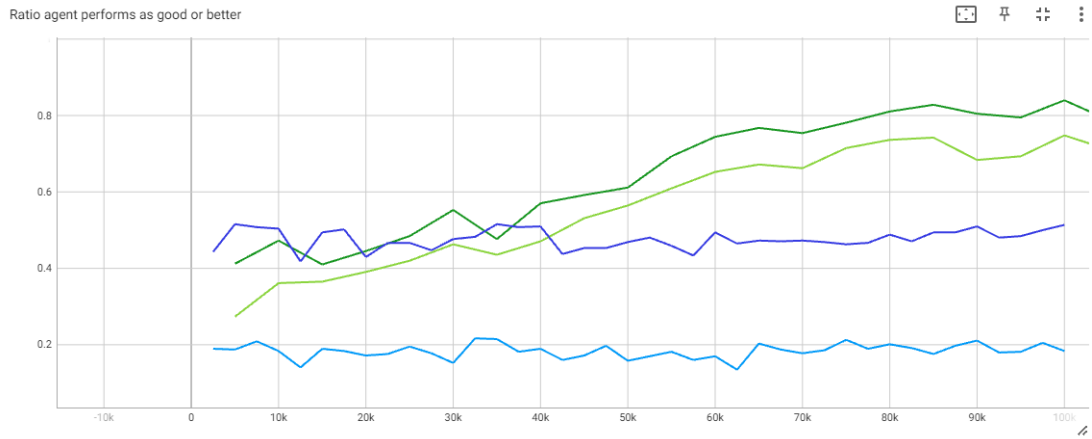

Figure 7.8: *Ratio that the agent performs better than all the implemented heuristics plotted over the number of training episodes. Blue lines represent the model with two actions and the green lines represent the model with additional actions. The dark lines represent the ratio that the model performs as good as all the heuristics used in the model. The light lines represent the ratio that the model performs strictly better as all the heuristics used in the model.*

It can be observed from Figure 7.7 that the test reward is better when the model is allowed to use the additional actions. Figure 7.8 shows the ratio that the algorithm (strictly) outperforms the heuristics used in that algorithm. The ratio that the model with additional actions outperforms the four heuristics is clearly higher than the ratio that the model with two actions outperforms its two heuristics. The

learning curves are still increasing after 100,000 training episodes. The model will be trained for more episodes and compared to the optimal solution in the next section.

As of the second step, the heuristics do not behave like the original heuristics. In the second step, NN2 and MR2 consider the initial start city as the viewing point for the next move, whereas NN and MR consider the added city as the viewing point for the second step. This could result in different average values between the original and the new heuristics. To test this, all four heuristics are applied to 10,000 instances. The average test rewards of the original and new heuristics are close to each other for both NN and MR, as shown in Table 7.1. The ratio that the reward of the original heuristic is greater than the reward of the new heuristic is approximately equal to the ratio the other way around, which implies that none of the two heuristics outperforms the other for more instances. This is the result for both NN and MR and is shown in Table 7.1. So, the newly added heuristics are similar to the initial heuristics. They behave almost the same as the initial heuristics and have the same average reward. On their own, the heuristics do not add any value. However, allowing the agent to add cities from the start of the tour significantly improves the overall reward.

Heuristic	Average reward
NN	-44.22
NN2	-44.25
MR	-43.35
MR2	-43.35

Table 7.1: *Average reward of 10,000 test instances for the different heuristics.*

Heuristic	original > new	original < new	original = new
NN	0.318	0.310	0.372
MR	0.339	0.343	0.318

Table 7.2: *Fraction of the best performing heuristics for both NN and MR. In the first row, NN is compared with NN2. Where the first colum is the ratio that the reward of NN ¿ reward of NN2. The second column the other way around and the last column where the rewards of NN and NN2 are equal. In the second row the same comparison is made for MR.*

7.4.1 Comparison to optimality

The algorithm will be trained longer in this section because the learning curve has continued at 100,000 training episodes. Moreover, this section will compare the algorithm to the optimal value. From the previous section, it is clear that the algorithm is learning, but it is not clear how well it performs compared to its potential. 500 TSP instances are generated and solved optimally using Google OR-Tools (Furnon and Perron, 2024). This Python package can be used to solve combinatorial optimization problems, such as the TSP. The trained algorithm tests these instances and each instance is tested 128 times. Each instance is tested more than once because

an action might have multiple options, resulting in different rewards for the same instance. Testing multiple times generalizes the reward the algorithm is able to obtain.

The average test rewards of the algorithm trained for 500,000 training episodes are given in Figure 7.9. This figure also contains the average rewards of NN and MR. The black line in this figure is the average optimal reward of the 500 generated instances. For all test solutions, the optimality gap is calculated.

Three different measurements are used for the gap. 1: Average minimum gap. For each instance, the minimum gap is used. These minima are averaged. 2: Average gap. 3: Average maximum gap. Results are given in Table 7.3.

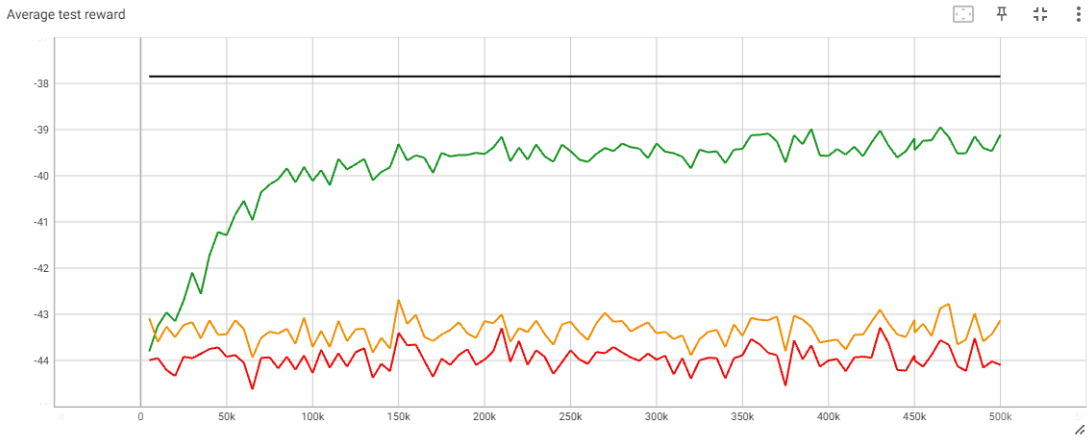

Figure 7.9: *Average test reward plotted over the number of test episodes. Red is the model's reward, green is NN reward and blue is MR reward. The black line is the average optimal reward found for the 500 generated instances.*

Measurement	Gap
1. Average minimum gap	0.0242
2. Average gap	0.0405
3. Average maximum gap	0.0622

Table 7.3: *Optimality gap with respect to the true optimal value for 500 instances.*

On average, the optimality gap is 0.04 with the true optimal solution. The difference between different gaps in Table 7.3 is explained by the fact that, in some cases, the action has multiple options. The average minimum gap estimates the optimality gap obtained when the agent chooses the optimal option for each action with multiple options. Therefore, if the algorithm performs at its best, there is still an optimality gap of 0.02. However, the possibility exists that the true optimal solution cannot be reached. Figure 7.10 shows a small example in which the algorithm cannot optimally solve the instance with the four implemented heuristics. Figure 7.11 shows the explicit step where the algorithm fails to generate the optimal solution. At this point, none of the actions can lead to the true optimal solution. Therefore, the model could be improved by further extending the action space.

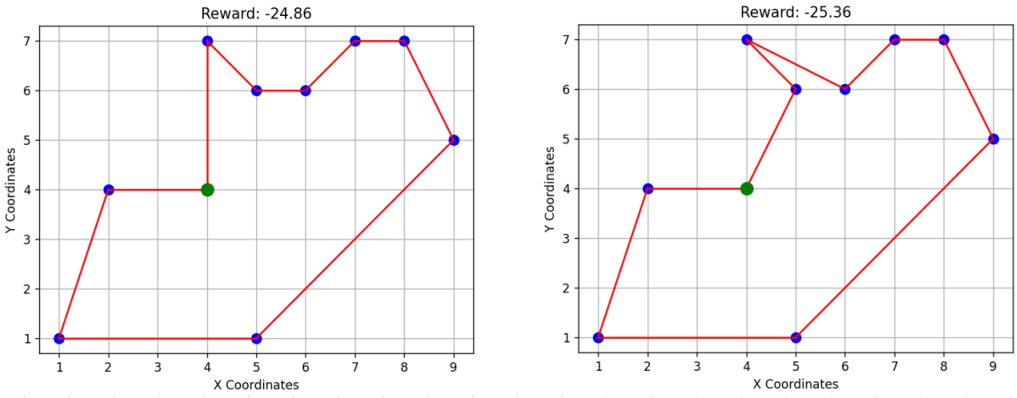

Figure 7.10: *Small exmaple for which the algorithm cannot find the optimal solution with use of the four implemented heuristics. On the left is the optimal solution found with use of Google OR-Tools. On the right is the best achievable solution found by the model. The green node is the start city.*

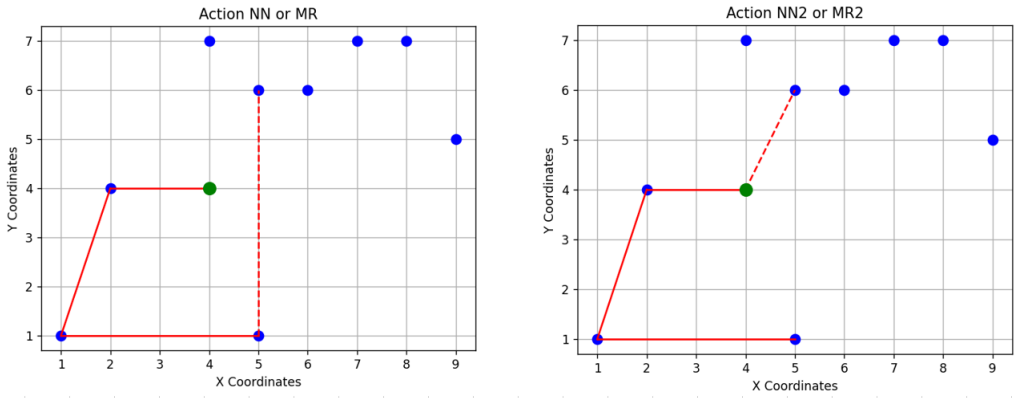

Figure 7.11: *Partial tour of the example where the model cannot solve to optimality. The solid line is the current partial tour. On the left, the dashed line is the added line if either NN or MR is chosen in the next step. On the right, the dashed line is the added line if either NN2 or MR2 is chosen in the next step.*

7.4.2 Action distribution

The action distribution of test episodes after for the algorithm trained for 500,000 training episodes are given in table 7.4. An interesting result is that all heuristics are used in the final model. This implies that the model learns to effectively combine the heuristics in order to construct a better solution.

Heuristic	Fraction
NN	0.176
NN2	0.304
MR	0.271
MR2	0.249

Table 7.4: *Action distribution of test episodes after 500,000 training episodes.*

7.5 Summary results

The TSP is tackled using an RL-based hyper-heuristic. Pooling layers are added, which reduces the computation time. Initially, NN and MR were implemented. After NN2 and MR2 were added to the action space, the algorithm's performance significantly increased. The model reaches an optimality gap of 0.04, where in the final model, all four heuristics are used, indicating that the model effectively learns to combine the heuristics. However, it is shown by an example that these heuristics cannot guaranty that the optimal solution can be reaches. Therefore, it is likely that an further extension of the actions space improves performance.

Chapter 8

Results

This chapter will discuss the results. In Chapter 9, the methods and modelling choices will be discussed together with their limitations. In Chapter 10, the results will be linked to the original problem of improving the existing PPO algorithms using the test environments. Furthermore, recommendations for future research will be given.

8.1 Results

The first environment was the Maze, where the first challenge encountered was an imbalanced exploration-exploitation process. The model converged too quickly to a local optimum: walking back and forth in the maze without reaching its goal. This imbalanced exploration-exploitation process is solved in literature by introducing an entropy regularization term (Haarnoja et al., 2018). The entropy regularization term penalizes a policy that results in a converging probability. This entropy regularization term was included in the actor loss function. It was scheduled toward zero, which encourages exploration in the early training but does not withhold the model from exploiting after the entropy term reaches zero. After fine-tuning the newly introduced entropy weight hyperparameter, this entropy term helped the model explore at the start while maintaining its ability to converge. This fine-tuned entropy term for the 5×5 maze increased the model's solvability from 0.3 to 0.96 for 5×5 mazes after 50,000 training episodes. The entropy regularization term is a valuable addition to the model. It helps balance the exploration-exploitation process.

Continuing on the 5×5 mazes, the next challenge encountered was that the model struggled to solve the last 4 per cent of the mazes. The model is able to learn the first 96 per cent, but learning to solve the last 4 per cent is difficult. After this, it takes many new training episodes to increase solvability slightly. This problem is solved by introducing a second model which learns solely from the instances the initial model cannot solve. The idea behind this second model is that the second model will learn the characteristics of these complex mazes, allowing the model to solve the mazes that the initial model cannot solve. Learning will be more efficient since the second model focuses solely on the problematic mazes. After training both models, new instances are passed in the initial model. If the initial model cannot solve the problem optimally, the problem is passed into the second model. Combining the use of two models increased the overall solvability to 99.7 per cent. This was

compared to the initial model, which was trained longer and reached a 98.4 per cent solvability. This indicates that combining both models increases overall solvability. Besides the increase in solvability, it was found that the second model did not perform strictly better than the initial model. Indicating that the second model learns characteristics other than those of the initial model. Therefore, the most important conclusion that can be drawn from these results is that both models learn different characteristics of the maze. Combining both models enhances the total solvability that the initial model could not achieve.

The configuration of the initial 5×5 model is used for larger mazes. This model was able to solve 96 per cent of the 7×7 mazes. For the model used to solve 10×10 mazes, the memory size and the batch size were increased. The change of these hyperparameters led to a significant increase in performance. The trained algorithm was able to solve 93.8 per cent of the mazes. A noteworthy result is that for the largest mazes, not all mazes were solved in the minimal number of steps possible. The model is able to solve 93.0 per cent of the mazes optimally. This difference was not visible for the smaller mazes, which could be explained by the fact that the larger mazes have more possible solutions.

The second test environment involved the Traveling Salesman Problem. For this problem, a hyper-heuristic is trained by PPO. Initially, the available heuristics were the nearest neighbour and maximum regret with respect to the end of the tour. After allowing the model to also use these two heuristics with respect to the start of the tour, the model's performance significantly increased. The new model performed at least as well as each of the four heuristics in approximately 90 per cent of the instances. 500 test instances were solved to optimality using Google OR-Tools. These instances were tested using the trained model. The trained model reached an average optimality gap of 0.04. Furthermore, one notable observation from the trained model is that it uses all four heuristics. Indicating that all heuristics contribute to the final solution. In conclusion, the hyper-heuristic effectively learned to combine the proposed heuristics, resulting in a significantly higher average reward compared to individual heuristics. Moreover, incorporating pooling layers led to a three-fold reduction in total computation time.

Chapter 9

Discussion and limitations

This chapter will discuss different modelling choices and aspects of the research, together with the limitations that arise from these methods. It starts by discussing the general points of this research. After these are discussed, points specifically to the methods implemented in the environments will be discussed.

9.1 General discussion

The first topic of discussion is the hyperparameter choices. In this research, most hyperparameter values remain untouched from the start. The values chosen were mainly based on two papers (Schulman et al., 2017), (Andrychowicz et al., 2021). Without further research, these values were used for all models. It is unclear if the model could have performed better by choosing other values. Andrychowicz et al. (2021) recommended initial hyperparameter values but argued that these recommendations are not necessarily the best choices. Because of the limited time for this thesis, the initially defined hyperparameters are not optimized for the environment. Nevertheless, it has to be kept in mind that a more extensive optimization of hyperparameters could lead to faster convergence and better model performance. However, faster learning does not necessarily imply that the model will reach the highest performance.

The second point of discussion is the training time for the models. In this research, the models did not have any predetermined stopping criteria for training, they were trained until the training stagnated. This choice is made since there was no required performance the model had to reach, the aim of this thesis was to improve the model. Therefore, the main objective for the methods was to compare the methods and configurations deployed.

9.2 Maze discussion

In the maze environment, a second model was introduced. In the research, the models were compared based on the amount of training. In this research, training time was expressed number of training episodes. However, comparing based on the

number of training episodes is arguable because this does not represent the number of network updates. The total number of network updates is more when the solvability of the model is lower. If the model cannot solve a training episode, the memory is filled with 50 steps. If the model solves the training episode, the memory is filled with less steps. Since network updates are made when the memory reaches a specific size, the model with lower solvability updates its networks more often. From the results of that section, it can be observed that the second model learns slower. Therefore, the second model generates more steps and updates its network more often than the initial model in the same number of episodes. Nevertheless, since the initial model was trained for more episodes and the solvability of the initial model was not able to surpass 98.4, the conclusions drawn that the combination of the two models increases the solvability is reliable.

There is a limitation with using two models as implemented in this research. The resulting return is quickly available in the maze environment after the model is chosen. It was possible to run both models on the forehand and choose whichever model yielded the highest return. In practice, this cannot always be done because it is not quickly known what the return will be for the chosen model. In the context of the CSP, when considering the stacking of the containers from one ship as a single instance, it is impossible to run both models and choose the one with the best results. This is because each container's action must be chosen when the container is taken off the ship. If the action is chosen, the container must move and be stacked there immediately. It takes some hours to unload the entire ship. Next to that, the state transition function in a container terminal is stochastic. The accomplished return will be known after hours after all actions are taken. However, the containers are already stacked. Therefore, the model must be chosen before a ship's unloading starts. One could argue that simulations can be run before the ship arrives. Where the two models are simulated, and the best model is chosen. However, the same issue arises in this case when using a metaheuristic. Which is that there is so much uncertainty in the terminal, that the latest information is necessary to choose an action. This makes the simulation not an option. In conclusion, implementing multiple models that learn to solve different instances is able to improve overall solution quality. It is a valuable method for improving the models. It is proven in the maze environment that combining both models results in higher solvability. The two models are able to learn distinct characteristics of the same problem. However, before it can be used in practice, extra thought has to be made to determine which model to use for which instance. An idea to accomplish this is to introduce a third model specifically trained to choose between the two models.

9.3 TSP discussion

The state definition for the TSP is defined as a grid such that the model used for the mazes could be extended to the TSP. Subsequently, the modelling choice was made to draw cities from the grid and consider these grid points as true points. The downside of this choice was that it occured that actions had multiple options, and in these situations, one of these options was chosen randomly. This led to the fact that optimal strategy could result in different solutions. This problem was visible

in the optimality gaps, where the average optimality gap ranged from 0.02 to 0.06, depending on the specific actions taken. Another option was to draw cities as floating points and only use the grid definition as state representation.

Another point of discussion is that the model's performance is only compared to the true optimum, neglecting the optimum the model can achieve by combining solely its heuristics. The absence of knowledge regarding the optimum solution that the model can achieve with these heuristics poses a challenge in understanding the optimality gap in this research. From this research, it can be concluded that the model learns, but not if the model learns its optimum. As discussed in the literature review, the possibility exists that this method could never reach the true optimum since it operates on the search space of heuristics instead of the search space of solutions (Bello et al., 2016). However, comparing the algorithm with the optimal value gives information about why the model does not reach the true optimal value. In case the optimum possible by using the heuristics is close to the true optimum, the algorithm's training should be improved. If the optimum possible by using the heuristics is close to the achieved reward by the algorithm, it suggests that the implemented heuristics are insufficient. In that case, more heuristics should be implemented. Because this comparison is not made, it is impossible to explain why the model does not reach the optimum value. In a small example is shown that the true optimum is not reachable by using the four implemented heurisitcs. However, its is unknown if this explains the entire gap.

Chapter 10

Recommendations and future research

This research aimed to improve the existing PPO model to make it more suitable for different combinatorial optimization problems. The motivation behind the development of this model is the challenges that arise in a container terminal and a warehouse or distribution centre. The higher goal is that the model could eventually be used to optimize container terminals and warehouses. Multiple test environments are built on which the model is developed. These test environments' purpose was that they were smaller and more straightforward problems. The model could be monitored and developed more easily on these environments. Another advantage is that aspects of these environments are related to the container terminal and warehouse. Furthermore, the model used for these environments could form a basis for future optimization problems. Before the start of this thesis, the implemented model was not able to solve the environments. The techniques implemented in this research allowed the model to solve almost all mazes up to size 10×10. For the TSP holds that the model is able to effectively learn to combine the implemented heuristics to significantly improve the solution's reward compared to that of the individual heuristics. However, it is shown that the TSP's action space should be further extended. This could be done with other constructive heuristics, or it could be helpful to implement insertion heuristics, where a city is not necessarily added to the tour's start or end. Due to the limited time for this thesis, this was not further investigated, but it is a good next step for increasing the TSP model's performance.

The introduction mentioned the different test environments used before implementing the model at the container terminal. These environments were the Maze and the TSP. Future steps could be the VRP and the CSP. The order of these environments is chosen based on their expected difficulty. The purpose of these environments is that the model should be able to solve all these problems since the container terminal is more complex than these individual environments. It is not necessary for one model configuration to be able to solve all these instances, but there should be a model that can solve these problems individually. Furthermore, another purpose of these environments is that they could be used as a starting model for new problems.

The environments could be used as building blocks for the container terminal

and warehouse. The transportation of containers from the ship to the stacking location by an AGV can be related to the Maze. The TSP can be used as a starting point in a warehouse, such as order picking. Looking ahead to future environments, the VRP can be related to the AGV routes. The VRP can be used to determine which AGV will be used for each container. It is efficient if the AGV travels as few as possible distance and will not travel without a container. Furthermore, the VRP model could be used for warehousing problems such as order picking, which is the VRP in essence. The TSP is a specific case of the VRP, with one truck and unlimited capacity. Therefore, the TSP model can be used as a basis for the VRP. Furthermore, the CSP will logically link to the stacking problem in a container terminal. In conclusion, the environments can be used as building blocks for optimising a container terminal. With the implemented methods and RL techniques, this research improved the models used for these building blocks. Besides using the building blocks for the container terminal, these environments could be used as a basis for new optimisation problems that ICT Group may want to solve in the future.

Besides the possibilities of using these environments in practice, the techniques used in this research provide a valuable foundation for future research. Introducing a second model helped the model solve instances that could not be solved initially. Combining these two models resulted in a solvability rate that the initial model was unable to achieve. However, the implementation of the second model, as in this research, has its limitations. Therefore, in order to effectively combine both models, some more research has to be done on the choice of the model.

A hyper-heuristic approach allows the model to use expert knowledge to build a solution. In context with the container terminal, currently, a common practice is to use a decision-based algorithm based on expert knowledge. A hyper-heuristic trained by PPO could be a promising addition because the knowledge currently used will not be lost. Moreover, the limitation that a hyper-heuristic cannot guarantee that the optimal solution could be obtained may not be any problem in practice since the aim is to generate better solutions than the current solution and not necessarily build the optimal solution.

Taking all aspects into account, this thesis marked a step forward in the development of a Proximal Policy Optimization algorithm. Besides the improvements it made for solving the Maze and the TSP, using a second model and a hyper-heuristic approach gives valuable insight for future research.

Bibliography

Aiswarya, G., Mariah, M., Katragadda, R., & Makam, R. (2023). Control of Self-Driving Cars using Reinforcement Learning. *2023 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. https://doi.org/10.1109/conecct57959.2023.10234763

Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., & Bachem, O. (2021). What matters in On-Policy Reinforcement Learning? A Large-Scale Empirical Study. *HAL (Le Centre pour la Communication Scientifique Directe)*. https://hal.inria.fr/hal-03162554

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural Combinatorial Optimization with Reinforcement Learning. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1611.09940

Bresson, X., & Laurent, T. (2021). The transformer network for the traveling salesman problem. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.2103.03012

Chen, Y., Shih, W.-Y., Lai, H.-C., Chang, H.-C., & Huang, J.-H. (2023). Pairs Trading Strategy Optimization Using Proximal Policy Optimization Algorithms. *2023 IEEE International Conference on Big Data and Smart Computing (BigComp)BIGCOMP*. https://doi.org/10.1109/bigcomp57234.2023.00015

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., & Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *arXiv (Cornell University)*, *30*, 6351–6361. https://doi.org/10.48550/arXiv.1704.01665

D'Eramo, C., & Chalvatzaki, G. (2022). Prioritized Sampling with Intrinsic Motivation in Multi-Task Reinforcement Learning. *2022 International Joint Conference on Neural Networks (IJCNN)*. https://doi.org/10.1109/ijcnn55064.2022.9892973

Dong, H., Zhang, D., Shi, Q., & Li, K. (2022). Entropy regularized actor-critic based multi-agent deep reinforcement learning for stochastic games. *Information Sciences*, *617*, 17–40. https://doi.org/10.1016/j.ins.2022.10.022

Euchi, J., Moussi, R., Ndiaye, F., & Yassine, A. (2016). Ant colony optimization for solving the container stacking problem. *International Journal of Applied Logistics*, *6*(2), 81–101. https://doi.org/10.4018/ijal.2016070104

Furnon, V., & Perron, L. (2024, March 7). *Or-tools routing library* (Version v9.9). https://developers.google.com/optimization/routing/

Gholamalinezhad, H., & Khosravi, H. (2020). Pooling methods in deep neural networks, a review. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.2009.07485

Gunawardhana, J. A., Perera, H. N., & Thibbotuwawa, A. (2021). Rule-based dynamic container stacking to optimize yard operations at port terminals. *Mar-*

itime Transport Research, *2*, 100034. https://doi.org/10.1016/j.martra.2021.100034

Gutin, G., Goldengorin, B., & Huang, J. (2007). Worst case analysis of Max-Regret, Greedy and other heuristics for Multidimensional Assignment and Traveling Salesman Problems. *Journal of Heuristics*, *14*(2), 169–181. https://doi.org/10.1007/s10732-007-9033-3

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1801.01290

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *Advances in Neural Information Processing Systems 30 (NIPS 2017)*.

Hou, Z., Lee, T., & Keidar, M. (2022). Reinforcement learning with safe exploration for adaptive plasma cancer treatment. *2023 IEEE transactions on radiation and plasma medical sciences*, *6*(4), 482–492. https://doi.org/10.1109/trpms.2021.3094874

Jin, X., Duan, Z., Song, W., & Li, Q. (2023). Container stacking optimization based on Deep Reinforcement Learning. *Engineering Applications of Artificial Intelligence*, *123*, 106508. https://doi.org/10.1016/j.engappai.2023.106508

Kallestad, J. V., Hasibi, R., Hemmati, A., & Sörensen, K. (2023). A general deep reinforcement learning hyperheuristic framework for solving combinatorial optimization problems. *European Journal of Operational Research*, *309*(1), 446–468. https://doi.org/10.1016/j.ejor.2023.01.017

Kendall, G., & Li, J. (2012). Competitive travelling salesmen problem:A hyper-heuristic approach. *Journal of the Operational Research Society(2013)*, 208–216. https://doi.org/10.1057/jors.2012.37

Khachay, M., & Neznakhina, K. (2019). Complexity and approximability of the Euclidean generalized traveling salesman problem in grid clusters. *Annals of Mathematics and Artificial Intelligence*, *88*(1-3), 53–69. https://doi.org/10.1007/s10472-019-09626-w

Kool, W., Van Hoof, H., & Welling, M. (2019). Attention, learn to solve routing problems! *International Conference on Learning Representations*. https://openreview.net/pdf?id=ByxBFsRqYm

Kozlica, R., Wegenkittl, S., & Hirländer, S. (2023). Deep Q-Learning versus Proximal Policy Optimization: Performance Comparison in a Material Sorting Task. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.2306.01451

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R., Hubbard, W., & Jackel, L. D. (1989). Handwritten Digit Recognition with a Back-Propagation Network. *neural information processing systems*, *2*, 396–404. http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.pdf

Li, Q., Gama, F., Ribeiro, A., & Prorok, A. (2019). Graph Neural Networks for Decentralized Multi-Robot Path Planning. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1912.06095

Liang, X., Ma, Y., Feng, Y., & Liu, Z. (2021). PTR-PPO: Proximal Policy Optimization with Prioritized Trajectory Replay. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2112.03798

Liu, K., Quan, W., Cheng, N., Xu, Z., Deng, J., & Gao, D. (2022). PPO-based reliable concurrent transmission control for telemedicine real-time services. *ICC 2022 - IEEE International Conference on Communications*. https://doi.org/10.1109/icc45855.2022.9839084

Monaci, M., Agasucci, V., & Grani, G. (2021). An actor-critic algorithm with policy gradients to solve the job shop scheduling problem using deep double recurrent agents. *arXiv (Cornell University)*. https://doi.org/10.1016/j.ejor.2023.07.037

Müller, F. M., & Bonilha, I. (2021). Hyper-Heuristic based on ACO and local search for dynamic optimization problems. *Algorithms*, *15*(1), 9. https://doi.org/10.3390/a15010009

Narasimhan, K., Kulkarni, T., & Barzilay, R. (2015). Language understanding for textbased games using deep reinforcement learning. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Nazari, M., Oroojlooy, A., Takáč, M., & Snyder, L. (2018). Reinforcement learning for solving the vehicle routing problem. *Neural Information Processing Systems*, *31*, 9861–9871. https://papers.nips.cc/paper/8190-reinforcement-learning-for-solving-the-vehicle-routing-problem.pdf

OpenAI. (2019). Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1912.06680

Osmankovic & Konjicija. (2011). Implementation of Q — Learning algorithm for solving maze problem. *2011 Proceedings of the 34th International Convention MIPRO*, 1619–1622. http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.ieee-000005967320

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T. P., & Silver, D. (2020). Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609. https://doi.org/10.1038/s41586-020-03051-4

Schulman, J., Chen, X., & Abbeel, P. (2017). Equivalence between policy gradients and soft Q-Learning. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1704.06440

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015). Trust Region Policy optimization. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1502.05477

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., & Abbeel, P. (2015). High-Dimensional continuous control using generalized advantage estimation. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1506.02438

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy optimization Algorithms. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1707.06347

Sharma, A., Kaur, H., & Prashar, D. (2023). Maze Solver: A Federated Reinforcement Learning Approach. *14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. https://doi.org/10.1109/icccnt56998.2023.10307671

Su, K., & Lu, Z. (2021). Divergence-Regularized Multi-Agent Actor-Critic. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2110.00304

Trott, A., Zheng, S., Xiong, C., & Socher, R. (2019). Keeping your distance: Solving sparse reward tasks using Self-Balancing shaped rewards. *arXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1911.01417

Wang, J., Xiao, C., Wang, S., & Ruan, Y. (2023). Reinforcement learning for the traveling salesman problem: Performance comparison of three algorithms. *The Journal of Engineering*, *2023*(9). https://doi.org/10.1049/tje2.12303

Appendices

Appendix A

Maze entropy weight

A.1 2 models 5×5

In this appendix, diffent entropy weights are tested for the second model used for the 5×5 mazes. In Figure A.1, the green line suffers from poor exploration at the start of training. Different entropy weights are tested, results are given below. An etnropy weight that starts at 0.1 and decays over 10,000 episodes yielded the best results.

The following entropy weights are used: Green starts at 0.1 and dacays over 5,000 episodes. Pink starts at 0.1 and decays over 10,000 episodes. Dark blue starts at 0.2 and decays over 10,000 episoded.

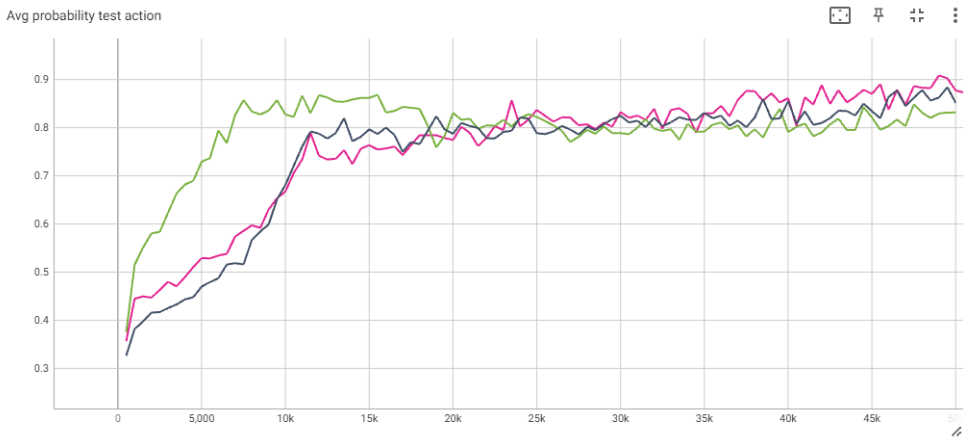

Figure A.1: *Average probability of test action taken plotted over the number of test episodes for different entropy terms.*

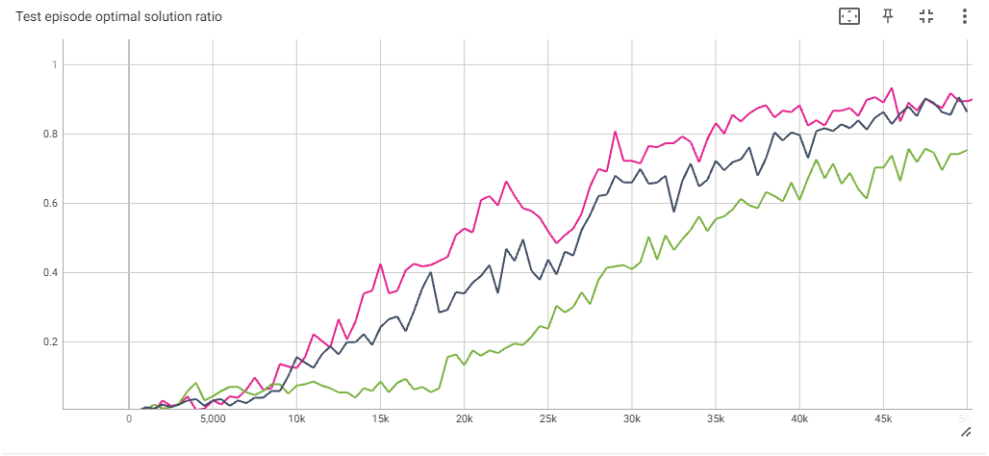

Figure A.2: *Ratio of test instances that the model optimally solves the maze plotted over the number of test episodes for different entropy terms.*

A.2 10×10 mazes

In this appendix, diffent entropy weights are tested for the model used for 10×10 mazes. An entropy term that starts at 0.1 and decays over 50,000 episodes yielded the best results.

The following entropy weights are used: Black starts at 0.1 and dacays over 5,000 episodes. Green starts at 0.1 and decays over 20,000 episodes. Blue starts at 0.1 and decays over 50,000 episodes. Red starts at 0.1 and decays over 100,000 episodes.

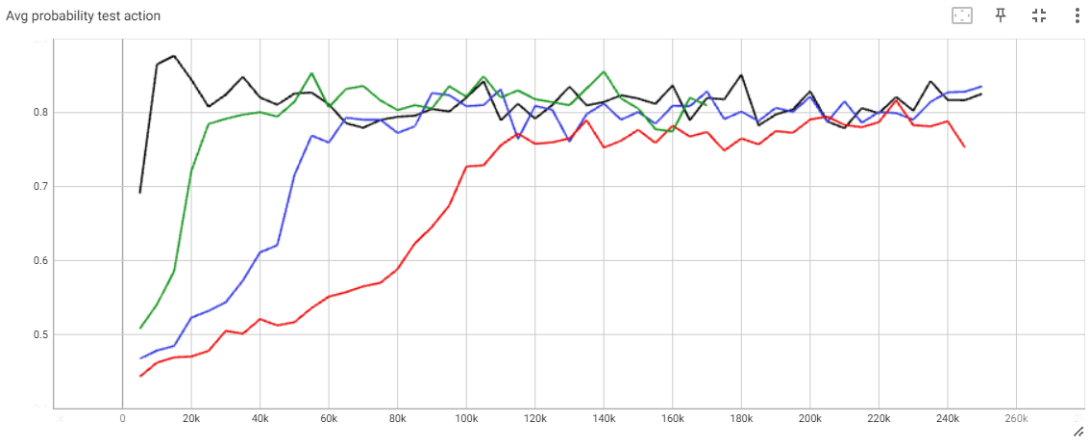

Figure A.3: *Average probability of test action taken plotted over the number of test episodes for different entropy terms.*

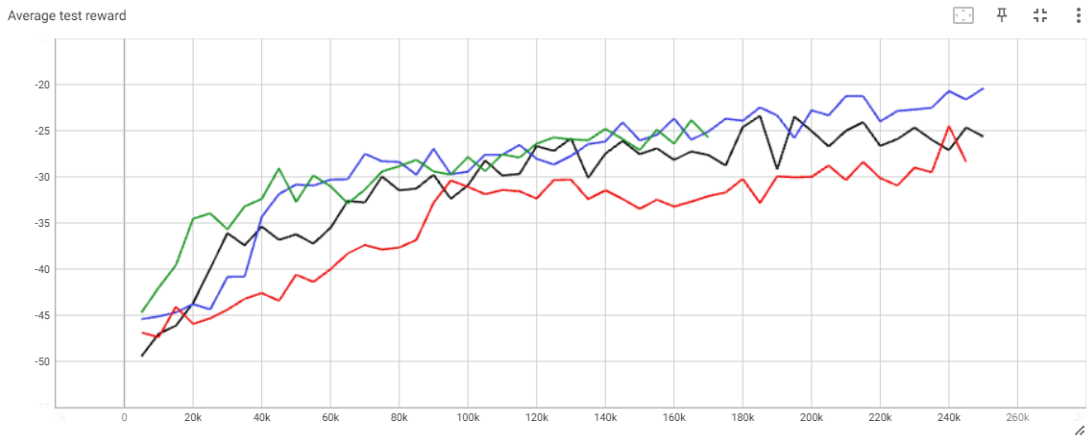

Figure A.4: *Average test reward plotted over the number of test episodes for different entropy terms.*

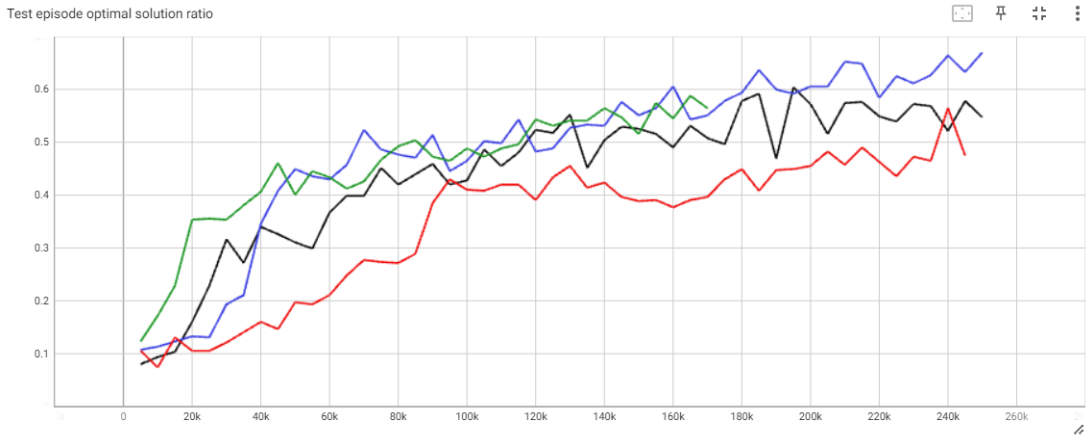

Figure A.5: *Ratio of test instances that the model optimally solves the maze plotted over the number of test episodes for different entropy terms.*

Appendix B

Maze memory size and batch size

In this appendix, different values for the memory size M and the batch size B are implemented for the algorihtm trained on 10×10 mazes. From these results, there will be continued with a memory size of 16,384 and a batch size of 256.

Line color	M	B
Orange	4096	64
Pink	16,384	64
Blue	16,384	128
Green	16,384	256

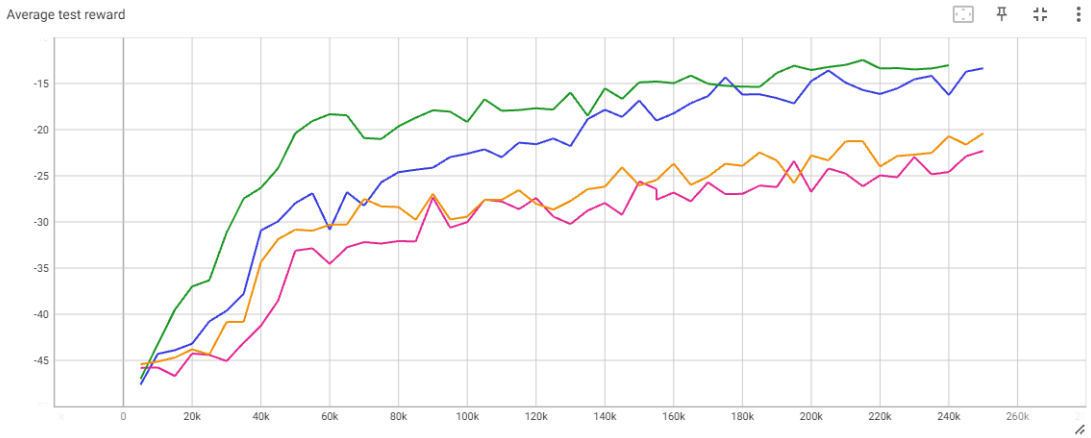

Figure B.1: *Ratio of test instances that the model optimally solves the maze plotted over the number of training episodes for different values for M and B. The value for M and B for the different line colors is stated in the table above.*

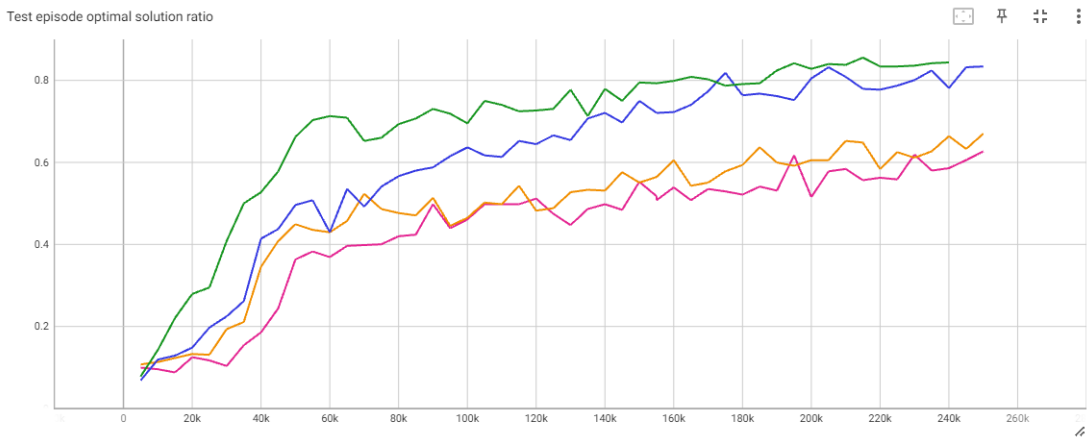

Figure B.2: *Ratio of test instances that the model optimally solves the maze plotted over the number of training episodes for different values for M and B. The value for M and B for the different line colors is stated in the table above.*