Deep Reinforcement Learning for Intelligent Frontier Ranking in Search-and-rescue Scenarios

by

Taylor Bergeron

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Masters of Science

in

Robotics Department

by

May 2022

APPROVED:

Dr. Carlo Pinciroli, Advisor

Dr. Berk Calli, Assistant Professor, Robotics Engineering

Dr. William Michalson, Professor, Robotics Engineering

Abstract

Robot-based search-and-rescue is a compelling application due to its dangerous and time-critical nature. However, exploring real-world environments efficiently is difficult because of their complex and dynamic features, and of the need to pick the most promising unexplored areas. This research focuses on reinforcement learning for ranking frontiers (i.e., unexplored areas) for a search-and-rescue application in an indoor environment. My approach is based on Advantage Actor Critic (A2C), a reinforcement learning method which combines two prominent reinforcement learning (RL) algorithms. I implement the method using Robot Operating System (ROS) and the Gazebo simulator, and the OpenAI Gym to setup the reinforcement learning environment. I train a singular agent to find the optimal point to navigate to in a given environment, with the goal of exploring the environment as quickly as possible. Then, I implement the model on each robot in a swarm, to have the swarm explore the environment as quickly as possible. Experimental evaluation shows that my approach reduces exploration time by 8.4% with respect to traditional approaches. In addition, the software I developed in this project is the first to integrate popular SLAM ROS packages and the Gazebo simulator with OpenAI ROS Gym, providing the means to further this line of research with more sophisticated approaches.

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support.

I would like to thank Professor Carlo Pinciroli, whose expertise and support was invaluable in formulating a thesis research topic. His feedback on my research progress throughout the year helped me guide it to its final state. I would also like to thank my committee members, Professor Berk Calli and Professor William Michalson, for providing constructive feedback on my thesis.

I would also like to thank my collaborators on the project: Ashay Aswale, Tyler Ferrara, Connor Mclaughlin, and Peter Nikopolous. They were always willing to let me bounce my ideas off of them, and they helped build the swarm capabilities that allowed me to integrate my thesis into a swarm. Without them, I would not have been able to test my thesis on hardware.

I would like to thank my family: Mum, Dad, Kate, Nanie, and Grampy, for always supporting me in every way possible. I wouldn't be where I am right now without all the support. From playing Nancy Drew games together to sharpen my problem solving skills, to coaching Lego league so that I could learn my first coding 'language', to science team coaching and chaperoning so that I could investigate my true passion in the sciences, I have always had the love and support necessary to grow my passion for engineering.

I would also like to thank my boyfriend Luke, and my friends, Paulaine, Dan, Michaela, and Nick, for supporting me when I needed to study but also always being there to have fun when I wanted to decompress.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robot simultaneous location and mapping (SLAM) solutions have been a topic of research and development for the last 40 years due to SLAM solutions' ability to have a robot map out an area whilst keeping track of the robot's current location. SLAM solutions allow for a robot to explore an environment and create a map of the area during exploration by using the robot's various sensors to determine the contours of the area it is in. By using a SLAM solution, the robot is also able to estimate its location in the map by using the data from the onboard sensors in conjunction with historical data the robot collects during the exploration.

Robots need SLAM for various applications. Robots need SLAM to travel from one location to another to complete tasks, and to explore a scene for search-and-rescue purposes. While a robot is exploring it could constantly be using its camera to scan the scene, and if the camera feed is fed to a computer vision machine learning model, the robot could identify victims. If the robot identifies a victim, the robot can report back the location of the victim along with the victims image to first responders. This allows for first responders to stay safer, as they can directly go rescue the victim rather searching the entire environment to find the victim.

In the case of SLAM, a map is a human-understandable representation of the environment. Each part of the environment is represented by small square. Each square has a value associated with it, and that value represents if that small sliver of the environment is occupied by an object, unoccupied, or if that part of the environment is unknown since the robot has not travelled there yet. A frontier is the boundary between the explored areas of the map and the unknown areas of the map.

There are many environmental situations that can impact SLAM performance. Sensor noise increases the chances that the resulting map is not an accurate representation of the environment. Complex environments can suffer from loop closure errors, which is when a robot arrives at a previously mapped region of the map and fails to recognize that it is the same place. The robot is then unable to use the recognition of the loop closure to reduce uncertainty in the map estimates. Complex environments also cause longer exploration times, because the robot has more frontiers to choose from for exploration, and it might have to retrace steps multiple times in order to visit every place on the map. It can also be slow for a single robot to explore the environment due to having to move slow enough to gather accurate data. Hazardous environments, such as a building on fire or a building with a water based sprinkler system activated, pose risks to the robot. These risks could break the robots chassis or sensors, rendering it unable to explore the remainder of the environment.

Multi-robot SLAM provides the unique opportunity to remedy a lot of the single robot SLAM failures and drawbacks. By having multiple robots exploring the environment, there is no single point of failure. If multiple robots fail in some way during exploration, although the exploration will take longer by relying on a fewer number of robots to map the area, the map is still able to be completed. The parallelism that multi-robot SLAM provides decreases the total time until map completion. For complex environments, multi-robot SLAM is especially helpful, as robots can search separate areas of the map which allows for a decrease in the amount of times a robot re-explores the same area.

However, multi-robot SLAM is not without challenges. There exists the problem of which section of the map each individual robot should explore. This extends into which frontier each robot should choose to explore, so that each robot explores a separate area. This will reduce the amount robots will revisit an area, and it allows for the entire environment to be explored quicker. Traditional approaches to frontier selection are not appropriate because they lack optimization for the specific environment being explored. Often, a hard coded weight is chosen to use in the calculation to rank the frontiers best to worst. This weight is a constant value throughout the entire exploration and does not change, and it is used to rank frontiers. However, this weight is not going to always be the best weight at every point in exploration. It would be advantageous if the weight could change depending on the current state of the map.

1.1 Problem Statement

This thesis frames the frontier selection process as a reinforcement learning problem. I chose reinforcement learning over other applicable methods to solve the problem because I wanted to investigate if I could apply reinforcement learning to this scenario and create a usable solution.

Reinforcement learning is based on a framework in which an agent receives rewards for optimal actions taken, and receives a penalty for unwanted actions. During the training phase of the model, the robot can choose a weight to use in a frontier ranking calculation. The robot then can travel to the lowest cost frontier. The robot can gather metrics during the travel, such as time to the goal, total distance moved, and the end occupancy of the desired goal point. All these traits of the selected frontier can be fed back to the model and can inform future decisions. The exploration of an environment begins when the robot is first placed in the new environment, and spans until the map is complete and no frontiers are left. Each of these exploration cycles during training is referred to as an episode. The reinforcement learning model is trained to optimize the exploration by reducing time to map completion.

1.2 Contributions

This thesis provides a method of creating a reinforcement learning model that can output a changing weight for frontier ranking calculations. After the robot reaches a frontier point, the robot runs the current observations through the model to receive the best weight for that instant. The weight is not the same throughout the exploration. This frontier selection approach leads to a quicker total exploration time than traditional frontier selection approaches. The changing weight, which is used in the frontier ranking calculation, leads to a total reduction in search time of 8.4%.

This thesis also provides the following software contributions. I created a custom OpenAI Gym environment for TurtleBot3 reinforcement learning training. I forked the Gmapping repository, which is the specific ROS SLAM package used for this thesis. The implementation allows for Gmapping to be completely reset with a ROS message, which is published when the OpenAI Gym environment is reset at the end of an training episode. Since Gmapping dynamically changes the size of the map based on what is explored, and the model requires an input of a constant size, I implemented a map resizing node that buffers around the map if the provided map is smaller than the largest possible map size for the maze to be explored. I also implemented a re-settable version of Move Base, a ROS navigation package, to allow Move Base to be fully reset at the end of an episode.

Both industry professionals and researchers often use Move Base for guidance and path planning. Move Bases uses both the global and local occupancy grid to avoid obstacles effectively on hardware. The model is trained using Move Base to move point-to-point, which allows for the model to best fit the weight for Move Base. Training on Move Base allows for the robot to be trained on the full navigation pipeline that will be used in deployment. From current literature, another reinforcement learning model that includes Move Base in its training does not exist, instead other approaches use A* and other less robust methods. Other researchers use these methods since the methods can be directly written by the researcher, which allows for easier debugging. However, A* and other less robust path planning methods are more likely to encounter delays when exploring an environment due to unforeseen obstacles appearing when the robot is sent to a final point that is in an unknown area of the occupancy grid. Move Base handles sending a robot into a completely unknown area and avoiding unforeseen obstacles well by using the local and global occupancy grids to quickly plan new paths. Global occupancy grids is the entire occupancy grid, while local occupancy grids are a smaller square occupancy grid centered on the robot's current position.

One of the advantages of this generalized input is that it can be used on various sizes of maps as well as various map inputs. This allows for both individual robot maps to be provided to the model as well as the merged map created from all the robots maps.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Machine Learning Overview

Machine learning (ML) has been gaining traction in recent years and has become a buzz word due to its popularity. Machine learning is a discipline that you can use to create model to provide a desired output when given a specific input. This input can be, for example, an image, an audio file, or an array of integers, among other things. The input is then manipulated through various layers. "A layer is a container that usually receives weighted input, transforms it with a set of mostly non-linear functions and then passes these values as output to the next layer" [9]. The final layer can output various information, such as a classification of an object, a number which can be used in calculations, or a location of an object in an image, among other things. The generality of machine learning allows it to be applied to various problems.

Reinforcement learning (RL) is an unsupervised learning subset of machine learning. Unsupervised learning is when "the model is not provided with the correct results during training," instead feedback to the system is given through other methods [11]. For reinforcement learning it is through rewards. Reinforcement learning does not require labelled input and output data for training and testing purposes. Reinforcement learning focuses on training an intelligent agent to take actions in an environment which maximizes a cumulative reward. This rewards process rewards desired actions and penalizes unwanted actions [5]. The main intention is for the agent to observe its environment and learn the best action to take given an environment state, through trial-and-error. The model is trained through many episodes. In order to train through episodes, the environment and the agent (but not the model) must be reset after each episode. When an environment and agent is reset, they are both set to their initial states. The model trains on episodes repeatedly to converge to the best desired behavior given an observation state. The agent seeks long term and maximum overall reward to achieve the best solution.

The specific reinforcement learning method used for this thesis is advantage actor critic. In order to understand why this thesis uses actor critic, it is important to note the two main categories of RL methods. There are value-based methods and policy-based methods [21]. Policy-based and value-based methods are also referred to as on-policy and off-policy, respectively [29]. Value-based methods attempt to approximate the optimal value function. The value function π is the expected discounted reward when starting in a state and following policy π. The policy tells the agent what action to execute given the state in the environment. The higher the output of the value function, the better the action is to take. The policy gradient which updates the policy parameters is shown in Equation 2.1 [3]. J is the long-term reward function to maximize. θ is the policy neural network parameters, N is a mini batch size, which is the number of samples that are processed before the model is updated. $Q(s, a)$ is the state-action value function. s is the state and a is the action. π is the policy.

$$\nabla_\theta J = \frac{1}{N} \sum_{i=1} \nabla_a Q(s,a) \Delta_\theta \pi(s) \tag{2.1}$$

An example of a value-based RL approach is Q-learning. There is also policy-based learning which attempts to find the optimal policy without using the Q value as an intermediary step in the calculation. Essentially, the Q value is a maximum future reward [21]. An example of policy-based algorithms are Policy Gradients and REINFORCE.

These methods both have advantages and disadvantages. Policy-based approaches tend to work best with continuous data and have a faster convergence. On the other hand, value-based approaches are more sample-efficient and steady to convergence. The largest drawback of policy-based approaches is their inefficiency [29]. Policy gradients have a high variance of gradient estimates [53]. This variance is especially common in models with high-dimensional action spaces. This high variance is caused in part due to difficulty in value assignment to the actions which affected the future rewards [53]. Policy gradients are often estimated by Monte Carlo rollouts. The rollouts "estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy" [45]. To reduce the high variance of gradients, Wu et al. derived a "bias-free action-dependent baseline for variance reduction. Their baseline fully exploits the structural form of the stochastic policy itself and does not make any additional assumptions about the" Markov Decision Process (MDP), which models decision making and is the formal foundation of reinforcement learning [53]. However, reducing variance in policy-based models still remains a challenge for many researchers. Value-based methods have the disadvantage of instability, so the function approximation is easily perturbed by small changes to the inputs to the model.

Realistically this means value-based approaches require extensive hyperparameter tuning to produce stable behavior. Hyperparameters are arguments to a model that control the learning process, and the hyperparameters are set before the learning process begins and remain unchanged during training [32]. Although there have been implementations that create optimal outputs, there is ”little theoretical understanding of how deep Q-learning could obtain near-optimal objective values” [29].

2.1.2 Actor-Critic Methods

Scientists merged policy and value-based methods to reap the benefits of both methods while reducing the disadvantages. This is how the Actor-Critic (AC) method was created [21]. Figure 2.1 shows a schematic representation of the actor and critic interactions. The main idea is that value-based and policy-based methods are both used —one computes the optimal action based on the state observation and the other produces the Q values of the action. The actor part of the model comes first. The actor takes the input of the state, and the actor outputs the optimal action. The actor is policy-based, as the actor learns the optimal policy. The critic evaluates the action by computing the value function, which means the critic is the value-based entity. As the model trains, each of the models, the actor and the critic, train together and individually get better. In simple words, the actor decides what the best action to take is, and the critic responds to the actor with how optimal the action was and in what way the actor should adjust its model. Another reason Actor-Critic methods have become popular is because they use value approximators which replace the policy-based rollout estimates which impact the policy gradients variance. Although this replacement reduces the variance, it also results in an increased bias [29]. Bias is the difference between the average prediction of the model and the best value that is trying to be predicted. The trade-off between bias and variance is a common challenge in machine learning [51]. Actor-Critic models have proven that they are able to train and learn large and complex environments [21].

The actor is a function approximator and, as previously stated, produces the best action for a state. The actor can be a type of neural network, such as a convolutional neural network. The critic is a secondary function estimator, whose two inputs are the output action of the actor and the state observation of the environment. The critic outputs a Q value for the given information. Gradient descent is used to update both the actor and the critic's weights. "Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient" [17]. The formal equation for gradient descent is displayed in Equation 2.3 [17]. Gradient descent has two parameters: m, the weight and b, the bias. Gradient descent calculates the partial derivatives of the cost function, in Equation 2.2, with respect to the parameters.

$$f(m,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (mx_i + b))^2 \tag{2.2}$$

$$f\prime(m,b) = \begin{bmatrix} \frac{df}{dm} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N}\sum -2x_i(y_i - (mx_i + b)) \\ \frac{1}{N}\sum -2(y_i - (mx_i + b)) \end{bmatrix} \tag{2.3}$$

The weights of the network are updated at each step. As the actor and critic train, the actor learns the policy so it outputs better actions, and the critic increases its capability of evaluating those chosen actions.

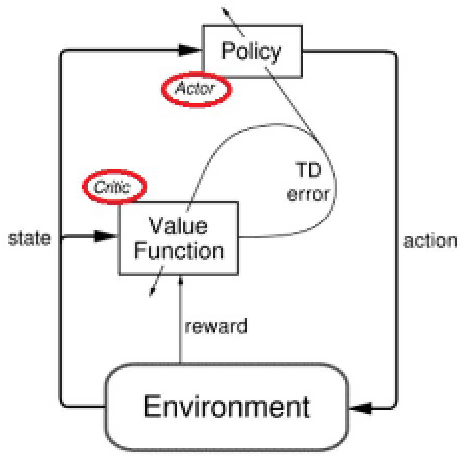


Figure 2.1: Actor Critic (AC) Schematic Representation [42]

To improve upon the Actor-Critic, Minh et al. introduced Advantage Actor-Critic (A2C) [27]. The Boltzmann exploration is used so the Q-values are decomposed into advantage function and value function [33]. The "Boltzmann exploration is a classic strategy for sequential decision-making under uncertainty, and is one of the most standard tools in Reinforcement Learning (RL)" [6]. For a simple implementation of the Boltzmann exploration, the probability of choosing an action is proportional to an exponential function of the empirical mean of the reward of that action. Advantage functions represent how optimal an action is when compared to other actions for a given input state. As previously mentioned, the value function represents how optimal it is to be at the input state. In Actor-Critic the critic learns the Q values, so it learns the advantage function and value function. However, in A2C, the critic only learns the advantage values. This means that the critic evaluates not only how beneficial an action is, but also how much better it could be. By training the critic on the advantage function, the high variances of gradient estimates that stem from policy-based models are reduced and the critic is more stabilized.

One of the drawbacks of almost any machine learning model is that the model takes time to train. In 2016 Google's DeepMind published the novel Asynchronous Advantage Actor-Critic (A3C) [26]. A3C maintains a policy and an estimate of the value function. A3C relies on parallel actor learners and accumulate updates which improve the training stability. A representation of the parallel workers can be seen in Figure 2.2, and Figure 2.3 shows how the original actor critic fits into the A3C model. Each of these learners are independent agents/networks and have their own individual weights. Each learner interact with a different copy of the environment, but in parallel. This means each learner's states are not the same at each timestamp, but, by training in parallel, they can explore more state-action pairs in a shorter time. This parallelization improves one of the main challenges of reinforcement learning, when the amount of state-action pairs are too large for the model to address each pair. When there are too many state-action pairs, the model is unable to provide an optimal action for every state. Since there are many learners exploring the state-action space at once, after training the end model is more optimal.

A3C uses the same methods to update the policy and value functions as A2C, but at a different point in time. The policy and value functions, meaning the actor and the critic, are updated after a certain amount of actions are reached or when a terminal state occurs. The learners periodically update a global network which has the combined weights. The updates do not happen at the same time, hence the asynchronous nature. After each update time or terminal action is encountered, the actor and critic update the global network then continue individually training. To train effectively, when an agent updates the global network, it then takes the global network's weights and updates its own. This allows each separate actor-critic pair to benefit from the other actor-critic pairs.

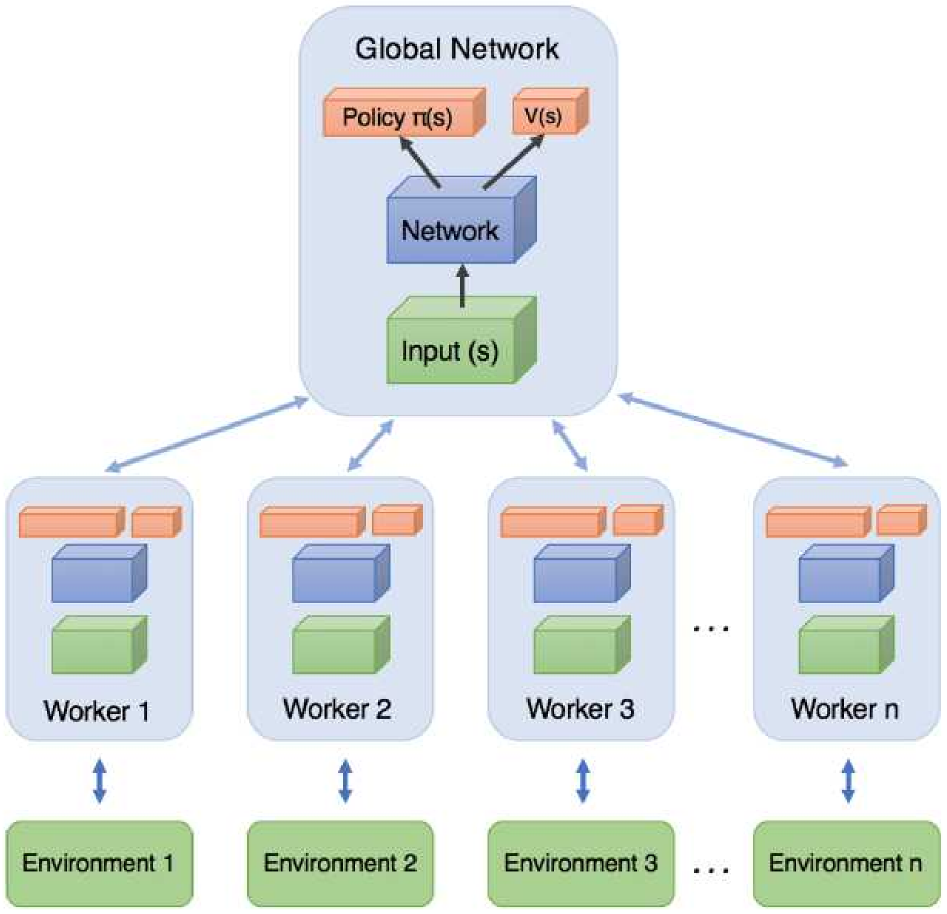


Figure 2.2: Asynchronous Advantage Actor Critic (A3C) Representation [42]

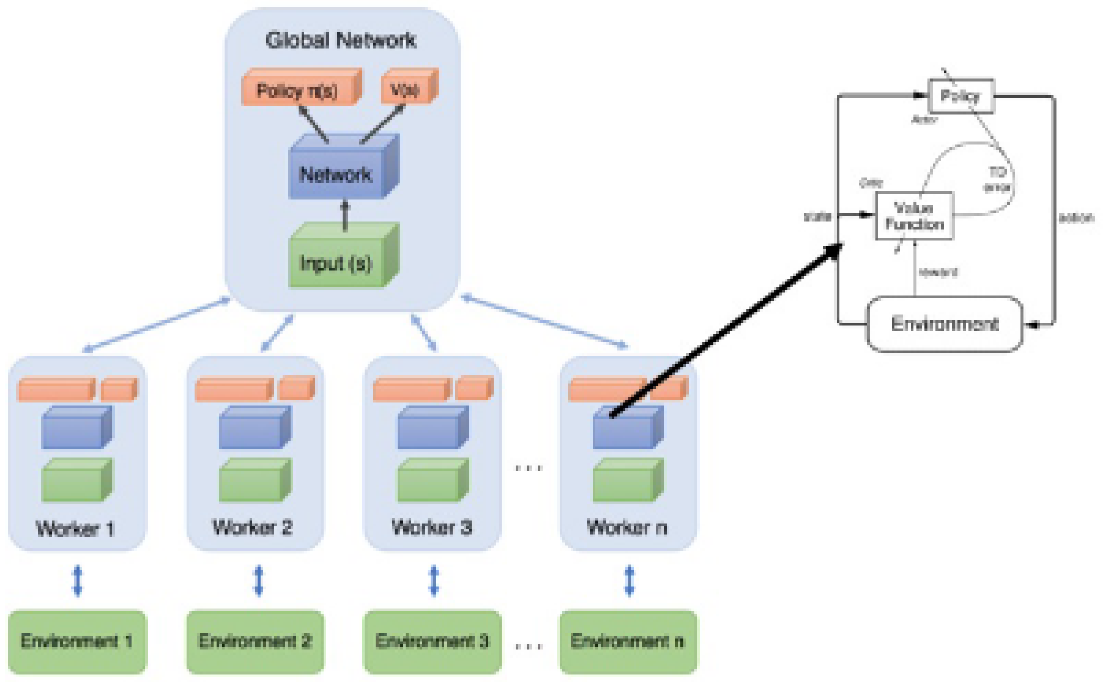


Figure 2.3: Asynchronous Advantage Actor-Critic (A3C) Representation including Advantage Actor-Critic [42]

2.1.3 A* Algorithm

Many of the works in the Related Work section of this paper refer to the A* algorithm, either as part of their implementation or as a baseline for path planning implementation evaluations. A* is a pathfinding algorithm which is used to find the shortest path between two points [1]. A* is optimal, so it can find a path as short as Dijkstra's Algorithm, but in shorter time [24]. A* combines logic from Greedy-Best-First-Search and Dijkstra's algorithms. It favors points on the path that are close to the starting point and vertices that are close to the goal. It uses $g(n)$ which is the cost from the starting point to a location and $h(n)$ which is the estimated cost from a location to the goal. $h(n)$ is the heuristic. A* balances both functions as it creates a path from the starting point to the goal point [39]. Each time it ensures that a vertex has the lowest cost, $g(n) + h(n)$. Pseudocode for the A* algorithm is shown in Algorithm 1 from [39]. Neighbors are graph nodes that are directly connected.

Algorithm 1 A*

```
1: frontier = PriorityQueue()
2: frontier.put(start, 0)
3: came_from =
4: cost_so_far =
5: came_from[start] = None
6: cost_so_far[start] = 0
7: while not frontier.empty() do
8:     current = frontier.get()
9:     if current == goal then
10:        break
11:    for next in graph.neighbors(current) do
12:        new_cost = cost_so_far[current] + graph.cost(current, next)
13:        if next not in cost_so_far or new_cost < cost_so_far[next] then
14:            cost_so_far[next] = new_cost
15:            priority = new_cost + heuristic(goal, next)
16:            frontier.put(next, priority)
17:            came_from[next] = current
```

2.2 Related Work

2.2.1 Frontier Selection and Environment Exploration

Although there has been significant research in the search-and-rescue space, there still exist shortcomings when it comes to speed of exploration. The concept of frontiers was introduced in research conducted by Yamauchi [54]. Yamauchi defined the frontier as the boundary between the explored open cells and the unexplored cells. This work is what much of the classic robotic exploration methods are modelled after. The author introduces a method for detecting frontiers in occupancy grids and navigating to those frontiers. Yamauchi tested his approach on a physical robot and proved it to work in cluttered environments with obstacles, proving the approach suitable for large spaces as well as tight crevices. If a robot can move to a point then it is accessible, and accessible space is contiguous, since a path must exist from the robot's initial starting position to every accessible point. This means that if all frontiers are explored then all of the environment will be explored. Yamauchi shows that frontier-based exploration is possible on a robot with noisy sensors and imprecise movements. In Yamauchi's implementation, the robot moves to the nearest accessible frontier to explore the environment. Although the approach works and is effective, the efficiency of the exploration could be improved with more optimization of frontier election.

Li et al. introduce an intelligent machine learning based approach to exploring an unknown environment [23]. They implemented a general exploration framework that consists of 3 modules: decision, planning, and mapping. The most novel part of Li et al.'s research is the deep reinforcement learning algorithm that uses a deep neural network to learn the best exploration strategy from the map at any point. The results showed that their algorithm, fully conventional Q-network with an auxiliary task (AFCQN), had a better learning efficiency and adaptability for unknown environments than the traditional deep Q-network (DQN) or fully convolutional Q-network (FCQN) [23]. The authors viewed exploration as a sequence decision-making tasks. The framework for the decision making module is a markov decision tuple consisting of the finite set of state, set of actions an agent can take, the transform distribution over the next state given the reward from the past state using the action, the reward from taking the action, and the discount factor. The discount factor balances how much the agent prioritizes rewards in the distant future versus those in the immediate future. They intended to maximize the expectation of the long term cumulative reward. The authors outlined a common problem plaguing model based exploration. When compared to empirical features, original sensor data has a very high dimension, which is how many attributes the dataset has [10]. High dimensionality increases the state space greatly. This makes it difficult for the model to cover all potential state-action combinations. Models with high dimensionality take an impractically long time to train. However, even if the model successfully trains after many runs in simulation, since the noise and sensor data differ greatly between simulation and physical systems, the model likely will not perform well when run on the physical system. The model will likely take sub-optimal actions. Also, if the model learns a control law in simulation, it is uncertain that the robots will have near perfect performance with the control law on hardware. In order to solve the control law transition from simulation to hardware complication, the authors separated the control and exploration logic. They put the control logic in the planning module and put the exploration logic in the decision model which provides the goal point. The planning model chooses the path from a robots current position to the goal point. This way, the control policy can use traditional optimal control, and the decision module can implement a deep reinforcement learning algorithm for choosing the goal point. The input to their model is the map, current robot position, and past robot positions. The current robot position and past robot position are derived from their mapping module. The paper's representation of the action space is particularly novel. The action space for the reinforcement learning model is discrete and consists of uniformly chosen points on the occupancy grid. Points in unknown space, free space, and obstacles are all valid actions, as seen in Figure 2.4 Their reward setup intended for the robot to explore safely and effectively, so they wanted to encourage the robot to choose a goal exploration point from the free space. Therefore, they penalize the robot when it chooses unknown space or an obstacle. The authors implemented a convolutional neural network (CNN) that estimates the value of each point in the grid map. The raw occupancy map is provided to encoder layers which provide an output with the same dimension of the action space. To estimate the Q-values they created a FCQN. The fully connected final layer in the FCQN estimates the advantage value of the terminal action and state value. The Q-value of each action point is found, and the action with the best value is the action that should be taken. As the robot explored, the input space to the model increased rapidly, since the map grew. The agent would need more time to analyze every possible state-action combination, which would dramatically increase training time and complicate the training process. The authors implemented an auxiliary task to enhance prior knowledge which improves training efficiency. The authors added this final branch to the the FCQN network, making the network a fully convolutional Q-network with an auxiliary task (AFCQN). The authors proved that the AFCQN model explored the region at an average faster rate, but the FCQN had a shorter average path length and a higher mean explored efficiency, which means the model avoids redundant movement. A disadvantage of this work is the authors only implemented this method on a singular robot navigating the space, and this means that their approach could not work on a merged map as the approach and training is not generalized enough.

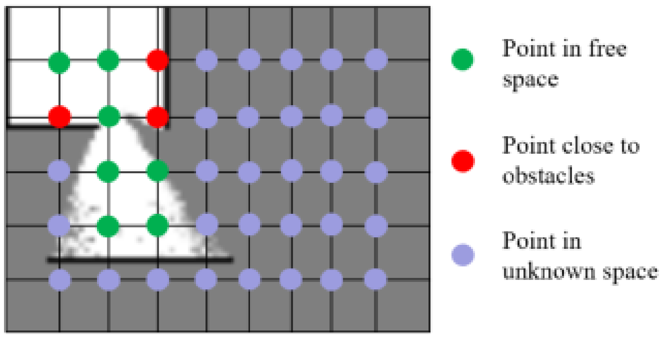


Figure 2.4: Sampled Action Space from Map [23]

Niroui et al. conducted additional research on using reinforcement learning for robot exploration [30]. The authors focused on a cluttered urban search-and-rescue (USAR) environment, as the environments are often unpredictable, to challenge their robot exploration approach. They determined that deep learning techniques would be the most appropriate for the environment and produce better exploration results than other state-of-the-art techniques. The authors defined better exploration as exploration that results in a quicker time to map completion. The researchers combined the traditional frontier-based exploration approach with deep reinforcement learning so that the robot efficiently autonomously explores unknown environments. The authors proposed the first deep learning frontier based approach for specifically search-and-rescue applications. The deep reinforcement learning (DRL) network the authors implemented integrates with the frontier exploration, which allows the robot to learn from the map exploration. The DRL approach is also robust enough to generate exploration strategies with varying degrees of clutter in the unknown environments. The main goal of the research was to maximize the robots information gain early in the exploration process. The researchers trained the model on various sizes and layouts of maps so that the robot exploration stack will be robust to various environments. The results showed that the authors' deep learning frontier exploration method was able to explore the map at a faster pace than traditional exploration approaches. The authors noted that when implemented in the real world, their research will enable a robot to identify victims quicker.

For the implementation, the authors used various modules which interacted with one another, as seen in Figure 2.5. The authors used a TurtleBot for the implementation, and ROS and TurtleBot Stage to simulate the environment [52]. The world model contained a 2D occupancy grid of the environment which was created from a 3D sensor. The exploration model uses the occupancy grid and the robots pose to choose the best frontier location to travel to. That desired point is sent to the Navigation module which creates a path to the robots position to the goal point by using A*. The robot uses Move Base to traverse from point to point along the A* path.

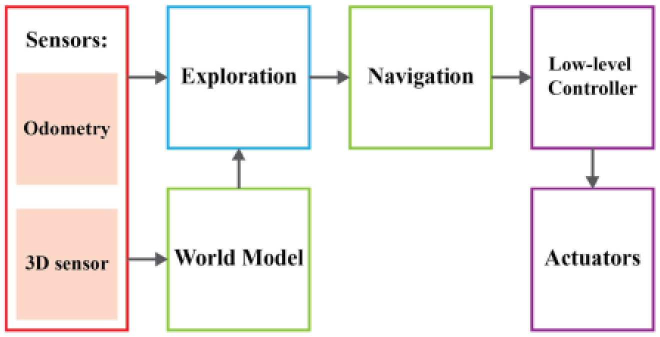


Figure 2.5: Modules for Robot Exploration in USAR [30]

The deep learning logic was contained within the exploration module. The authors implemented an A3C network for the frontier exploration, with the architecture represented in Figure 2.6. The input to the model is 3 64x64 matrices. The three matrices represent the map, the robot's location in (x,y), and the frontier locations in (x,y). The authors scale any size occupancy grid to 64x64 by separating the map into regions and separating the cells into three categories: open, occupied, or unknown. Niroui et al. then multiply the number of cells in each category by a weight: $weight_{\text{open}}$, $weight_{\text{occupied}}$, and $weight_{\text{unknown}}$. The region is assigned to the cell category with the highest weighted sum.

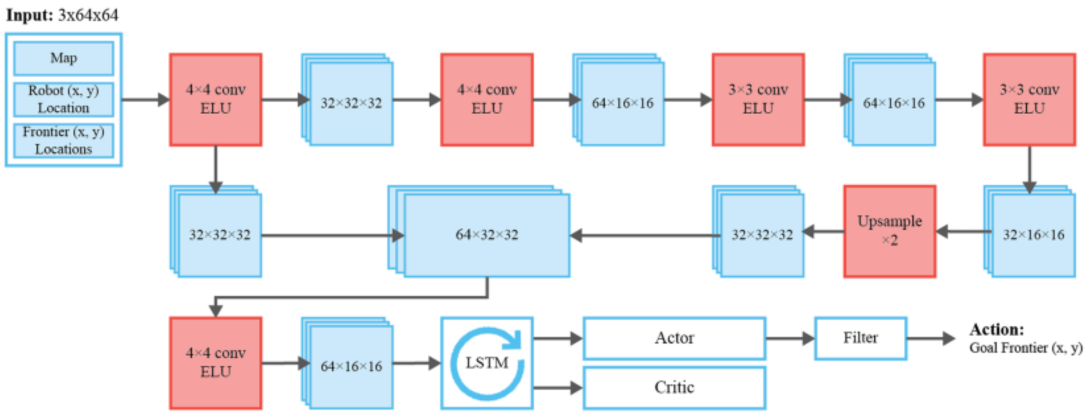


Figure 2.6: A3C Network for Frontier Selection in USAR [30]

To find the frontier locations, the authors split the open and unknown cells from the occupancy grid into two different layers. In order to get the frontier boundary, the unknown layer is dilated to overlap with the with the open cells layer, and element wise operations are conducted between the two layers. The frontier clustering algorithm clusters various frontier boundaries into groups using k-means. k-means clustering separates n observations into k clusters [38]. The end result is a list of locations, with each location representing a frontier boundary.

The network architecture begins with multiple convolutional layers. After each convolutional operation, the authors call an exponential linear (ELU) activation. To use previous robot states in the model's decision making, Nuroui et al implemented a long short-term memory (LSTM). The Actor-Critic layer uses the output of the LSTM. The actor layer outputs a single parameter, and that parameter represents the $weight_{\text{output}}$ that is used in the frontier calculation cost, in Equation 3.1. The filter layer uses the $weight_{\text{output}}$ to rank the frontiers and choose the best one, meaning the one with the lowest cost. d in the equation represents the normalized distance from the robots current location to each frontier location as measured by A*. g is the normalized potential information gain at the frontier location, which is the number of unknown cells that encapsulate the frontier boundary in the occupancy grid. The robot chooses the frontier with the lowest cost to explore. The model aims to maximize the authors' objective function, which is the total information gain along the robots exploration path.

With regard to network specifics, the authors have 10 workers in parallel training the A3C model. They implemented a test agent to monitor the training. The test agent was a single robot starting at a random location in a random cluttered environment. The authors implemented the randomization to avoid overfitting to one environment. A state transition for the advantage actor critic model occurs when the agent navigates to the selected goal point. The exploration occurred until the map was completed or a maximum step size was reached. The authors set the rewards structure so the highest rewards occur when an agent completes the map and when the robot has a high information gain. Therefore the $weight_{\text{output}}$ of the network is tuned to maximize information gain. Information gain is the amount of map explored during traversal from one frontier to another. The authors' model took 119 hours to train with the training agents executing a sum of 3,570 episodes. This is an example of high efficiency training for reinforcement learning. The authors proved their approach was scalable on various size maps, as the approach worked on 20x20 m^2, 40x40 m^2, and 80x80 m^2 USAR environments. Although the authors proved that their method worked, their implementation was vague which made it difficult for me to replicate. Their approach proved to work on a single robot, but they did not implement it on a swarm of robots, so it is unknown if their trained approach would work on a merged map of multiple robots executing search-and-rescue exploration at once.

Chen et al. implemented a novel approach to utilize the local structure of the environment while predicting the best location for the robot to traverse to next [7]. The optimal location is the action that allows for optimal sensing. The trained DRL model produces optimal or near-optimal exploratory sensing action pattern with improved computation efficiency.

The robot learns the exploration policy using an information-based reward given during exploration. The robot is trained on various indoor environments. The robot's potential future reward is also considered in the action selection decision. This means the action choice "balances short-term maximization of mutual information (MI) with the potential for long-term maximization of future rewards." [7] The authors aimed to leverage the shared characteristics between various indoor environments, such as office buildings, factories, and parking garages. The authors implemented Recurrent Neural Networks (RNN) to train the robots through a continuous map input [41]. The authors modified the traditional Deep Q-learning (DQN) algorithm by adding a dropout layer to avoid overfitting. To accelerate the policy training process, they implemented a Bayesian exploration strategy. The approach includes a training phase and a testing phase. In the training phase the agent optimizes the parameters at each time-step using a reward, by using reinforcement learning with the input of the occupancy grids. During the testing phase the robot uses its learned policy to predict the best sensing action, a position to traverse to, as a function of the current state observation. The framework for training and testing can be seen in Figure 2.7.

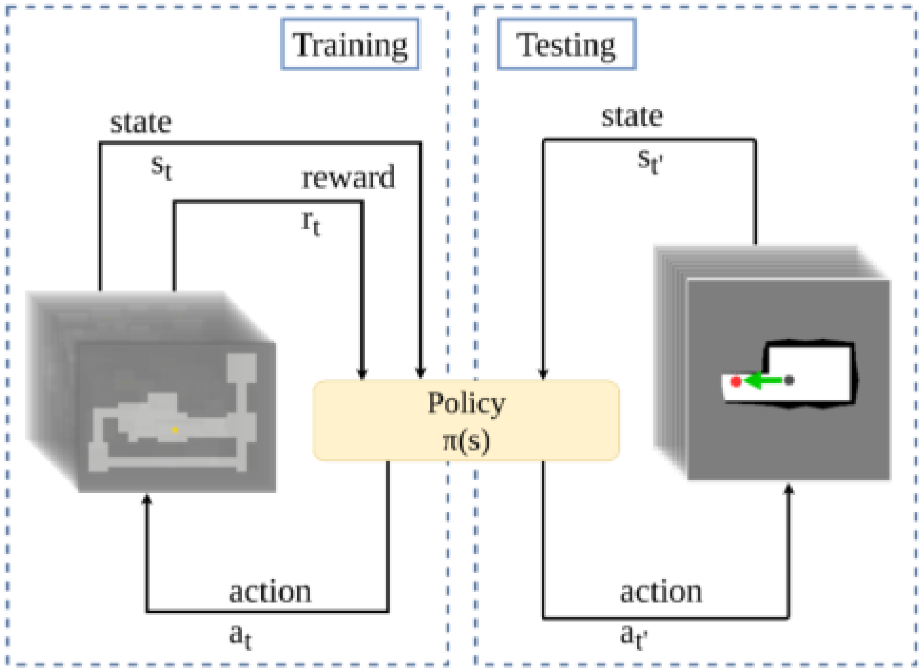


Figure 2.7: Training and Testing Frameworks For Self-Learning Exploration [41]

The authors use only a small occupancy grid for the input to the model, since with a full map input their model would not converge. The desired action point for optimal sensing is a open point on the occupancy grid within a specific radius of the robot. This radius is the limit of the locally visible portions of the 2D occupancy map. The authors chose this approach since all open locations within the visible nearby portions are viable uniformly sized regions, and any point in the overall map might not be viable. This approach is scalable and computationally efficient. Regardless of the size of the environment to be explored, the sensors will always have the ability to sense the same radius, so the input size to the model does not need to change. This approach is computationally efficient, since there is a small set of input data to the model, so little transformation needs to be done on the input local map in order to get the DQN to converge on the best solution. The input map and the possible actions can be seen in Figure 2.8.

The authors compare their results to an approach purely based on the Mutual Information (MI) strategy for finding the best action. The MI strategy "explicitly evaluates the expected information gain of every possible sensing acting, by projecting the sensor's rays throughout the map at each candidate configuration, and choosing the section action that offers maximum expected MI" [7]. However they do not compare it to any other methods that train on the global map, so it is difficult to ensure how quickly the robot actually maps out the environment.

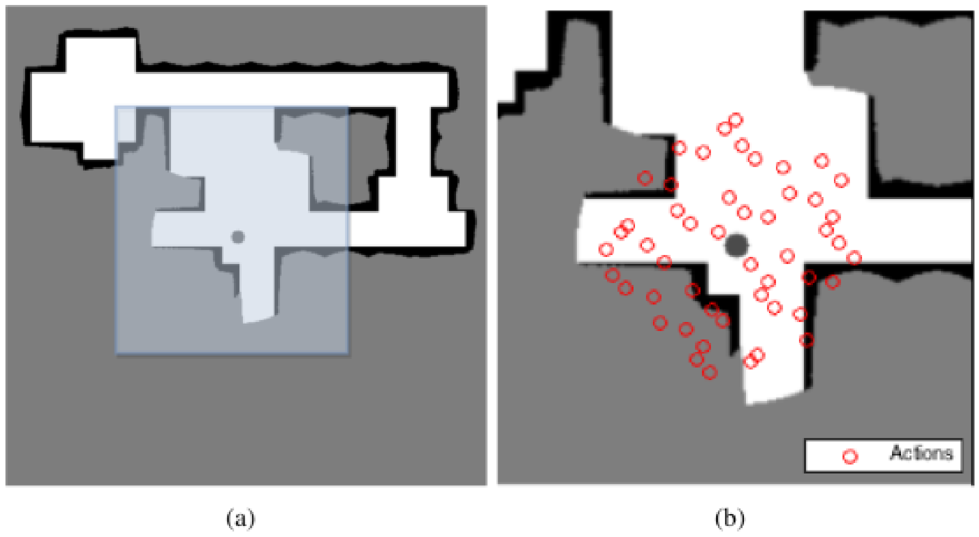


Figure 2.8: a) Shows the Local Map in Grey Overlay Extracted from the Global Map which Represents the State Space b) The Robot's Action Space in Red [41]

Research continued on ways to use machine learning to determine the best goal candidates for frontier based robot exploration [13]. Faigl et al. researched this topic by representing the occupancy grid as a self-organizing map on which to conduct the travelling salesman problem (TSP) with neighborhoods. The authors studied the exploration as a repeated coverage problem of the map frontiers, during which the minimal number of goal candidates is calculated along with the expected cost of travelling to the goal candidates.

The authors represented the frontiers as Yamauchi did [54]. The authors acknowledged that utility functions for ranking the frontiers are more useful than selecting the closest frontier; however, they present a drawback of the utility function approach which only considers the immediate benefit of selecting the next best goal at a specific step, without planning for the overall search.

For this research, the authors extended their two previous works by using self-organizing maps (SOM) as a solution to the watchman route problem [12] and the TSP with neighborhoods [14] and applied it to the robotics exploration problem. At the time of the paper, the authors determined this was the first SOM based solution to the robot exploration problem with limited environment visibility.

The authors represent the environment as an occupancy grid. The exploration process is a simple state machine. At each timestep the robot calculates the most recent occupancy grid. The robot detects the frontier cells using Yamauchi's logic in [54], and then from that determines the goal candidates. The next goal the robot selects is the goal with the lowest TSP distance cost. The robot navigates to the goal, gathers the newest sensor data for the occupancy grid, and repeats the exploration process. Once there are no reachable frontiers, the process terminates.

Faigl et al.'s research specifically addressed the generation of new goal candidates in the state machine. The optimization value the authors consider is the time necessary to explore the entire environment. The value is the distance travelled by the robot until all the frontiers are covered. They refer to this length as L. Although L is not directly optimized, the approach is based on the selecting the shortest TSP paths at each timestep, so this approach produces an improved performance when compared to the utility function approach.

The advantage to SOM based approaches is the robot does not need to travel to each of the frontier points. The method is designed to determine goal candidates that are on a shortest closed path that connect them all, and along this path all frontiers can be covered. The design is based on the idea that it is not necessary to visit a specific frontier to explore new area round it. Therefore, the goal candidates are locations at a specified distance $\rho\prime < \rho$ from the frontier boundaries. $\rho\prime$ is a goal candidate's distance from the frontier boundaries, and ρ is the maximum allowed distance a goal candidate can be from the frontier boundary. Each frontier cell is represented as a set of possible grid cells from which the frontier can be explored. Therefore the problem is able to be formulated like a TSP neighborhood to find the shortest path that allows coverage of all the frontiers.

The SOM is created by two layered competitive neural networks (NN). The input layer is a 2D vector of frontiers and the output is a 1D array of neurons. The frontiers used as input are coordinates in the occupancy grid, and the array of neuron weights as output represents a path in the occupancy grid. This path is denoted as a ring and it is the sequence of shortest paths between the ordered neuron weights. The output layer is a list of straight line segments, the ring, that is the shortest path in the environment that connects the various neurons. A visual of the logic can be found in Figure 2.9.

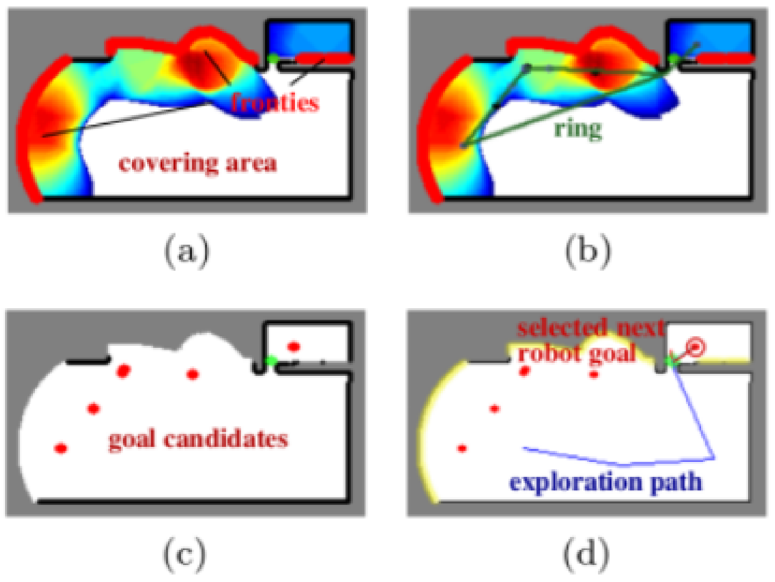


Figure 2.9: The SOM and TSP Based Exploration Process [13]

To get the best location for exploration, the authors implement a virtual neuron. The virtual neuron is the closest point on the ring to the frontier, if the goal candidates do not already include the closest point on the ring to the frontiers. As seen in Figure 2.9, the end of the exploration path does not stop at a goal candidate, but rather a point that was on the green ring.

The results of the paper indicate that there is an exploration efficiency improvement by using the TSP-based frontier approach. The authors' SOM-based exploration method proved more efficient than a greedy and k-means method of frontier ranking and exploration. The SOM-based method proved better in open space environments with a long sensor range, and in maze like office environments with shorter sensor ranges. This approach, although clever, does not seem to always provide the best location for exploration. Instead it is a complicated center of the frontier, which may not be the best at all times.

Lei et al. [46] introduced an exploration approach based on a DQN framework for obstacle avoidance and robot control. The DQN is separated into a supervised deep learning model and a Q-learning network. The authors conducted the experiments in simulation using ROS and Gazebo and using a TurtleBot robot. They show their approach works on various corridor-based environments. The final environment that they show their approach on is different than the pre-training environment. The authors mention that this was the first approach that used raw sensor information has been used directly with reinforcement learning to create an exploration strategy.

The input to the network is the depth image from the robot's RGB-D camera. The first three layers in the network are convolution layers with pooling and rectifier layers. After, feature maps of each input are fully connected then the authors send the connected feature maps to the softmax layer. The softmax layer chooses one of the three directions for the robot to go. An image representing the model can be seen in Figure 2.10

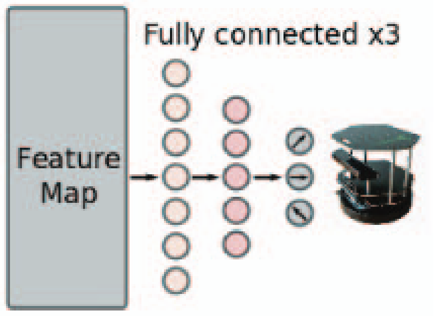


Figure 2.10: The Model Representation that Outputs The Robot Direction [46]

The reward structure for the model rewards non-collision states with the same score. This could be improved by rewarding non-collision states that lead to a quicker exploration with a higher reward.

Although this approach does not consider frontiers, it is interesting that the robot both learns a reliable control method and also an exploration strategy. The authors prove that their training process works and their robot can explore an environment without colliding with obstacles. However, the authors do not compare their approach to another baseline approach, such as the work of Yamauchi in [54], so it is difficult for the reader to understand how quick and efficient their exploration is comparatively.

2.2.2 Collaborative Multi Robot Exploration

Although strides have been taken in research towards decreasing search time for a individual agent, researchers can create a large increase in exploration efficiency by using multiple robots to search environments.

In 2000, Burgard et al. introduced a novel way of coordinating a team of multiple robots to effectively explore an environment [4]. Their goal was the same as common single robot exploration strategies: to minimize the overall exploration time. The authors focused on calculating optimal goal points for each individual robot to traverse to and explore so that the robots are each searching separate parts of the environment.

The authors use a probabilistic approach for the selection of exploration points. The multiple robots consider the costs of reaching the goal location along with the utility of the goal location. The authors calculate the utility cost by using the size of the unexplored area that a robot is able to sense once it has reached the point. When a point is assigned to a robot, the utility for the same point is reduced for all the other robots in the group. This method is how the robots choose different target locations while searching.

The authors implement a method to merge the individual robot maps into a global map for all the robots to access and choose their goal locations from. To create the list of goal points for the robots to choose from, the authors use the same frontier approach as [54]. The system is centralized. To estimate the utility, the centralized unit estimates the expected amount of area that will be discovered when the robot arrives at the goal location. The centralized unit does this based on observations seen in practice. A robot exploring a large open section of the environment can find more than a robot exploring a narrow part of this environment. The robot can find more because once a laser scan from a LiDAR unit hits a wall, its observation in that line of sight end. The method continues as it next minimizes the utility of other unexplored frontier points near the selected goal location. The method then uses these reduced utilities to calculate the goal frontier locations for the other robots in the group.

The technique was tested in both real world in simulation. The authors gather data on a robot exploring individually, an uncoordinated robot team exploring, and a coordinated robot team exploring. As hypothesised, the slowest exploration time was the individual robot, followed by the uncoordinated team, and the best exploration time was the coordinated robot team.

A disadvantage of this approach is that, because each robot separately explores the environment and the robot team does not consistently meet back at a central location, there is not much map data redundancy. Although not directly addressed, the assumption is that the map is merged at each timestep. However, this assumes the robots will always have communication. If a single robot is out of communication and then fails, all of that individual's data is lost, and the overall search time will increase dramatically since another robot will have to re-explore the previously explored space in order to complete the map.

In [43], Simmons et al. created an exploration algorithm that coordinates robots by minimizing the potential of overlap in mapping for the various robots in the group. The authors contain many of the authors of [4] and this paper was published three months after [4], so many similar ideas are used. Some key differences is that rather than have the central unit tell each robot what goal point it should go to, the robots place bids. Each time the robot receives an update from the central mapper, it constructs a new bid. The bid consists of frontier cells along with the costs and information gains associated with each frontier cell. The central executive unit considers each bid, and after all bids are received it tasks each robot to one of the listed frontier cells. The unit takes into account their bid and the overlap in coverage. So although a robot might select a certain frontier as the best one to explore, the central unit might assign it another one for reasons such as overlap in exploration with another one, little information gain, etc. Since the bidding occurs each time the map is updated, a robot could be going to one frontier point, but will re-bid on the frontier points, and could be tasked to another one mid point execution. The exploration of the entire map ends when there is no more information to be gained.

This approach was thoroughly tested in simulation and hardware, as part of DARPA's Tactical Mobile Robot project. The testing occurred during a 5-day period with the robots exploring an empty hospital. The authors proved the same results as in the previous paper [4] but with an interesting twist. With their algorithm and a maze-like environment full of narrow hallways with no obstacles, they found that a two robot team is quicker on average completing the map than three robots. This situation occurs because the three robots end up interfering with one another. This method of exploration, although more complex, still would suffer from the same map data redundancy failure as [4].

With regard to robot collaboration, some of the best algorithms are inspired by nature. Kamalova et al. created a robot exploration method based off of biology [20]. They implemented an algorithm that creates way points for a swarm to explore based off of the movements of whales and grey wolves. They also implemented a third algorithm, the particle swarm optimization algorithm [37]. The authors focused on the problem of exploring an area with uncertain conditions where precomputation and prediction are impossible. To find the optimal solution for the robot, stochastic optimization techniques were implemented, which are techniques that generate and use random variables. As the robot explores the environment it updates its map. ”The points of the frontier line are assumed as the swarm population with their own positions and costs, which allows computation of the next global waypoint” [25]. The nature-inspired algorithms determine the waypoints.

The objective of the authors implementation was to constantly have the robot moving in unknown environments without any initial parameters, and build a map through the process. The robot will move to each of the waypoints, and through that movement a map of the environment will be created.

The robot uses the waypoint algorithm avoid obstacles and explore. The robot creates 5 local waypoints, seen in Figure 2.11. The robots use the local waypoints to change their current travel direction as a last minute obstacle avoidance measure. Their global waypoint for exploration purposes still remains the same. After the robot arrives at the local waypoint, it will continue to a new global waypoint. For the exploration process, the robot begins exploring in a straight line with no global goal point. The first global waypoint is only created once a robot encounters an obstacle. The computation for the global waypoint is based on the frontier points. The input to the global waypoint generation algorithm is the frontiers, calculated in the same way as in [54], but the global waypoint generation relies on one of the three biology inspired algorithms. All of the three biology-inspired algorithms use the same global and local waypoint method for obstacle avoidance. The flowchart for the waypoint switching strategies can be seen in Figure 2.12.

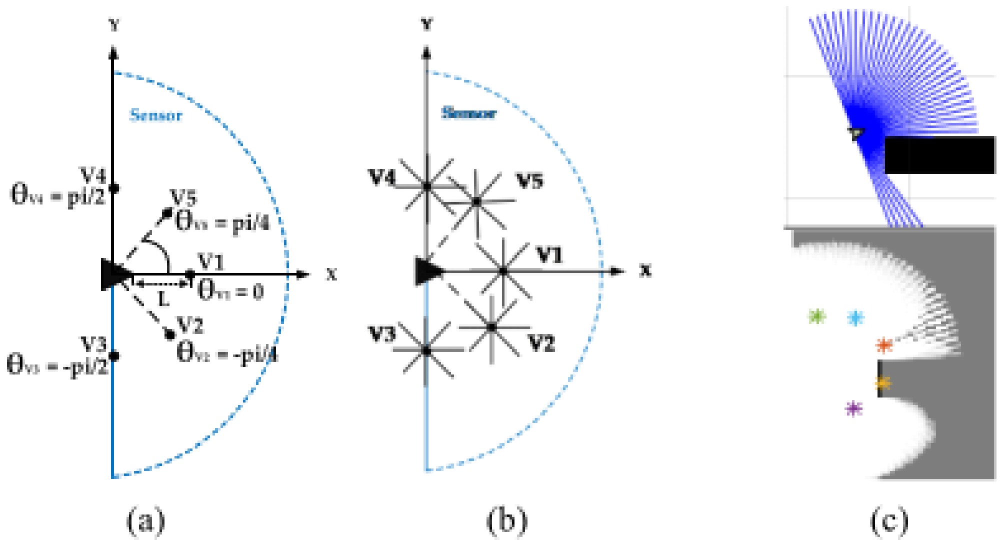


Figure 2.11: a) The Robot's Position and Five Local Points in Distance L with angles θ b) The Additional Ray Beams for Monitoring Obstacles in Intervals c) Local Waypoints Example with An Obstacle [20]

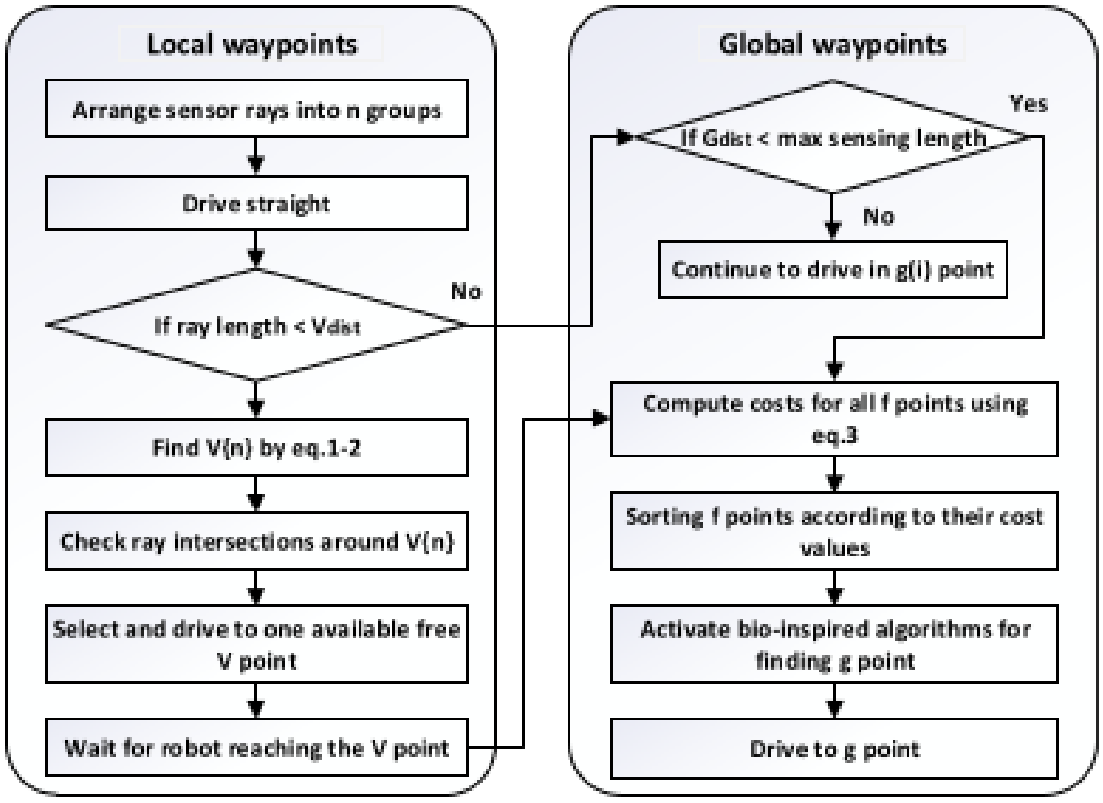

Local waypoints
Arrange sensor rays into n groups
Drive straight
If ray length < Vdist
No
Find V(n) by eq.1-2
Check ray intersections around V(n)
Select and drive to one available free V point
Wait for robot reaching the V point
Global waypoints
If Gdist < max sensing length
Yes
No
Continue to drive in g(i) point
Compute costs for all f points using eq.3
Sorting f points according to their cost values
Activate bio-inspired algorithms for finding g point
Drive to g point

Figure 2.12: Waypoint Switching Flowchart [20]

The first of the nature-inspired algorithms, the grey wolf algorithm, is based on hunting strategies of wolves, specifically the roles and the stages of the hunt. The authors implement social hierarchy in the algorithm, along with the four stages of hunting: encircling prey, hunting, attacking prey, and the search for prey. The robots have three different leadership levels: alpha, beta, and gamma. The remaining robots that are not assigned a leadership role are denoted as omega and follow the three leaders. The encircling prey method is the robots' attempt to find the global solution, and spread out across the map as much as possible to "encircle" the map. The hunting operator represents that the three leader wolves know the strategy better than the other wolves. Each of the leaders decides their own solution as to the best path and decision, and three leader robots determine the priority of the waypoints. There is no priority difference between the three leadership levels. The attack for prey stage is when the robots attempt to converge to one location and come together, and the search for prey is when the robots try to diverge from one another and explore more of the map.

The whale algorithm is based on the population-based meta heuristic algorithm [25], which replicates the foraging behavior of humpback whales. The algorithm uses three models: encircling prey, spiral bubble-net attacking algorithm, and searching for prey. The intricacies of the algorithm differ from the grey wolf based approach. Particularly, there is no hierarchy in this approach, but the goals of each model are the same. The hunt begins with the robots searching for prey and the swarm of robots will follow a randomly selected robot. Searching for prey means the robots diverge, and encircling the prey entails the robots surrounding the prey on the perimeter of the map. The attack stage is when they all converge and move into the same space.

Finally, the authors implemented particle swarm optimization (PSO) [37]. This is a stochastic optimization method that replicates the social behavior seen in bird flocks and fish schools. A single robot is denoted as a particle. Each particle has a position and velocity, and this velocity causes the swarm to move in a particular way. The robot shares its personal best solution and the global best solution. At each iteration a new particle is evaluated by the objective function which returns the cost of the solution. This cost is then compared with the prior personal and global best solutions. If the new cost is better than the prior, then the particle is stored as the personal best solution. If that personal best solution is better than the other particles' personal best solution then the best personal best solution becomes the global best solution. These costs relate to the velocity and allow the swarm to move together in the same direction or diverge depending on the cost.

All algorithms were implemented and compared in simulation. The grey wolf optimizer algorithm was the only algorithm tested in hardware. All the methods resulted in the entire map being explored. The PSO exploration had the least amount of waypoints and the total distance travelled by the swarm was less than the other methods. For evaluation, authors split the waypoints into avoidance and exploration waypoints. The grey wolf algorithm had more waypoints than the other algorithms but the robot reached them quickly because the distance to the waypoints was minimal. The obstacle avoidance method caused the extra waypoints and therefore minimizes the efficacy of the exploration.

Although the robots explore the environment, it is uncertain to understand the overall efficacy with no comparison to a baseline, such as random walk. Therefore, it is uncertain how these algorithms compare to other state of the art exploration methods for search-and-rescue scenarios.

Christianos et al. decided to increase the exploration speed of DRL based exploration by implementing the DRL approach on a swarm with shared actor-critic reinforcement learning [8]. During the training episodes, as the robots are exploring the map, the experience is shared amongst the worker agents. The algorithm, Shared Experience Actor-Critic (SEAC), implements experience sharing by combining the gradients of the different robot learners. The experience sharing is especially useful in environments with sparse rewards since sometimes, even for a single robot, it is difficult for the model to converge. The authors evaluated SEAC in various sparse reward environments and used a swarm of robots to explore the space. SEAC consistently outperforms baselines and state of the art algorithms by training in less steps and SEAC converges to choosing actions with higher rewards. The authors state that because of the difficulty of the environments, the shared experience between the multiple agents is what causes the model to converge and the swarm to effectively map the area.

The authors compare SEAC to baselines, which makes it clear how their algorithm performs in comparison. One of the baselines is the Predator Prey algorithm which is similar to the grey wolf algorithm from [25]. Although the authors explained SEAC, they did not outline the inputs and outputs to the model, nor the makeup of the inner layers. This would make implementing the authors version of SEAC difficult for other researchers.

2.2.3 Deep Q-learning for Path Planning

The majority of Q-learning research for swarms of robots has focused on dynamic path planning and obstacle avoidance. Although not directly related to the research for this thesis, the concepts and the improvements some of the authors make with regards to training time, such as pre-training, are pertinent.

Bae et al. created a multi-robot path planning algorithm for exploration that uses deep Q-learning along with CNNs [2]. This approach focuses on the path planning and obstacle avoidance aspect of exploration. Each robot navigates independently but collaborates with the other robots for efficient exploration. The coordination relies on the robots' current state, such as their pose and velocity. Unfortunately, many current methods for multi-robot exploration struggle with robot movement planning, since the robots have difficulty classifying other robots in their sensor range as either obstacles or other robots. The authors apply DQN to the robot exploration algorithm and CNN for analyzing the situation so the robots understand their surroundings more effectively. The CNN analyzes the state a robot is in by using the information from the robot's camera as it explores. The robot makes navigation decisions using the DQN which analyzes the current state of a robot.

In order to have the CNN effectively identify features, a pre-processing step occurs. The CNN is trained to collect and learn certain features in the image.

The first step in the authors algorithm is to have the image information from the robots camera to be passed to the CNN. First, the CNN extracts feature information from the image. This means that relationships between features and objects can be found. The algorithm uses the goal value of the DQN to calculate the loss of all actions taken by the robot during the training phase. For feedback to the DQN, the algorithm compares exploration distance to what it would have been with using A*. This comparison value is referred to as the compensation value. The agent learns by trying to always increase the compensation value. Therefore, if the current searching travelling distance increases to more than what it would be with the A* algorithm, the compensation value decreases. The authors intend for the model to converge to behavior with a large compensation value, which is a shortest distance exploration, akin to A*.

The authors tested their approach on different maps. The robot begins in one of four randomized orientations, and the robots starting position is in the upper north east corner of the environment. The number and location of the objects in the environment are randomly chosen. The authors' CNN and DQN approach to exploration is compared to A* and D*.

The results show that the authors' proposed algorithm, which uses DQN and a CNN, creates flexible movement of each robot in the swarm when compared to conventional swarm exploration methods in similar environments. The proposed approach performed well in dynamic environments with moving obstacle. Since A* would not work well as moving obstacles were not accounted for in the original A* path, the authors only tested their own approach in the dynamic environment. The authors' approach worked and the robots avoided both moving obstacles and other robots moving in the environment. In static environments, A* performs better, since the DQN and CNN approach takes unnecessary movements. Based on the results it is inconclusive how the authors' DQN and CNN approach would compare with other state of the art obstacle avoidance and exploration methods, especially for a swarm with limited moving obstacles. Move Base would have been an appropriate baseline to compare their implementation too since it is so widely used. The authors also did not detail the communication and coordination aspect of the swarm navigation algorithms.

Yang et al. also conducted research on path planning for multi robot systems based on a DQN algorithm [55]. The authors focused their research on optimal performance of a scheduling system in a warehouse which uses robots that handle items from one location to another. Their DQN algorithm combines Q-learning, an empirical playback mechanism, and productive neural networks to create Q values for their multi-robot system. The authors Q-learning goal is two solve two main problems of current multi-robot path planning —slow convergence and excessive randomness. Pre-training using prior knowledge and rules are used to improve the DQN approach.

The authors use prior knowledge to initialize the Q value table. The authors use prior knowledge because it avoids exploration of the system and weights from scratch which reduces exploration due to randomness. Before the robot uses the model, the robot will run A* with a single robot in a static environment, in order to determine the path from the starting point to the goal point. The robots use this path to guide the subsequent path planning calculations. This trick allows the robots to have an understanding of the environment from the beginning and not be completely unaware of the situation, which reduces training time for the model. The algorithm for training can be seen in Figure 2.13.

The authors test the DQN path planning method in simulation. The results show that the authors' DQN algorithm converges faster than classic DQN approaches and the authors' DQN algorithm can learn the solutions to path-planning problems in less time. The authors propose an interesting method for reducing training time for exploration and path planning systems. Although it accounts for moving obstacles, it is not suitable for an environment that is not known beforehand, since this is not a map exploration technique.

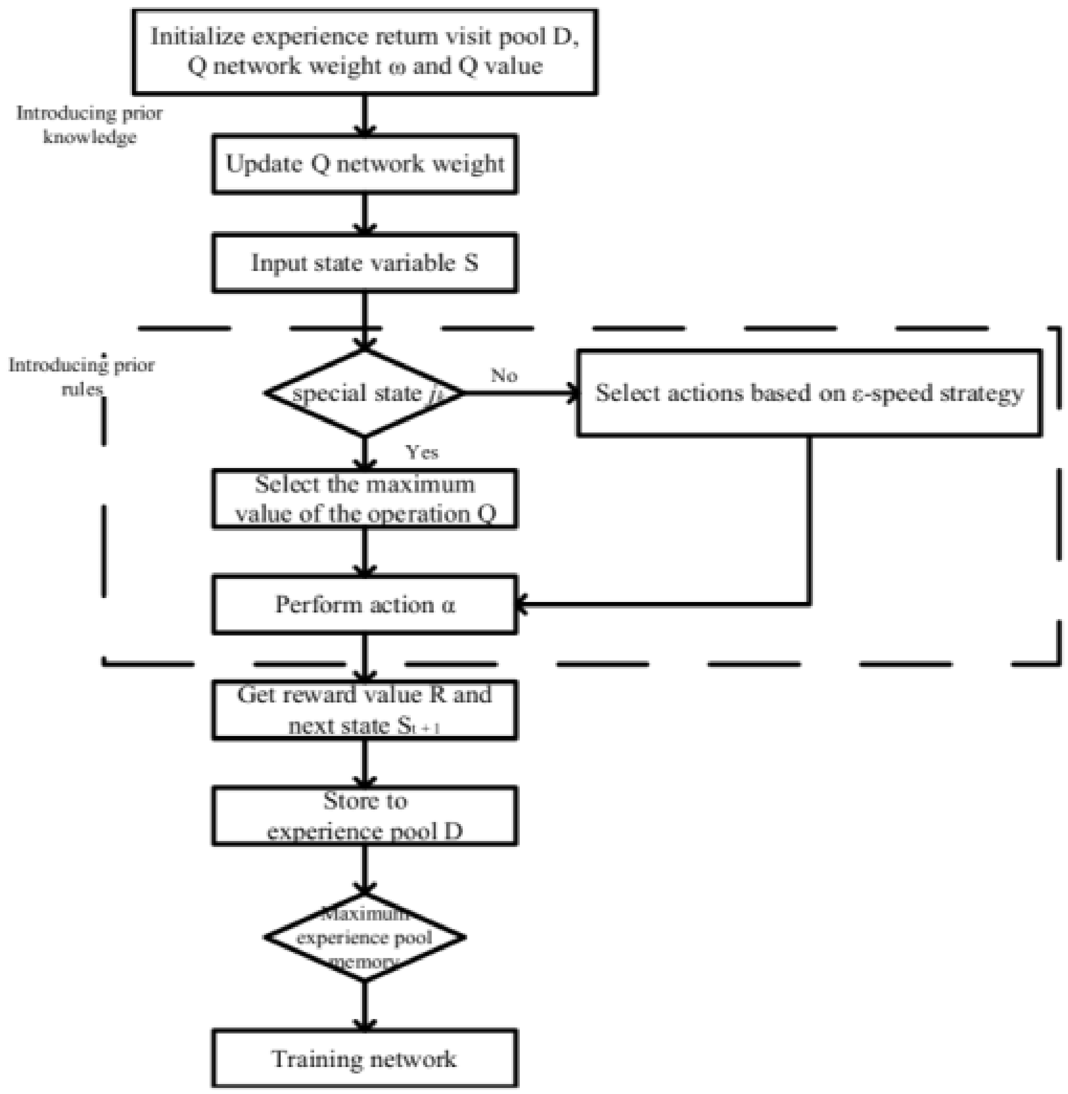

Initialize experience return visit pool D, Q network weight ω and Q value
Introducing prior knowledge
Update Q network weight
Input state variable S
Introducing prior rules
special state
No
Select actions based on ε-speed strategy
Yes
Select the maximum value of the operation Q
Perform action α
Get reward value R and next state S_{t+1}
Store to experience pool D
Maximum experience pool memory
Training network

Figure 2.13: DQN Algorithm for Path Planning Training [55]

2.3 Contribution of This Work

After surveying the current state of the art research of reinforcement learning models for increased exploration speed, I determined that this thesis contains novel attributes. The first is this is the first reinforcement learning approach specifically targeted at increasing the search speed for a single robot with the goal of scaling the approach up to a swarm of robots, while using the same reinforcement learning model. This thesis is also the first approach to use Move Base during both training and during exploration run time. This thesis is also among the first research to be done on deep reinforcement learning based exploration that trains on optimized frontier searching, rather than deep reinforcement learning based exploration that only trains on obstacle avoidance while exploring the scene. Finally, this thesis provides a full OpenAI ROS Gym training pipeline that is integrated with commonly used ROS packages.

Chapter 3

Methodology

3.1 Problem Formulation

This thesis aims to reduce the total exploration time to map completion for a single robot and extend the approach to a swarm of robots. The individual robot explorers use their maps to determine the best frontier to traverse to. The frontiers are defined as the boundary between the explored open cells and the unexplored cells [54]. The most common way for robots to rank frontiers is by proximity to the robot's current location, as in [54] or by another cost function that takes into account both the size of the frontier and distance to the frontier. Popular approaches have a set weight used for the balancing of the two frontier characteristics, distance and size. This leaves an opportunity for optimization. A set weight cannot always perform the best in every situation. Based on the environment, sometimes a closer frontier would be better than a farther frontier, such as in a room exploration with lots of obstacles. Since the robot would have a lesser chance of getting stuck and confused while navigating to a closer frontier, the total exploration time could be shortened. But if there is a large empty room, the largest frontier could be higher priority since much more information would be found in a shorter amount of time. When a frontier is better than another, it means the algorithm selecting that frontier over another frontier would reduce the total exploration time. If the weight changes depending on the structure of the environment, it could lead to quicker search times for both an individual robot and a swarm of robots. From the reading and researching of current state-of-the-art exploration techniques, there is no other implementation of an optimized weight for frontier ranking implemented on a single robot and extended to a swarm of robots for a search-and-rescue scenario. The goal of the thesis is to use A2C to optimize the w_{output} in Equation 3.1 depending on the current state of the robot. d is the distance to the frontier, which is the euclidean distance from the robots current position to the center of the frontier. g is the size of the frontier, which is the number of cells in the frontier.

$$\text{cost} = w_{\text{output}} \cdot d + (1 - w_{\text{output}}) \cdot (1 - g) \tag{3.1}$$

Individual robot explorers use their current map data to determine the best frontier to traverse to. For the extension, the swarm uses a commander and explorer structure to search the environment. The commander uses a list of its available frontiers to determine where the best meeting point will be. The commander robot calculates these frontiers individually, so it uses the same logic as the single robot implementation. The commander then sends the swarm of robots off to individually explore for a predetermined amount of time, after which they will reconvene. This individual exploration uses the individually calculated frontiers to have each robot explore the environment.

3.2 System Overview

I setup the system using Robot Operating System (ROS) and Gazebo for simulation, as well as RViz for visualization. Move Base sends the robot to the locations chosen by the frontier selection algorithm. For the ML pipeline, I used OpenAI Gym to set up the Gazebo simulation and reset episodes. OpenAI Gym allowed for a straightforward implementation of a reinforcement learning (RL) training pipeline, by supplying a class structure for interaction between the training agent and the Gazebo simulation. By using OpenAI Gym I was able to devote more time to the RL development.

TurtleBot3 Burgers were the robot of choice for this project. They are equipped with the basic sensors necessary for this project, and are straightforward to integrate with ROS from the extensive robotics research and documentation on them.

3.2.1 OpenAI ROS

OpenAI is a company that provides OpenAI Gym, which is a "complete Reinforcement Learning set of libraries that allow to train software agents on tasks, so the agents can learn by themselves how to best do the task" [36]. OpenAI has been used for complex training scenarios, for example, training an agent to play Dota2. One of the best components of OpenAI is the Gym, which allows for an Environment to train on [35]. However these environments that Gym provides are not Gazebo based, so there was a gap in the package based reinforcement learning pipeline for developers. That is why Ezquerro et al. created OpenAI ROS [35].

The OpenAI ROS package requires very little setup for a basic reinforcement learning implementation, without other packages. The OpenAI ROS package has a class hierarchy , which can be seen in Figure 3.1. The Gazebo Environment class provides communication between the coded OpenAI programs to Gazebo. This is connected to the RobotEnvironment classes. These classes exist pre-made for popular robots, so the Turtlebot3 is included. This layer provides the connection between the Gazebo simulation and the OpenAI Algorithm environments. This layer handles obtaining sensor information from the robot and interacting with the environment, such as sending ROS messages for the robot to move. It is intended to be a transparent layer to the OpenAI algorithms and the developer. Unfortunately, as later explained, due to it not supporting many common packages, the transparency is not as seamless as it appears. This increases development time for the person creating the RL pipeline. The task environment layer is the layer used to train the robot. This level interacts down to the RobotEnvironment. For example, the TaskEnvironment layer has the set action, get observation, and check if the episode is done methods. It processes the information from the RobotEnvironment, such as the data from the occupancy grid, in order to complete the previously mentioned methods. The final uppermost layer is the learning algorithm and training script. These are python scripts that can use popular machine learning libraries such as Pytorch [40] or Keras [22], to implement reinforcement learning models. It is in this TaskEnvironment layer that I implemented the logic for resetting environments. This uppermost level is also where the TaskEnvironment is initialized. The system overview is described as: the TaskEnvironment inherits from the RobotEnvironment, the RobotEnvironment inherits from the GazeboEnvironment, and the GazeboEnvironment inherits from the GymEnvironment. The GymEnvironment is the basic environment structure from OpenAI, and is of type `gym.Env` [35].

The GazeboEnvironment class is intended to connect the simulated gym environment to the Gazebo simulator. This layer is where the logic for environment resets lives, such as resetting the simulator after each episode and resetting the controllers. There are four functions required by OpenAI Gym for resetting the gym environment: `step`, `seed`, `reset`, and `close`. This layer implements these functions to include the necessary steps to replicate this behavior in the Gazebo simulation as well. This class also publishes the latest episode reward to a specified topic.

The RobotEnvironment class is where the ROS functionalities that control the robot reside. This is also where checks are which ensure all the ROS nodes are functioning correctly and as intended before starting up the training process. The user does not need to set this up for a specific robot. It is already implemented for many popular robotics platforms, and the list can be found at [35]. The robot is selected within the RobotEnvironment class. For example, the TurtleBot2 robot is implemented in the `turtlebot2_env.py` class in the `openai_ros` package.

The TaskEnvironment class provides the context for the actions the robot should learn. This means it depends on the task and on the robot, so it is customized by the user. It provides the necessary information up to the training script. The training script only needs to call the TaskEnvironment methods to understand the current state of the robot throughout the episode. The TaskEnvironment requires the following methods for training: `set_action`, `get_obs`, `compute_reward`, and `is_done`. `set_action` takes the selected action from the training script and applies this to the actual robot. It contains the logic for taking the action representation from the training scripts model and putting it into robot actions. `get_obs` collects the observation of the robots state when the action is concluded. `compute_reward` computes the reward received at the end of the action. Often it uses the get observation function. `is_done` detects if the current training for the episode is finished, also often using the observation function. The TaskEnvironment class also includes functions for initializing environment variables and setting the initial pose.

The training script sets up the reinforcement learning algorithm for the robot to learn. The script selects the task/action for the robot to do, as well as setting the robot to be used. The script also sets up the training environment using the environment classes above by using configuration files. This script is referred to as the RL node, since it contains most of the RL logic and instantiates all of the OpenAI ROS classes.

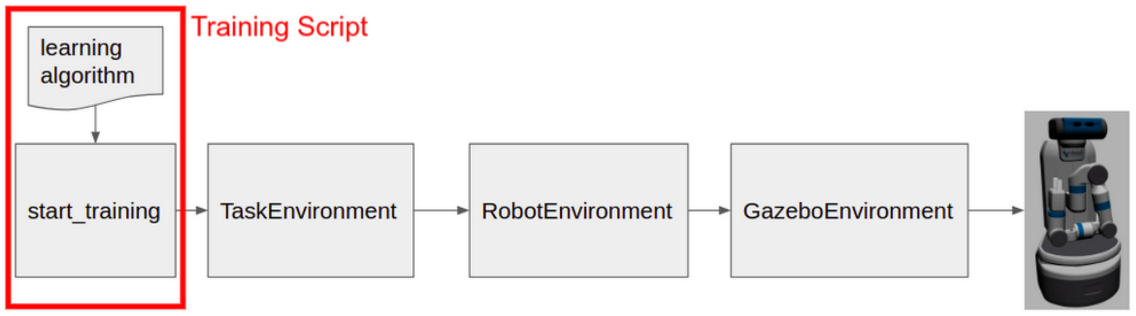


Figure 3.1: OpenAI ROS Pipeline [36]

A noteworthy contribution of this thesis is the creation of a reinforcement learning actor critic OpenAI Gym pipeline. As of the writing of this thesis, the only public implementation of a reinforcement learning model with OpenAI Gym for ROS and Gazebo is Q-learning. By implementing A2C, I created much more complexity to the system as the observations and action space were much larger than Q-learning. Also, since this implementation of actor critic required Gmapping and Move Base, whereas the more simplistic Q-learning example available did not, it required some customizations to the normal procedures of running those packages.

3.2.2 ROS Package Set Up

The simulation was implemented in ROS and Gazebo, with RViz for visualization.

I implemented Move Base for navigation. Move Base is a ROS package that provides path planning for a robot when given a goal pose [28]. It uses both a global and local occupancy grid to plan the navigation task. Due to Move Base's utilization of both the global and local occupancy grids for path planning, as well as the in depth monitoring of the robots path planning and navigation status on various ROS topics, it made sense to use this package rather than do a hand built A* implementation for navigation. Move Base takes into account dynamic obstacles, so the robots in the swarm will be able to navigate around one another if they are in each other's path.

Gmapping was used for mapping the environment, since it is a package that implements laser-based SLAM [16]. Gmapping creates a 2D occupancy grid for a robot and it publishes it to a dedicated topic. Each robot used Gmapping to create their individual map. A separate team working on this same project but with a different focus, worked on a map merging node, which combines all the robots individual maps into one merged map which is published to a specific topic [15].

Ferrera et al. [15] created the environment in Gazebo with a maze structure, to replicate the hallways found in an office building environment. In order to aid in the scan matching when traversing up and down the long hallway, I added pillars were to the hallway, as seen in Figure 3.2. The pillars represent unique objects found in office hallways that help with scan matching: plants, chairs, paintings, etc.

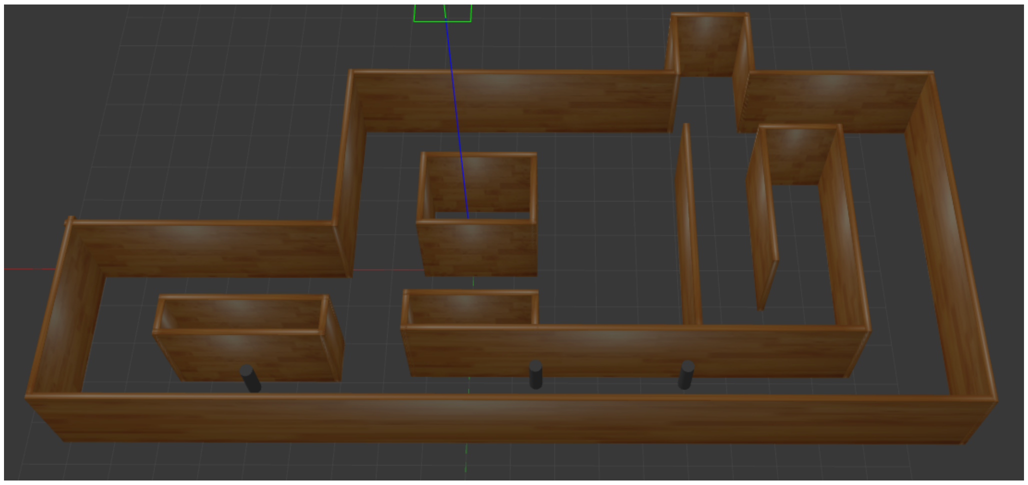


Figure 3.2: Gazebo Maze With Pillars

3.3 Reinforcement Learning Set-up

Reset Function for Required Packages

The intention behind using OpenAI ROS for this project is it is supposed to remove the ROS specific development work so that the programmer can concentrate on the Machine Learning application. Unfortunately, this was not entirely the case, and I needed to implement many modifications to the supplied TurtleBot3 environment in order to make it useful for this specific reinforcement learning application. OpenAI ROS was not created for the ease of use with other SLAM ROS packages when training reinforcement learning episodes. Gazebo worked as intended and was reset correctly at the end of the episode. However, I used other packages such as Gmapping or Move Base in the training pipeline, OpenAI ROS cannot fully reset the episode, as it does not have the capability to reset these packages. It is important that Gmapping resets at the end of an episode, else the map will not clear when the robot is placed back at its start position and is intended to begin exploring again. If Gmapping is not reset then the old completed map is still in the Gmapping memory, but the robot is suddenly teleported back to its starting location as the Gazebo simulation is reset. This confuses Gmapping as the scan matching is now broken. Therefore, it was necessary that I implement a reset function for Gmapping that allows it to clear the map upon a signal.

A similar issue occurred with Move Base needing to be reset in order to operate correctly over multiple episodes. The robot is somewhere in the maze when the episode resets, and the robot is suddenly teleported to the starting location for exploration. This confuses Move Base since it recognizes that the robot did not move its wheels or navigate to another location, but suddenly all the LiDAR readings change. Therefore, I needed to implement Move Base with a reset signal as well.

In order to solve the Gmapping issue, I forked the Gmapping Repository. The Algorithm for this shown in Algorithm 2. In the main function, 1) the node is initialized, 2) the Gmapping class is instantiated, 3) live slam is started, and 4) ROS spins up. Instead of including 2-4 in the main loop, instead another method is added, which includes steps 2-4, and is called by main. This new method is `main_loop`, which is seen in Algorithm 2. In here, the function first subscribes to a topic `/syscommand`. It is on this topic that the RL node will send the reset command. In the callback to this topic, the Gmapping node receives the String "reset" it sets a Boolean called `shouldReset` to true. Then, while ROS is running and while not `shouldReset`, Gmapping is instantiated, live slam is started, and ROS spins once. Once ROS has spun once, the method checks to see if the while condition of not `shouldReset` is still true. If the reset signal was sent, then the while loop exits. When that inner while loop exits, the outer while loop, which checks if ROS is okay, is still true. Therefore, the method re-instantiates Gmapping, starts live slam for that instance, and ROS spins once. That process continues until ROS is shutdown at the end of all the training episodes.

Algorithm 2 Gmapping Reset Functionality

```
1: procedure MAIN_LOOP
2:     ros subscribe to "/syscommand"
3:     while ros::ok() do
4:         instantiate SlamGmapping gn
5:         gn.startLiveSlam()
6:         while not shouldReset do
7:             ros::spinOnce
8:             ros::Duration(0.2).sleep()
9:         shouldReset = false
```

I also needed to reset Move Base, and in order to accomplish that I implemented a python subprocess for the Move Base ROS launch files. If any node or program sends `start` to `/start_move_base` then the Move Base node creates a Python subprocess for the move base launch file. When `stop` is sent to `/stop_move_base` then the kill signal was sent to that original Move Base sub process.

These solutions worked and they both needed to be added to the RL pipeline. In order to integrate the Gmapping reset function in the reset function for the OpenAI Gym environment, I implemented the call to `/syscommand` with the value of `reset`. The same was done for Move Base; the reset function for the OpenAI Gym environment sends a String of `stop` to `/stop_move_base`.

3.4 Dynamic Map Resizing

Gmapping dynamically resizes the occupancy grid map while the robot is exploring. It does this to ensure that the map can grow if the robot explores more environment. Gmapping tries to estimate a reasonable starting map size based on the robots LiDAR scans upon startup, but often the occupancy grid map grows in width and height throughout the exploration as the robot finds more area. However, the size of the array input to a ML model must be pre-defined upon initializing the model and must stay constant. In order to handle this dynamic map resizing, I created a ROS node named `publish_inputs_for_learning`. This node uses two variables, a desired max dimension in x (width), and a desired max dimension in y (height), to expand the map. In order to expand the map, the node pads zeros to the upper left part of the occupancy grid. It pads as many zeros as necessary to make the current occupancy grid the desired dimensions in x and y. I chose these desired dimensions by experimenting on the chosen environments, specifically sending the robot to the four corners of the map via Rviz tasking. I set the desired width and height to the size of the occupancy grid after all four of the points have been visited. Figure 3.3 shows the raw map upon startup from Gmapping without any padding, while Figure 3.4 shows the map with the padding to make it a square 250 by 250 map.

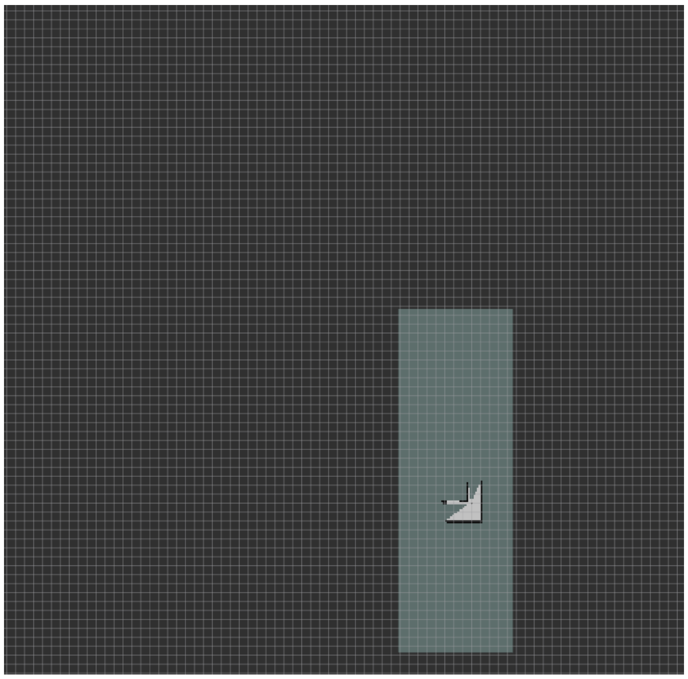


Figure 3.3: Raw Occupancy Grid Map

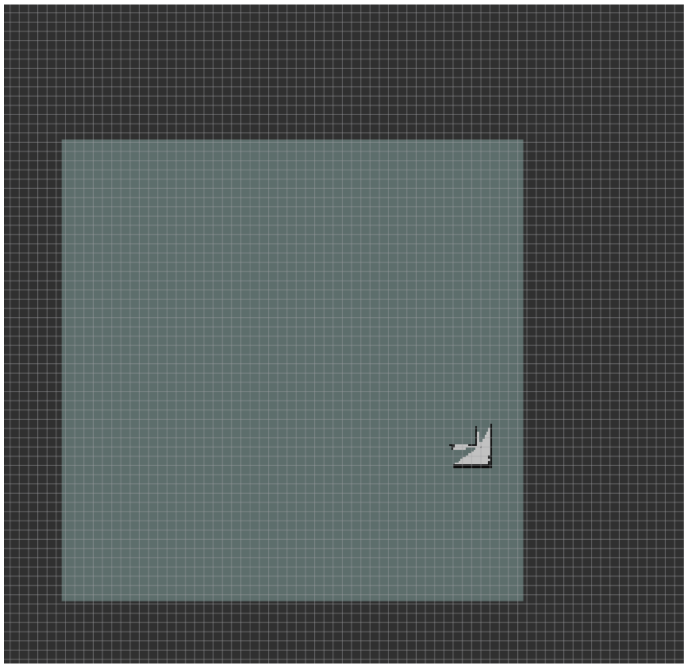


Figure 3.4: Padded Occupancy Grid Map

3.5 Frontier Calculation

The frontier calculation pipeline can be seen in Figure 3.5.

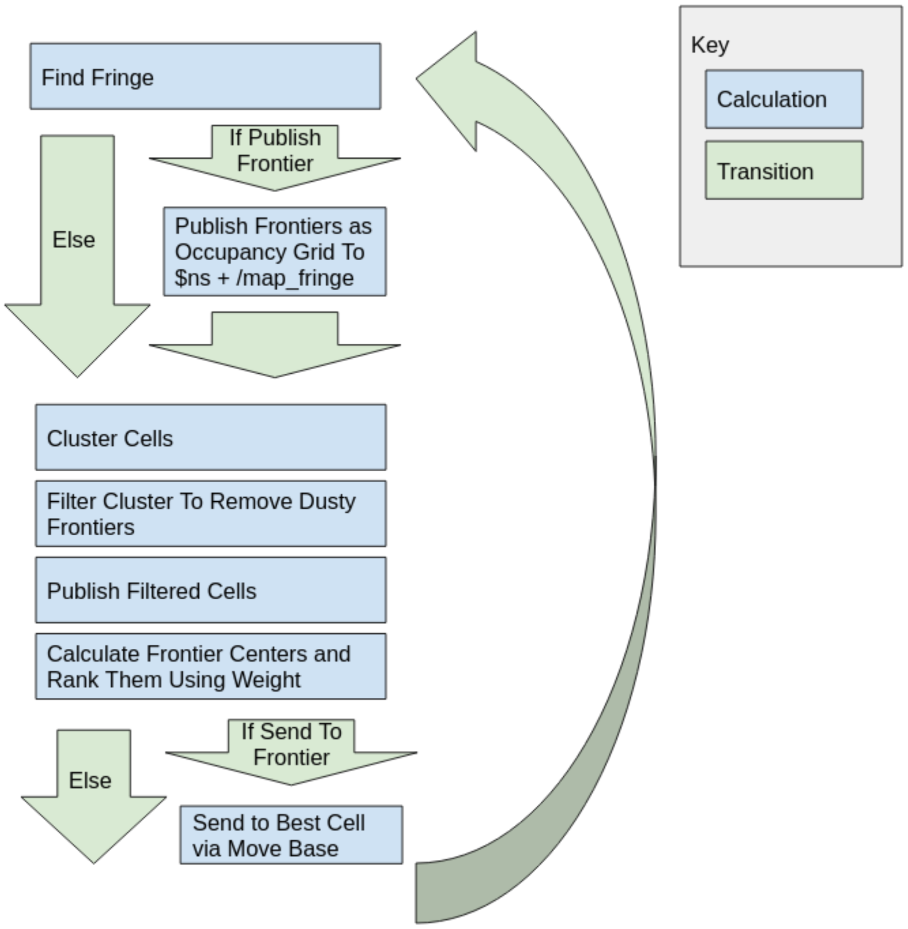

Find Fringe
If Publish Frontier
Else
Publish Frontiers as Occupancy Grid To $ns + /map_fringe
Cluster Cells
Filter Cluster To Remove Dusty Frontiers
Publish Filtered Cells
Calculate Frontier Centers and Rank Them Using Weight
If Send To Frontier
Else
Send to Best Cell via Move Base
Key
Calculation
Transition

Figure 3.5: Frontier Calculation Diagram

The find fringe frontier detection node works in two main steps: Sobel edge detection and cell filtering. Using the Sobel edge filter, the fringe detection node can fine the known cells that border the unknown cells. Then the node will compare the cells and only pick those that are predicted to be free space. To find all cells that are unoccupied and immediately adjacent to unexplored nodes, the node takes a logical AND over the two sets. This results in a list of fringe cells.

Next, node must cluster the fringe cells, and it creates a dictionary to hold the clusters. The key to the dictionary is a cell in the cluster. The node clusters the cells by first starting out with all the clusters in a list. Each round, all the frontiers attempt to be added to a list that exists within the dictionary. It can only be added if it is 8-connected to another cell in an existing list. If it is eight connected the node adds the cell to the list and removed from the list of cells to be clustered. The in depth logic for clustering is showing in Algorithm 3. The list of `grid_cells` is provided to the clustering algorithm from the find fringe cells detection node, explained above. `cluster_dict` is the dictionary that holds the clusters. The key to the dictionary is the first cell that was added to the cluster, and the value is the list of cells in that cluster. The node creates the dictionary and the node adds the fist cell in `grid_cells` to an empty list, and then that list is added to the dictionary with the key of the first cell. That first cell is then removed from the list of `remaining_cells`. `remaining_cells` is a list that is used to track which cells are left to be assigned to a cluster. The dist function is the euclidean distance between two cells, and `same_cluster_threshold` is the max distance two cells can be apart to be within the same cluster. For this implementation, I set it to 2 times the map resolution. This means they need to be within a two cell distance from one another to be in the same cluster, which is essentially 8-connected.

Algorithm 3 Clustering Algorithm

```
procedure CLUSTER(grid_cells)
    for a_cell in grid_cells do
        if remaining_cells is not empty then
            if cluster_dict is not empty then
                for each key of cluster_dictionary do
                    cells_in_cluster = cluster_dict.get(key)
                    for a_cluster_cell in cells_in_cluster do
                        if dist(a_cell, a_cluster_cell) < same_cluster_threshold then
                            if a_cell not already in cluster then
                                add a_cell to cells_in_cluster
                                remove a_cell from remaining_cells
                            for potential_cell_for_current_cluster in remaining_cells do
                                if dist(potential_cell_for_current_cluster,a_cell) < same_cluster_threshold then
                                    if potential_cell_for_current_cluster not already in cluster then
                                        add potential_cell_for_current_cluster to cells_in_cluster
                                        remove potential_cell_for_current_cluster from remaining_cells
                    update dictionary cluster_dictionary[key] = cells_in_cluster
    return cluster_dict
```

Now that the frontiers are clustered, another round of filtering begins. Any cluster with four or less cells are filtered out of the list of viable clusters. This is because there is simulated noise with the LiDAR sensor, and Gmapping will sometimes leave a single unknown cell surrounded by lots of known cell. This floating unknown cell is deemed a frontier as it is an unknown cell bordering known, so the four connected cells around it are deemed the frontiers. Once the clusters are created, these small frontiers can be filtered out as they do not aid in system performance and are not large enough frontiers to be searched. Oftentimes, over time, Gmapping either corrects the floating unknown cells to open, or the robot passes over that area of the map to go to a different frontier, and now that there is a new scan of that area Gmapping updates the map and corrects that cell to open. This solution is in place to ensure that even if Gmapping does not fix itself that there are no unnecessary frontier clusters in the list.

Next the frontiers need to be ranked from best to worst. In this case, best means the frontier with the lowest cost. The cost of the frontier is calculated as in Equation 3.1. The size of the cluster is the amount of cells in the cluster, and the distance is the euclidean distance from the robot's current position to the center position of the cluster. The center position of the cluster is all of the cells x values averaged and y values averaged. The $weight_{\text{output}}$ is either a set weight the entire run, per the more classic frontier approaches, or changes based on the output of the ML model. The $weight_{\text{output}}$ is a value between 0 and 1 that is the trade-off between the distance to the frontier and the size of a frontier. Once all the costs of the clusters are determined by the frontier selection node, the cluster dictionary is sorted by the value of the cost, and returned.

The final list of frontiers is the list of center locations of the frontier clusters ranked from lowest cost to highest cost.

3.6 Map Representation of Frontiers and Pose

Some of this thesis's implementations of the ML solutions to exploration, discussed later in this paper, require the pose and the frontiers as inputs. In order to keep consistent with the map input, these inputs were both represented as arrays of data in occupancy grid fashion. For the frontier input, all frontier points are represented as 0s, and all the remaining spaces in the grid are -1. For the pose output, the robots current pose is represented as 0 and the remainder of the spaces are -1. These two maps also are processed through the resizing map node, `publish_inputs_for_learning`, so that they can always be the same size as the desired map size. Figure 3.6 shows the frontiers and robot's current pose as the size of the raw robot occupancy grid, and Figure 3.7 shows the resized padded maps to make them the desired size.

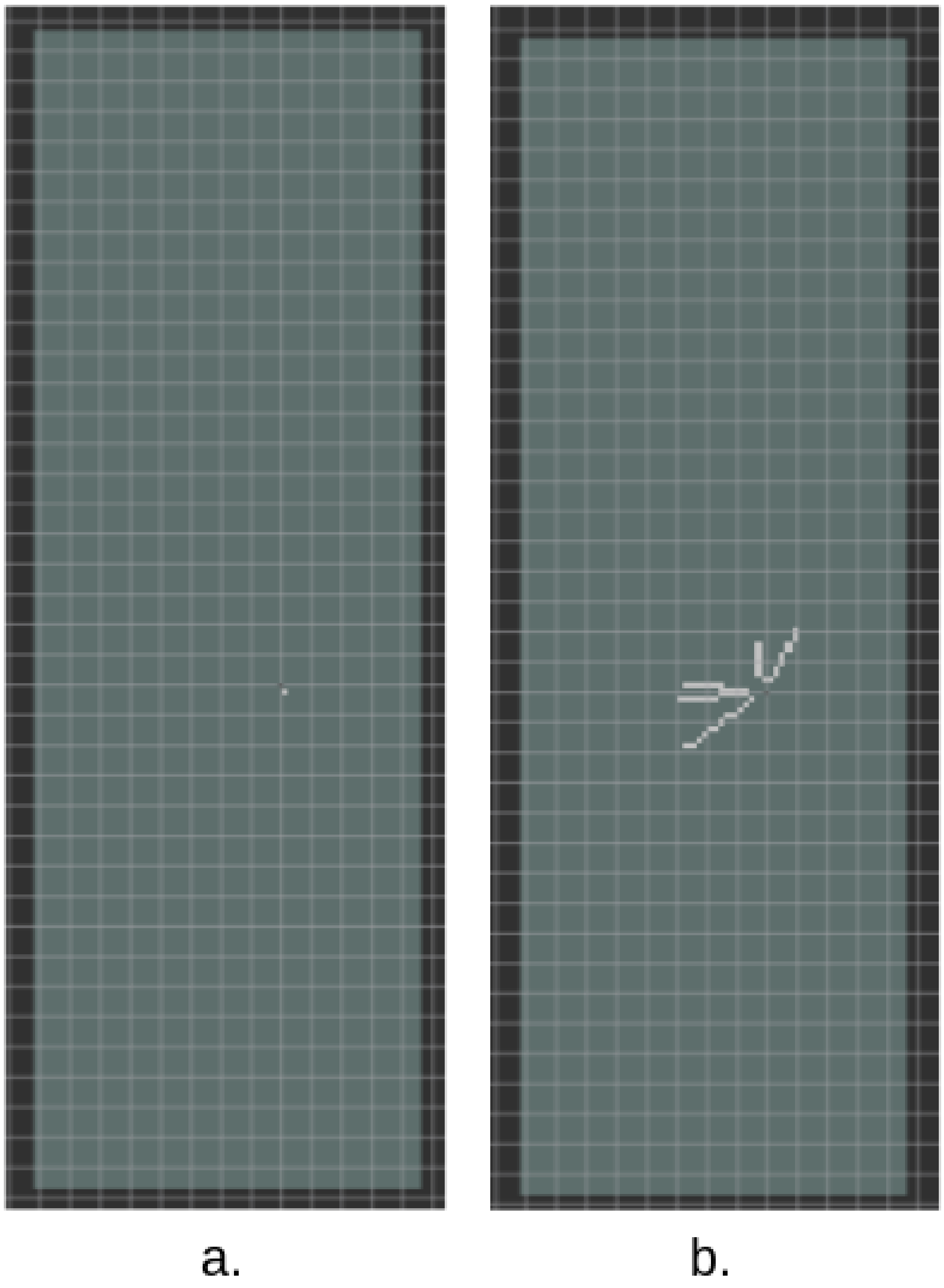


Figure 3.6: a. Frontiers in Occupancy Grid Map Representation b. Robot's Current Pose in Occupancy Grid Map Representation

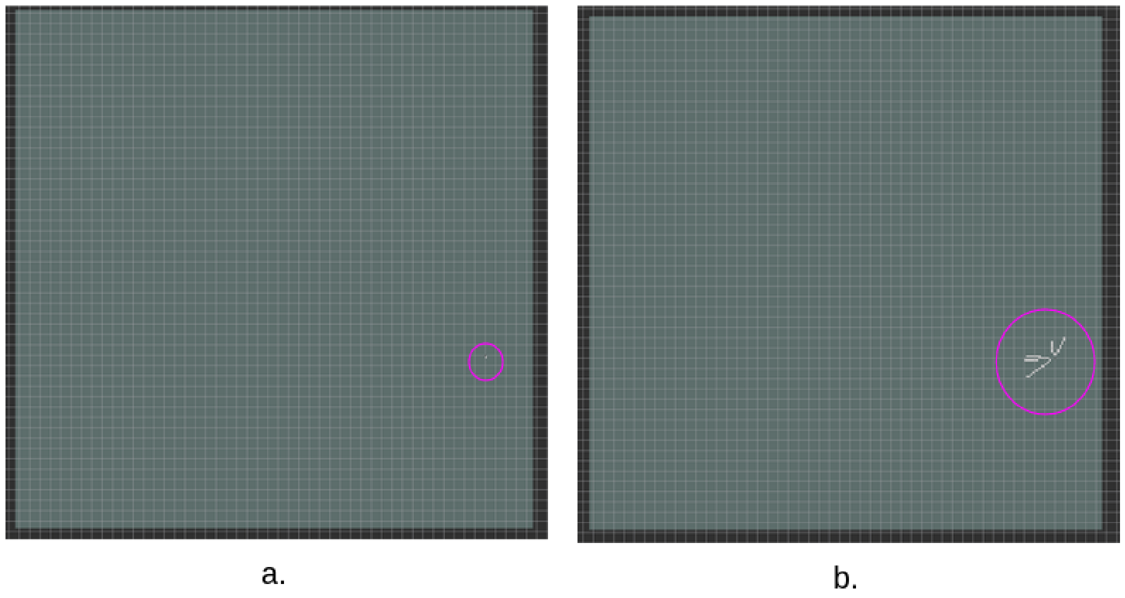


Figure 3.7: a. Frontiers in Padded Occupancy Grid Map Representation b. Robot's Current Pose in Padded Occupancy Grid Map Representation. Both Outlined by the Pink Circle

3.7 Initial Test of Environment

In order to test the setup of the OpenAI ROS Environment, I implemented a simple Q-learning algorithm for a single robot learning obstacle avoidance. The available actions were one of three actions: left, right, forward. The input to the Q-learning algorithm was the laser scan messages from the 360 LiDAR. In order to allow for enough state action pairs to be tested, I segmented the scan message into 6 values, and cast those three values to integers between -4 and 4.

The main functionality to be tested with this implementation is that the `get_observation`, `get_reward`, and `is_done` functions were implemented correctly, along with the environment resetting working as expected. It is through this test that I discovered the Move Base and Gmapping complications. Once I implemented the solutions for resetting Gmapping and Move Base completely at the end of an episode, I implemented the reinforcement learning model based approach for having the robot efficiently search the environment.

3.8 A2C Implementation

During this thesis, I implemented A2C, with a variety of input sizes and two different output representations. The model intends to update the policy gradient by using Monte Carlo updates, so random samples are taken to update the policy parameter.

The actor critic model desires to minimize the total loss, L, in Equation 3.10 [3]. The loss value is the sum of the value loss (from the actor, Equation 3.8), the policy loss (from the critic, Equation 3.7), and the entropy regularization term (Equation 3.9). a_t is the action at time t and s_t is the state at time t. π is the policy and θ is the policy neural network parameters, so π_θ is the policy using the θ parameters. $A(s_t, a_t)$ is the advantage function for state s_t and action a_t. The advantage function is in Equation 3.6. The advantage function combines the value function, $V(s)$, and action-value function, $Q(s, a)$. The value function represents the long-term average reward for the provided state. The action-value function measures how optimal it is to take action a in state s, and follow policy π after. $V(s)$ and $Q(s, a)$ are known as the Bellman equations [3] and are shown in Equations 3.4 and 3.5. s is the current state, $s\prime$ is the next state, a is the current action, and $a\prime$ is the next state. γ is the discount factor, which is a value between 0 and 1. γ ensures that the agent is neither too short-sighted nor too far-signed, by ensuring the agent prioritizes the agent's actions to maximize the total discounted rewards [3]. The Bellman equations are updated by the transition probability, $P_{s,s\prime}$, and the expected reward, $R_{s,s\prime}$. The transition probability and expected reward equations are shown in Equations 3.2 and 3.3. The transition probability and expected reward equations assume the MDP, which is that the transition to the next state at $t+1$ only depends on the state and action at the current time, t [3]. E is the expectation over samples.

$$P_{s,s\prime} = Pr(s_{t+1} = s\prime | s_t = s, a_t = a) \tag{3.2}$$

$$R_{s,s\prime} = E[r_{t+1} | s_t = s, s_{t+1} = s\prime, a_t = a] \tag{3.3}$$

$$V(s) = \sum_a \pi(s,a) \sum_{s\prime} P_{ss\prime}[R_{ss\prime} + \gamma V(s\prime)] \tag{3.4}$$

$$Q(s,a) = \sum_{s\prime} P_{ss\prime}[R_{ss\prime} + \gamma \sum_{a\prime} \pi(s\prime, a\prime) Q(s\prime, a\prime)] \tag{3.5}$$

$$A(s,a) = Q(s,a) - V(s) \tag{3.6}$$

$$L_v = \sum (V^{target} - V(s_t))^2 \tag{3.7}$$

$$L_p = -\sum log \pi_\theta(a_t | s_t) A(s_t, a_t) \tag{3.8}$$

$$L_e = -\sum \pi_\theta(a_t | s_t) log \pi_\theta(s_t, a_t) \tag{3.9}$$

$$L = 0.5 * L_v + L_p - 0.001 * L_e \tag{3.10}$$

In order to update the actor and critic from the loss values, I used `backward`, a Pytorch function, to get the gradients [48]. I used `step`, another Pytorch function, to update the parameters [47].

3.9 Different Model Configurations

The inputs to the various models listed below are one of two inputs: only the map, or the map, frontiers, and pose appended to one another. The frontiers and pose maps creation are outlined in the subsection Map Representation of Frontiers and Pose. All maps are represented as inputs by by their data array component of the occupancy grid structure. Gmapping created the map, and the resulting map was resized to fit the desired dimensions. This is outlined in the subsection Dynamic Map Resizing. However, this map had one last alteration before it was fed to the model.

The raw representation of the occupancy grid map's data array was difficult for a reinforcement learning model to understand. The values of an RL input should contain order, numerically, in this case from either worst to best or best to worst. The reinforcement learning model needs to learn a gradient, so it needs to learn what values of the occupancy grid are bad, satisfactory, and best. In this case, the robot should find a gradient between the best and safest cells to go to. Since the goal was for total exploration time to decrease, the robot should go to cells that allow for the greatest environment discovery but that are also safe enough that there are not complications while travelling to the point. The known unoccupied cells are safest to go to, since there is no obstacle in that cell for the robot to get stuck on. The unknown cells are the next safest to go to. There is the chance that once the robot makes it to the unknown cell that it ends up being in an obstacle, which creates delays and complications with Move Base. There is also the chance that the selected unknown cell is purely unreachable. However there is also the chance it is an open cell that ends up in a large amount of the environment discovered with no complications. Finally, the worst cell to go to is a known occupied cell. This will end in little to no information gain, and will cause lots of complications and delays with Move Base.

The traditional numeric ordering of the occupancy grid is unknown, unoccupied, and occupied, represented by -1, 0, and 100, respectively. The numeric ordering should be altered so that the order is, worst to best, occupied, unknown, unoccupied. I did this by replacing the values in the maps data field when inputting it into the model. All 100s in the data, which are the occupied spaces, are replaced with -1. All -1s in the data, which are unknown cells, are replaced with 0. All 0s in the data, which are unoccupied spaces, are replaced with 1. Now, all of the 3 cases are numerically spaced the same amount and should aid the model in converging to the best cells for the robot to go to.

Since the frontiers and pose inputs are created by custom ROS nodes, upon creation they are bound to [0,1], so they will have a similar value range to the map.

During the thesis research, I implemented four different models, with the goal that one of them would converge to a usable solution after training. The four models are as follows:

1. The first version of the model uses the input of the map, frontiers, and pose. The maps are input to the model as a 1D array, with the three inputs appended together, in the order of map, frontiers, pose. The output of the model is the size of the map, so 1/3 the size of the input. For any input, the `argmax` of the array is the point on the map the robot should go to. The index to point algorithm converts between the index of the 1D map array to the position in the 2D occupancy grid.

2. The second version of the model uses the input of the map. The map is input to the model as a 1D array. The output of the model is the size of the map, so the input size and output size are the same. For any input, the `argmax` of the array is the point on the map the robot should go to. The index to point algorithm converts between the index of the 1D map array to the position in the 2D occupancy grid.

3. The third version of the model uses the input of the map, frontiers, and pose. The maps are input to the model as a 1D array, with the three inputs appended together, in the order of map, frontiers, pose. For any input, the `argmax` of the array is the best value the robot should use as $weight_{\text{output}}$ in 3.1. The output array is of size 100, so the weight inputted into the frontier calculations is best output/100.

4. The fourth and final version of the model uses the input of the map. The map is input to the model as a 1D array. For any input, the `argmax` of the array is the best value the robot should use as $weight_{\text{output}}$ in 3.1. The output array is of size 100, so the weight inputted into the frontier calculations is best output/100.

The final model, 4, is the only model that converged to a useable solution. Figures 3.8 and 3.9 show the inputs and outputs the actor and critic for model 4.

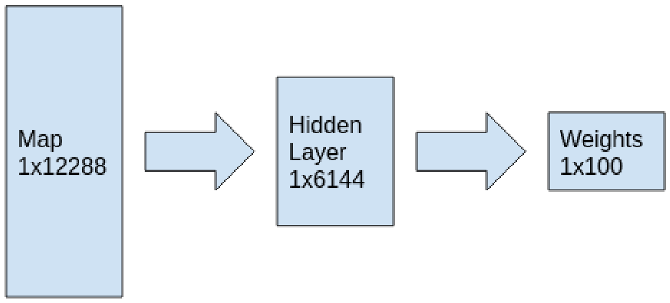


Figure 3.8: Diagram of the Inputs and Outputs of the Actor for Model 4

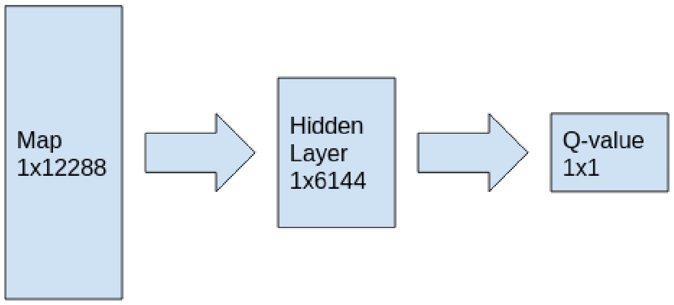


Figure 3.9: Diagram of the Inputs and Outputs of the Critic for Model 4

3.10 Model Output to Occupancy Point Conversion

For models 1 and 2, the RL node converted the model output into a point for the robot to go to. The model output is the size of the robots current map, which the robot uses to navigate. The output that is chosen from a specific input is the `argmax` out of all the outputs, so the output with the highest weight. The output is a 1D array, and the size of the array is the size of map width times the size of the map height. Since the map width and height are included in every map update from Gmapping, it is simple to transform this 1D index to a 2D map coordinate. The equations for this transformation are seen in Equations 3.11 and 3.12.

$$x = index\%int(map\_width) \tag{3.11}$$

$$y = (index - x)/(map\_height) \tag{3.12}$$

3.11 Rewards Structure

In order to train the robot I implemented a specific rewards structure. In Table 3.1 and Table 3.2 the rewards structure is shown for the 4 different reinforcement learning model versions. The OpenAI Gym implementation gives rewards to the robot agent at the end of an action completion.

As shown in Table 3.1 models 1 and 2 give four additional rewards to the agent: unoccupied cell, unknown cell, occupied cell, and near wall rewards. The descriptions are as following. Since models 1 and 2 specifically pick actions that are points on the occupancy grid there needs to be a reward given for the specific cell they pick, which are the four aforementioned rewards, and then rewards based on travelling to that cell. The unoccupied reward is a positive reward the OpenAI Gym node gives to the agent if the action it selects is a cell that is unoccupied, meaning has the value of 0. The robot should be positively rewarded for selecting a known open cell because there is less of a chance it could get stuck. The OpenAI Gym node gives an unknown reward to an agent when the action it selects is a cell that is unknown, meaning it has the value of -1. This is given a higher positive reward than the unoccupied cell because the robot should be rewarded higher for trying to explore more unknown spaces. The reason why the unoccupied cell and unknown cell rewards have only a 100 point difference is because the desire is for the robot to determine the best place to explore is an unknown cell close to unoccupied cells, therefore exploring cells that are, in a sense, frontiers. The occupied cell reward is a negative reward given to the agent when the action it selects is a cell that is occupied, meaning it has the value of 100. Finally, the near wall reward is the negative reward the OpenAI Gym node gives to the agent if it selects an action cell that is four connected to an occupied cell. This is given to the agent because during exploration there is an increased chance for the robot to get stuck if it explores very close to obstacles. Since the point of the exploration is to search the space as quickly as possible, the model should avoid putting the robot in situations where it is more likely to get stuck. The remaining four rewards are discussed next, as they are shared by all models.

Table 3.2 shows the rewards for models 3 and 4. These same rewards are also shared by models 1 and 2 with the same values for the rewards. The OpenAI Gym node gives the end episode reward to the agent at the end of the action, if, at the end of the action, there are no more frontiers left. During the travelling from the starting point to the action cell point, in the case of models 1 and 2, or the selected frontier center, for models 3 and 4, the open cell found reward is given to the agent per open cell it finds. The reward is the open cell found reward value times the amount of open cells that were found during travelling. The occupied cell found reward follows the same logic, meaning the reward is the occupied cell found reward value times the amount of occupied cells that were found during travelling. Finally the travel time is a negative reward the OpenAI Gym node gives to the agent for how much time it takes from travelling from the robots current position to the action cell point or the selected frontier center. The travel time negative reward is multiplied by the total time in seconds of travel time.

Rewards Setup	
Reward Reason	Value
Unoccupied Cell	400
Unknown Cell	500
Occupied Cell	-500
Near Wall	-50
End Episode	500
Open Cell Found	50
Occupied Cell Found	25
Travel Time	-50*s

Table 3.1: Rewards Given to the Robot Agent at the End of an Action: RL Models 1 and 2

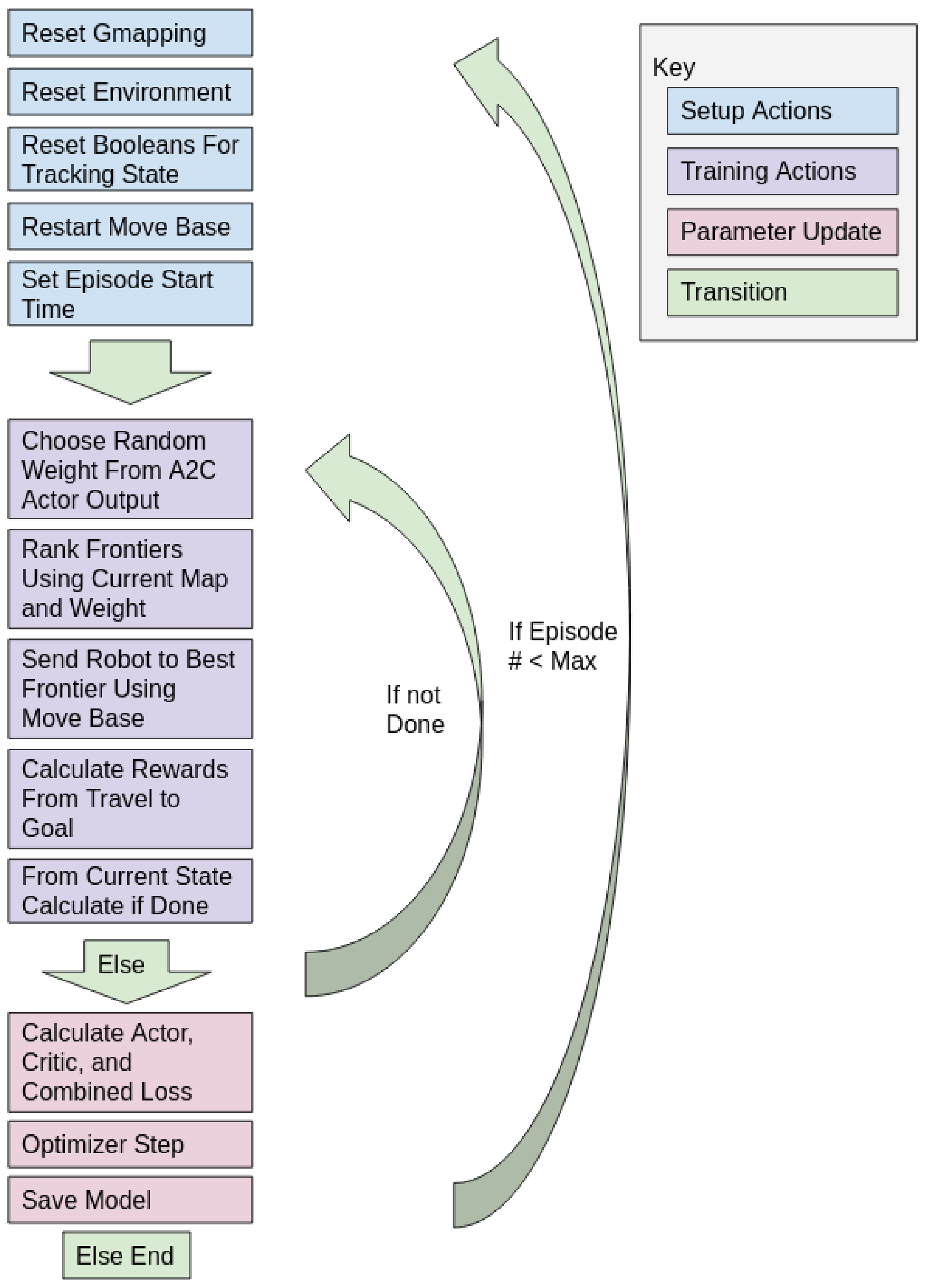
Rewards Setup	
Reward Reason	Value
End Episode	500
Open Cell Found	50
Occupied Cell Found	25
Travel Time	-50*s

Table 3.2: Rewards Given to the Robot Agent at the End of an Action: RL Models 3 and 4

I trained and implemented the models using a single robot exploring, and extended my research by implementing the solution on a swarm.

3.12 Model Pipeline

The training pipeline for model 4 is shown in Figure 3.10.



Reset Gmapping
Reset Environment
Reset Booleans For Tracking State
Restart Move Base
Set Episode Start Time
Choose Random Weight From A2C Actor Output
Rank Frontiers Using Current Map and Weight
Send Robot to Best Frontier Using Move Base
Calculate Rewards From Travel to Goal
From Current State Calculate if Done
Else
Calculate Actor, Critic, and Combined Loss
Optimizer Step
Save Model
Else End
If not Done
If Episode # < Max
Key
Setup Actions
Training Actions
Parameter Update
Transition

Figure 3.10: Training Pipeline for Model 4

3.13 Training Pipeline on Cluster

To make this application available for various members of the team to run, each with different operating systems, it was necessary to dockerize the solution. The other project teammates, Ferrera et al. [15] created a docker container that builds around the project's source folder. It uses noVNC [31] to have a visual interface to the running docker container. This allows for ROS to be run in the Docker container but for the user to be able to interface with RViz and Gazebo interfaces.

The WPI Turing cluster does not support Docker, but supports Singularity [19]. Singularity is a container platform which allows users to build and run containers which allow for entire software environments to be run on various system setups. Singularity can create Singularity containers from Docker images. Therefore, to get the Docker container onto the cluster, I uploaded the Docker image to a private Docker Hub page, then pulled onto the cluster using Singularity's Docker Hub interface. Unfortunately, Singularity requires that the container permissions be inherited from the user permissions. Because the cluster only allows Admins to have sudo permissions, sudo permissions are never given to the Singularity container. This caused complications. However, the fix was put all the source code for the project is put under root. Before the Docker container is pushed to dockerhub, I made root recursively readable, writable, and executable by all groups on my local machine, which has sudo permissions. I also conducted all installations and other processes that require sudo before pushing the container to Docker Hub. Then, once I pulled the image down to Singularity and built the Singularity container using the Docker image, all of root is accessible by any user, so the ROS code can be accessed and run.

In order to run the code on the clusters vast array of CPUs and GPUs, I initiated the job in `sinteractive` mode. This allows the user to interact with the cluster and the code during the job, instead of just submitting a job script and waiting for the results. Due to the roslaunch files, it is necessary to do it this way. This allows for me to restart the ROS nodes in the middle of the job if something unexpected occurs. I ran the training on the cluster and the job was granted the following resources: 8 CPU cores, 2 A100 GPUs, and 48 gb of RAM. The maximum training time was limited to 7 days, as that is the longest job that is allowed.

I then ran the Singularity container in writable mode in order to access and edit all the files. I ran a singular launch file to start up all of the required simulation tools, ROS packages, and training scripts. I saved the models resulting from training as pth files to the cluster. To easily transfer these files between machines, I used `scp`, and to easily load the models during run time, I used the Pytorch function `load_state_dict`.

3.14 Swarm Extension

3.14.1 System Node Interactions

The system is broken down into various components, which individual students or groups of students took ownership of. The overview of all of the components and their interactions is seen in Figure 3.11. This thesis focuses on the development of the "Explore a Frontier" box and the blue "Choose Frontier" diamond in Figure 3.11. A box is an action and a diamond is a datatype. Figure 3.12 shows the in depth swarm state machine. For the swarm extension, this thesis implementation provides the list of ordered frontier centers when the blue Frontiers diamond is called.

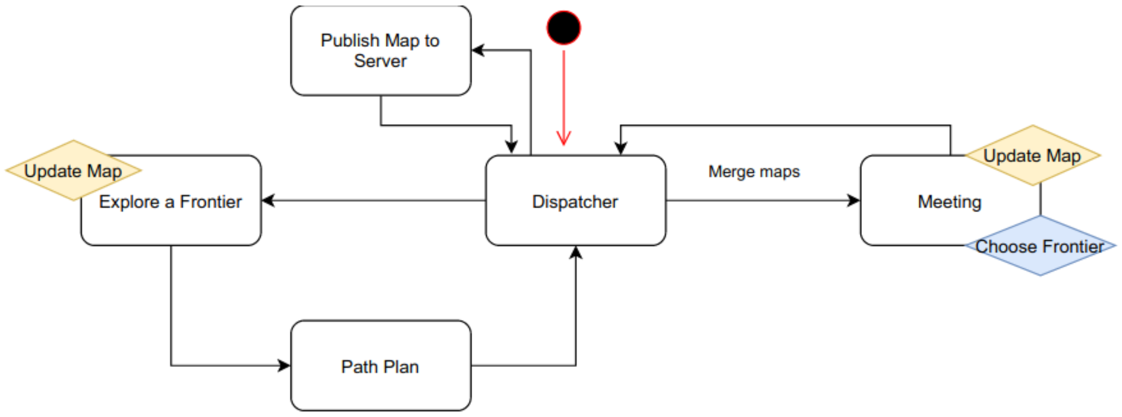


Figure 3.11: State Machine Overview

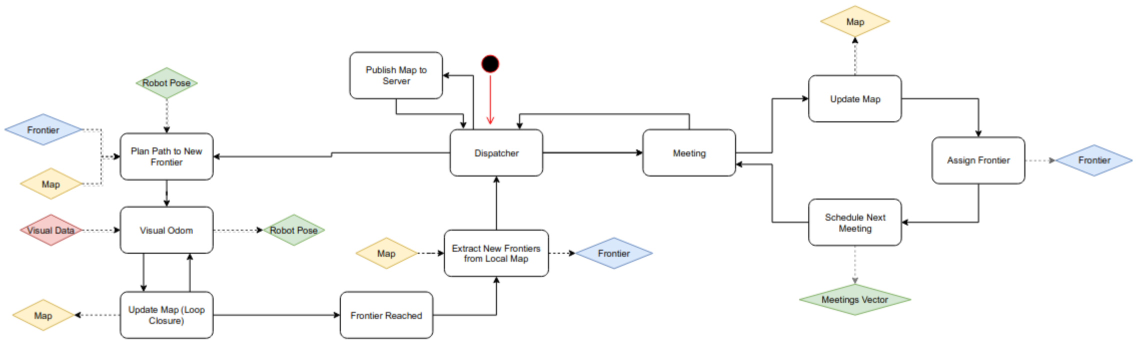


Figure 3.12: State Machine Overview More Detailed

3.14.2 Request and Response with State Machine

I connected the frontier ranking node to the rest of the system by using ROS topics for request and response. When the state machine desired the frontier centers, it would publish a request to the topic `/frontier_request`, consisting of a String of the robots namespace. The frontier node will wait until the occupancy grid for that robot is published, then runs the frontier calculations explained in the Frontier Calculation subsection. The top ten frontiers are returned to the state machine via the topic `/frontier_list`. The find fringe frontier detection node returns the frontiers as a PoseArray, which is an array of poses, with each one being the center of a frontier cluster. Each frontier pose is in the frame of the requested robot's map. A generalized state machine diagram of this interaction can be seen in Figure 3.13

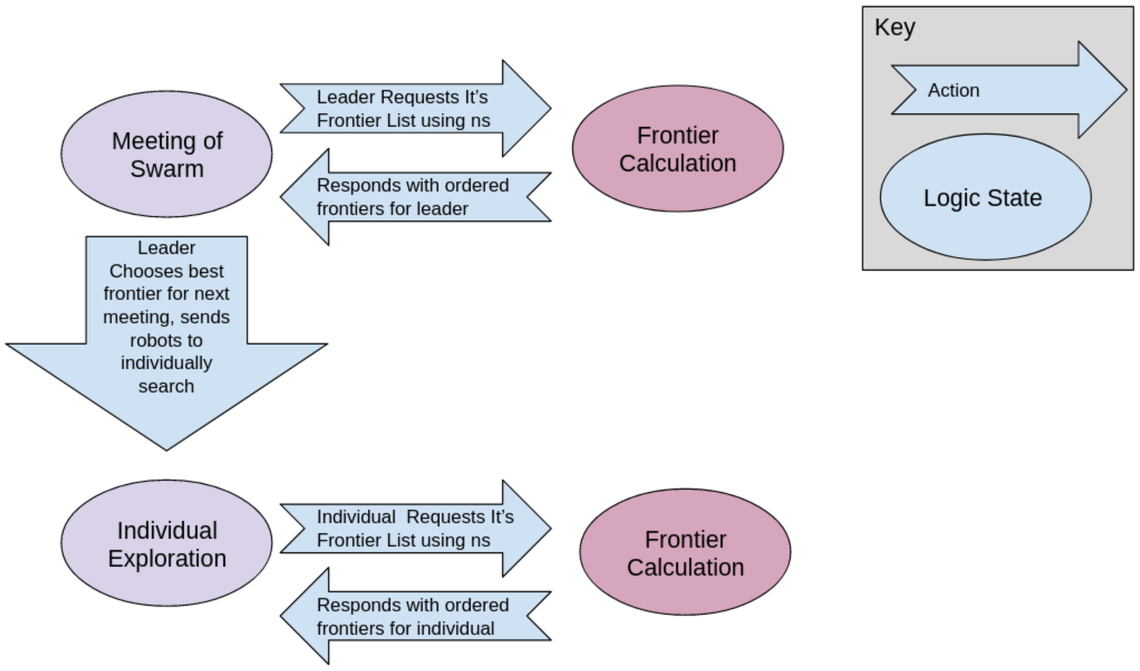

Key
Action
Logic State
Meeting of Swarm
Leader Requests It's Frontier List using ns
Frontier Calculation
Responds with ordered frontiers for leader
Leader Chooses best frontier for next meeting, sends robots to individually search
Individual Exploration
Individual Requests It's Frontier List using ns
Frontier Calculation
Responds with ordered frontiers for individual

Figure 3.13: The State Machine Request and Response with the Frontier Calculation Node

The state machine requests this list from the frontier node during two phases of exploration. It requests the list during the meeting phase, when the group leader decides which is the best frontier for the robots to conduct their next meeting at. Robots each also request the list of their own frontiers when they are independently exploring.

Chapter 4

Evaluation

4.1 Single Robot

4.1.1 Data Gathering

In order to gather data efficiently, I created an automated testbed must for both the set weight method and the model determined weight method. The testbed requires a complete environment reset once the environment is completely mapped. Fortunately, I already created most of this logic and workflow for the reinforcement learning pipeline.

In order to keep everything the same during both methods runs, I used the exact same launch files for the simulation setup. For gathering data for the model weight version, I used the same class for the training, with some edits to remove the training code. I used this class because it had the simple interface for knowing when the exploration was done, and for handing the correct inputs to the model. The class also had the code implemented to easily send the robot to the chosen frontier, as robots were sent to frontiers during training. The baseline for the evaluations is a set weight in the frontier calculations. I implemented data gathering for the set weight approach using the OpenAI ROS Environment. Using the OpenAI ROS Environment keeps the two implementations setups the same, to reduce discrepancies in timing due to simulation setup differences, and the Environment makes it easier to easily collect data due to the episode reset functionality. Both methods of data collection use the following logic. I set the max episode limit to 120, which was the desired amount of data points to collect. The desired amount of data points for evaluation were 100, but the additional 20 points allowed for outliers and such to be removed from the larger dataset. At the beginning of every 'episode', the data collection node recorded the start time. At the end of every 'episode,' when there were no frontiers left and the environment returned done, the data collection node recorded the system end time. The data gathering node appended these times to a csv of results, and then the environment reset and a new 'episode' for data collection began. This made for a simple streamlined data collection pipeline.

I collected the data for the two frontier ranking methods on a GS65-Stealth-Thin MSI laptop. The laptop has 16 GiB of RAM, a 1 TB disk capacity, i7-8750H processor, and a NVIDIA GeForce GTX 1070 graphics card. The computer was running no other tasks during the training, and was on a laptop cooling pad to reduce CPU slowdown over time as the computer's core temperature rose.

4.1.2 Statistical Significance Evaluation Setup

In order to prove the validity of the results, a statistical test must be run on the data. A comparison test should be run to investigate the differences between data set means. The only predictor for the data sets is time, which is categorical. The outcome variable is quantitative, and and both groups come from the same population. For both method evaluations, I used the same single TurtleBot3 robot, with the same environment, and mostly the same software setup, with the only main difference having been the different algorithm implementations. As previously mentioned, both methods were run on the same computer, which was doing no other tasks during the training time. There are also exactly two groups being compared. This leads to a paired t-test being the most appropriate statistical test [44].

Before running any of the evaluations, I created a hypothesis on which data set would have an overall lower exploration time. Since the model should use the current state of the map for determining the best frontier weight, it stands to reason that the model derived weight exploration method is more optimized to the specific map configuration then the set weight method. This leads to the hypothesis that the model derived weight method would end up with overall quicker exploration.

In the following graphs, the only comparisons were between the set weight method and the model derived weight method. I determined the set weight method was the baseline. I determined the baseline is because Niroui et al. in [30] determined that the second best weight for the frontier ranking calculation is 0.75. All the other weight setups fell far below their custom machine learning implementation and set 0.75 weight implementation. Their results can be seen in Figure 4.1. The 0.75 set weight method can be seen in red, and their reinforcement learning implementation can be seen in blue.

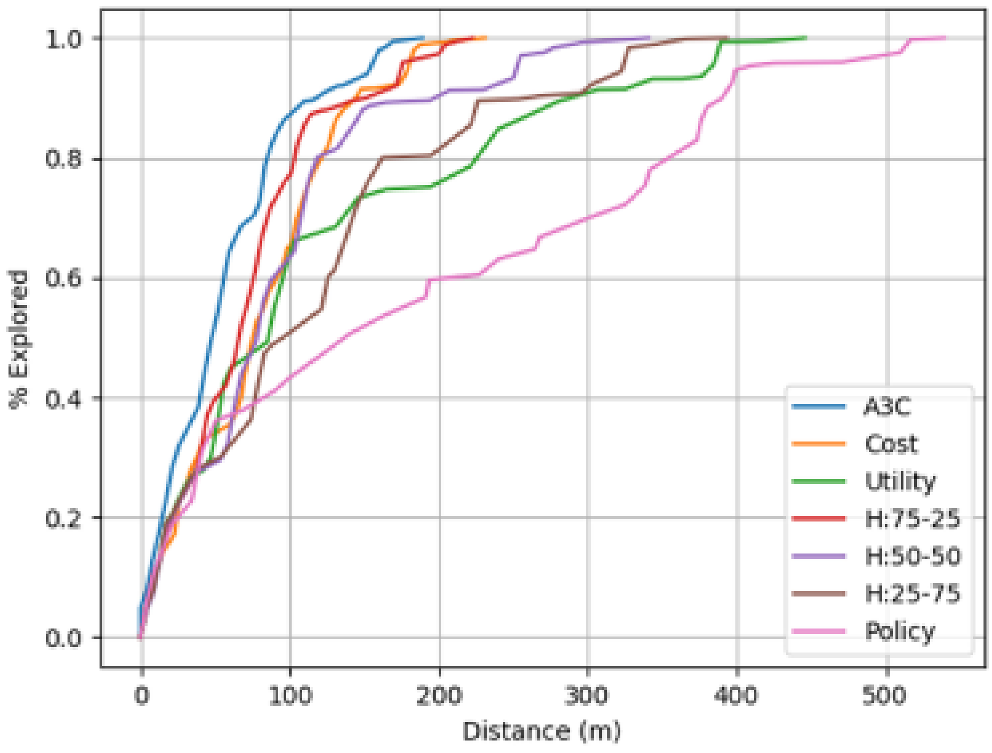


Figure 4.1: Mapping Results From Various Frontier Ranking Implementations [30]

4.1.3 Results

It is first important to prove the statistical significance of the two data sets. As mentioned above, before running the data collection, I outlined the pre-conceived notion that the model derived weight method would lead to an overall shorter time to map completion. With paired t-tests, either one-tailed P values or two-tailed P values can be compared. Both one-tailed and two-tailed P values are based on the same null hypothesis [34]. This null hypothesis is that the populations are the same and that the difference between the means of the samples is due to circumstance and chance. A one-tailed P value is appropriate when the difference in means between the two groups can only go in a single direction. A one-tailed P value should be used in two conditions. The first is when the data set with the larger mean can be predicted before data collection. The second is if the data set with a larger mean ended up being the opposite one of the pre-formulated hypothesis, and this difference would be attributed to chance rather than admitting that the opposite data set actually had a higher overall mean. This is the case in this situation, so the one-tailed P value was used to either accept or reject the null hypothesis.

I collected the data in two files. I merged these files and used LibreOffice Calc to run the paired t-test. The results of the paired t-test are in Table 4.1. The constants I used for comparison to accept or reject the null hypothesis are listed in Table 4.2. The one tail P-value will show if the null hypothesis should be accepted, and it is compared with alpha [49]. In this case, the P value was 0.0015, which was significantly less than alpha, which was 0.05. A value of 0.05 for alpha is common practice for statisticians. Since the P value was significantly less than alpha, this means there is very strong evidence that the null hypothesis is rejected, so the two groups are different. This means the following results are statistically significant, since the two methods having different average exploration times was not due to chance but due to one being overall faster than the other.

Paired T-Test Results		
Measurement	Set Weight Method	Model Weight Method
Mean	702.1808	643.2350
Variance	21845.4212	11820.7527
Observations	100	100
Pearson Correlation	-0.1137	
Observed Mean Difference	58.9458	
Variance of the Differences	37319.6592	
df	99	
t Stat	3.0512	
P (T¡=t) one-tail	0.0015	
t Critical one-tail	1.6604	

Table 4.1: T Test Results for Two Data Groups: Set Weight for Frontier Ranking Calculations and Model Derived Weight for Frontier Ranking Calculations

T-Test Constants	
Constant	Value
Alpha	0.05
Hypothesized Mean Difference	0

Table 4.2: T Test Constants For Comparison

The next results show the average time for both methods to fully explore the environment until map completion. This is exploration until the map has no frontiers remaining. The overall results are seen in Table 4.3 with the differences between the total exploration time of the two methods shown in Table 4.4. The first column of results for Table 4.3 shows the total 'raw' time for 120 runs, with no outliers removed. The model derived weight method performed significantly better, as it completed in 243 less seconds on average. This is a 24% improvement. The second column of results for Table 4.3 shows the total time for 100 runs, with outliers removed. The outliers I removed in this case are the total exploration times over 1000 seconds. 1000 seconds was chosen as the upper threshold since when watching the data collection, I noted that exploration times over 1000 seconds tended to occur when Move Base had complications with guiding the robot, when the chosen frontier center was close to a wall, and when Move Base would oscillate between two best path solutions for a while, which would delay exploration. For the two groups of data with outliers removed, the Model Weight Method did better again. This implementation lead to a 59 second decrease in total search time. The overall percent of time decrease was less than with outliers, as the overall total search time decrease from the set weight method to the model derived weight method was 8.4%.

Average Time to Map Completion		
Method	Average Time (outliers)	Average Time (no outliers)
Set Weight Method (0.75)	1008	702
Model Weight Method	765	643

Table 4.3: Average Time for the Robot to Complete the Entire Exploration

Time Improvement Using Model Weight Method		
	Average Time Decrease (outliers)	Average Time Decrease (no outliers)
Time	243	59

Table 4.4: Total Exploration Time Improvement Using the Weight From the Model Output

The following results are a graphical representation of the data collected from the two methods. Figure 4.2 shows the total time until completion for both methods in a box and whisker plot format. The data for this method includes the run data with the outliers over 1000 seconds removed. The set weight method is shown in blue and the model based method is shown in orange. The lower quartiles for both methods are essentially the same, but the upper quartile for the set weight method is significantly higher. The mean, denoted by the x in each datasets box, is lower for the model based method.

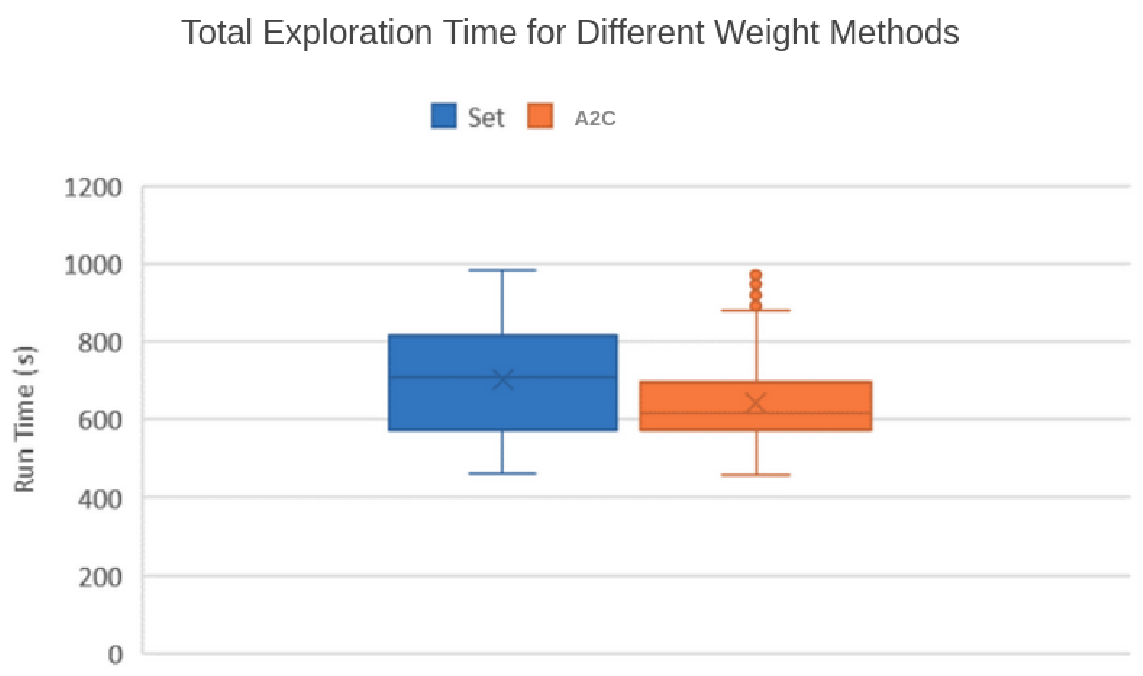


Figure 4.2: Box and Whisker Plot Showing Two Approaches With Outliers Over 1000 Seconds Removed

Figure 4.3 shows the total time until completion for both methods in a box and whisker plot format. The data for this method includes the run data with the outliers over 2000 seconds removed. The Set weight method is shown in blue and the model based method is shown in orange. The lower quartiles for both methods are essentially the same, but the upper quartile for the set weight method is significantly higher. The mean, denoted by the x in each datasets' box, is lower for the model based method.

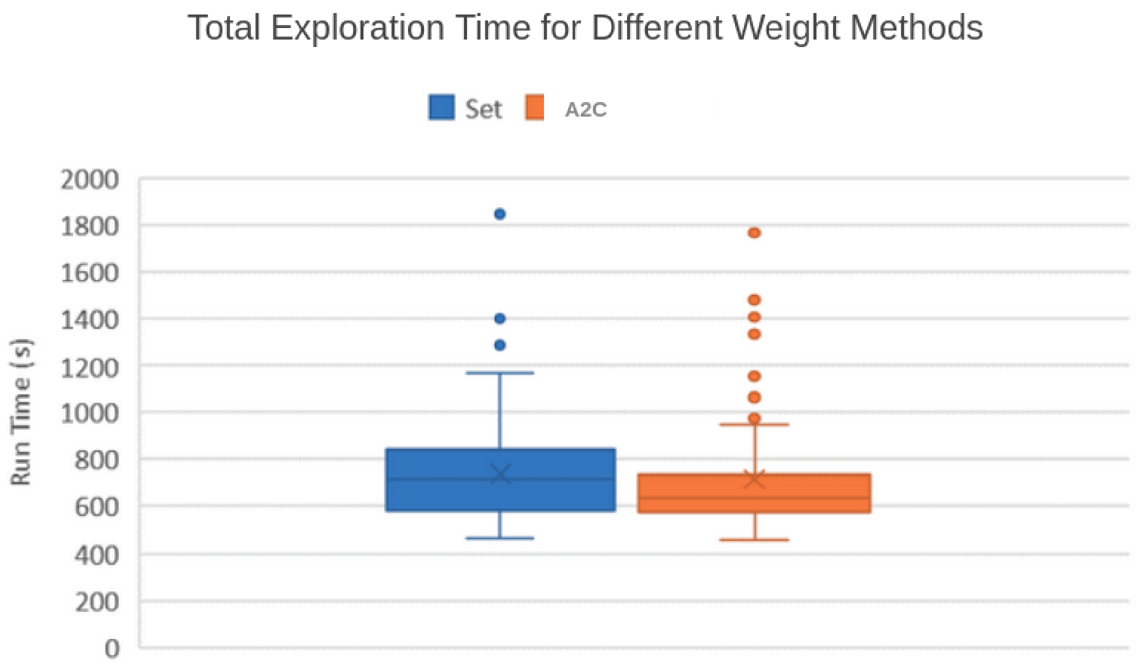


Figure 4.3: Box and Whisker Plot Showing Two Approaches With Outliers Over 2000 Seconds Removed

Figures 4.4 and 4.5 show each data point that was used to create the box and whisker plots in Figures 4.2 and 4.3, respectively. It can be seen that the orange dots representing the model derived weight run times settle lower on the graph than the blue dots representing the set weight method data. This is the case for both Figure 4.4 and Figure 4.5.

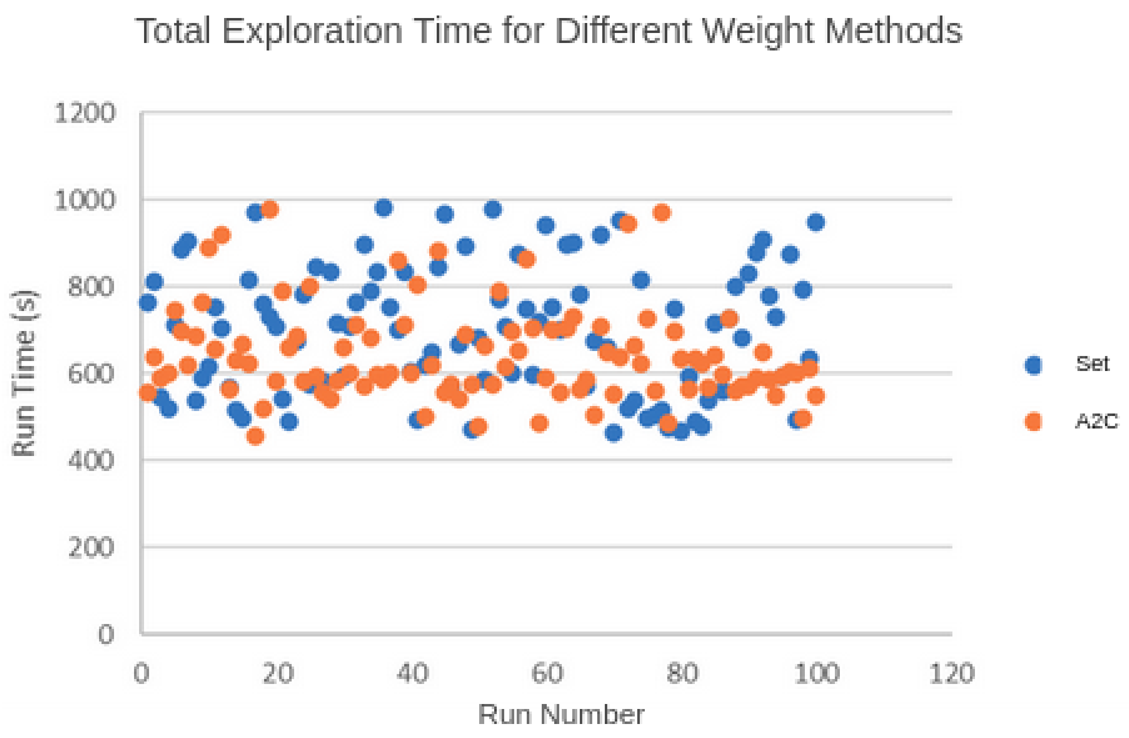


Figure 4.4: Scatter Plot for Box and Whisker Data. Outliers Over 1000 Seconds Removed

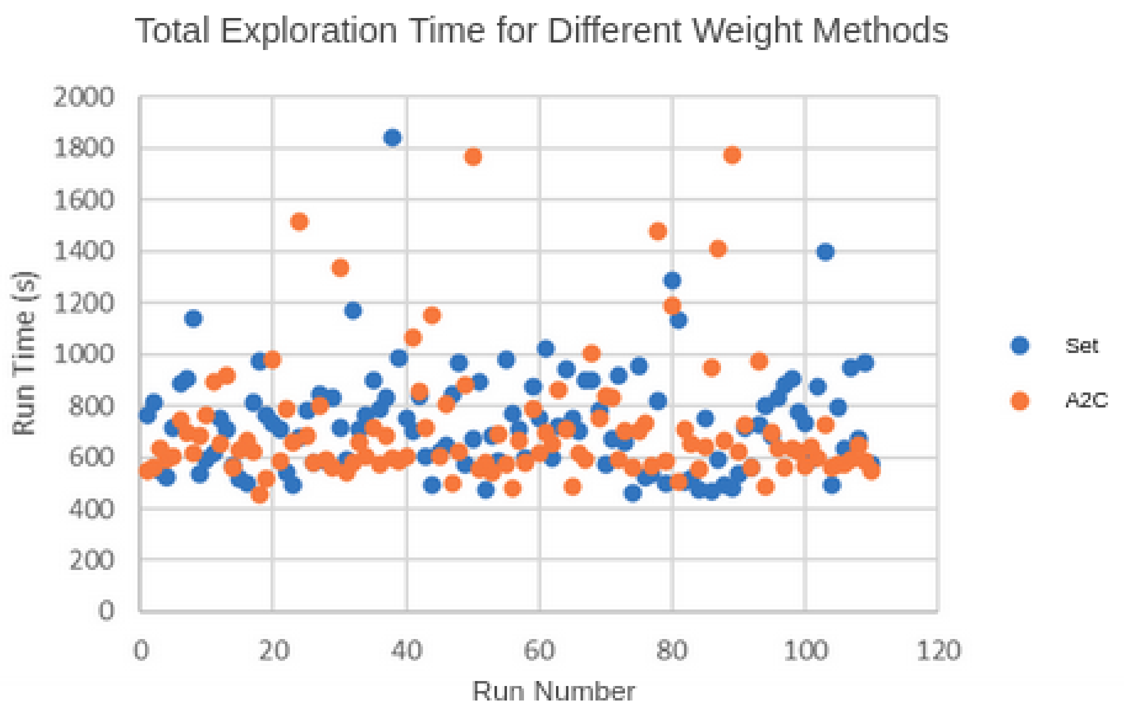


Figure 4.5: Scatter Plot for Box and Whisker Data. Outliers Over 2000 Seconds Removed

Figures 4.6 and 4.7 visually show the amount and distribution of the outliers for each dataset. Figure 4.6 shows the outliers in red for the complete data set of 120 runs of the model derived weight method. Figure 4.7 shows the outliers in red for the almost complete data set of 119 runs of the set weight method. One of the runs for the set weight method ran for 31,494 seconds. As this was fully automated, it is impossible to know the exact reason for this. A likely possibility is the robot got stuck multiple times around a corner and it took a while to recover. Figure 4.8 shows the complete data for all 120 runs of the set weight method, but the graph is difficult to read due to the large difference between the outlier and the rest of the data. Removing this outlier allows for the data to be viewed better.

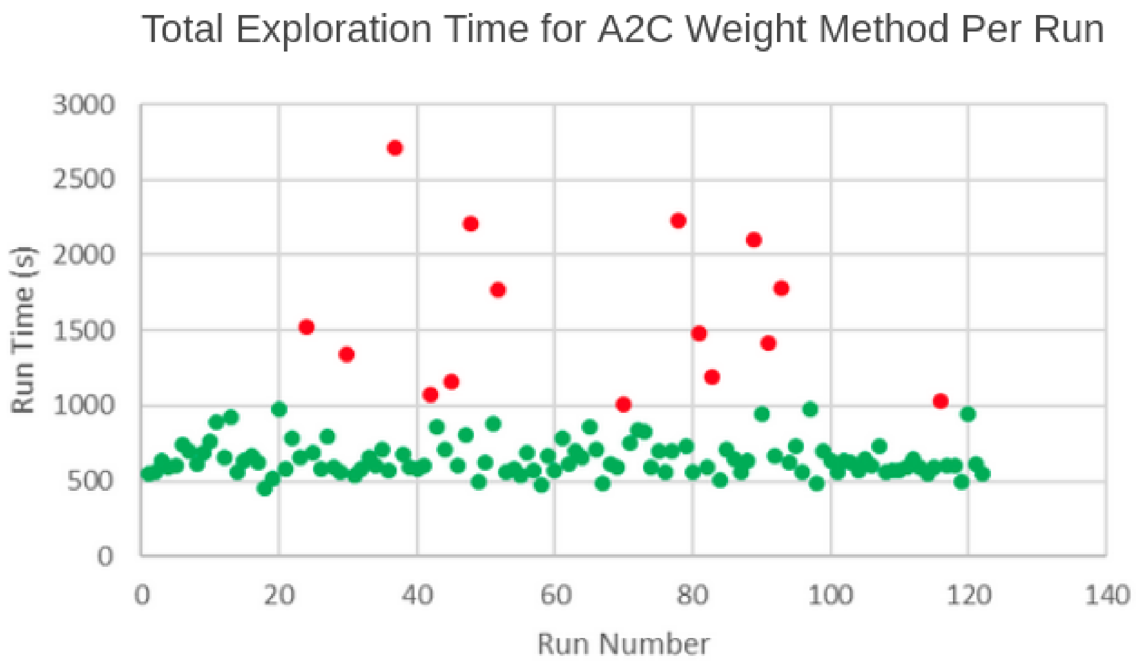


Figure 4.6: Scatter Plot of Total Exploration Time Until Map Completion using the Model Based Weight Method. Outliers in Red

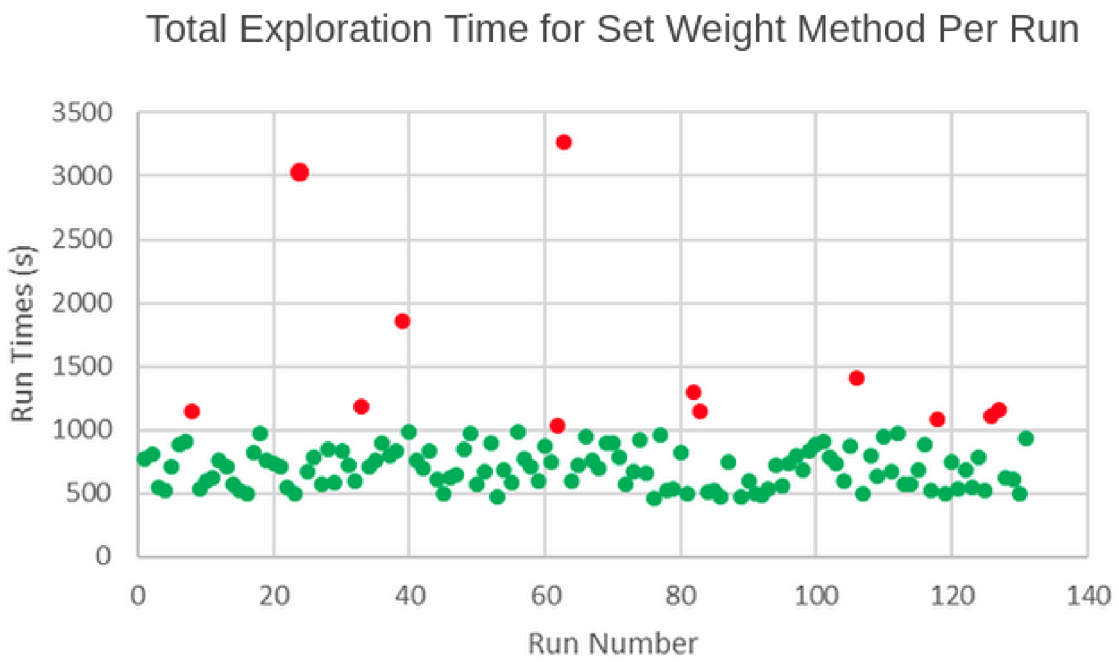


Figure 4.7: Scatter Plot of Total Exploration Time Until Map Completion using the Set Weight Method. Outliers in Red

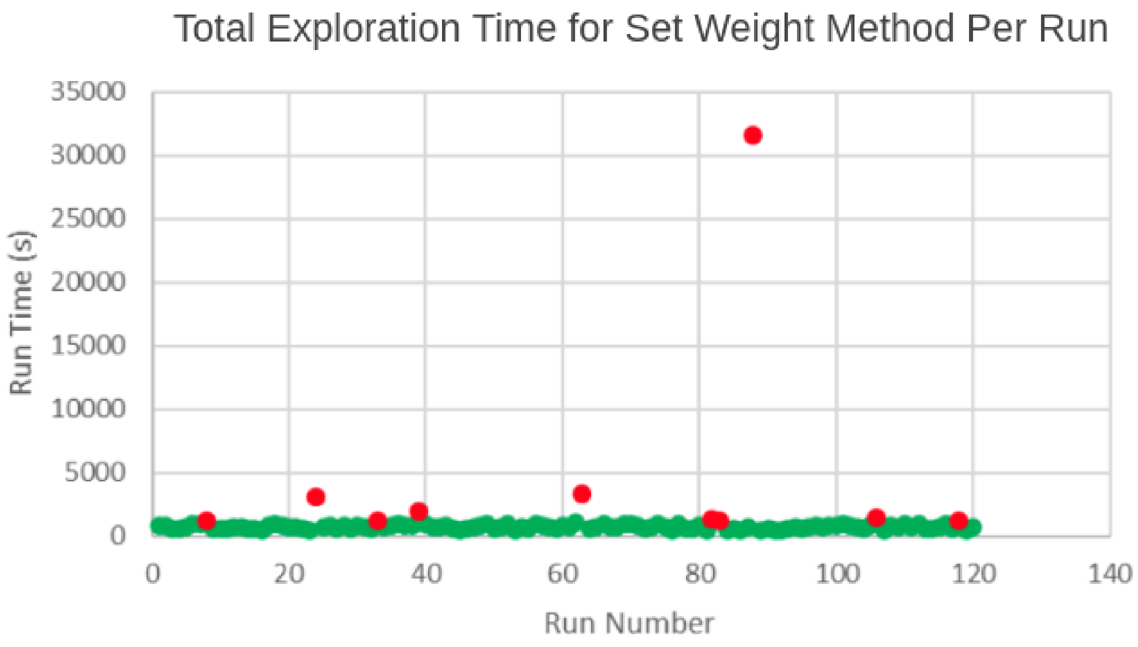


Figure 4.8: Scatter Plot of Total Exploration Time Until Map Completion using the Set Weight Method. Outliers in Red. Extreme Outlier Impacts Scale

4.2 Discussion of Results

Every possible statistical measurement leads to the logical conclusion that the model derived weight frontier ranking approach leads to a quicker exploration time than the set weight frontier ranking approach.

4.3 Observations

While watching both methods gather data, it was clear that at the beginning of the model versions run, it would take about 20 seconds upon initializing to load the map. Since the set weight method did not use a machine learning model, it did not suffer from this time delay.

Chapter 5

Conclusion

5.1 Achievements

This thesis has achieved numerous goals. I created a RL pipeline for complex training scenarios, and implemented resetting functionality to commonly used ROS packages. I show how various input sizes affect the quality of the model's output and which models can work for this given scenario. I have shown the need for reduced dimensionality for the models inputs and outputs in order to receive a quality output by the end of the training. This thesis has provided the steps to reproduce a working RL model that outputs an optimal weight for the frontier ranking calculation based on the current map of the robot. The results show that the optimized weight model reduces exploration time by 8.4%, or, on average, 59 seconds, for a single robot searching the scene. This time decrease in searching is significant for the swarm's intended purpose: search-and-rescue in a fire scenario. A small flame can turn into a roaring fire in less than 30 seconds, so this increase in speed could save someones life [18].

5.2 Future Work

There is much future work to be done in this research space. For this scenario, an interesting approach would have been to have a set number of discrete points from the map to test on as action spaces. For example, 25 spaces would be uniformly chosen by the training script from the map. Then, the robot trains on which of the 25 points is best to go to at any given point. That would have been interesting to test and compare to the current working solution.

There are other machine learning approaches that would have been interesting to investigate further, if given more time. Pointer networks were introduced by Vinyals et al. in 2016 [50]. Pointer networks do not train on the exact values in an array, but rather they train on the location in the sequence. Pointer nets have been used for natural language processing (NLP) approaches, specifically translation applications. For example, in French the phrase ”le ballon vert” would, word for word, translate to ”the ball green” in English. This ordering of noun and adjective does not follow traditional English language rules. A pointer net could use a dictionary to learn the word for word translation, and use reinforcement learning to rearrange the sentence to the correct order using the indexes of the words. Therefore the input to the pointer net could be ”le ballon vert” and the output could be ”the green ball.” Pointer nets have also been used to solve TSP problems, and computing Delaunay triangulations. It would be interesting to investigate using the same consistent weight for the frontier ranking calculation, and given data about the current state of the robot, such as the current map or pose, train on what index in the frontier array would be best at a given time. I could attempt to answer the following question: is the highest ranked frontier always the best to go to, or could the second ranked one actually receive the best reward?

5.3 Lessons Learned

The WPI Turing Computing cluster is not well documented and has many opportunities for improvements. In order to learn about the cluster and how to run jobs, it was necessary for me to reach out to an admin. The documentation for logging into the cluster and such even requires having it be sent by an admin. Fortunately the admins try their best to be helpful but having basic outline hosted on the same site that users sign up to access the cluster would have been very helpful. It took a week to receive the initial instructions on how to use the cluster after the account had been approved. This was a blocker for that entire week, and not having complete documentation only added to that delay. The cluster also forcing Singularity to inherit the sudo permissions of the user is unnecessary and caused another week delay for creating a workaround for that issue that did not break ROS. If the cluster supported Docker, that would make students lives much easier. Fortunately I built the time into the schedule to handle multiple week-long delays, but it still had an unfortunate impact on the project.

Another lesson learned is that using ROS in a Docker container and creating it so the user is able to view Rviz and Gazebo GUI windows takes a significant amount of effort and time.

Another lesson I learned is that OpenAI ROS did not support resetting of popular ROS SLAM packages. OpenAI ROS developers advertised the system as an inclusive out of the box solution to creating reinforcement learning training pipelines. It was intended to have the machine learning programmer not deal with the lower-level ROS setup. Unfortunately diagnosing the cause of many training problems being the result of the packages not properly resetting took time, along with the solution implementation. If I had known this complication beforehand it would have helped during the project planning process.

The process for creating the data gathering pipeline took a bit of time after the model was trained. Although fortunately I took a lot of the setup for an automated data gathering pipeline from the reinforcement learning pipeline. The time for gathering over one hundred runs with real time simulation data for both of the methods took over a week. Fortunately I anticipated this during the planning pipeline, but it was a bit of a hindrance. To ensure nothing altered the data, such as starting up a software that consumes a lot of the CPU in the middle of a run, I could not use the computer gathering the data.

I also learned some important lessons about reinforcement learning from this thesis. The first is that to get a model to converge, I need to select the rewards very carefully. Since the reward tied to each characteristic of the observation, the rewards relative value to one another must be considered, since they add up to an overall reward which the agent uses to adjust the model. I needed to choose weights that made the model prioritize time reduction over other improvements. Therefore, the longer the robot took to get to one location to another, the larger the penalty it got, and that negative reward had a much larger magnitude than the other rewards. The other lesson I learned about reinforcement learning is that it is more difficult than I anticipated to create a model that converges to a working solution. Even if the rewards are reasonable and should lead the model to convergence, too much dimensionality in the input will make the model either not converge at all, or take an unreasonable amount of time to train to converge to a useable solution. Therefore, I needed to go through four iterations of training different model setups in order to create one that converged. Fortunately complications like this were anticipated in the project timeline, but it took more time than I thought.

Overall this thesis taught me a lot about reinforcement learning applications for robot exploration, and I am looking forward to the future strides that will be made in the field.

Bibliography

[1] A* comparison.

[2] Bae, H., Kim, G., Kim, J., Qian, D., and Lee, S. Multi-robot path planning method using reinforcement learning. *Applied Sciences 9*, 15 (2019).

[3] Balakrishnan, K. *TensorFlow Reinforcement Learning Quick Start Guide: Get Up and Running with Training and Deploying Intelligent, Self-learning Agents Using Python.* Packt Publishing Ltd, 2019.

[4] Burgard, W., Moors, M., Fox, D., Simmons, R., and Thrun, S. Collaborative multi-robot exploration. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)* (2000), vol. 1, pp. 476–481 vol.1.

[5] Carew, J. M. Reinforcement learning, March 2021.

[6] Cesa-Bianchi, N., Gentile, C., Lugosi, G., and Neu, G. Boltzmann exploration done right, 2017.

[7] Chen, F., Bai, S., Shan, T., and Englot, B. Self-learning exploration and mapping for mobile robots via deep reinforcement learning. *AIAA Scitech 2019 Forum* (2019).

[8] Christianos, F., Schäfer, L., and Albrecht, S. Shared experience actor-critic for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 10707–10717.

[9] Dettmers, T. Deep learning in a nutshell: Core concepts, 2015.

[10] Dimensionality & high dimensional data: Definition, examples, curse of, 2016.

[11] Donalek, C. Supervised and unsupervised learning.

[12] Faigl, J. Approximate solution of the multiple watchman routes problem with restricted visibility range. *IEEE Transactions on Neural Networks 21*, 10 (2010), 1668–1679.

[13] FAIGL, J., VANĚK, P., AND KULICH, M. Self-organizing map for determination of goal candidates in mobile robot exploration. In *Proceedings of the 22nd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning* (Apr. 23 2014).

[14] FAIGL, J., VONÁSEK, V., AND PREUCIL, L. Visiting convex regions in a polygonal map. *Robotics Auton. Syst. 61* (2013), 1070–1083.

[15] FERRERA, T., MCLAUGHLIN, C., AND NIKOPOULOS, P. Online map synchronization for multi-robot slam.

[16] gmapping.

[17] Gradient descent.

[18] Home fires.

[19] Introduction to singularity.

[20] KAMALOVA, A., KIM, K. D., AND LEE, S. G. Waypoint mobile robot exploration based on biologically inspired algorithms. *IEEE Access 8* (2020), 190342–190355.

[21] KARAGIANNAKOS, S. The idea behind actor-critics and how a2c and a3c improve them, November 2018.

[22] Keras: the python deep learning api.

[23] LI, H., ZHANG, Q., AND ZHAO, D. Deep reinforcement learning based automatic exploration for navigation in unknown environment, 07 2020.

[24] MARTELL, V., AND SANDBERG, A. Performance evaluation of a* algorithms, 2016.

[25] MIRJALILI, S., AND LEWIS, A. The whale optimization algorithm. *Advances in Engineering Software 95* (2016), 51–67.

[26] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning* (New York, New York, USA, 20–22 Jun 2016), M. F. Balcan and K. Q. Weinberger, Eds., vol. 48 of *Proceedings of Machine Learning Research*, PMLR, pp. 1928–1937.

[27] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T. P., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. *Asynchronous Methods for Deep Reinforcement Learning.* arXiv, 2016.

[28] move_base.

[29] Nachum, O., Norouzi, M., Xu, K., and Schuurmans, D. Bridging the gap between value and policy based reinforcement learning, 2017.

[30] Niroui, F., Zhang, K., Kashino, Z., and Nejat, G. Deep reinforcement learning robot for search and rescue applications: Exploration in unknown cluttered environments. *IEEE Robotics and Automation Letters 4*, 2 (2019), 610–617.

[31] novnc info.

[32] Nyuytiymbiy, K. Parameters and hyperparameters in machine learning and deep learning, 2020.

[33] O'Donoghue, B., Munos, R., Kavukcuoglu, K., and Mnih, V. Combining policy gradient and q-learning, 2016.

[34] One-tail vs. two-tail p values.

[35] Openai gym.

[36] openai_ros.

[37] Particle swarm optimization.

[38] Piech, C. K means.

[39] Pinciroli, C. Cs4341: Search, 2020.

[40] Pytorch.

[41] Sak, H., Senior, A., and Beaufays, F. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition, 2014.

[42] Sarkar, A. A brandom-ian view of reinforcement learning towards strong-ai, 2018.

[43] Simmons, R. G., Apfelbaum, D., Burgard, W., Fox, D., Moors, M., Thrun, S., and Younes, H. L. S. Coordination for multi-robot exploration and mapping. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (2000), AAAI Press, p. 852–858.

[44] Statistical tests.

[45] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction.* The MIT Press, 11 2017, p. 152.

[46] Tai, L., and Liu, M. A robot exploration strategy based on q-learning network. In *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)* (2016), pp. 57–62.

[47] torch.optim.optimizer.step.

[48] torch.tensor.backward.

[49] Understanding p-values — definition and examples.

[50] Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks, 2015.

[51] What is the difference between bias and variance?, 2022.

[52] Wiki, R. turtlebot stage, Sep. 25 2018.

[53] Wu, C., Rajeswaran, A., Duan, Y., Kumar, V., Bayen, A. M., Kakade, S., Mordatch, I., and Abbeel, P. Variance reduction for policy gradient with action-dependent factorized baselines, 2018.

[54] Yamauchi, B. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'* (1997), pp. 146–151.

[55] yang, y., Li, J., and Peng, L. Multirobot path planning based on a deep reinforcement learning dqn algorithm. *CAAI Transactions on Intelligence Technology 5* (06 2020).