University of Southern Denmark

Start Date: February 3rd 2025

Due Date: June 2nd 2025

Bachelor project: "Drone Swarm Control"

Submitted by:

Kasper B. Mathiesen
kmath22@student.sdu.dk

Søren V. Væde
sovae22@student.sdu.dk

Supervised by:

Anders Christensen
Kasper Andreas Rømer Grøntved

Contents

1 Abstract

The deployment of autonomous drones in search-and-rescue operations represents a transformative advancement in disaster response, offering substantial benefits in scenarios characterized by hazards, time sensitivity, or logistical complexity. This study introduces a machine learning methodology designed to enable a swarm of autonomous drones to execute a search-and-rescue mission effectively within a known environment. By conceptualizing the environment as a grid-based heatmap that accentuates areas with a higher likelihood of locating survivors, the system trains agents to prioritize areas of high significance. The primary challenges addressed include inter-agent coordination, path planning, and task distribution. The proposed framework ultimately seeks to diminish search duration and personnel needs, thus enhancing the overall effectiveness and safety of rescue operations.

2 Introduction

In recent years, the development of search-and-rescue drones has emerged as a major innovation in the field of disaster response and management [10]. Drones offer advantages in scenarios where human intervention may be risky, time-sensitive, or logistically challenging. When dealing with a search-and-rescue mission, drones can access areas that are otherwise difficult to reach and can carry sensors such as thermal cameras to make the search more effecient, as they can cover vast areas simultaneously and relay real-time data to ground teams. Thus, if multiple drones can be deployed, the time to detect a distressed person will decrease and the chance of survival will increase. To increase the effectiveness of such operations, the aim of this project is to enable a swarm of drones to effectively cover a known environment using a learning based algorithm. This will reduce the time it takes to execute a search-and-rescue mission, and the number of people needed. For this machine learning model, it will be studied how to improve coordination between agents, path planning, and task distribution. The environment in which the machine learning model operates is defined as a grid, where each cell describes the expected amount of information, and is often visualized as a heatmap, as seen in fig. 1. The heatmap

is created by an algorithm [3] that analyzes the geographic location and identifies the regions with the highest likelihood of locating a survivor [7]. This will be our way of determining the importance of different locations within the environment, where we will directly map the likelihood of finding a survivor, to a reward for that location. These rewards will encourage the agent to prioritize navigating through high-probability areas during its traversal.

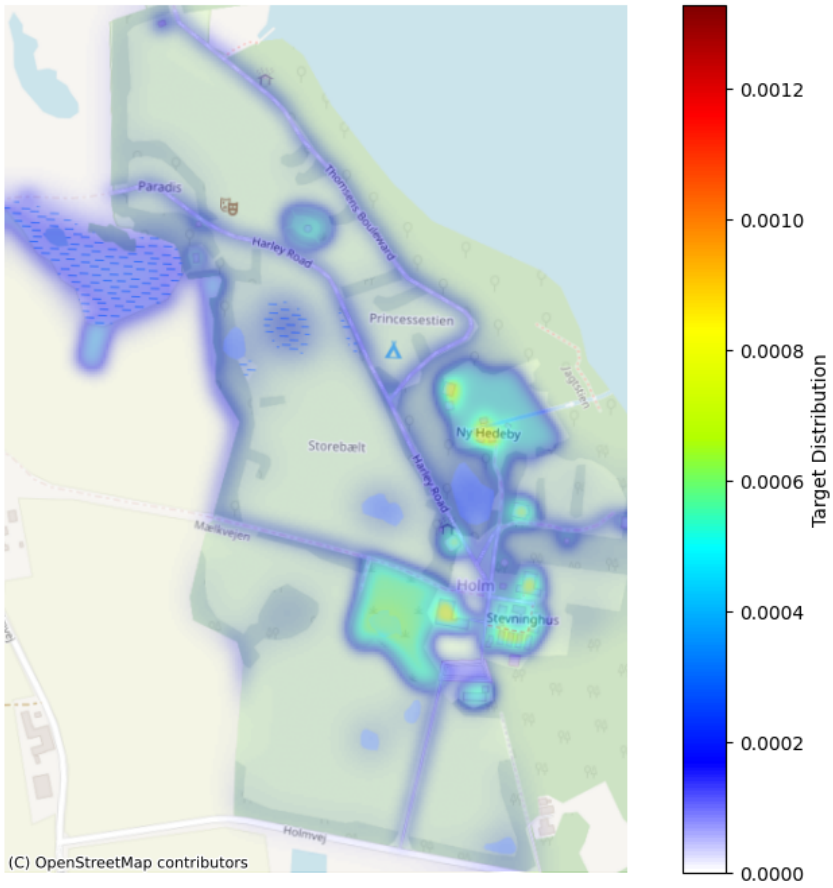


Figure 1: Heatmap of an area, where blue colored areas have a lower probability of having a survivor and red colored areas are highly likely to have one.

3 Problem Statement

To efficiently cover search areas, we aim to enable drones to autonomously coordinate search-and-rescue missions while optimizing predefined objectives. Assuming an environment of the area is provided that indicate regions of higher interest, the challenge involves covering the required area with minimal redundancy, prioritiz-

ing high-interest zones, and simultaneously managing resources. The multi-drone coordination must effectively distribute tasks among all active agents.

3.1 Goals

The goals of this project is to get a functional multi-drone search-and-rescue model functioning in a simulated environment. This model has to achieve the following:

- Search points of highest interest first, while still covering all points.
- Avoid clustering, by not having too many drones in the same area.
- Avoid redoing tasks.
- Manage battery life to avoid crashing while covering the area.

4 Reinforcement Learning

Reinforcement Learning is a method in which an agent learns to make decisions by interacting with an environment. This involves a set of states $\mathcal{S}$ and a set of actions $\mathcal{A}$ [11]. The agent receives feedback in the form of rewards r on the action $a \in \mathcal{A}$ it takes to transition to a new state $s \in \mathcal{S}$, reinforcing a behavior. The behavior that the agent learns to follow is called a policy π, which maps states to actions, and leads to a higher cumulative reward over time. Unlike supervised learning [8], reinforcement learning [6] does not rely on defined input-output pairs, but instead relies on learning through trial and error.

There is no universally perfect policy; optimality depends on the environment's constraints and the agent's objectives. In reinforcement learning, the agent's interaction with the environment is typically divided into episodes. An episode is a sequence of time steps that begins in an initial state and ends in a terminal state or after a fixed number of steps [11]. This episodic structure facilitates the agent's learning from comprehensive trajectories of behavior and enables the resetting of its experience after each episode. Episodes are especially relevant in environments where tasks have a clear beginning and end, such as task completion scenarios. To behave optimally,

one must clearly define what constitutes "optimal" in a given context.

Specifically, it is important to define how the agent should account for future rewards, known as the return G_t. If the sequence of rewards after time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3} \ldots$ [11], then the return can be defined as the cumulative sum of rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

Three models account for most of the work in this area [6], namely the finite-horizon model, the infinite-horizon model, and the average-reward model. These three models optimize rewards gained in different ways, the finite-horizon model optimizes the expected return over a fixed number of future steps, denoted by a horizon h. At each time step t, it aims to maximize r_t, the expected reward over the next h steps. The agent's policy may change over time as the remaining horizon shrinks. This model is applicable only when the agent's lifetime is known in advance, which is often not the case. In those cases, the two other models can be used. The *infinite horizon model* takes future rewards into account, but does so using a discount factor, denoted γ, (where $\gamma \in [0, 1]$).

This discount value determines how much the model should value future rewards; a high γ means the model puts a stronger emphasis on future rewards. The *average-reward model* will have the agent take actions that are intended to optimize the average reward obtained in the future. This can be challenging in certain scenarios, as two routes may yield the same average reward, yet differ significantly in their reward distribution—one offering a high initial reward, and the other providing more consistent rewards over time. Different implementations of this model handle those situations differently, but they all strive to get the highest average.

In this project, we adopt the infinite-horizon reinforcement learning framework, as it aligns with the objective of maximizing long-term cumulative rewards in a static and known environment. This formulation encourages agents to plan ahead and prioritize high-value regions over the course of extended trajectories. We employ the Q-learning algorithm, which operates under this infinite-horizon model to iteratively estimate the

optimal action-value function without requiring a model of the environment. Its off-policy and model-free characteristics make it particularly well-suited for coordinating multiple agents in structured environments, where rewards are spatially distributed and the goal is persistent exploration of high-importance areas.

4.1 Q-Learning

Q-learning is selected for this project due to its capacity to learn a policy that prioritizes high-reward regions through greedy action selection. For this, Q-learning is suitable, as the learned action-value function Q, directly approximates $q*$, the optimal action-value function [11].

A key advantage of Q-learning in this context is the low inference time. Once training is complete, the agent can make decisions by simply selecting the action with the highest Q-value for its current state. This involves a lightweight table lookup operation, which can be executed with minimal computational overhead.

In real-world search-and-rescue operations, where drones must respond quickly and reliably, this low-latency inference is essential. It allows each drone to operate autonomously in real time, even with limited onboard computational resources. This makes Q-learning not only an effective learning strategy, but also a practical choice for deployment in time-critical, resource-constrained environments.

At each time step, the agent observes its current state s, takes an action a, receives a reward r, and transitions to a subsequent state s'. The expected future return is estimated using the maximum Q-value associated with s', denoted $\max_{a'} Q(s', a')$, where a' ranges over all possible actions in the next state. The Q-value for the current state-action pair $Q(s, a)$ is then updated using the temporal difference (TD) learning rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where $\alpha \in (0, 1]$ is the learning rate and $\gamma \in [0, 1]$ is the discount factor that determines the weight of future rewards. This approach allows the agent to steadily

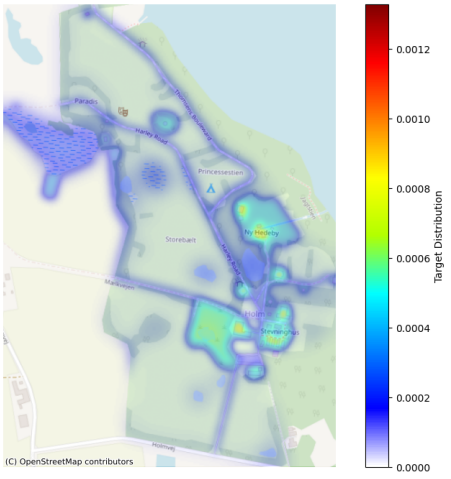
improve its estimates of the long-term outcomes linked to each state-action pair.

The rewards defined by the heatmap, will draw the agent to prioritize moving near high priority areas first. When a location is visited, the reward is gathered and then set to -2, this is an indirect way of letting the agents now, that this location has been visited. If the same location is visited again, the reward will decrement by -2 per revisit. e.g. a location with a value of 10, would go to -2 when visited the first time and then -4 when visited a second time.

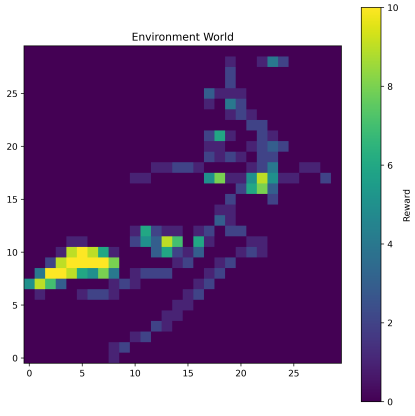
5 Environment

We define our environment to be a grid with the size 30×30, giving us a total of 900 locations, which the agents are able to visit. We choose this environment size for experiments, due to long training time of larger environment. We aim to make the agent able to perform missions on environment of all sizes. As mentioned we use the heatmap fig. 1 and scale it down to our environment's size, this can be seen in **??**. To ensure a standard for experimenting, the rewards from the heatmap are rescaled to a corresponding interval from 0 to 10, $r \in [0, 10]$.

Note that in fig. 2b the environment is flipped, this is only visual and does not affect the model.



(a) Heatmap from the algorithm fig. 1



(b) The environment

Figure 2: Heatmap and environment side by side

To ensure consistency between experiments, the same environment will be used throughout development of the machine learning model, unless stated otherwise. This approach makes it easier to observe the impact of changes on the model's variables, making it more clear how each parameter adjustment influences the model's behavior. The environment is built on top the Gymnasium OpenAI Gym Library [2]. This environment structure does not support multiple-agent simulations, which is crucial for this project. Therefore the structure is extended to support the simultaneous simulation of multiple agents, enabling swarm drone simulations. Furthermore, it is a deterministic and static world.

One way to evaluate the agent's performance is by measuring the percentage of total rewards it accumulates in an episode. This is done by comparing the rewards the agent accumulates to the total available in the environment. Using this percentage makes it easier to compare different versions of the agent. If an agent accumulates 80 percent of the possible rewards during a simulation, it will be terminated as we deem it to be a successful mission. Accumulating 100 percent of the available rewards represents full coverage of the areas with the highest probability of locating the missing person, as defined by the heatmap-derived reward signals.

A mathematical expression of this can be seen in eq. (1) Where W is a matrix containing the rewards of all locations in the environment and V is a matrix of the same size which describes what locations have been visited during the episode.

$$\text{Probability of finding survivor} = \frac{\sum(\mathbf{W} \circ \mathbf{V})}{\sum \mathbf{W}} \tag{1}$$

6 Agent Dynamics: Actions and Observations

We want an agent to be environment agnostic and have no global knowledge of the world, thus is only able to learn from what it can observe. This means the agent decides what actions to take, based on the information it can immediately observe. Agents do not have prior knowledge of the environment's layout, such as the locations of rewards or obstacles. These elements become known only when the

agent encounters them directly. This design encourages agents to learn and adapt through interaction with the environment, relying entirely on observable cues rather than any external or pre-defined information.

6.1 Action Space

The agent is able to perform both orthogonal and diagonal movements in all directions. This is chosen instead of only orthogonal movement, to increases the realism of its navigation strategy. An action to direction mapping table can be seen in table 1. These directions are visualized in fig. 3.

Action	**Direction**	**Vector** $(\Delta x, \Delta y)$
0	Right	$(+1,\ 0)$
1	Up	$(0,\ +1)$
2	Left	$(-1,\ 0)$
3	Down	$(0,\ -1)$
4	Up-Right	$(+1,\ +1)$
5	Up-Left	$(-1,\ +1)$
6	Down-Left	$(-1,\ -1)$
7	Down-Right	$(+1,\ -1)$

Table 1: Action-to-Direction Mapping

$$\begin{array}{ccc} (5)\ \nwarrow & (1)\ \uparrow & (4)\ \nearrow \\ (2)\ \leftarrow & C & (0)\ \rightarrow \\ (6)\ \swarrow & (3)\ \downarrow & (7)\ \searrow \end{array}$$

Figure 3: Action Space Visualization

6.2 Observation Space

To define a state, the agent makes an observation consisting of *visited states nearby* and *reward nearby*. *visited states nearby* is a 3×3 matrix, initialized with zeros and defines the agent's surroundings, with the agent positioned at the center. An index in this matrix is set to 1 if the corresponding location has been visited, it remains 0 if the state is unvisited.

reward nearby is a vector that is 8 units long, one for each direction the agent is able to move. With a observation range of its immediate surrounding locations, it checks in which direction the reward is the highest, and sets the corresponding index to 1, while the rest stays at 0. E.g. if the highest reward around the agent is directly to its left side, the reward vector will look like this: $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$. The reward would be in index three since that is the corresponding index for moving left

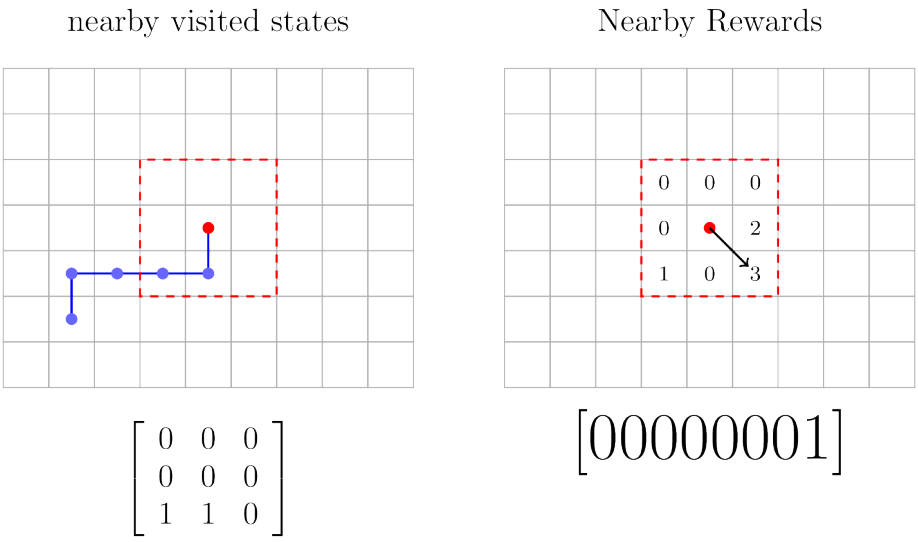


Figure 4: **Left:** Observation of previously visited states via path history.
Right: Detection of rewards nearby. Index 8 is set to 1 as the highest reward is in the down-right direction, see table 1.

in the action space, see table 1.

An illustration of the agent's observation can be seen in fig. 4.

7 Boltzmann Distribution

The Boltzmann Distribution, also known as the softmax function, is commonly used in reinforcement learning to introduce controlled stochasticity into action selection [9]. Instead of always choosing the highest-valued action, this method assigns probabilities based on estimated action values, allowing a balance between exploration and exploitation.

Agents navigate the environment by selecting one of the eight possible actions, from the defined action space. The probability p_i of selecting action i is given by:

$$p_i = \frac{e^{\beta x_i}}{\sum_{j=1}^{n} e^{\beta x_j}} \tag{2}$$

Here, x_i represents the value of action i, and β is a temperature parameter that adjusts the sensitivity of the distribution. Higher values of β make the agent more

deterministic, while lower values increase randomness in selection. This formulation encourages exploration by assigning a probability to all actions while favoring those with higher Q-values

8 Singular agent

With an agent able to observe nearby rewards and visited locations in its vicinity, we can deploy and experiment with a baseline for the model. When training and evaluating the agents, they are limited to taking the number of steps needed to exactly cover 100% of the environment. Meaning if the world is 30×30 the agent can take a maximum of $30 \times 30 = 900$ steps, after which the training episode will terminate. Furthermore, when evaluating, the episode will end if 80% world info is gained. Using these rules the goal of the agent is to maximize the info gained in each environment with the least amount of steps. Note that despite knowing the precise lifetime of an agent, we still use an infinite-horizon model as mentioned in section 4. The reason for this, is that the number of steps taken is not part of the agent's observation, and it does not change behavior as remaining steps decreases, effectively making it act accordingly to the infinite-horizon model.

With reward nearby and visited near as its only observations, it is often able to traverse the map and cover most of the reward, seen in fig. 5d. However, if the rewards near it have all been taken, it has no sense of where to go. Either way is equally as good according to the agent. This can occasionally lead to the agent looping around in the same area or trapping itself in a surrounded area of visited locations. It can be seen in fig. 5c that the agent is stuck in a loop, repeating the same moves in the corner. Note that in all four plots, the agent uses all of its 900 steps unless 80% info is gained in the environment, meaning that the agent in fig. 5a and fig. 5c are taking all 900 steps in circles or into the wall. Considering the reward for a location once it is visited, become increasingly more negative. This means the reward after the entire episode is a very large negative number, and the agent does not learn to avoid this looping behavior. After investigating this issue, it is found that, this is caused by the agent not being able to observe or know where the border

of the environment is, it takes an action, gets a negative reward for it, but does not learn to avoid it. At this stage of development, when the agent tries to walk outside of the world, it is clipped back to where i came from, but it still gets the reward for landing in the same location again. We do not wish to solve this issue by making the world borders part of its observation, since that would lead to a larger Q-table and thus slower learning. Therefore, the world borders are added to the individual agents internal information, together with its location. This enables us to externally mask bad action, while keeping the learning environment clean. When the agent is choosing what action to do next, we check if an action will lead the agent outside of the world, and give those actions a weight of zero.

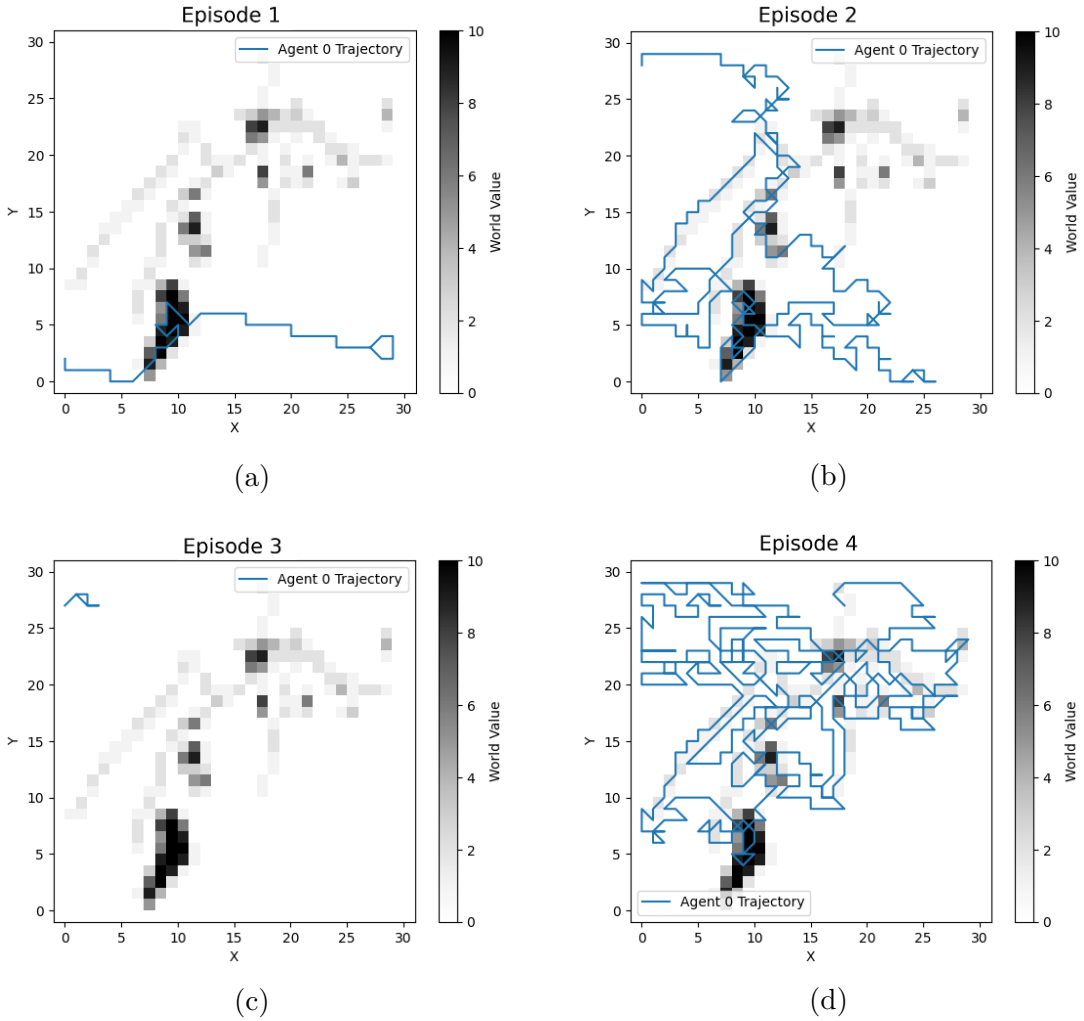


Figure 5: 4 evaluations of a single agent

8.1 Addition of point of interest observation

As seen in fig. 5b and fig. 5d, when the agent has no reward nearby, it starts to go in random directions. In order to change this behavior, we implement an additional observation that points in the direction of a point of interest (POI). These points will be placed in areas of the map, where there is a high density of rewards or one particularly high reward. It does not point to the exact location, just the direction of the nearest POI. This should help the agent learn to follow this indicator, when no rewards are present nearby.

In the following experiments we use the same environment and the same parameters as before fig. 5, but now with POIs implemented. It can be seen in fig. 6 that in the four evaluations, the agent always finds rewards and starts collecting them.

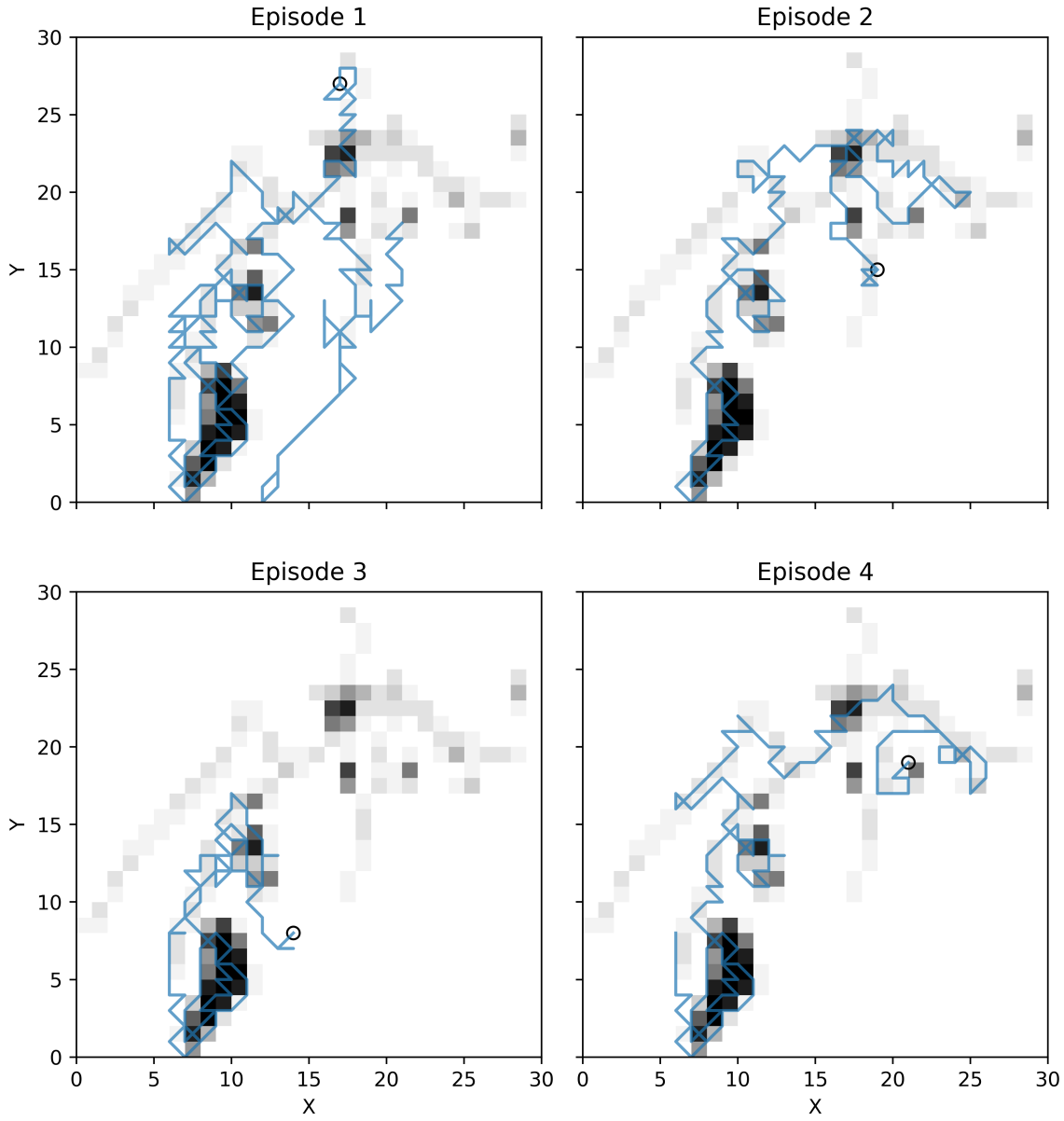


Figure 6: Single agent trajectory with 4 POIs

However this could also be due to chance, considering the agent already had a behavior of collecting rewards nearby. Also in all four episodes, it happens to either spawn near or inside rewards. Therefore this is not definitive proof that the POI system is actually helping.

In order to experiment the POI system. A very simple world with only 3 rewards, each with its own POI, is created. Once a POI is reached, it is deleted, as to not lead the agents towards already scanned areas. As seen in fig. 7, it shows that the

agent does learn to follow the POI observation, at least for a simple world with no other agents.

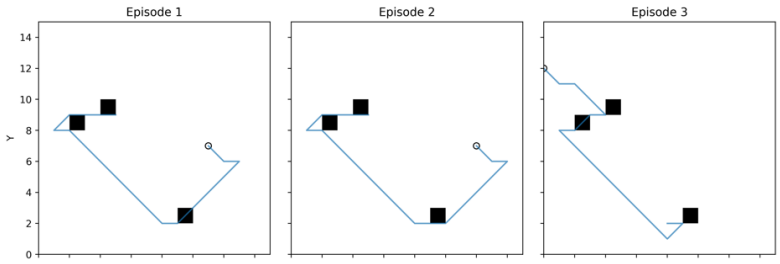


Figure 7: Single agent in environment with only 3 POIs

This shows the idea with the POI system works as intended. It points the agent in the direction of rewards without giving it global information of the environment. This has been done with manual placement of the points, the following section will describe how they will be placed automatically.

8.2 Automatic POI placement

An automatic function to choose where to place the POIs is designed. This will enable the POI system to work on any given environment without manually placing points of interest. When the environment is generated from the heapmat, it is scanned through by a 3x3 matrix from left to right. The average reward within this area is compared to the average reward of the environment, excluding rewards of zero. When this is true a POI is placed, this will place many POIs and many of them being close together. We decide when running a simulation, how many POIs we want placed for that simulation, and a minium distance between each point. This is to avoid clustering all the POIs inside the area with the highest rewards. E.g if it chosen to have two POIs with a minimum distance of 3 steps in a simulation, it will pick the two POIs with the largest average reward and have a minimum distance of 3 steps between every POI. A visual representation of this can be seen in fig. 8. Where the purple circle is a potential POI, but is not chosen due to being too close to the other possible POI which has a higher average reward around it. And then a last POI placed at the bottom left corner.

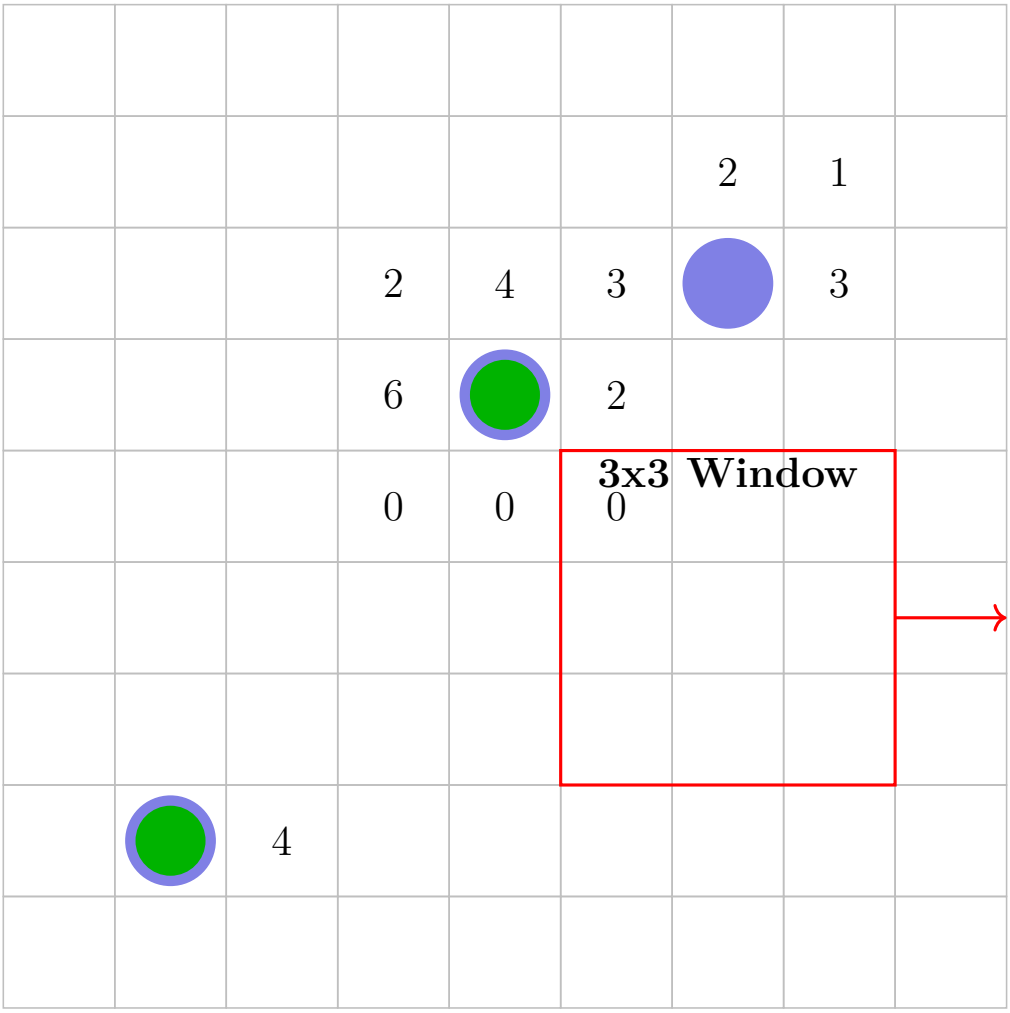


Figure 8: Scanning for POIs with 3x3 window and automatic selection

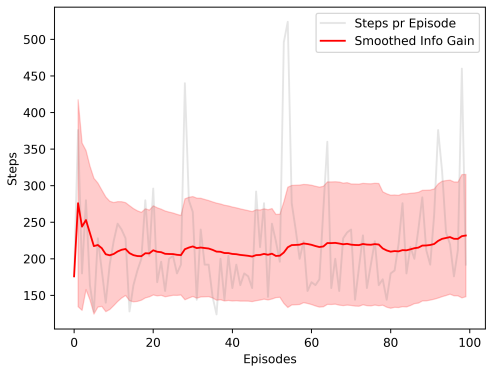
8.3 Changes to the observations

Currently we have the two observations: reward nearby and visited nearby. We realize that visited nearby is almost a redundant observation, since the reward nearby obs would already point away from a visited location, as it has a negative reward. It is decided to remove "Visited nearby" observation. Which informed the agent, what locations around it that had already been visited by any agent. The reasoning for removing this, is that it can be implemented within the reward nearby observation, making the Q-table even more compact while containing the same amount of information. The reward nearby observation originally would only set the index of the direction of the highest reward to 1, and keep the rest at zero. This meant, that it had no way to distinguished between visited locations and locations

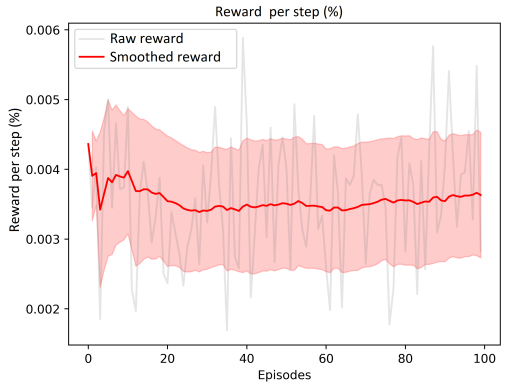
with a reward of zero. This is why ”visited nearby” obs was created in the first place. This is changed, by removing the visited nearby obs, and making the reward nearby doing more than just index the direction of the highest reward. Now the reward nearby obs can set an index to a value in the interval [0, 2]. Where if the value of an index is 2, means that direction has a positive reward, 1 meaning the reward is zero and index value of 0 means the reward is a negative number. Since a reward of a location is set to -2 upon visit, this effectually gives the same information as the visited nearby obs did before. While still pointing to where the positive rewards are. This idea could be extended even further, by making the reward near indexing value range from zero to 3, where a value of 3 would point to the highest of the nearest reward, 2 meaning any positive reward, 1 being zero reward and 0 being negative. This idea will be tested, to see if the learning rate is higher, lower or unchanged, and to see if there are any other major changes in behavior.

At this point in experimentation, both versions reach goal of 80 percent coverage very consistently, so comparing those is meaningless. However it is worthwhile to compare the amount of steps that are needed to complete each episode, and how much info the agents gain per step they take. A comparison between these two metrics , using 10.000 training episodes and 100 evaluation episodes, can be seen in fig. 9. It can be observed that the amount of steps used per episode are almost the same, being stable at around 225 steps used. There is a slight difference in info per step, with the max index of 2 being around 0,004, and max index of 3 being slightly lower at 0,00035. It is therefore concluded that the difference this change to the reward obs is insignificant and slightly in favor for the index with a max value of 2. This is also the option with the smallest Q-table, so the reward nearby obs will now have a range of [0, 2].

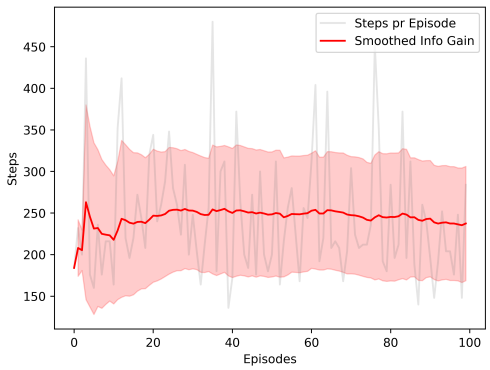
(a) Info per step with max index of 2



(b) Step per episode with max index of 2



(c) Info per step with max index of 3



(d) Step per episode with max index of 3

Figure 9: Comparison of different ranges of index values for the reward nearby obs

These changes to the observation space will reduce the size of the Q-table, here is a breakdown of how much smaller it has become. The original observation space was composed of the following:

Component	**Format**	**Possibilities**
Visited Nearby	3×3 binary matrix (8 cells)	$2^8 = 256$
Reward Nearby	8-length binary array	$2^8 = 256$
POI Direction	8-length binary array	$2^8 = 256$

Table 2: Original observation space components

Total observation space size: $2^8 \times 2^8 \times 2^8 = 2^{24} \approx 1.67 \times 10^7$

After the changes:

Component	Format	Possibilities
Visited Nearby	Removed	—
POI Direction	8-length binary array	$2^8 = 256$
Reward Nearby (new)	8-length array, values $\in \{0, 1, 2\}$	$3^8 = 6{,}561$

Table 3: Updated observation space components

Total new observation space size: $2^8 \times 3^8 = 256 \times 6{,}561 \approx 1.68 \times 10^6$

Thus, the observation space and consequently, the Q-table is reduced by:

$$\frac{1.67 \times 10^7}{1.68 \times 10^6} \approx 10$$

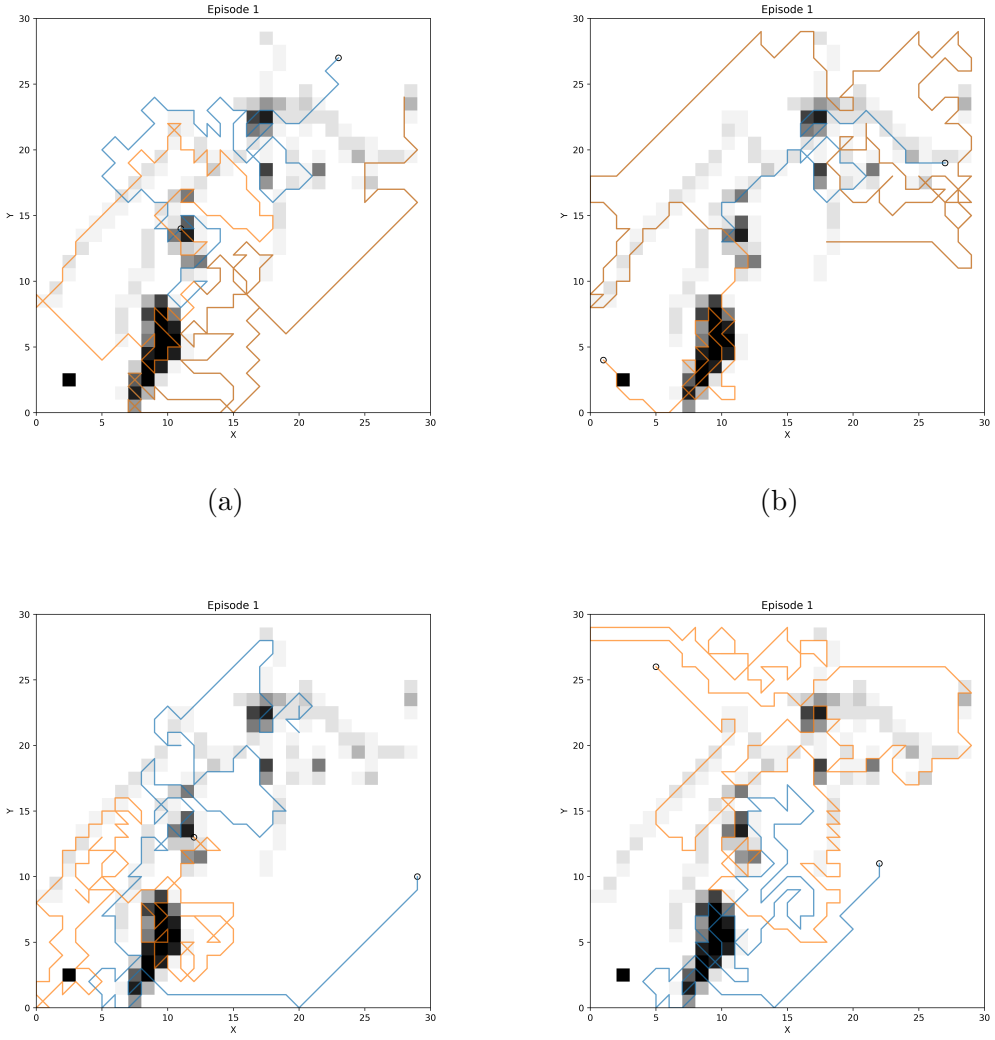
These changes will still convey the same information to the agent, while having a smaller Q-table.

9 Implementation of Swarm Behavior

Having established a satisfactory behavior for a single agent, our objective now is to simulate swarm behavior. Initially, the existing single agent model will be employed in its current form, with multiple agents concurrently situated within the environment. These agents will utilize a shared Q-table during both the training and evaluation phases. This configuration will function as a baseline, allowing us to observe the model's performance without any changes related to multi-agent behavior. It remains uncertain whether the present parameters and configurations, which have been optimized for a solitary agent, will facilitate effective swarm behavior. This methodology is intended to evaluate the performance of the unmodified single-agent model within a multi-agent framework. Contingent upon the performance observed, subsequent modifications will be contemplated and implemented to enhance coordination and overall efficiency.

9.1 Single-Agent Dynamics in Multi-Agent Scenarios

To evaluate the singular agent model in a multi-agent setting, two agents are deployed in the same environment, as seen in fig. 10. Both agents accumulate rewards with the same behavior as a single agent has done in previous experiments. However, we notice in coordinate $(11, 13)$ in fig. 10b that if two agents move to the same coordinate at the same time, they will take the exact same path until they terminate. This is because they share they same Q-table and thus will always take the exact same action in the same state. For this reason, the agents will be spawned separately while this issue persist. If they spawned together, as they would in a real life mission, they would all take the exact same actions throughout the whole episode.



(a) (b)

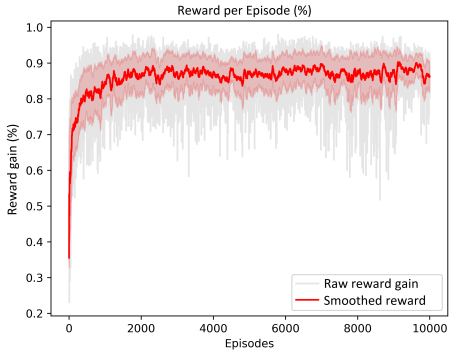
(c) (d)

Figure 10: Trajectories of 2 agents with single agent behavior across 4 episodes

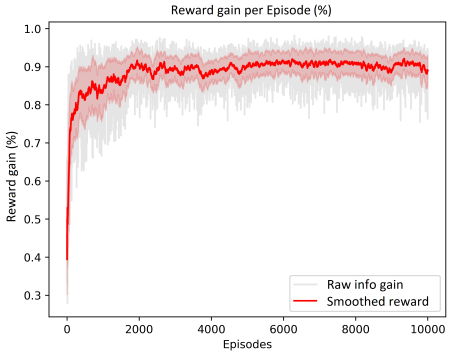
In the current setup, agents are unaware of each other and operate independently, with no mechanism for observing the presence of other agents. As a result, their behavior remains unchanged in a multi-agent environment, thus they do not adapt or coordinate their actions.

This lack of awareness can lead to inefficient behaviors. For example, if multiple agents detect the same reward, they may all attempt to collect it, unaware that others are doing the same. Since all agents interact with the same set of POIs, a POI collected by one agent is removed from the environment for all agents, potentially

causing conflicts or wasted movement.



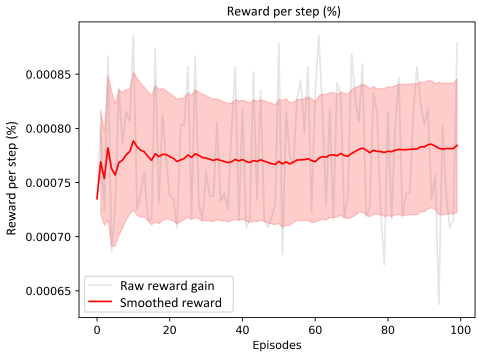
(a) SA with POI



(b) MA with POI

Figure 11: Comparison of rewards per episode in both single agent and multi agent training

Furthermore, the reward per episode can be seen in fig. 11. When comparing fig. 11a with fig. 11b it can be seen that the difference in rewards per episode while training is minuscule. To see if there is an actual change in performance the reward per step can be viewed.



(a) Reward per step for 1 agent

(b) Reward per step for 2 agents

Figure 12: Comparison between 1 and 2 agents' reward percentage per step, over 100 episodes, both using the same POIs.

The reward gained per step in fig. 12b is approximately five times better than fig. 12a when experimenting with two agents. This indicates an improvement in performance when increasing from one to two agents, to see if this improvement scale linearly the experiment is done using four agents, see fig. 13

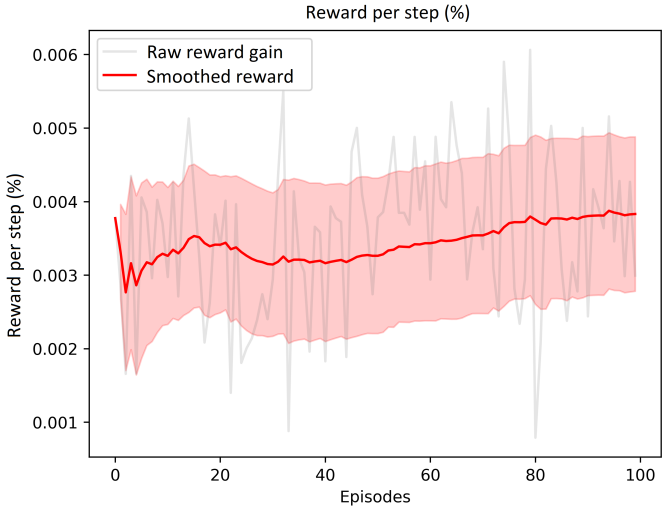


Figure 13: Reward per step for 4 agents

This reveals, experimenting with four agents sees no improvement from two agents. This indicates that more agents does not linearly improve performance. This concludes that the use of multiple agents with single agent dynamics are capable of completing missions with increased performance.

9.2 Cooperative Swarm

For the agents to be able to cooperate on covering the environment, they must be able to see each other. To achieve this, a new observation is needed: *Nearby agents*. This observation will allow the individual agents to see if any other agents are nearby. The observation will be in the same form as POI and reward near, meaning it will be an array with a length of 8, with each index representing a direction. If any agent is two cells or closer to the individual agent, it will set the matching direction to 1 in the nearby-agents, multiple directions can be set at the same time if more agents are closer than two cells. A visual representation of this observation can be seen in fig. 4. When evaluating the model with the *nearby agents* observation the number of revisits and agent overlaps is expected to decrease, as they can now see each other when in close proximity. This aims to improve cooperative interactions between agents, particularly by addressing the challenge of agents taking the same action, when at the same location, as seen in fig. 10b.

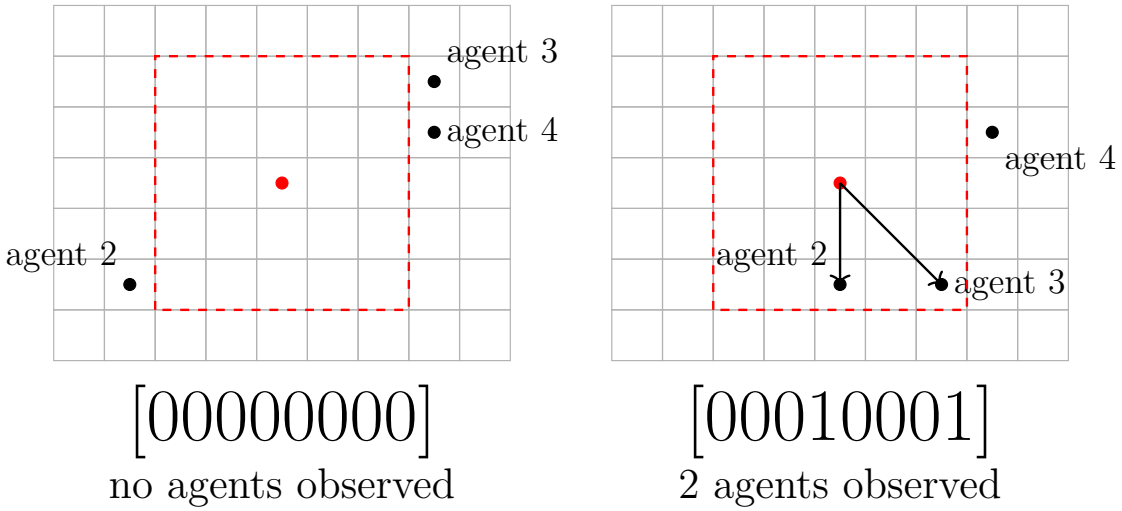
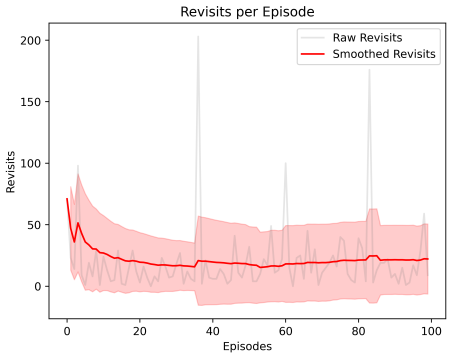
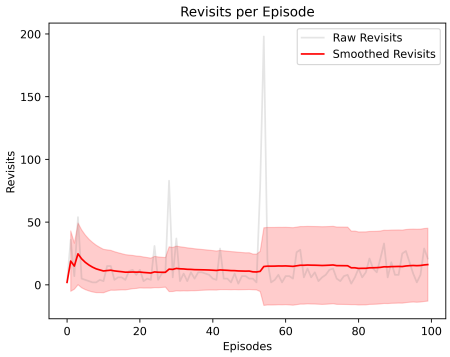


Figure 14: Nearby agent observation. Each index in the array corresponds to a direction. A value of 1 indicates that at least one agent is within two cells in that direction.

In order to test the impact of the new observation, experiments will be performed to compare the performances. An experiment to compare the number of revisits between four agents with and without *nearby agents* is performed. When comparing the two graphs in fig. 15, it is evident that the number of revisits only decreases slightly when the agents can observe each other. Though the average number of revisits is similar, there are fewer episodes with extreme revisits when the agents can observe each other, and the confidence interval is smaller. These differences indicate a slight improvement in performance. The reason for not seeing a larger decrease in revisits could be due to the agents already attempting to minimize revisits.



(a) Multi agent without *nearby agent* observation



(b) Multi agent with *nearby agent* observation

Figure 15: Comparison of revisits per episode between multi agent with and without *nearby agent* observation.

When looking at the trajectories of the agents fig. 16 it can be seen that they now attempt to avoid each other and sometimes spread out. Looking at episode 2 in fig. 16 it is clear that even when the agents get clumped up, they still attempt to avoid overlapping.

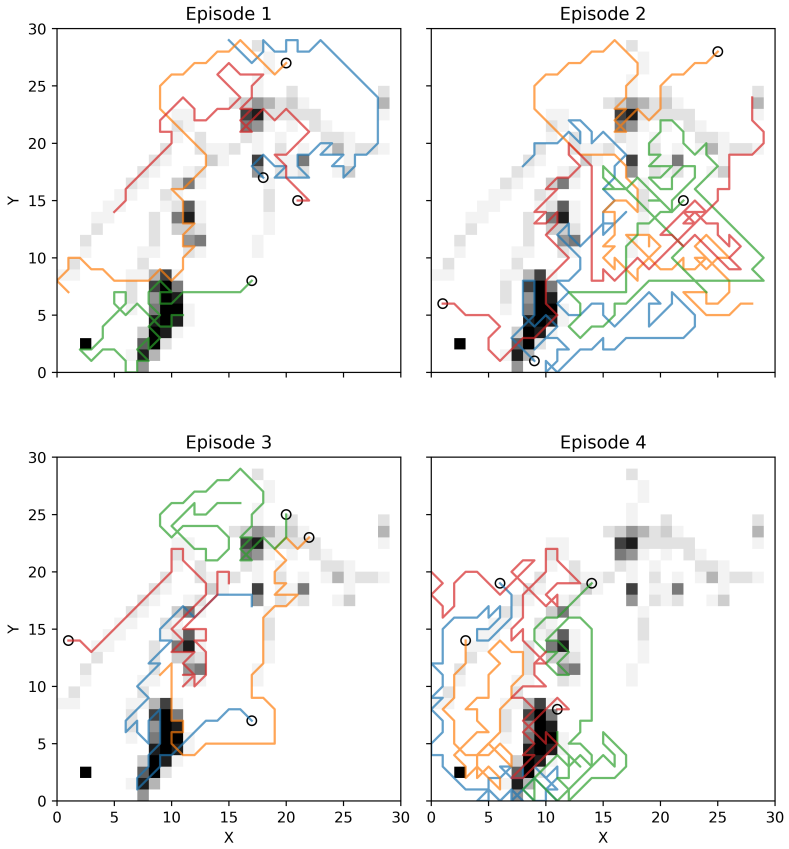
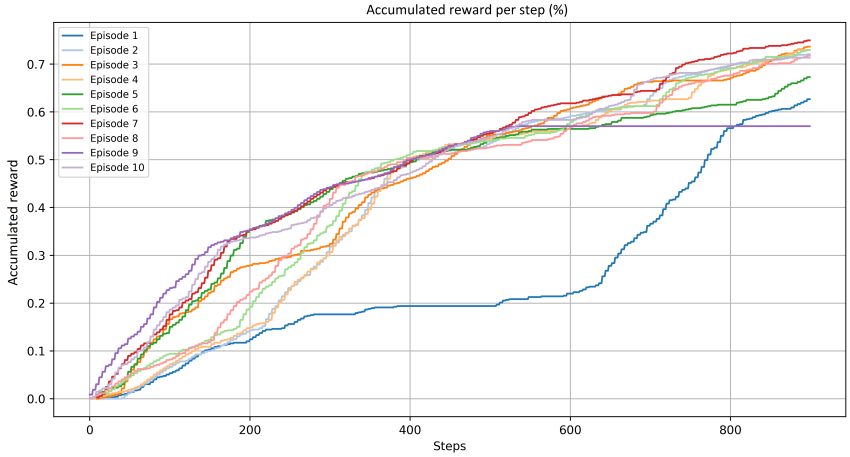


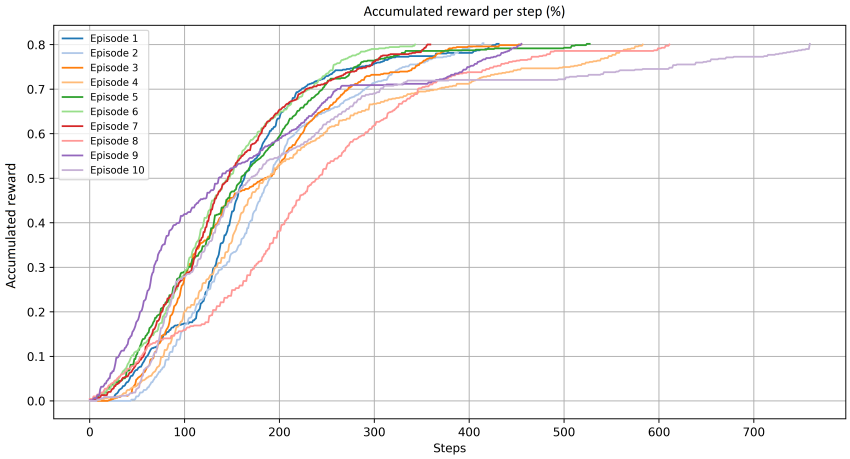
Figure 16: MA with nearby agent observation

To validate if performance increases with the nearby agent observation, we have to look at the reward accumulated each step over both experiments.

With the observation implemented it can be see in fig. 17a all of the episodes are terminated due to reaching maximum steps. Where in fig. 17b that the episodes are terminated due to the agents accumulating more than 80% of the potential rewards in the environment. It can be concluded that implementing *nearby agent* observation leads to a significant increase in performance, leading to a reliable and stable behavior. Thus the *nearby agent* observation will be implemented in the model as the performance gain



(a) Accumulated reward per step with no nearby agent observation



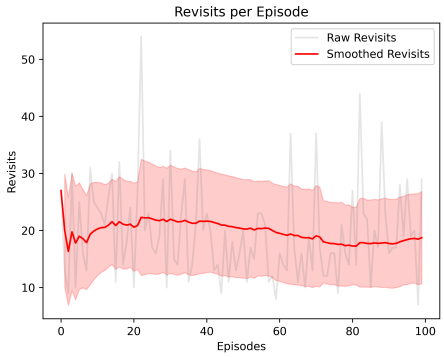
(b) Accumulated info per step with nearby agent observation

Figure 17: Comparison of revisited locations between MA with and without nearby agent observations in 100 evaluations

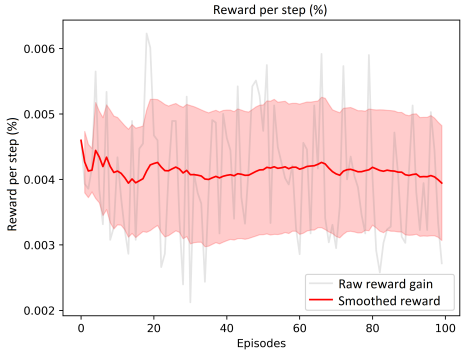
9.2.1 Uniform spawn

With the *nearby agent* observation implemented we will see if the behavior of agents in close proximity has changed. To test the behavior, the same experiment will now be conducted with the nearby agent observation, but with a uniform spawning of the agents, which means that all agents spawn in the same cell. This is to see if the agents can spread out and cover efficiently. When using the drones in a live scenario, the drones will be deployed from the same location. This is what the uniform spawn is trying to imitate. We expect a similar performance or slightly worse, considering

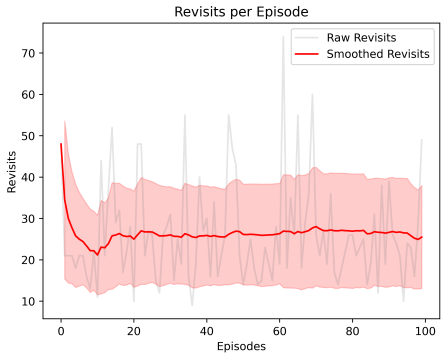
spawning the agents in different areas of the map would allow each agent to cover separate parts of the environment. Whereas when spawned together, all start from the same end.



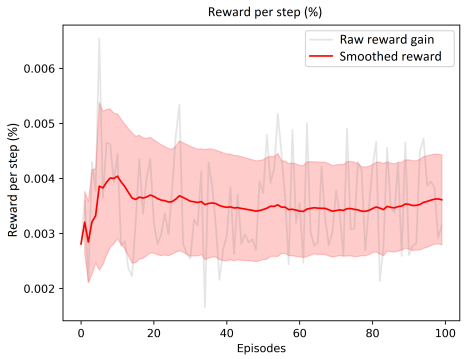
(a) Revisits with uniform spawning



(b) Reward per step with uniform spawning



(c) Revisits with random spawning



(d) Reward per step with random spawning

Figure 18: Comparison between revisits and reward per step between random spawning and uniform spawning

It can be seen in fig. 18 that the difference in reward per step and number of revisits is insignificant. This suggests that the *nearby agent* observation effectively prevents the coordination issues previously observed in close-proximity situations, fulfilling its intended purpose. To further investigate this, we will look at the trajectories for the agents, to see how they behave when in close proximity of each other.

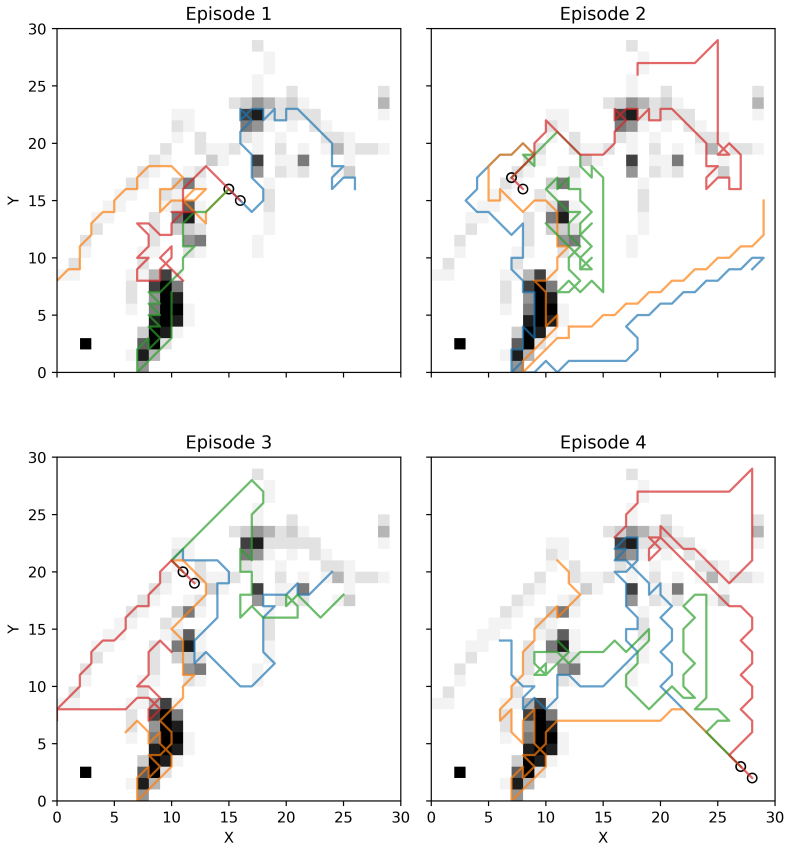


Figure 19: Multi agent trajectories with uniform spawn

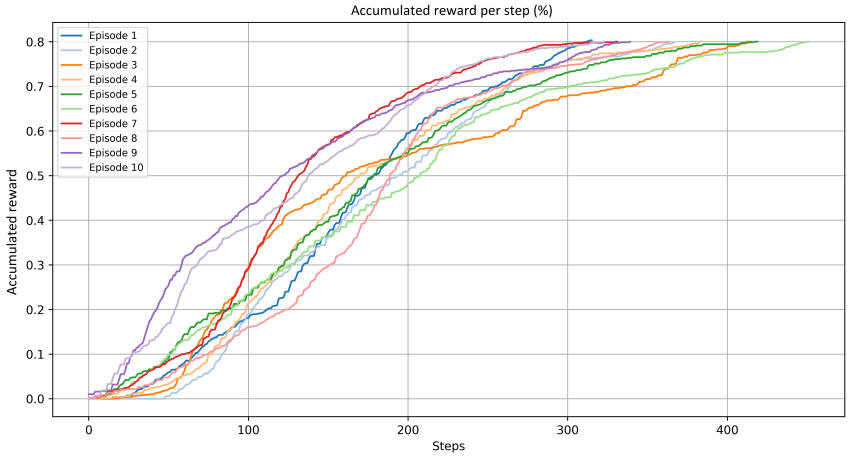
When looking at the trajectories in fig. 19 it can be seen that the agents spread out and cover different areas. Recalling the situation in a previous experiment, where agents in the same location would take the same action fig. 10. It is prevalent in fig. 18 episode 4, that despite all four agents spawning together, they start to spread out. Another clear sign of improvement can be seen in episode 1. Where all agents spawn in the center of the map, but each agent takes a different path diverging from the spawning point. Note that despite there being two circles side by side in the spawning point, the agents are still spawned in the same cell. Thus it can be concluded that spawning the agents in the same location does not decrease performance.

9.3 Swarm with individual POI

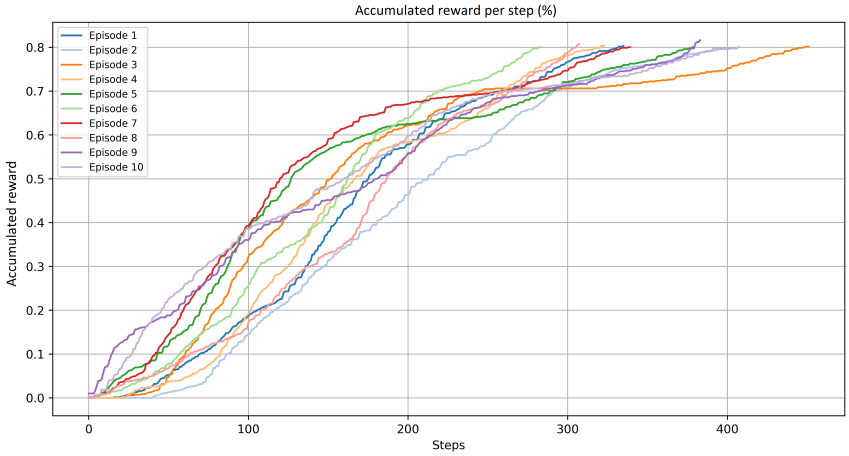
The introduction of POIs resulted in a noticeable performance improvement, suggesting that they help guide agents toward reward dense areas. However, we hypothesize that having all agents share the same set of POIs may be less efficient than assigning each agent a distinct subset. In a shared setup, one agent may head toward a POI, only for another agent to reach it first and claim the associated rewards.

In this experiment, each agent is assigned a distinct subset of POIs to determine whether this setup leads to improved performance. The POIs are evenly distributed amongst the agents, ensuring that no two agents share the same POI.

To determine if the performance is better with individual POIs we will look at the accumulated reward per step.



(a) Accumulated reward per step with no individual POIs



(b) Accumulated info per step with individual POIs

Figure 20: Comparison of revisited locations between MA with and without nearby agent observations in 100 evaluations

Looking at fig. 20 and comparing both graphs reveals that both setups perform similarly. In both cases the agents accumulate 80% reward within $300 - 450$ steps. We can conclude that individual POIs does not affect performance, thus we will proceed with a shared POI system.

10 Implementation

To implement the model on a real system the heatmap program [3] would be used as the user interface to allow an operator to fine tune the search-and-rescue mission.

If the mission includes an area in which humans cannot access, the operator could increase the prioritization of this area with the sliders. This interface can be seen on fig. 21.

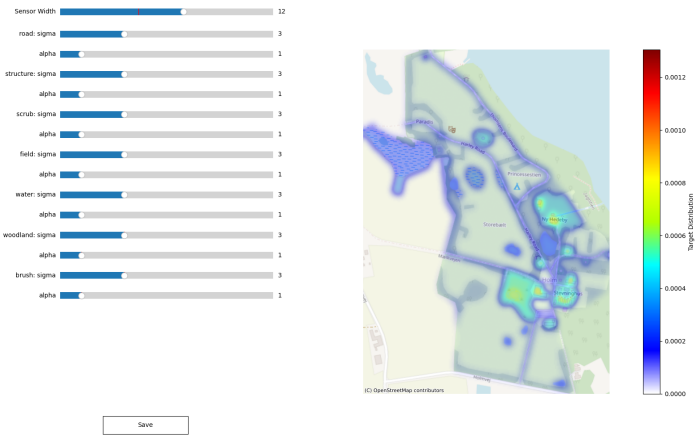


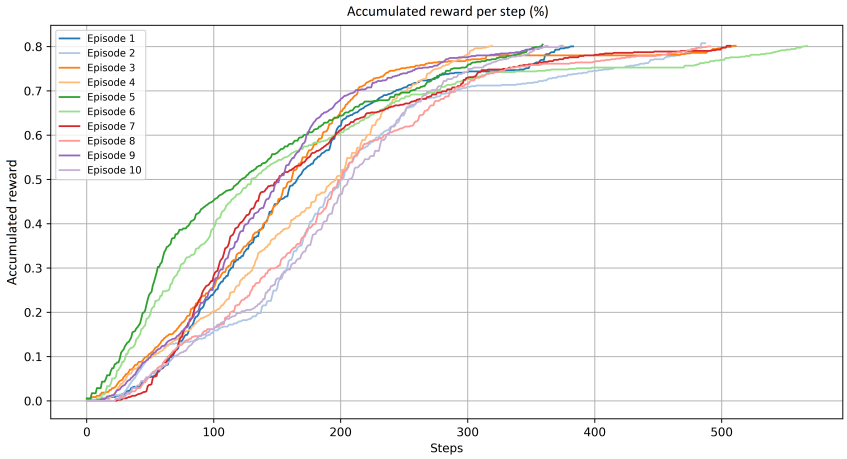
Figure 21: User interface

To implement the model on the drones, the Q-table generated from training will be stored locally on each drone, this is to allow the drones to independently take actions in the environment. The drones would need to share information between each other, such as the drones current location, what locations the drone have visited and what POIs have been visited. The drones should send this information to each other as often as possible to ensure that each observation the agents get is up to date. To ensure this, a peer to peer network [1] could be used. To make the drones able to move in the real world they would need coordinates, these coordinates can be extracted directly from the heatmap program [3].

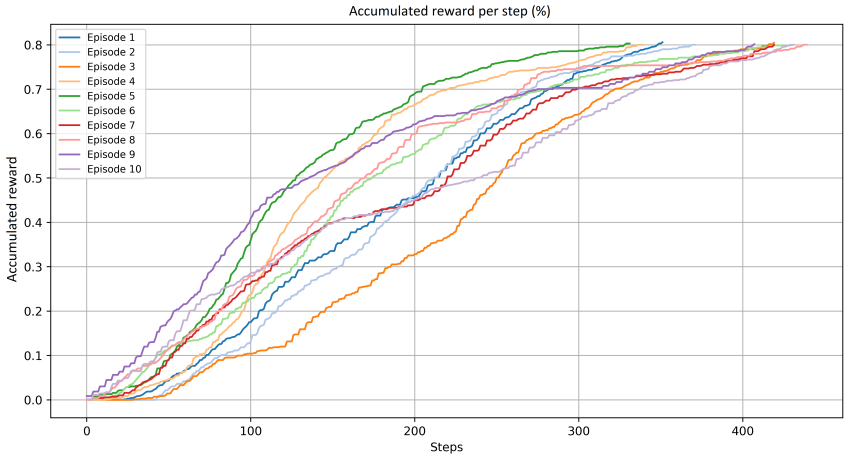
10.1 Battery Management

When moving around in a real life scenario, the drones will be expending their battery, once depleted, they will need to return to a fixed point and recharge. This experiment will implement a battery parameter that sends the agent to their home point if their battery goes below 15 percent. The home point is the location where the agents start their episode. The experiment aims to see if the agents will be able to continue their mission even when placed back at their home point. The battery

parameter will not be an observation and will therefore not affect their learning.



(a) Accumulated reward per step with no battery.



(b) Accumulated reward per step with battery.

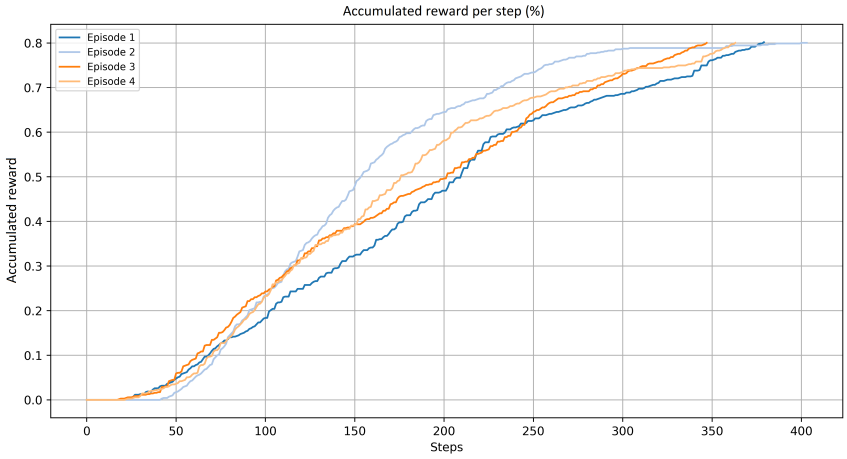
Figure 22: Comparison between a model with a battery and a model without one.

According to fig. 22, implementing a battery function slightly affects performance but the agents are still able to complete the missions within $300 - 400$ steps. To further improve performance, some improvements could be made. Distance to charging station and battery percentage could become an observation, making the agents change behavior accordingly. This could include deciding when to explore further away from home to maximize returns, and when to stay closer to home to avoid crashing. An algorithm from [4] that optimizes charging time to maximize drone flight time could enhance the search-and-rescue model. An additional efficiency gain

could be achieved by enabling the agent to record its last exploration point and resume from that location after recharging. These changes will not be explored further in this project.

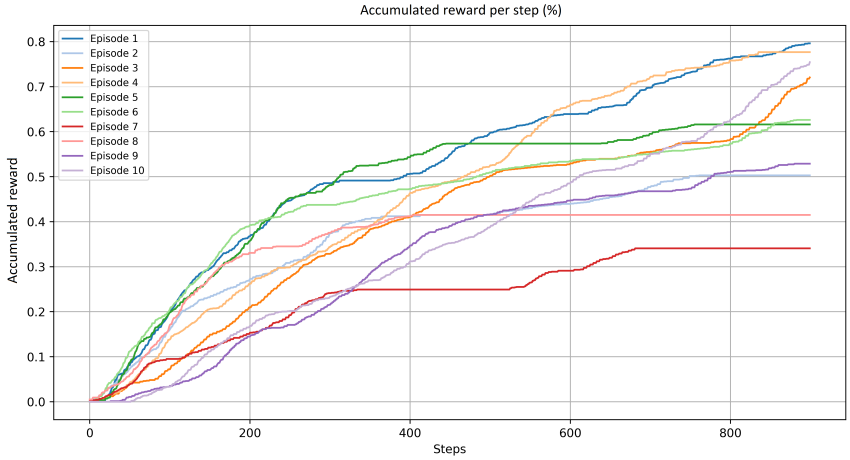
10.2 Testing on different environments

To verify that the agents' behavior generalizes beyond the training environment, we evaluate their ability to navigate previously unseen environments using the learned policy. We will train the agents in one environment, and test their performance in an unseen environment. As a reference point the agents are trained and evaluated in the same environment, seen in fig. 23a. It can be seen that in all 4 episodes the agents accumulate 80% of the environments' rewards after approximately $300 - 400$ steps.

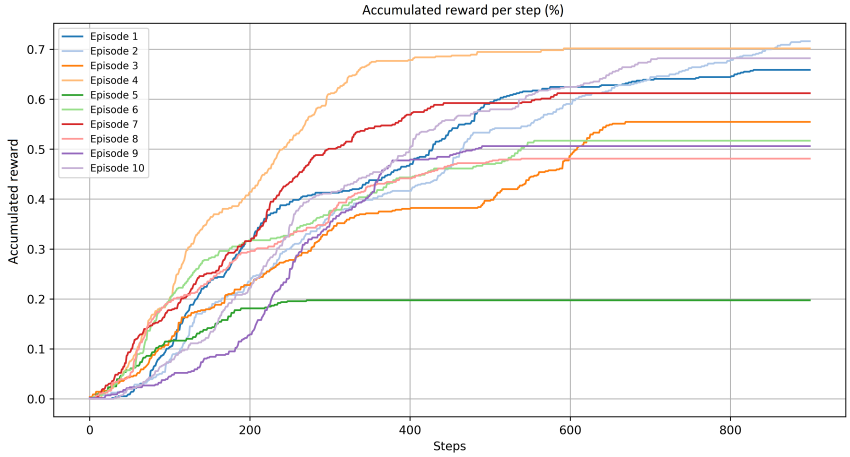


(a) Evaluation on the same environment used in training

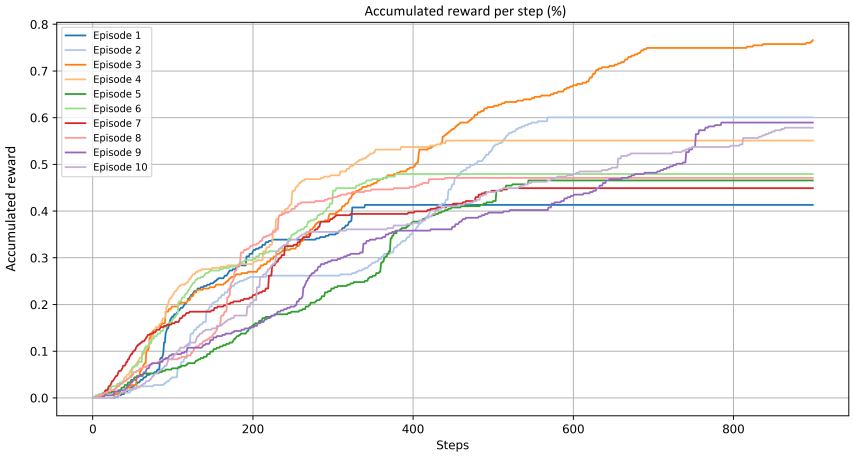
Figure 23: Testing the performance of the model on new unseen environments (1/2).



(b) Evaluation on unseen environment 1.



(c) Evaluation on unseen environment 2.

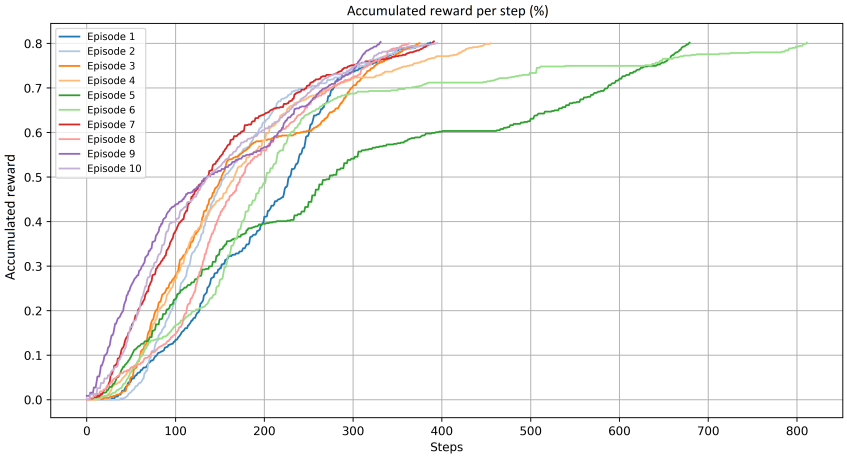


(d) Evaluation on unseen environment 3.

Figure 23: Testing the performance of the model on new unseen environments (2/2).

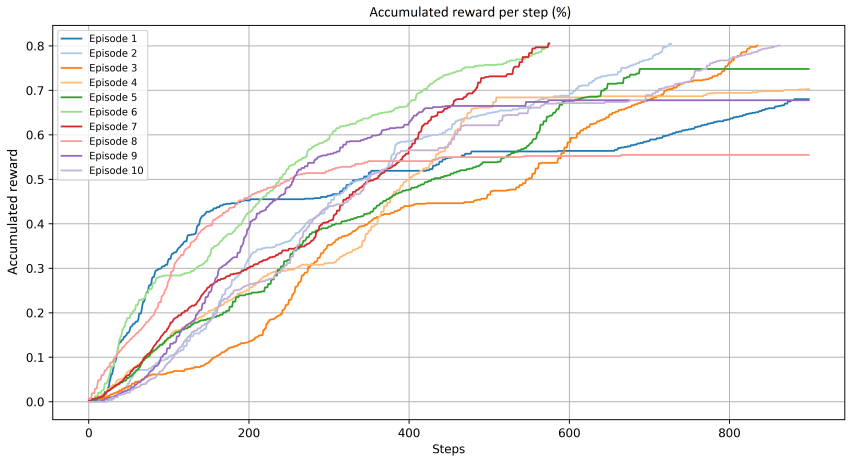
An examination of fig. 23b, fig. 23c, and fig. 23d reveals that, in the majority of episodes, the agents utilize all 900 steps, with only two episodes showing that the agents achieve 80% of the potential rewards. Analysis of fig. 23c further illustrates that, in most episodes, the agents cease accumulating rewards after a certain number of steps and remain stationary for the duration of the episode. This behavior suggests that the agents encounter unfamiliar states, resulting in undefined and inefficient behavior. Nonetheless, it is evident that the model exhibits effective behavior, as the cumulative reward demonstrates growth patterns comparable to the reference plot in fig. 23a. To ascertain the model's capability to manifest robust behavior suitable for implementation in any untrained environments, we will attempt to train the model across diverse environments and subsequently evaluate it in new, distinct settings.

The following is an experiment with the model being tested on 7 diverse environments and evaluated on 4 new environments, this should expose the models performance.

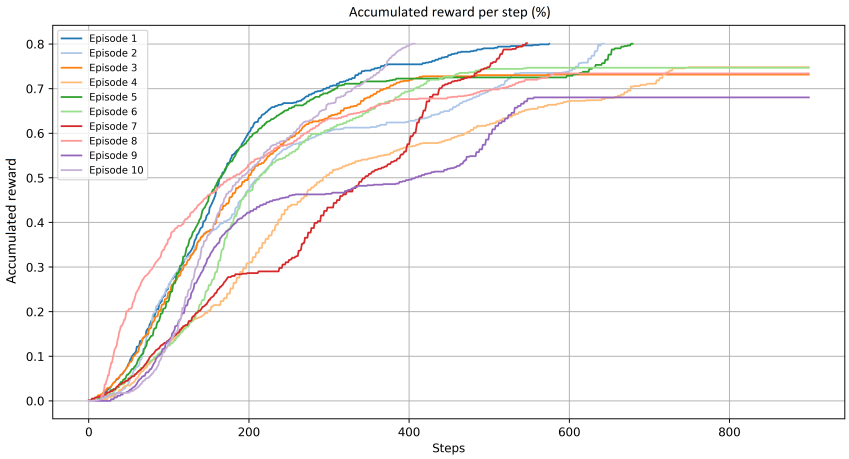


(a)

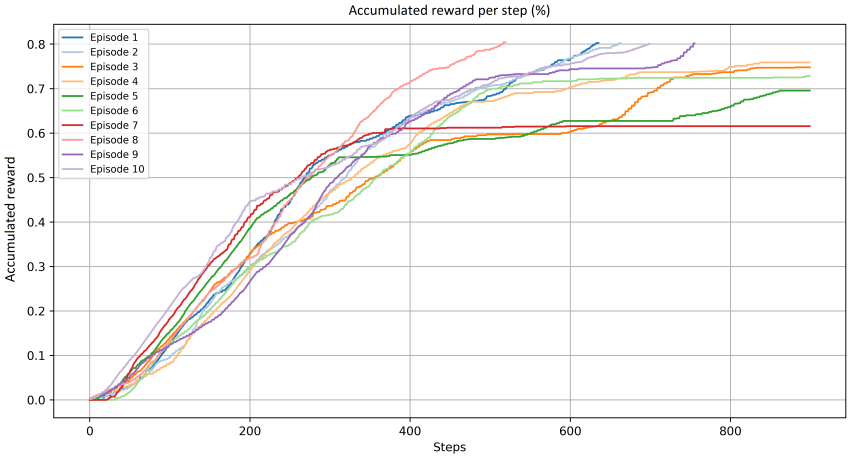
Figure 24: Testing the performance on different environments after training the model on 7 different environments (1/2)



(b)



(c)



(d)

Figure 24: Testing the performance on different environments after training the model on 7 different environments (2/2)

As shown in fig. 24, agents now complete most episodes within 900 steps, though some still end with stationary rewards likely due to reaching unfamiliar states. It is important to note that different environments exhibit varying reward distributions, which implies that the average number of steps required to achieve 80% coverage can differ across environments. Therefore, a higher step count compared to the reference does not necessarily indicate poorer performance. However, failing to reach 80% coverage entirely is indicative of suboptimal behavior. The experiment can conclude that the model can train in diverse environments and later succeed in search-and-rescue missions in new environments. Training in multiple various environments could help the model avoid unfamiliar states and improve mission efficiency, but this will not be explored in this project.

11 Our model compared to brute force.

The aim of the model is to provide a more efficient way to complete a search-and-rescue mission than using a method which sweeps all locations in the environment. An illustration of the brute force behavior can be seen on fig. 25.

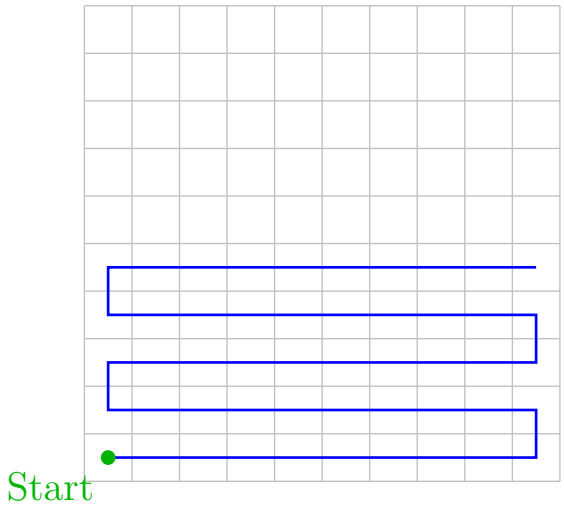
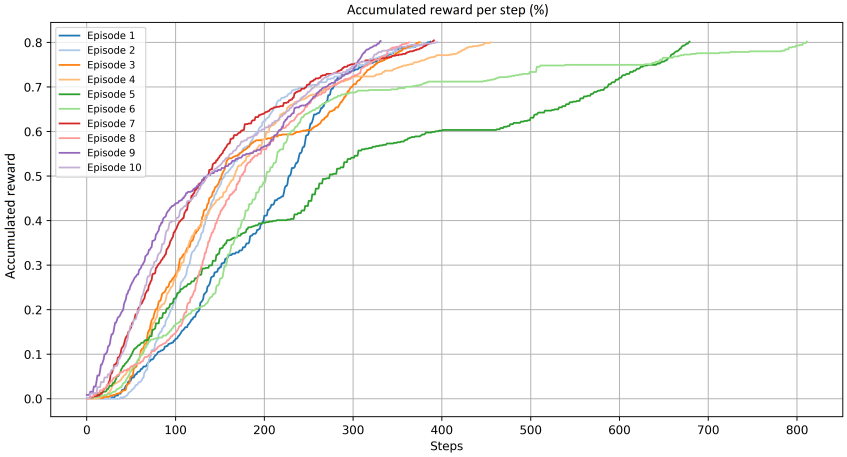
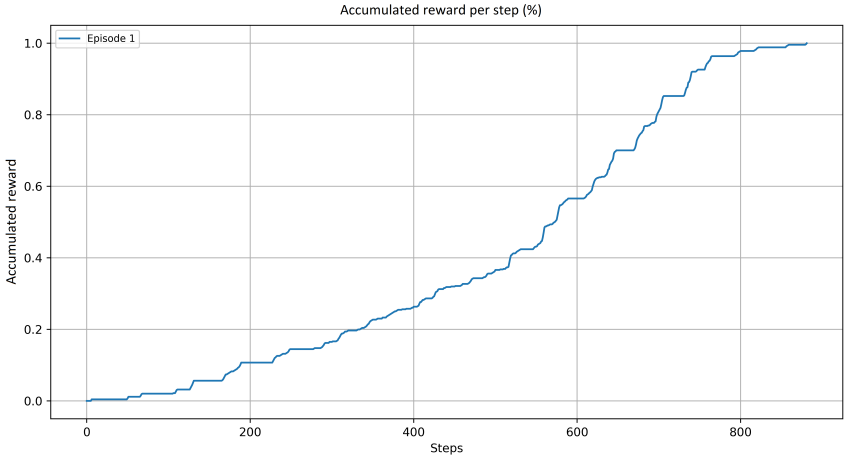


Figure 25: Brute force sweep

We will now compare our model to a brute force sweep method to reveal the pros and cons of both methods.



(a) Our model performing a mission on an untrained environment, prioritizing areas with higher likelihood of finding a distressed person



(b) Brute force sweep with an agent going from side to side, covering every location in the environment and using all 900 steps

Figure 26: Comparing our model to a brute force sweep of the same environment. The graph shows the accumulated reward per each step.

In fig. 26a it is demonstrated how our model dynamically prioritizes areas with a higher likelihood of finding a distressed person. It can be seen that the model consistently accumulates rewards quickly, with most episodes reaching the coveted 80% coverage within $300 - 400$ steps.

In contrast, fig. 26b shows a brute force agent that systematically covers every area in the environment. While this guarantees full coverage, it is far less efficient. It

takes nearly all 900 steps to achieve the same level of information that the intelligent model often reaches in less than half that time. In a search-and-rescue mission, time is of the essence. The brute force method does not adapt to the likelihood of finding high priority areas, resulting in significant time and energy waste. The type of search-and-rescue mission needed is up to the operator as both the brute-force and the intelligent model has its uses across different cases. However, our model could in theory also perform an intelligent and efficient 100% coverage with multiple agents, meaning complete coverage while still retaining efficient behavior. This could be the agents covering high priority areas first and then spreading out to perform sweeps of locations yet to be visited, making it a hybrid of our model and the brute-force sweeping model. This will not be explored in this project.

12 Discussion

There are several things worth reflecting on after running the experiments and reviewing the results. While the approach works well in many cases, there are also clear limitations that show up when the environment changes or becomes more complex. It is also worth thinking about how the learning method affects coordination between agents, and whether other approaches could improve overall performance.

12.1 Tests and performance

The experiments carried out in this project showed promising results for using a decentralized reinforcement learning in swarm based drone control. The agents were able to achieve high coverage within short time relative to the size of the environment. Due to the agent's limited observation space and simplistic design, it has fast convergence rate compared to large deep learning models. This is a desired result for the mission, and achieves the set goals for the agent. However, our model's performance is influenced by the structure of the environment, particularly the distribution of rewards. In environments with a strong reward-gradient i.e. high reward areas radiate outward to nearby locations. Then the agent is more easily guided toward regions with higher rewards. This reward gradient gives the agent a

continuous sense of direction, even after all POIs have been visited. In contrast, in environments where the reward gradient is low or where high reward areas are isolated without rewarding surroundings, the agent struggles to maintain effective exploration. Once nearby rewards are exhausted and all POIs have been visited, the agent is left with no clear signals to guide its decisions. In such cases, it often increases the time to reach the 80% reward threshold, as it lacks both local information and a global strategy to navigate to the remaining unvisited areas. The agent still avoids visited locations, meaning it will reach the threshold, but is unguided in the search of rewards. Another approach to improve the agents coverage above 80% rewards, would be to add memory and more shared information between agents. Our model uses minimal communication between agents, where the only information shared, is where each agent has been. Furthermore, they are only able to see each other when within range. If they were able to share their location with each other, throughout the environment, or inform other agents of areas that have not been visited along their path. Then they would have the needed information to guide them, when no rewards are nearby and no POIs left.

12.2 Shallow learning vs deep learning

It is plausible that the improvements mentioned above could be implemented using a deep learning model. Unlike Q-learning, which relies on discrete state-action pairs and explicit observations to learn decision making, deep learning models are capable of learning without the need of manual feature designs that are application-specifically engineered [5]. Allowing the deep learning model to learn complex patterns and generalizing across similar but unseen states. A deep learning model also open up for shared latent representations, where instead of broadcasting positions or POI status, the agents could exchange learned features or encoded intent, enabling a more coordinated swarm behavior. Beyond improved coordination, a deep learning model could be capable of a richer set of features, including a memory of previously visited areas and creating a spatial layout of the environment as it explores. This would allow intelligent backtracking to cover unexplored regions more effectively. However, these benefits come with trade-offs. Deep learning models are typically slower to

train and require more data and computational resources.

12.3 Decentralized vs Centralized

One major reflection point is the comparison between centralized and decentralized control. A centralized system might allow a planner to optimize the full mission globally, potentially reducing overlaps entirely and balancing coverage in real time. However, it introduces a single point of failure and reduces flexibility. In contrast, the decentralized approach used in this project allowed each agent to operate independently, offering robustness and scalability, though not without some inefficiencies. The best approach in real-world applications might be a hybrid model. This could be decentralized agents that synchronize periodically with a central coordinator to resolve conflicts and update mission goals.

13 Conclusion

The results of our work demonstrate that the proposed machine learning model effectively enhances search-and-rescue operations by systematically prioritizing regions with a higher probability of locating a distressed person. The model exhibits strong performance in coordinating multiple agents and dynamically optimizing their trajectories across diverse environments. Due to its relatively simple architecture, our method achieved 80% coverage within 400 steps, where brute-force methods achieved similar coverage within 900. However, limitations emerge in feature-sparse environments, where the lack of informative features in the heatmap can lead to unstable or undefined agent behavior following minimal training. This highlights the model's dependency on sufficiently feature-rich input data to ensure reliable generalization and operational stability. We have made the firmware publicly available on GitHub under the MIT license. (https://github.com/TheJoboReal/Bachelor).

References

[1] Khadija Ashraf and Ashwin Ashok. "P2P-DroneLoc: Peer-to-Peer Localization for GPS-Denied Drones using Camera and WiFi Fine Time Measurement". In: *IEEE Xplore* (2025). Accessed: 2025-05-20. URL: https://ieeexplore.ieee.org/abstract/document/10227756.

[2] Farama Foundation. *Gymnasium Documentation*. Accessed: 2025-03-07. 2025. URL: https://gymnasium.farama.org/index.html.

[3] Kasper AR Grøntved, Maria-Theresa Bahodi, and Anders Lyhne Christensen. "Automated Task Generation for Multi-drone Search and Rescue Operations". In: *International Symposium on Distributed Computing and Artificial Intelligence*. Springer. 2024, pp. 266–271.

[4] Vikas Hassija, Vikas Saxena, and Vinay Chamola. "Scheduling drone charging for multi-drone network based on consensus time-stamp and game theory". In: *Computer Communications* 149 (2020). Accessed: 2025-05-20, pp. 51–61. DOI: 10.1016/j.comcom.2019.09.021. URL: https://www.sciencedirect.com/science/article/abs/pii/S014036641930948X.

[5] Christian Janiesch, Patrick Zschech, and Kai Heinrich. "Machine Learning and Deep Learning". In: *Electronic Markets* 31.3 (2021). Accessed: 2025-05-26, pp. 685–695. DOI: 10.1007/s12525-021-00475-2. URL: https://doi.org/10.1007/s12525-021-00475-2.

[6] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* 4 (1996). Accessed: 2025-05-20, pp. 237–285. URL: https://www.jair.org/index.php/jair/article/view/10166.

[7] Robert J. Koester. *Lost Person Behavior: A Search and Rescue Guide on Where to Look—for Land, Air, and Water*. 1st. Accessed: 2025-05-28. dbS Productions LLC, 2008. ISBN: 978-1879471399. URL: https://outdoorsafety.cl/wp-content/uploads/2020/12/Media-Kit-Lost-Person-Behavior.pdf.

[8] Vladimir Nasteski. "An Overview of the Supervised Machine Learning Methods". In: *Horizons* 4.1 (2017). Accessed: 2025-05-20, pp. 5–17. DOI: 10.20544/HORIZONS.B.04.1.17.P05. URL: https://www.researchgate.

net/publication/328146111_An_Overview_of_Supervised_Machine_Learning_Methods.

[9] Ling Pan et al. “Reinforcement Learning with Dynamic Boltzmann Softmax Updates”. In: *arXiv preprint arXiv:1903.05926* (2019). Accessed: 2025-02-28. URL: https://arxiv.org/abs/1903.05926.

[10] Sarah Scoles. “Drones Are Doing the Dirty, Dangerous Work of Search and Rescue”. In: *Scientific American* (July 2024). Accessed: 2025-01-26. URL: https://www.scientificamerican.com/article/how-drones-are-revolutionizing-search-and-rescue/.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf: The MIT Press Cambridge, Massachusetts London, England, 2014.