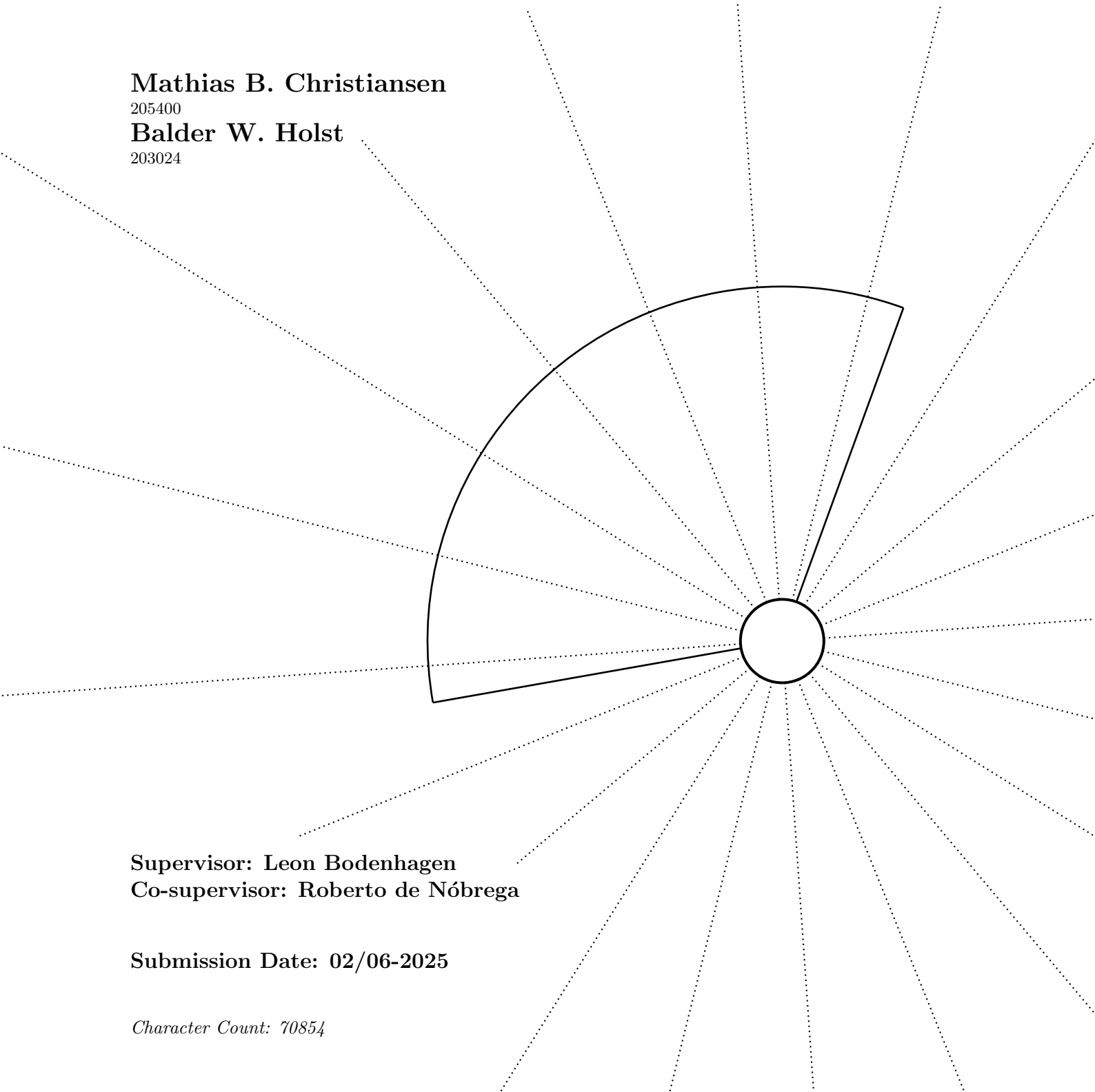
SDU

Coordinated Robot Search

Bachelor's Thesis in Robot Systems

Mathias B. Christiansen
205400
Balder W. Holst
203024



Supervisor: Leon Bodenhagen
Co-supervisor: Roberto de Nóbrega

Submission Date: 02/06-2025

Character Count: 70854

Abstract

This project explores behaviors which could be used to search a predefined and mapped environment. It includes the development of three algorithms: A gradient-based, a path-planning-based and a hybrid algorithm mixing the two. The algorithms were compared with a simple obstacle avoiding behavior as a baseline. It also contains an exploration of deep reinforcement learning as it applies to the topic of multi robot search. Algorithms were tested in Gazebo using ROS 2 as well as in a simple custom-built simulator. Experiments showed that the simple simulator was a good approximation of the more complex Gazebo environment in most cases, but that behavior performance did not always translate perfectly. Results gathered using the simple simulator showed that the pure pathing algorithm outperformed all other algorithms in coverage over time. The pure pathing also outperformed the gradient-based alternatives, on mean step computation time, albeit with more outliers. This indicates that gradient based methods could be used as a more stable backup behavior in corner cases where path-planning becomes too expensive.

Contents

1 Introduction **4**

2 Problem Statement **5**

3 Related Work **6**

4 Project Structure **7**
4.1 Simulated Hardware 7
4.2 Using Rust 7
4.3 Robot Behavior 8
4.4 Two Simulators 8

5 Gazebo Simulation **11**
5.1 ROS 2 11
5.2 Localization 11
5.3 Object detection 11
5.4 `botbrain` Integration 12

6 Simple Simulator **13**
6.1 Sensor Approximation 14

7 Search Algorithms **15**
7.1 The Roomba Algorithm 15
7.2 Target Contributions 15
7.3 Search Gradient 15
7.4 Proximity Grid 16
7.5 Gradient Based-Algorithm 17
7.6 Global Planner 18
7.7 Hybrid Algorithm 21
7.8 Pure Pathing Algorithm 22

8 Deep Reinforcement Learning **23**
8.1 Learning Framework 23
8.2 Reward Function 23
8.3 Trainer Implementation 24
8.4 Training a Minimal Agent 25
8.5 Training more Complex Agents 27

9 Results **28**
9.1 Botplot 28
9.2 Simulator Consistency 29
9.3 Search Algorithm Benchmarks 31
9.4 Computational Performance 32
9.5 Communication Load 33
9.6 Simulator Performance 34

10 Discussion **35**

11 Conclusion **37**

12 Further Development **38**
12.1 Real World Implementation 38
12.2 Advancing Deep Reinforcement Learning 39

13 References **41**

14 Appendix **43**
A Online Version of This Report 44
B `botbrain` Documentation 44
C GitHub Repository 44
D Repository README 45
E Turtlebot 4 Image 49
F `botbrain` Usage Example 50
G More Plots from `simple_sim` on Depot and Warehouse Maps 51
H AI Disclaimer 53

1 Introduction

Coordinated robotic search presents solutions to a range of real-world challenges. Applications include search and rescue in disaster zones, environmental monitoring, industrial inspection, and surveillance. Multi-Robot Systems offer significant advantages in these contexts, such as increased area coverage, improved robustness through redundancy, and the ability to operate in environments that are hazardous or inaccessible to humans.

Despite variations in specific mission objectives, many robotic tasks are governed by a common set of principles. Robots must autonomously explore known or partially known environments, interpret sensor data to detect targets or obstacles, and coordinate their behavior while operating under communication constraints. This project focuses on the development and evaluation of distributed algorithms for collaborative search within a mapped environment. All experiments are conducted in simulation.

To facilitate development and evaluation, a dual-simulator approach is employed. A custom lightweight 2D simulator supports rapid prototyping and visual debugging, while a more realistic 3D simulation based on Gazebo and integrated with ROS 2 provides a higher-fidelity testing environment. These environments allow for testing under close to realistic sensing and communication constraints.

The remainder of this report presents the problem statement, reviews relevant literature, and describes the hardware and software architecture used throughout the project. Subsequent sections introduce the simulator designs, detail the developed distributed search algorithms, and explore an alternative approach based on deep reinforcement learning, before evaluation of the results.

2 Problem Statement

This project focuses on developing distributed algorithms that enable collaborative robot search for a predefined object within a mapped environment. Both manually developed and deep reinforcement learning approaches are explored. During a search mission, the environment is progressively explored. It is assumed that map coverage over time is a measure of search efficiency. The effectiveness of each search algorithm is evaluated using two parameters:

- Map coverage over time
- The computational expense of the algorithm

All software developed in this project is tested exclusively in simulation; therefore, the specific communication technology is not defined. Robots are, however, constrained by a limited communication range, and the behavior algorithms should ensure that robots remain within this range to emulate real-world communication constraints. Each robot is equipped with a camera for detecting the search object and a LiDAR sensor for measuring distances to objects.

To evaluate the algorithms, demonstrate their feasibility, and support easy integration for ROS 2 based systems, they are implemented as ROS 2 nodes for simulation in Gazebo.

The project also includes the development of a simple simulation environment, which facilitates the development of search algorithms by providing visual debugging capabilities without depending on the ROS 2 stack, enabling a faster iteration cycle. This environment also facilitates training agents using reinforcement learning.

Three concrete hypotheses are presented to guide development and evaluation of the project:

- **Hypothesis 1:** A gradient-based search algorithm will outperform a basic obstacle-avoidance (baseline) algorithm in terms of coverage efficiency in a known map, but will incur higher computational cost.
- **Hypothesis 2:** A hybrid algorithm that combines gradient-based navigation with high-level path planning will achieve better coverage efficiency than a purely gradient-based method in a known map, but will incur higher computational cost.
- **Hypothesis 3:** A lightweight, custom-built simulator can approximate the key performance characteristics (e.g. coverage progression) observed in a high-fidelity simulator like Gazebo, while significantly reducing computational requirements.

3 Related Work

Multi-Robot Systems (MRS) represent a well-established field within robotics. MRS architectures are generally categorized as either centralized or distributed. In centralized approaches, a central server is responsible for assigning tasks and planning paths for all robots in the system. In contrast, distributed algorithms enable each robot to perform local planning and decision-making, often based on partial knowledge and limited communication.

Notable approaches to MRS control include market-based methods [1], dynamic programming [2], behavior-based strategies, and deep reinforcement learning [3]. In market-based task assignment, robots participate in an auction by bidding on tasks, with the highest bidder receiving the assignment. This method can lead to near-optimal task distributions, especially when task costs are well-defined. Dynamic programming approaches aim to compute optimal task allocations and trajectories based on known locations and capabilities of all robots. These methods typically require complete, up-to-date global information and are thus most suitable for centralized systems, or for distributed systems in which the full system state is shared [4]. Behavior-based approaches often draw inspiration from biological systems, such as ants and bees, and use indirect coordination methods like virtual "pheromones" or other spatial markers to guide robot behavior. In *Cooperative transport by ants and robots*, C. R. Kube and E. Bonabeau found that "a coordinated group effort is possible without use of direct communication or robot differentiation" [5]. Such strategies reduce the reliance on explicit communication and instead emphasize emergent, self-organizing behaviors.

This project adopts a similar approach by enabling robots to share a continuous stream of observations over a communication channel. These collective observations are used to update each robot's internal representation of the search map, which in turn influences its navigation decisions. This form of implicit communication allows for scalable coordination, as the time complexity of the algorithm does not increase with the number of participating robots [4].

Three algorithmic approaches are explored and implemented in this work: a gradient-based method, a frontier exploration algorithm using A* for path planning, and a hybrid strategy combining the two. A deep reinforcement learning approach is also explored, but not fully realized. While these methods may not always produce optimal exploration strategies, they are designed to be robust, computationally efficient, and adaptable to a variety of real-world scenarios.

4 Project Structure

The developed software is divided into three main components: the robot behavior implementation, a custom lightweight 2D simulator, and a Gazebo-based 3D simulation environment. Figure 1 shows the dependency structure of the `botbrain` behavior library and the two simulates which will be further explained in the following sections.

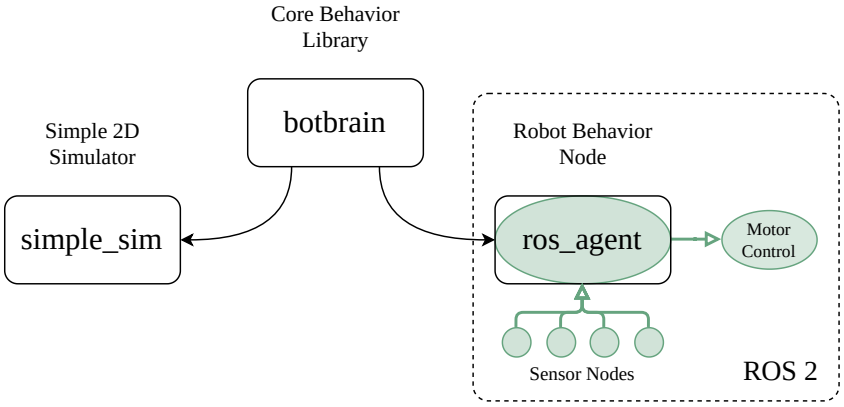


Figure 1: Software structure of the project. Black arrows indicate library dependencies, while green arrows denote ROS 2 communication through topics. Square boxes represent Rust crates and green circles represent ROS 2 nodes. The core library `botbrain` is ROS 2-independent and can be reused by the simple simulator without any dependency on ROS 2.

4.1 Simulated Hardware

A simulated differential-drive robot equipped with a camera, LiDAR, odometry, and communication capabilities is used as the target platform. LiDAR and odometry are used for localization via Adaptive Monte Carlo Localization (AMCL), as discussed later in section 5.2. The camera is used for object detection, and a communication channel is assumed for data exchange between robots.

To leverage existing 3D models and URDF files, the Turtlebot 4 [6] is selected as the simulated platform. Although the Turtlebot 4 lacks a dedicated communication unit, inter-robot messaging is handled by the simulator, rendering the system agnostic to the specific communication method. The software can be adapted to any robot platform that satisfies the sensor and communication requirements by configuring parameters in the source code.

4.2 Using Rust

This project is primarily implemented in Rust. While this choice was primarily based on developer preference, it brings several notable benefits. Rust's memory safety guarantees and strong static typing contribute to a more robust and maintainable codebase. The compiler ensures that object lifetimes are correctly managed, eliminating an entire class of memory-related bugs. These features make Rust particularly well-suited for collaborative development, where code clarity and correctness are critical.

Although ROS 2 does not officially support Rust, the `r2r` library [7] provides bindings to the ROS Client Library (RCL) [8], enabling message serialization, topic publication/subscription, and integration with other ROS 2 components. The initial build times are longer compared to C++, as Rust bindings are

generated during first compilation. ROS 2 uses the Colcon command line tool [9] for building, testing, and using multible packages. A simple CMake script was made to integrate `r2r` packages with ROS 2, with inspiration from [10].

This project also uses OpenCV [11] for simple object detection, elaborated in section 5.3. OpenCV was chosen for its mature and well-documented API, as well as its support for multiple platforms. The Rust bindings for OpenCV are available in the `opencv-rust` library [12].

4.3 Robot Behavior

Robot behavior is implemented in the `botbrain` library, which defines a common `Robot` interface that exposes a robot's internal state to external programs (see fig. 2).

```
pub trait Robot {
    fn set_id(&mut self, id: RobotId);              // Set the ID of the robot
    fn set_map(&mut self, world: Map);              // Set the world map

    fn input_pose(&mut self, pose: RobotPose);      // Input the robot's pose
    fn input_cam(&mut self, cam: CamData);          // Input camera data
    fn input_lidar(&mut self, lidar: LidarData);    // Input LiDAR data
    fn input_msgs(&mut self, msgs: Vec<Message>); // Input messages from other robots

    fn get_id(&self) -> &RobotId;                   // Retrieve the robot's ID
    ...
}
```

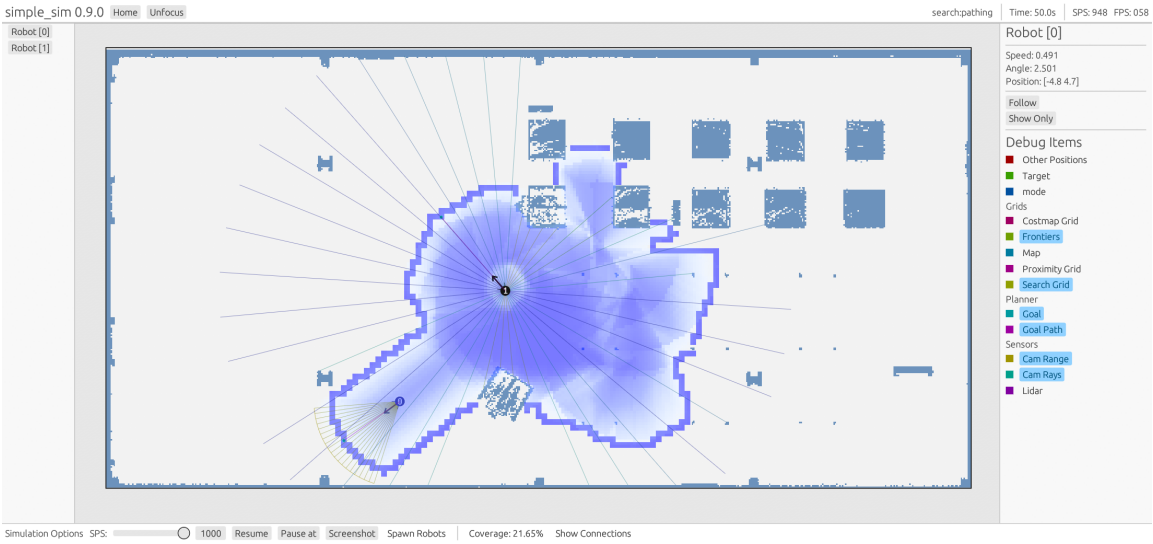
Figure 2: Methods of the `Robot` interface.

Methods prefixed with `set` are intended for initialization and must be called before any other interaction. The `input` methods are used to provide live sensor data and messages from other robots, while `get` methods retrieve state information without modifying it. Notably, the interface does not include a method that directly runs the robot's control loop. Instead, the `Robot` represents only the internal state. Control logic is implemented as an external function that reads from and writes to this state to produce control signals and a list of outbound messages. This separation allows for multiple behavior implementations to operate on the same robot state, simplifying algorithm comparison.

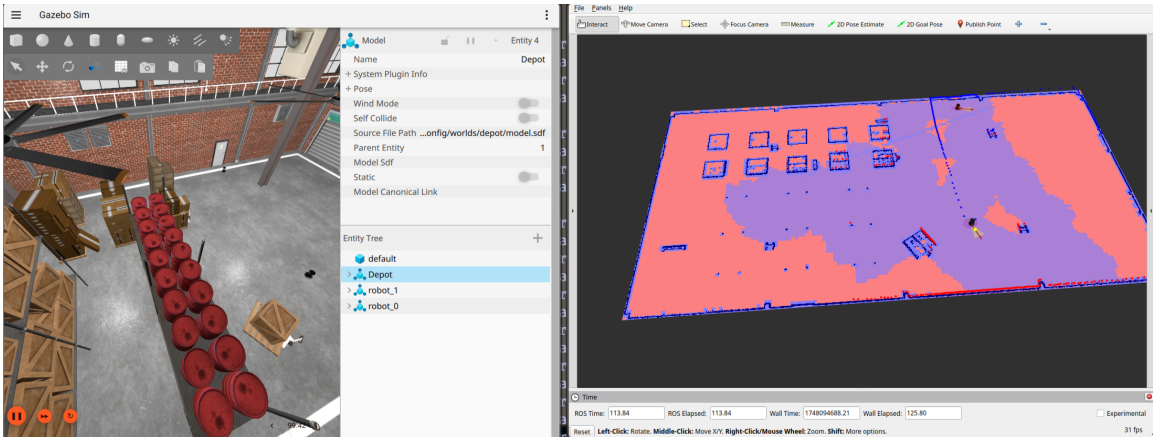
4.4 Two Simulators

This project uses a simulator-to-simulator approach rather than attempting direct deployment to physical hardware. Two simulators were used: a lightweight custom-built 2D simulator called `simple_sim`, and a more realistic 3D simulation in Gazebo integrated with ROS 2.

The 2D simulator is designed to be lightweight and fast although the simulation environment is very simple. The Gazebo simulation, by contrast, is computationally much more heavy and provides more extensive sensor emulation and more realistic robot dynamics. It is used to validate that behaviors developed in the simpler simulator transfer correctly to a close to real environment.



(a) Simple Simulator Graphical User Interface showing two robots, one in the middle and one in the lower left part of the map. Several debug items are visualized: Search Grid, Frontiers, and LiDAR and Camera approximations are shown. These will be explained in later sections.



(b) Gazebo Simulator in ROS 2. The left window contains the Gazebo visualization, which renders the 3D enviornment. On the right, RViz2 is visualizing data sent over topics which includes the current search grid as shown in red and blue.

Figure 3: Screenshots of the two simulators running the "depot" environment.

The double simulator architecture is built around the shared `botbrain` behavior library, which encapsulates each robot's internal state and provides a standardized interface for behavior control. This modular design allows the same behavior code to be executed in both the lightweight `simple_sim` environment and the more realistic Gazebo based simulation without requiring any code changes. By decoupling the behavior logic from the underlying simulation platform, the system supports consistent benchmarking and fair comparisons between algorithms. This separation also promotes cleaner, more testable, and reusable code, making it easier to iterate on behaviors and evaluate improvements.

4.4.1 Behavior Debugging

To facilitate development and validation, `botbrain` supports real-time debugging of both robot behavior and internal state, through the exposed debugging interface known as the `Debug Soup`. The `Debug Soup` allows behavior modules to annotate the simulation with structured debugging data. These debug items — such as occupancy grids, path vectors, target points, and internal decision states — can be visualized in both ROS 2 via RViz2 (fig. 3b) and `simple_sim` (fig. 3a).

RViz2 supports visualization of messages published on ROS 2 topics. `Debug Soup` elements are converted into ROS 2 message formats (e.g. `OccupancyGrid`) and published for real-time display. In `simple_sim`, debug items are located on the right panel in the user interface and their overlays can be toggled at will.

This enables introspection of robot behavior during simulations and is invaluable for development, debugging, and helps in interpreting emergent behavior of the swarm. The `Debug Soup` can be disabled to save memory and increase performance after development has finished.

5 Gazebo Simulation

To validate that the robot behavior models function under realistic conditions, a high-fidelity simulation environment is necessary. This project utilizes Gazebo Harmonic, a 3D physics-based simulation engine capable of emulating a variety of sensors such as LiDAR and RGB cameras. Gazebo is selected due to its flexibility, integration capabilities with ROS 2, and widespread adoption in the robotics community.

5.1 ROS 2

Deploying robot systems in real-world scenarios requires more infrastructure than simulation alone, including networked communication and physical hardware interfacing. ROS 2 (Robot Operating System version 2) is widely used for these purposes. It provides a rich ecosystem of tools, communication primitives, hardware drivers, and standard packages for robotic applications. ROS 2 is built on a publisher/subscriber communication model. Individual tasks are encapsulated as nodes, which communicate through topics, services, and actions. This modularity enables complex systems to be built from loosely coupled components.

In this project, each robot runs in its own ROS 2 namespace, allowing for the separation of topics and nodes. This approach supports scalable multi-robot simulation, where sensor data and localization outputs — such as those from AMCL, LiDAR, cameras, and transform frames — remain isolated per robot instance.

Gazebo also uses a publish/subscribe model internally. Through the `ros-gz-bridge`, Gazebo topics are bridged to ROS 2 topics, allowing control commands to flow into Gazebo and simulated sensor data to be published back into ROS 2.

The Turtlebot 4 is modeled using a URDF (Unified Robot Description Format) file that includes specifications for LiDAR, camera, and drive plugins. The robot's differential drive is simulated using the Gazebo Diff Drive Plugin [13], which accepts velocity commands and updates the robot's pose accordingly.

5.2 Localization

To enable functional behavior, robots require an estimate of their pose within the environment. This project assumes access to a pre-existing map for localization.

Localization is handled by the AMCL (Adaptive Monte Carlo Localization) algorithm [14, 15] from the ROS 2 `nav2` stack [16, 17]. AMCL uses LiDAR and odometry data to estimate the robot's pose through a particle filter, yielding a position and orientation with an associated uncertainty.

For environments without an available map, SLAM (Simultaneous Localization And Mapping) is employed to build the map while localizing the robot in real time. The `nav2` SLAM package is used for this purpose, particularly during map generation phases in 3D simulations.

5.3 Object detection

To locate a predefined target object in an environment, a simple rule-based object detection system is used in place of a machine learning-based detector, as object detection is not a focus of the project.

The camera feed is accessed through a ROS 2 topic. The object of interest is defined as a yellow ball, and the detection pipeline proceeds as follows:

1. Convert the input image from BGR to HSV color space (Hue, Saturation, Value), as HSV is more robust for color detection.
2. Apply a mask to isolate yellow hues (around 60° in hue).
3. Extract contours from the masked image.
4. Create minimum enclosing circles around the contours.
5. Calculate the average color of the circles to verify hue alignment with yellow.

If no circles are found, the frame is considered to contain no search object. If one or more circles are detected, their probability of being the target object is computed based on their average color proximity to yellow. This method is computationally inexpensive and sufficient for simulation, though it lacks robustness in poor lighting or cluttered environments.

5.4 botbrain Integration

As in simple_sim, inter-robot communication is essential for coordinated behavior. A custom ROS 2 message type is used to encapsulate communication between robots:

```
# file: AgentMessage.msg
uint32 sender_id
byte[] data
```

This message contains only a sender ID and a serialized, variable length, byte array of data which contains the message. This design choice avoids the need for multiple ROS 2 message types. Each robot serializes its outgoing messages and deserializes incoming data for processing via the shared botbrain interface. After receiving the latest sensor data, each robot's behavior is called. The resulting control command, consisting of linear and angular velocity, is published to Gazebo. The simulation then applies this input using its differential drive model.

6 Simple Simulator

The simple simulator was developed to enable rapid testing and debugging of robot swarm behavior in a controlled 2D environment. It simulates multiple robots operating in a discretized world, rendering them as circles with their respective IDs displayed at their centers. The simulator advances in discrete time steps at 60 updates per simulated second and executes in a multi-threaded fashion to maintain performance across large robot groups. The simulator can also be run in headless mode, without visualization, for even better performance.

The simulator supports loading map layouts from ROS 2-compatible `.pgm` files, allowing it to closely mirror environments used in the 3D Gazebo simulations. This compatibility facilitates seamless testing of behaviors across both simulation platforms.

Figure 4 shows the graphical user interface of the simulator, displaying a scenario where three robots are performing a coordinated search in the left region of the map.

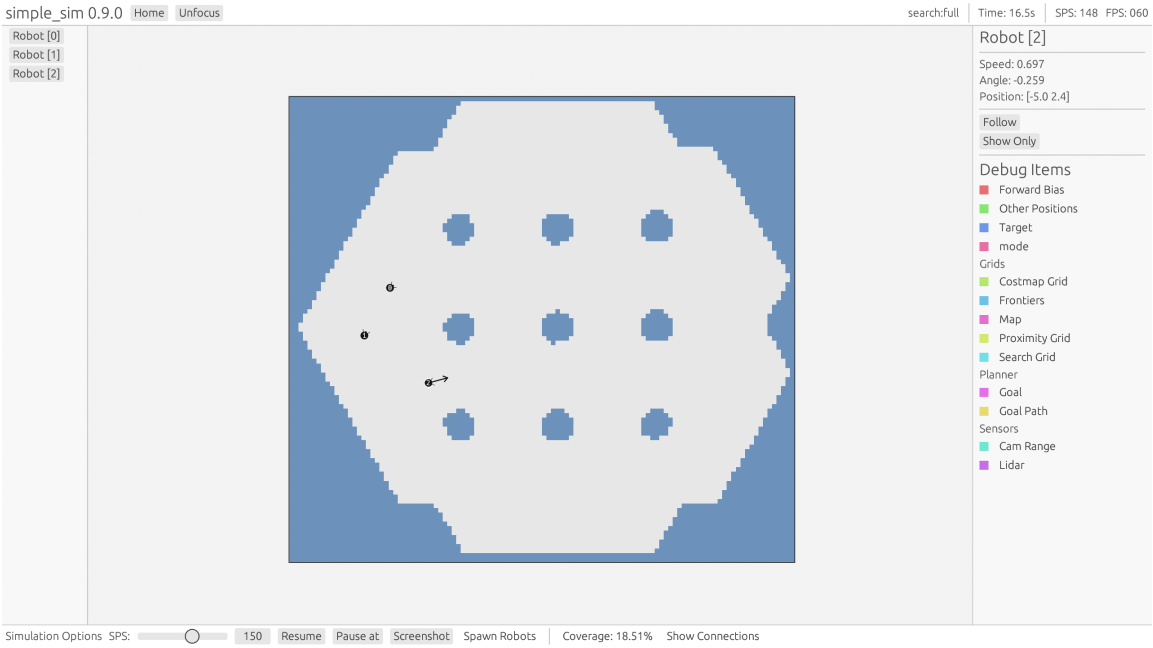


Figure 4: Simple simulator interface showing three robots operating in a shared environment.

The simple simulator is designed to be lightweight and performant. It supports deterministic execution, enabling reproducible tests for debugging. Simulation speed can be accelerated beyond real time, depending on CPU capacity and the number of robots simulated.

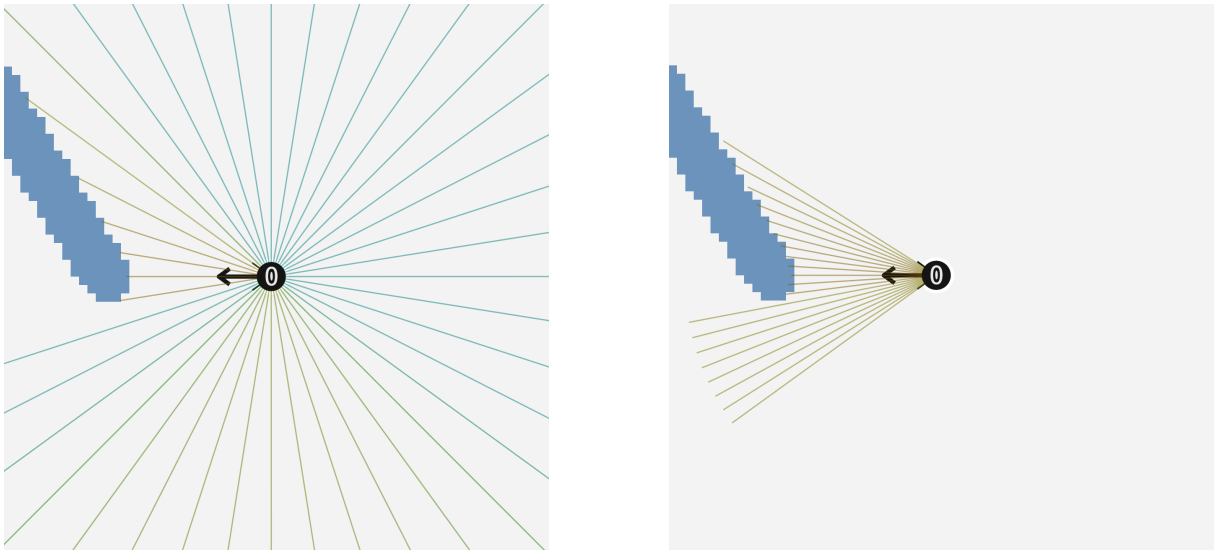
Multi-threading is used to parallelize sensor simulation and robot state updates, allowing the system to scale to larger swarms. In practical testing, the simulator is capable of 30 to 50 robots in real time on an Apple Macbook with M1 Pro [18] depending on the behavior. Because the environment is purely 2D and avoids expensive physics calculations, the performance overhead is minimal compared to full 3D simulation.

6.1 Sensor Approximation

To run behaviors, each robot requires three inputs: pose, LiDAR, and camera data. The pose is set directly by the simulator and requires no approximation. Both LiDAR and camera inputs are emulated using simplified ray casting techniques with a constant step size.

LiDAR is emulated by emitting a series of radial rays from the robot's position at fixed angular intervals. Each ray is advanced incrementally until it intersects with an obstacle in the environment. The resulting distance data mimics real-world LiDAR output and is illustrated in fig. 5a.

The camera emulation follows a similar ray-casting procedure, but is constrained to a defined field of view, replicating the properties of the Turtlebot 4's camera, see fig. 5b. Rather than applying a full object detection algorithm, the camera rays directly infer object identity upon intersection, allowing simplified yet functional input for the behavior algorithms.



(a) LiDAR approximation.

(b) Camera approximation.

Figure 5: Sensor input approximation in the simple simulator.

7 Search Algorithms

This section describes the development of distributed algorithms that enable a group of robots to collaboratively search a map for a target object. All algorithms are implemented in the `botbrain` library.

7.1 The Roomba Algorithm

The Roomba algorithm serves as a baseline for evaluating the performance of more advanced behaviors. This simple obstacle avoidance algorithm moves in a straight line until the robot encounters an obstacle, at which point it turns away until it can resume moving forward.

In its implementation, the robot uses LiDAR data to locate the nearest obstacle. If the obstacle is within a predefined safety threshold, the robot rotates away. The resulting paths from this behavior are shown in fig. 6.

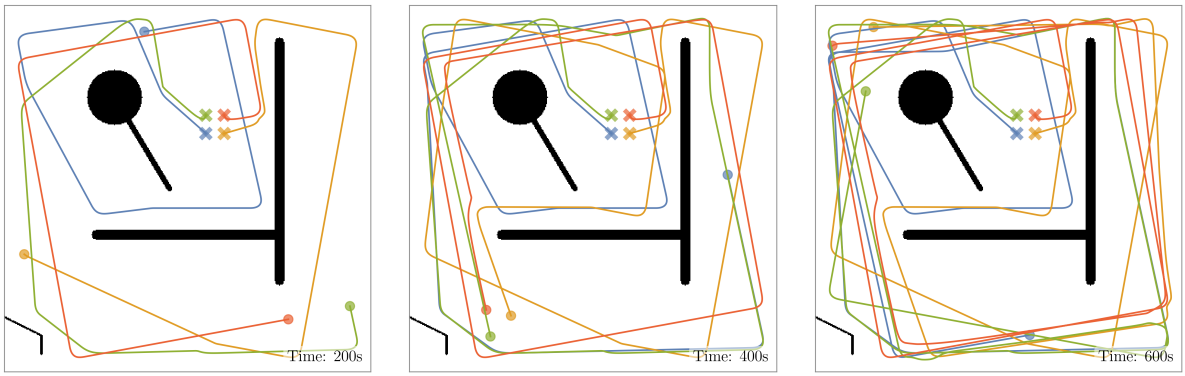


Figure 6: Paths of four robots running the Roomba algorithm at 200 s, 400 s, and 600 s in an example world simulated using `simple_sim`. Crosses mark starting positions, and circles end positions.

7.2 Target Contributions

Each robot maintains an internal state, which — combined with sensor input — drives its movement and exploration behavior. Conflicting objectives may arise; for instance, the robot may detect an unexplored area ahead, but find its path blocked by an obstacle. To resolve this, each behavior subsystem outputs a vector, a "target contribution", indicating its preferred movement. These vectors are summed to produce a single target vector, which dictates the robot's final motion. Subsystems can be weighted according to their importance, obstacle avoidance, for instance, is typically given higher priority.

7.3 Search Gradient

Each robot maintains a "search grid," which acts as a heatmap representing the exploredness of each cell in the map. This grid is continuously updated using the robot's own observations and messages received from other robots via the shared communication channel. Explored areas are considered "cold," while unexplored areas and search objects are "hot".

Assuming lossless communication, the search grids across robots should remain synchronized. Each robot computes a directional gradient from the search grid at its current position, encouraging movement toward hotter (less explored) regions.

In practice, computing the gradient involves several steps. First, the average heat under the robot (H) must be calculated, using the cells underneath the robot (C_H), as in eq. (1). N is the number of cells in C_H.

$$H = \frac{1}{N} \sum_{c \in C_H} \text{heat}(c) \tag{1}$$

Next, a vector $\mathbf{d}$ is computed from the robot's position P to every grid cell (C_∇) within a specified radius r:

$$\mathbf{d} = \text{pos}(c) - P, \quad \forall c \in C_\nabla \tag{2}$$

Each cell contribution to the final gradient, is based on the difference in heat between the cell and the heat under the robot, and its proximity to the robot in the direction of $\mathbf{d}$. The contribution fades linearly with distance, resulting in the following expression:

$$\nabla = \sum_{c \in C_\nabla} \underbrace{\mathbf{d}/\left\|\mathbf{d}\right\|}_{\text{Direction}} \cdot \underbrace{\big(\text{heat}(c) - H\big)}_{\text{Heat Difference}} \cdot \underbrace{\big(1 - \left\|\mathbf{d}\right\|/r\big)}_{\text{Nearness}} \tag{3}$$

This formulation works well in open areas but may behave undesirably near map edges. Since cells outside the map have no heat value and negative contributions repel the robot, the gradient can incorrectly point toward the map boundary if surrounding cells inside the map are colder than the robot position.

To address this, only positive heat differences are considered in the gradient calculation. In practice, this means that the gradient points towards the "hottest" direction, but not necessarily away from the "coldest" direction.

$$\nabla = \sum_{c \in C_\nabla} \begin{cases} \mathbf{d}/\left\|\mathbf{d}\right\| \cdot \big(\text{heat}(c) - H\big) \cdot \big(1 - \left\|\mathbf{d}\right\|/r\big) & \text{if heat}(c) > H \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

This modification prevents attraction toward map boundaries and ensures the robot continues to seek out unexplored areas. Cells occluded by obstacles are also excluded from the gradient computation.

7.4 Proximity Grid

To maintain simulated network connectivity, robots must stay within communication range of each other. Each robot uses a "proximity grid" to estimate which areas of the map are within range of the main robot network.

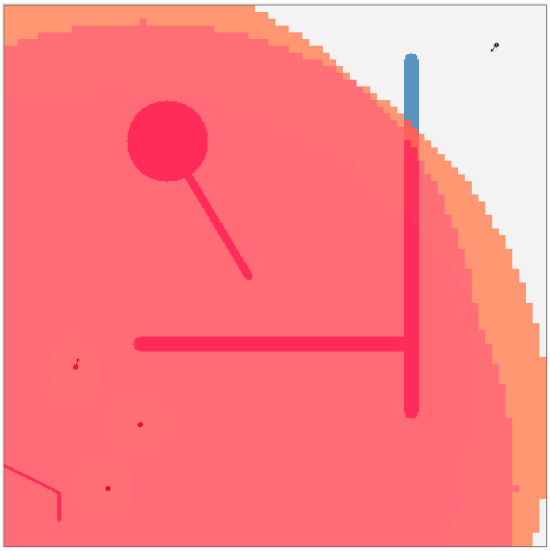


Figure 7: Proximity grid showing 3 robots forming the main network and a disconnected robot is heading towards the main network.

The proximity grid is computed by identifying the largest set of connected robots and "coloring" their communication range. Tiles connected by a single robot are half-shaded to indicate weaker signal reliability (see fig. 7). A gradient is computed over this grid and added as a weighted contribution to the robot's overall target vector.

Importantly, each robot excludes itself when calculating the proximity grid. This ensures that if a robot is the sole link between two clusters, it will move toward the main network and draw disconnected robots with it.

7.5 Gradient Based-Algorithm

The gradient-based algorithm calculates the robot's "target" vector using contributions from both the search and proximity grid gradients. However, when relying solely on these contributions, the resulting target vector often points sideways or even backward. This occurs because the robot continuously "cools" down the area in camera's field of view, which leads to excessive turning and minimal forward progress.

To mitigate this, a small constant forward vector is added to the total target vector to maintain movement and promote map coverage. Figure 8 illustrates the paths of four robots executing this behavior.

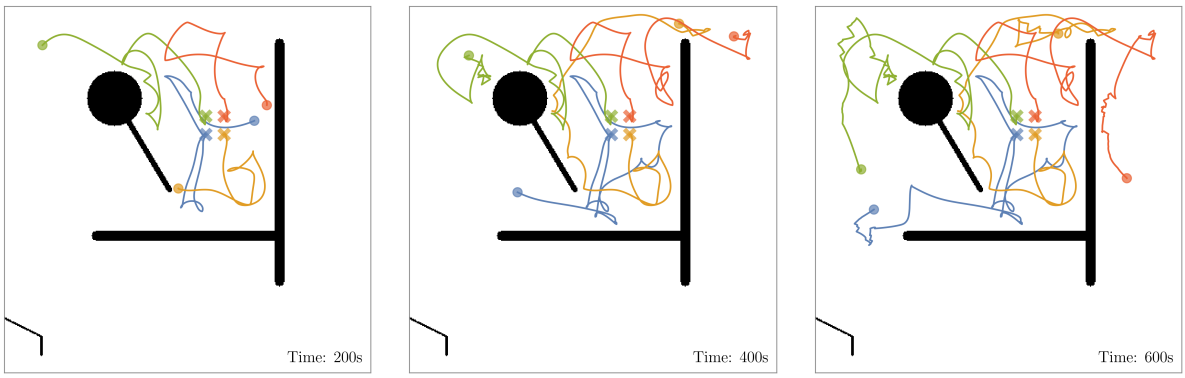


Figure 8: Paths of four robots running the gradient-based algorithm at 200 s, 400 s, and 600 s in an example world simulated using `simple_sim`. Crosses mark starting positions, and circles end positions.

The robots explore the environment effectively but tend to linger in already explored regions. This behavior may be due to the limited range of the search gradient. Additionally, robots exhibit more erratic paths when the gradient magnitude becomes small. In general, robots switch between smooth trajectories in unexplored areas and jagged paths when trapped in previously covered regions.

In fig. 9, the search grid is visualized. Here the robots are moving towards unexplored or hotter areas.

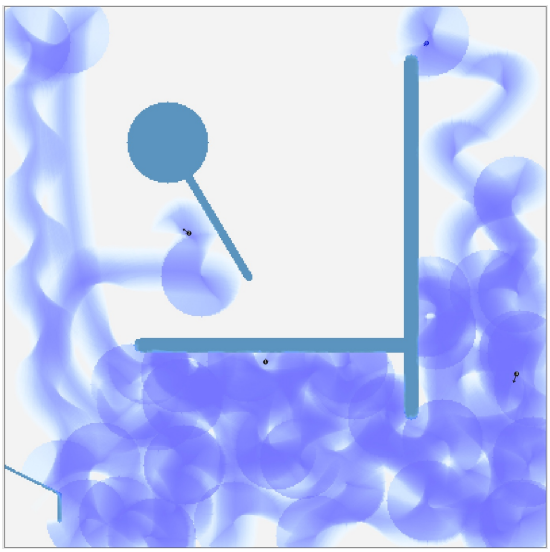


Figure 9: The search grid, where darker blue shows the cooler aeras..

To improve coverage, mechanisms that encourage movement toward globally unexplored areas would be beneficial.

7.6 Global Planner

When the search gradient fails to provide a clear direction — i.e., when the robot lacks a local preference — a global planner is used to guide the robot to a more promising unexplored region. The global planner operates according to the following procedure:

1. Generate a costmap (section 7.6.1).
2. Identify frontiers and group them into regions [19] (section 7.6.2).

3. Evaluate the frontiers and select a goal (section 7.6.2).
4. Plan a path using a straight line or the A* algorithm [20] (section 7.6.3).
5. Follow the path until the goal is reached or the path is invalidated (section 7.6.4).

7.6.1 Costmap

The costmap is a grid-based representation of the environment with a variable resolution. Each cell in the costmap is assigned one of three states:

- **Searched:** Cells identified as covered in the search grid.
- **Unknown:** Cells not yet visited.
- **Obstacle:** Cells corresponding to known obstacles or other robots, inferred from recent LiDAR readings.

The costmap is used in both frontier exploration and path planning. To ensure safe navigation, the robot must avoid collisions with obstacles as well as other robots. Therefore, when evaluating potential paths or frontiers, the surrounding cells are also checked to ensure they are free of obstacles. The number of surrounding cells considered depends on a configurable clearance parameter, which defines the minimum required buffer between the robot and nearby obstacles.

This clearance can be dynamically adjusted over time — for example, allowing the robot to take riskier, narrower paths during later stages of the search when unexplored areas become harder to reach. This adaptive behavior helps maintain efficiency without compromising early-stage safety.
An example of the costmap and its affect on path planning is shown in fig. 11 on page 21.

7.6.2 Frontier Exploration

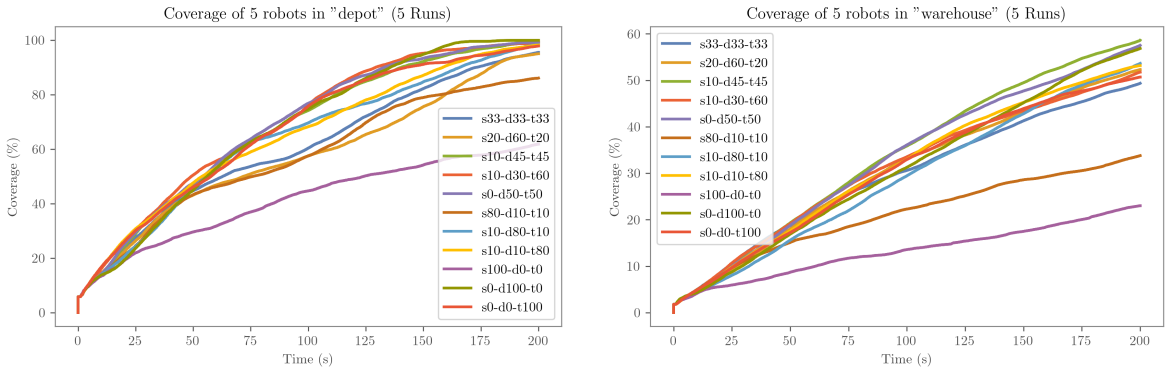
A frontier is defined as a boundary between explored and unexplored regions. These frontiers are discovered using a Breadth-First Search (BFS) starting from the robot's current position. This approach avoids a full costmap scan, thereby reducing computational overhead.

Detected frontier cells are grouped into regions and scored based on:

- Frontier region size
- Distance to the robot
- Required turn angle to face the frontier

As mentioned, frontiers must meet a minimum obstacle clearance threshold to be considered valid as a potential goal.

An experiment was conducted in two environments to evaluate the coverage performance with a range of different weights, see fig. 10. All tested configurations will eventually achieved full map coverage; therefore, performance is evaluated based on coverage speed.



(a) Simple map

(b) Complex map

Figure 10: Frontier evaluation parameters comparison. The weights are: `sXX` for frontier size, `dXX` for distance to the frontier, and `tXX` for the required turn angle.

Configurations that prioritized distance and turn angle produced the fastest coverage. This outcome suggests that minimizing rotation allows for higher sustained forward velocity. In contrast, configurations that emphasized frontier size alone often resulted in inefficient behavior, such as long detours and frequent turning, which slowed down overall progress. The final parameter configuration, `s10-d45-t45`, produced the overall best and most consistent performance across maps with various complexity.

7.6.3 Path Planning

Once a goal is selected, the planner first attempts to generate a direct, straight-line path to the goal. If this path violates the required clearance from obstacles or other robots, the planner falls back to the A* algorithm.

The A* algorithm explores the grid by combining the cumulative path cost with a heuristic estimate of the remaining distance to the goal. A min-heap (priority queue) is used to efficiently select the next cell to expand based on the lowest estimated total cost. Only collision-free cells that satisfy the clearance constraint are considered during expansion.

Once the goal is reached, the resulting path consists of the sequence of cells from the start to the goal. This path is then post-processed to remove unnecessary intermediate points, provided that the smoothed path still maintains the required clearance. This smoothing step reduces path length and ensures more natural robot motion while preserving safety.

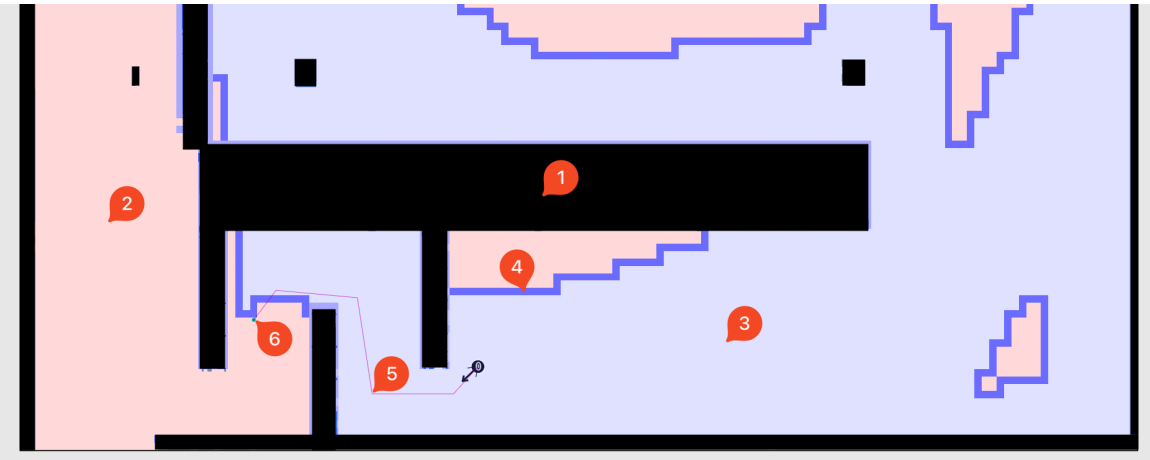


Figure 11: Screenshot of global path planner components in `simple_sim`: 1) obstacle cells, 2) unexplored cells, 3) explored cells, 4) frontier cells, 5) goal path, and 6) goal.

7.6.4 Path Following

The robot follows the planned path as a sequence of waypoints. As each waypoint is reached, it is marked as visited and removed from the remaining path.

To ensure robustness in dynamic environments, the path is continuously validated using the costmap. If the clearance to the next waypoint becomes invalid due to a newly detected obstacle, the robot does not immediately discard the path. Instead, it continues toward the waypoint, while incorporating reactive feedback from the LiDAR sensor.

LiDAR input takes precedence over the path-following command. For example, if a new obstacle appears directly in front of the robot, the repulsive contribution from the LiDAR cancels out the forward path-following command, causing the robot to stop. In less obstructed cases—such as when an obstacle appears on the front-right side—the robot will adjust its heading and veer left to navigate around it.

If the path fails clearance checks repeatedly, the planner attempts to replan a new path to the current goal. However, in some cases, no viable path may exist due to dynamic obstructions. In such scenarios, the robot will abandon the current goal and select a new one.

Additionally, if a robot becomes disconnected from the swarm, a new goal is generated to guide it back toward the network. This reconnection behavior uses the proximity grid to identify areas with higher robot density, helping to restore communication.

7.7 Hybrid Algorithm

The global planner enables a hybrid algorithm that combines both local and global strategies. As long as the local search gradient provides a sufficiently strong direction, the robot continues using the gradient-based method. When the gradient magnitude falls below a predefined threshold, the global planner is triggered to relocate the robot to an unexplored area.

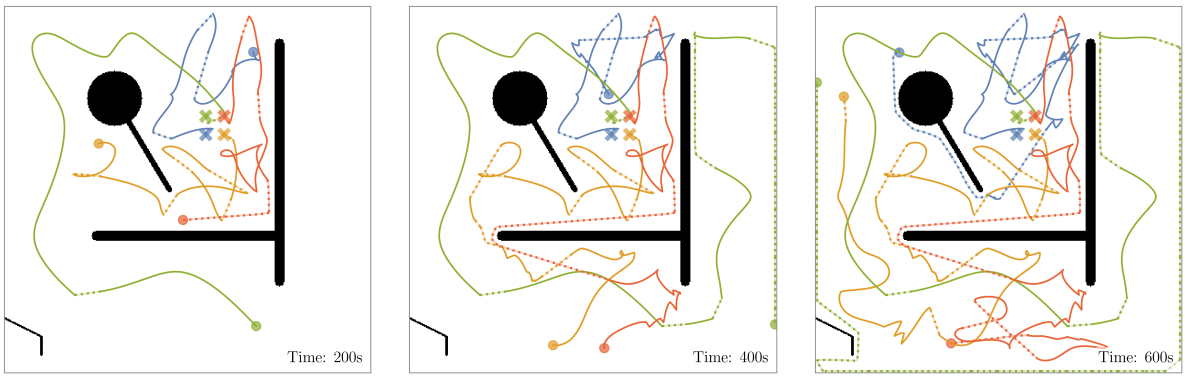


Figure 12: Paths of four robots running the hybrid algorithm after 200 s, 400 s, and 600 s in an example world simulated using `simple_sim`. Solid lines indicate gradient-based motion, dashed lines path-following. Crosses mark starting positions, circles end positions.

Figure 12 shows the resulting behavior, where solid lines represent gradient-based movement and dashed lines indicate global path-following. This behavior produces a more spread out swarm as compared with the pure gradient-based algorithm which may lead to better search efficiency.

7.8 Pure Pathing Algorithm

An alternative to the hybrid approach is to use the global planner exclusively. In this case, the robot continuously alternates between selecting a new goal and following the computed path. Figure 13 shows example paths. As with the hybrid algorithm, the trajectories are smooth and consistent, with little lingering.

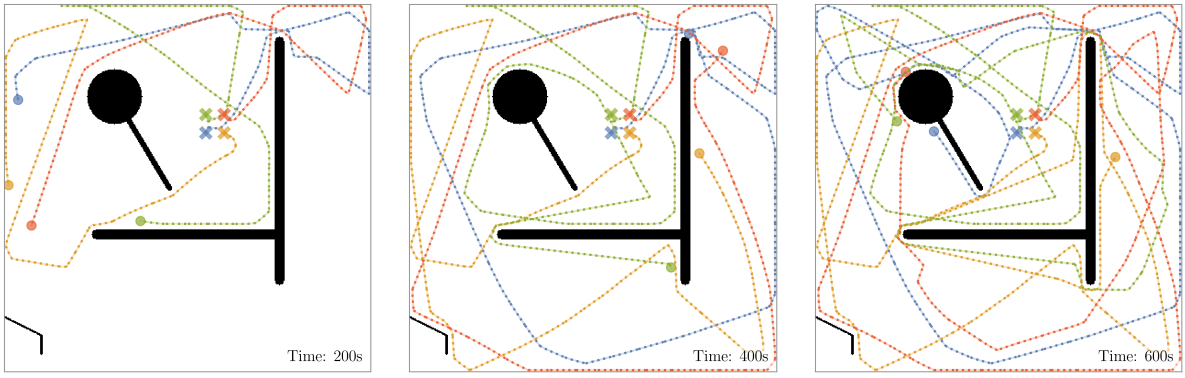


Figure 13: Paths of four robots using the pure pathing algorithm. Dashed paths indicate full-time path-following. Crosses mark starting positions; circles indicate end positions.

The pure pathing algorithm also achieves full map coverage, with highly structured and efficient motion. A comparative analysis of all algorithms is presented in section 9.

8 Deep Reinforcement Learning

In contrast to the manually developed algorithms, a Deep Q-learning approach was also explored. Reinforcement learning suits this problem well, as coverage can serve as a reward function. These types of systems are also very adaptable to new environments as they can be retrained on new data to expand and tune their capabilities.

Traditional Q-learning methods store a table of discrete state-action pairs and their expected reward in memory, which can be cumbersome in real world applications where state is often continuous. Another pitfall of traditional Q-learning is that the size of the Q-table grows with the state space. Neural networks solve both of these problems. The Q-table can be approximated by a fixed size neural network, mapping states to action values. These networks are called a DQNs (Deep Q Networks).

Designing a DQN involves several important considerations, including selecting the number of layers and neurons, choosing appropriate activation functions, and determining key hyperparameters. After the architecture is defined, the network is trained, evaluated, and iteratively optimized to ensure stable and effective learning. Finally, the trained model can be validated on unseen environments and integrated into the broader system for deployment.

8.1 Learning Framework

The framework for defining robots controlled by neural networks is implemented as a part of the `botbrain` library and can be enabled with the "rl" feature flag. The models are implemented using the `burn` [21] library, a Deep Learning Framework for Rust. `burn` provides a high-level API for building and training deep learning networks and has multiple backends available. This project uses the cross-platform `wgpu` backend [22] for training and evaluation.

`botbrain` defines the `RlRobot` which is generic across four parameters: the `burn` backend, the state type, which the robots provides as input to the network, the action type, which defines the discrete actions the robot can take, and the network, which defines how the input and output layers should be connected. Several state, action and network implementations are also provided, which can be combined in any combination to create specializations of `RlRobot`. This system facilitates composition and expansion of `RlRobot` components and makes it easy to swap out single elements and compare differing architectures.

8.2 Reward Function

The reward function is a central component of reinforcement learning, as it defines the value of agent actions. A well designed reward function encourages desirable behavior and penalizes undesirable outcomes.

In this project, the reward is primarily based on map coverage. At each time step, a positive reward is given proportional to the amount of new area explored by the agent. This incentivizes the agent to move into unexplored regions of the map. Additionally, a small negative reward is applied for each step to encourage efficiency and discourage the agent from idling or circling already explored areas. A further negative reward scales linearly with the time since new area was covered. Collision with walls also results in a negative reward to promote safe navigation. To maintain network connectivity among robots, a penalty could be introduced when the agent moves outside of the communication range of the rest of the swarm. This would help keep the robots together to allow them to communicate, but was not implemented.

8.3 Trainer Implementation

Training the DQN involves allowing the agent to interact with the environment over multiple episodes while incrementally improving its policy based on received rewards. The `trainer` is implemented as a separate program using `simple_sim` as a library and was inspired by the `burn-rl-examples` repository [23].

Each training episode starts with one or more robots placed in a randomly generated environment. The episode proceeds step by step, during which the agent observes its current state, selects an action using an ϵ-greedy policy for balancing exploration and exploitation, executes the action, and receives a reward. The transition tuple, containing state, action, reward and next state, is stored in a replay buffer. ϵ is exponentially decreased as a function of total steps taken, resulting in random actions in the beginning of training approaching pure on-policy actions as training progresses.

8.3.1 Target and Policy Networks

During training, two networks with identical structure called the target and policy networks are used. The policy network encodes the current best policy and is updated at each time step using gradient decent. The target network is used as a steady reference when calculating the loss. Each time step, the weights of the target network are updated in a small linear step towards the policy network. The size of the step is configured with the τ hyperparameter. These networks both continually approach the optimal policy; The policy network approaches faster and is used to get on-policy actions, while the target network trails behind, but provides more stable state action values.

8.3.2 Calculating Loss

Each time step, batches of transitions are sampled randomly from the replay buffer and used to train the policy network by performing gradient decent on the loss function. Loss (L) is calculated as the mean square error (MSE) between the value of an action, as determined by the policy network, Q_{policy}, and the expected value of the action Q_{expeted} in the current state. The expected value is the value of the best action from the next state, as given by the target network, $Q_{\text{target,next}}$, multiplied by the discount factor γ, plus the reward gotten from taking the current action R.

$$L = \text{MSE}(Q_{\text{policy}}, Q_{\text{expected}}), \quad \text{where} \quad Q_{\text{expected}} = \gamma \cdot Q_{\text{target,next}} + R \tag{5}$$

Training continues for a fixed number of episodes or until the performance converges. An episode is terminated when the agent has performed the maximum number of time steps, or when 95% of the map is successfully covered.

8.3.3 Generating Training Environments

As to not over fit the network, agents are trained on different environments. A simple program was created to generate worlds populated by line and circle obstacles. The generator is however quite simplistic and does not check that the entire map is reachable which could lead to problems during training. Figure 14 shows examples of generated environments.

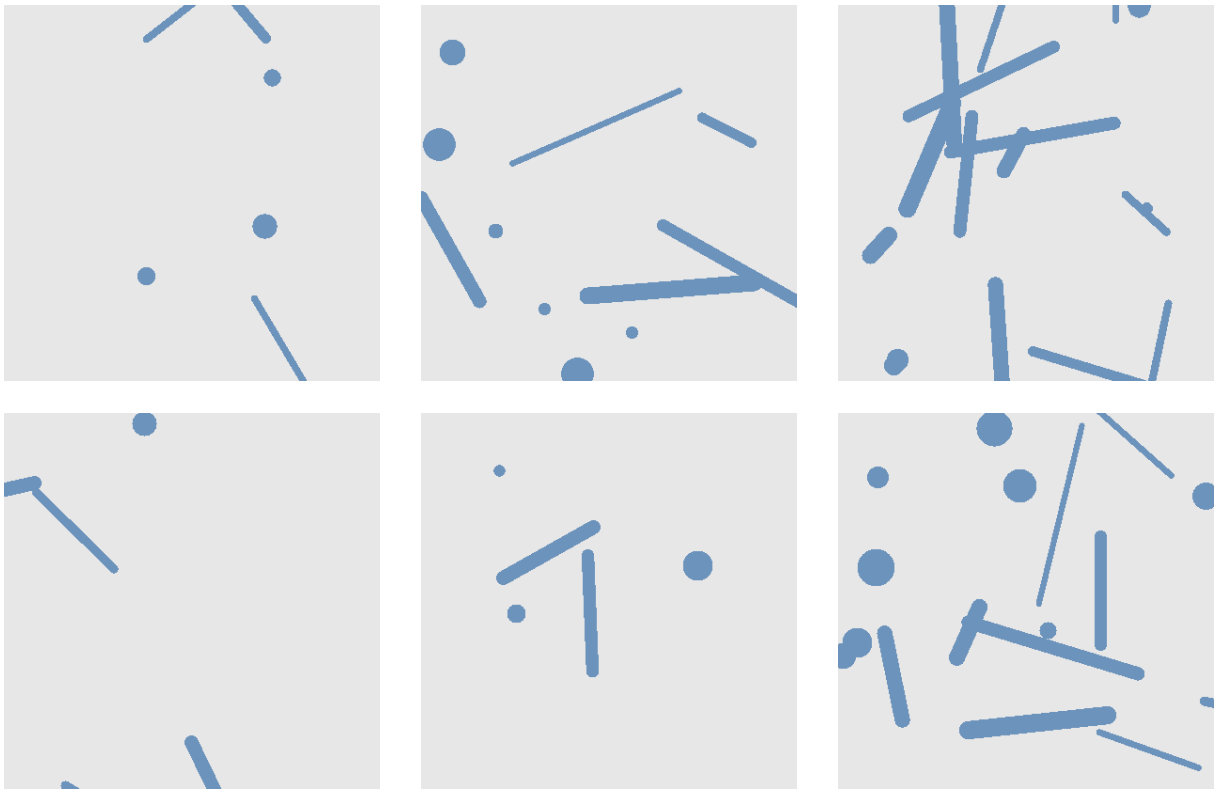


Figure 14: Examples of generated environments.

8.4 Training a Minimal Agent

To ensure that the training framework works, a simple agent was trained. The agent is provided the same state as used in the Gradient behavior: Robot angle, the angle and distance to the closest obstacle, and the angle and length of the search gradient. The action space consists of 7 discrete actions, spanning from full left to forward to full right control commands. A single hidden layer of 64 nodes connects the input and output layers. The sigmoid activation function is used for all layers.

The agent was trained on 200 episodes, each consisting of 240 simulated seconds. The training progress is shown in fig. 15.

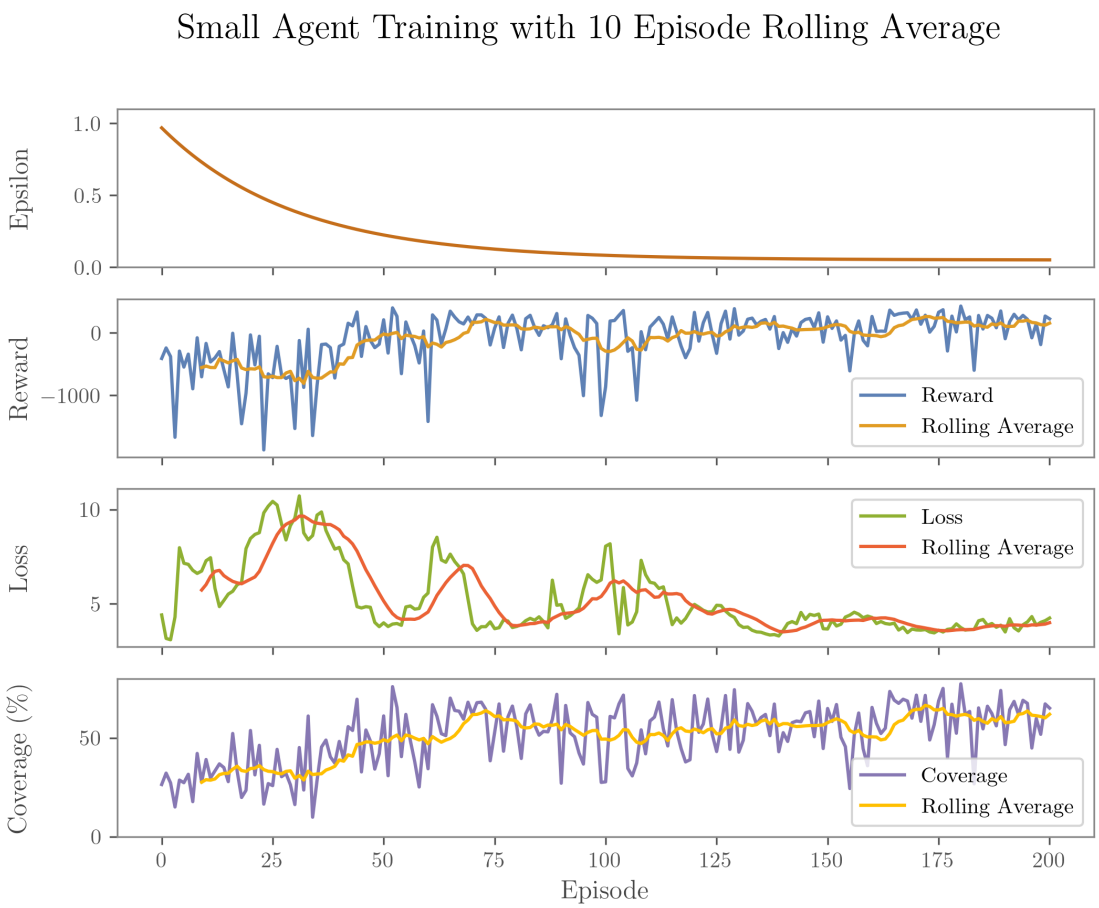


Figure 15: Plots of the training progress of a single minimal agent. The top plot show ϵ (random action chance), the second plot, reward, the third plot loss and the last plot shows the coverage. As training progresses, the agent learns to cover more of the map within the episode time of 240 s. The loss also decreases, indicating that the model improves its prediction of state action values.

As the agent is trained, the reward and coverage increases, indicating that the agent is learning to cover the map. The loss function also settles down eventually, which signifies that the model is predicting the reward function well. Running the behavior results in the path shown in fig. 16.

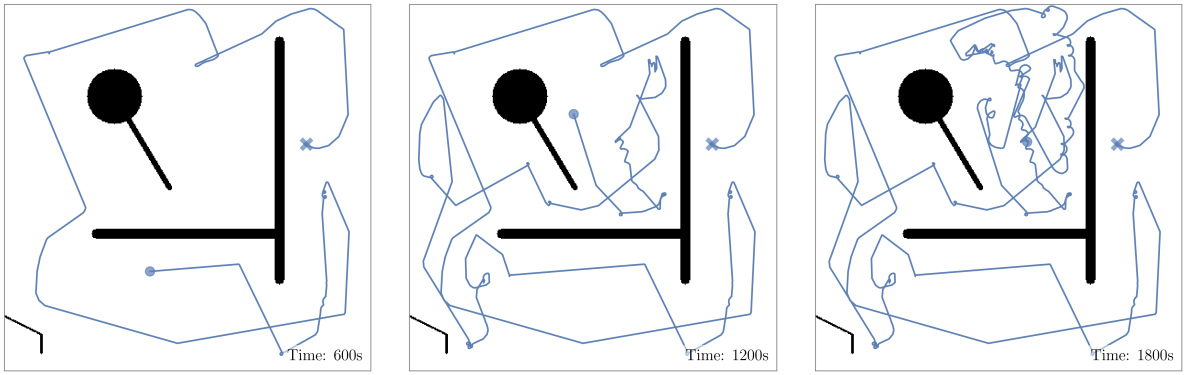


Figure 16: The agent path after 600, 1200 and 1800 seconds of simulation time. The agent learns to move around the map and avoid collisions with obstacles.

8.5 Training more Complex Agents

While the successfully trained agent show that the framework works, training is very computationally expensive. The minimal agent took approximately 12 hours to train. This could be an issue with the trainer or simulator implementation, especially as the simulation steps take longer as training progresses. The trainer was run with a profiler to investigate this, which showed that approximately 75% of the run time is spent within the `burn` framework. This leads to the conclusion that updating the model weights is most major factor in the training performance, and that faster hardware is needed to speed up training.

More complex agents are implemented in `botbrain` and training was attempted but never finished. An agent called `big-boy` was implemented to theoretically be aware of the robot swarm as a whole, and could therefore be trained to keep the swarm within communication range. The state input for the `big-boy` agent consists of the following fields:

- Robot angle
- Average swarm angle
- 12 LiDAR rays
- Search gradient vector
- Vector to the closest robot
- Size of the swarm (robot count)
- Center of the swarm
- Spread of the swarm in x and y direction

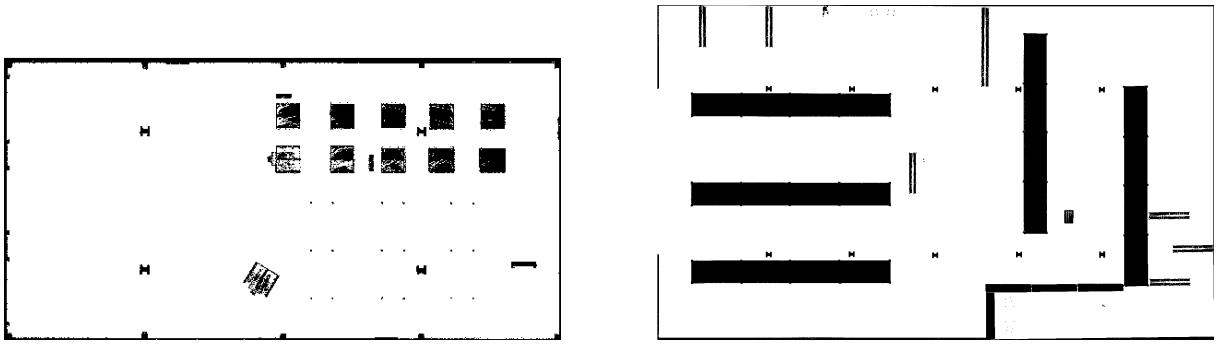
This results in 25 input nodes. The action space consists of 9 actions in the same manner as the minimal agent. This model contains three hidden layers of 96, 64 and 32 nodes respectively. The larger network should make the agent capable of learning a more intricate behavior and make better sense of the increased input nodes.

In theory, the network inputs could be expanded even more to also include global knowledge of the map to increase performance. This could be implemented in the current framework, but was not, as training was deemed infeasible.

9 Results

This section presents the results of simulations meant to determine the performance of the developed search algorithms. Algorithms were run on two maps: Depot and Warehouse [24]. Figure 17 shows the maps, with differing denseness of obstacles, used during testing.

Search algorithms are evaluated in terms of two key metrics: map coverage over time and computational expense. It is assumed that a higher coverage over time metric would lead to a shorter time to find a search item. Additionally, the consistency between the two simulation environments, `simple_sim` and the Gazebo ROS 2 setup is evaluated to confirm the robustness and portability of the behavior implementations.



(a) Depot. Dimensions: 30.2 m by 15.3 m.

(b) Warehouse. Dimmensions: 50.2 m by 30.2 m.

Figure 17: Maps used for benchmarking search algorithms. The Depot map is smaller and more open compared to the Warehouse map, which is larger and has more hard to reach areas.

9.1 Botplot

To enable efficient and reproducible experimentation, a custom Python package named `botplot` was developed. This tool provides utility functions for running and analyzing scenarios in both `simple_sim` and the Gazebo/ROS 2 simulation environments.

During each simulation run, data is collected in real time. In `simple_sim`, the data is recorded directly and stored in a `.ipc` (Apache Arrow) file. Similarly, in the ROS 2 setup, a dedicated logger node captures relevant simulation data, which is likewise serialized for later analysis.

Once recorded, the data is processed using `botplot`, which offers a suite of customizable plotting functions to visualize various performance metrics. These include map coverage, robot paths, communication patterns, and computational performance, among others.

To streamline the workflow, results are cached based on the simulation scenario. This allows for rapid re-plotting without the need to rerun the simulation, significantly improving the iteration speed when updating visualizations.

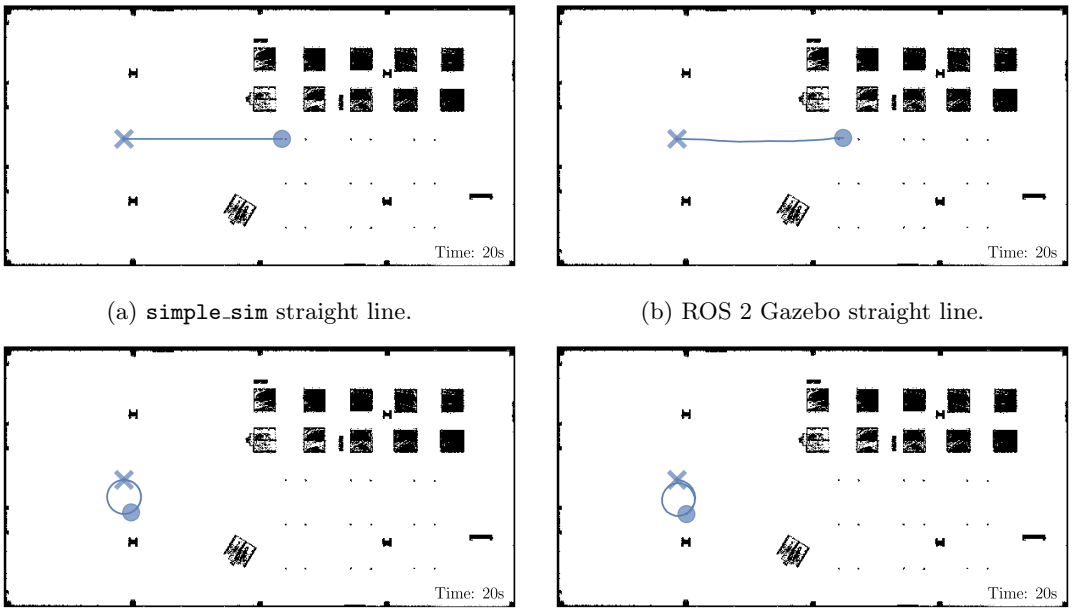
All plots presented in this report were generated using `botplot`.

9.2 Simulator Consistency

A central goal of the dual-simulator setup is to ensure that algorithms implemented using the `botbrain` interface yield consistent behavior across both simulators. While Gazebo includes more realistic physics and non-deterministic behavior due to its time step and sensor noise, consistency in qualitative behavior (e.g., coverage trend) is still expected.

9.2.1 Movement Calibration

To verify basic motion consistency, simple movement behaviors were executed in both simulators. Speed and steer commands were scaled to get the paths shown in fig. 18.



(a) `simple_sim` straight line.

(b) ROS 2 Gazebo straight line.

(c) `simple_sim` circular.

(d) ROS 2 Gazebo circular.

Figure 18: Paths produced by `simple_sim` and ROS 2 Gazebo.

The paths are visually similar, indicating that the velocity commands generated by the behavior logic are interpreted consistently across both environments.

9.2.2 Performance Consistency

To validate performance characteristics across simulators, the coverage of 4 robots was recorded over 6 runs for the behaviors developed in section 7 in both simulators. Figure 19 shows the results of this test.

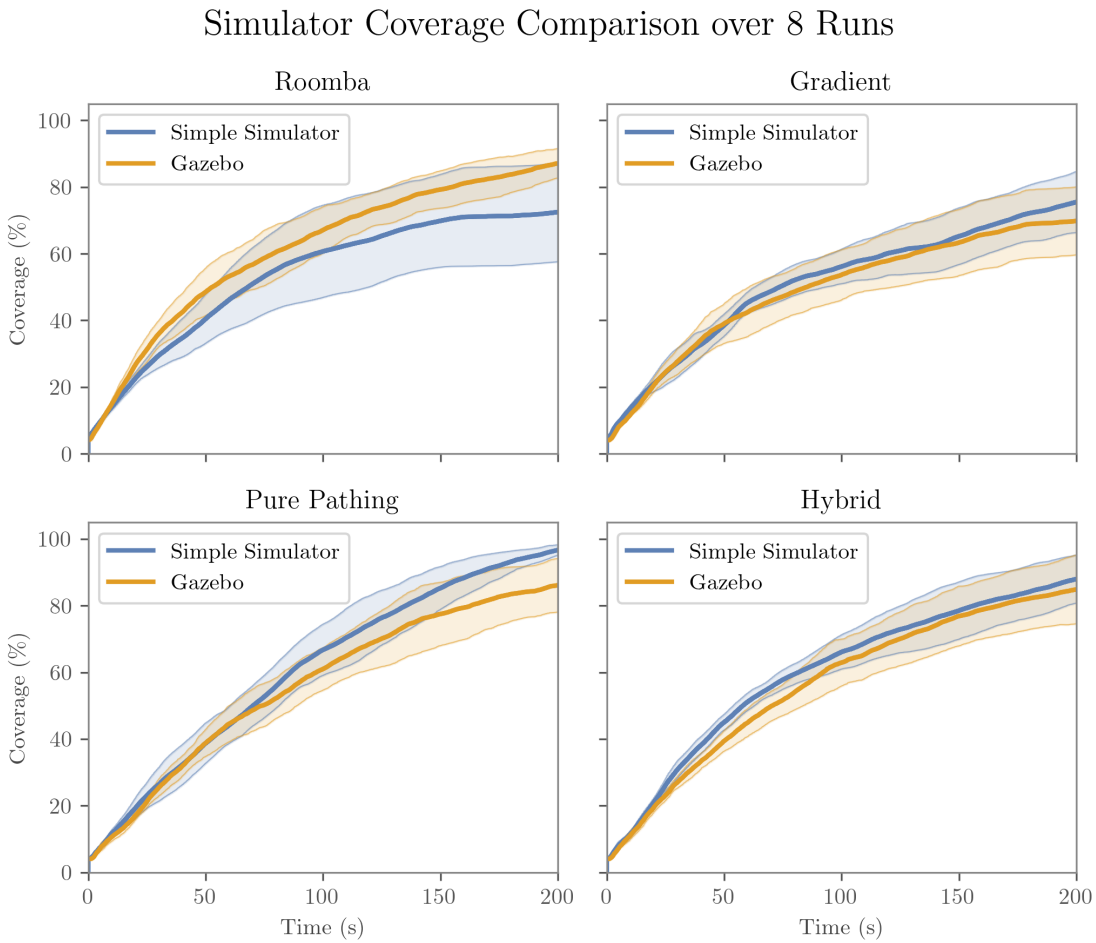


Figure 19: Comparison of coverage over time between ROS 2 Gazebo and `simple_sim`. Mean and standard deviation of coverage are shown.

The coverage performance seems to match well between the two simulators, with coverage mostly staying within 1 standard deviation of each other. The Roomba algorithm does, however, seem to perform better in the gazebo environment. Figure 20 shows the difference in coverage performance.

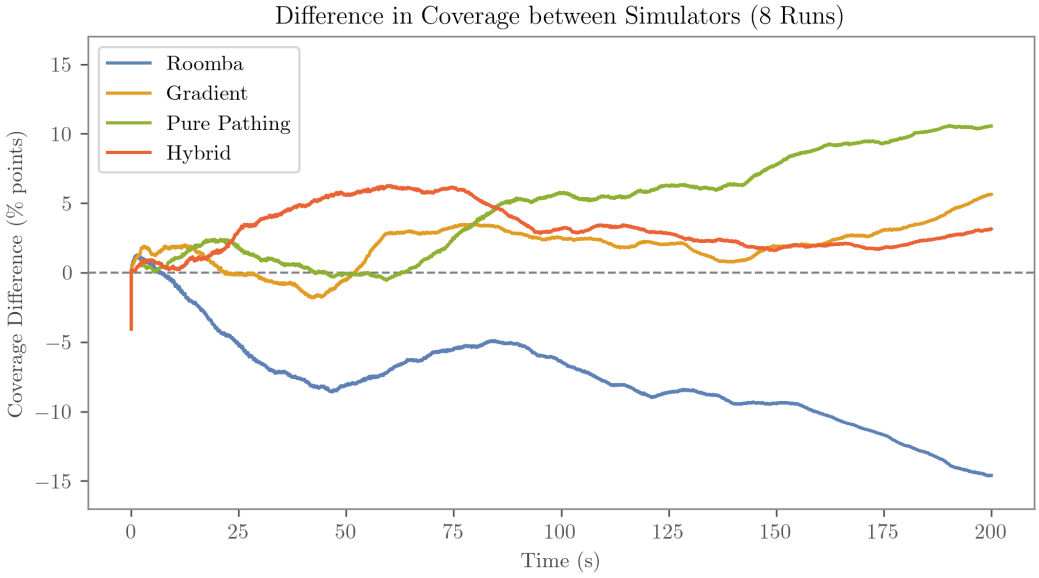
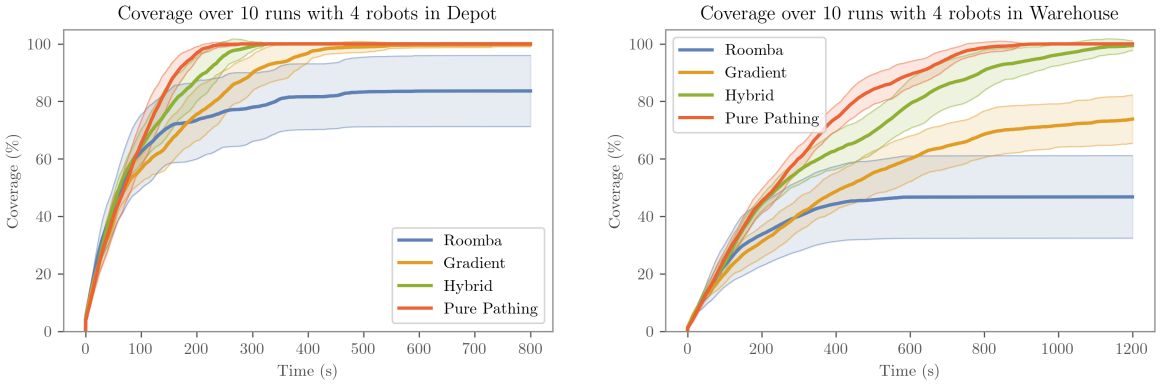


Figure 20: Comparison of difference in coverage over time between ROS 2 Gazebo and `simple_sim`. Positive values indicate coverage are faster in `simple_sim` compared to Gazebo.

9.3 Search Algorithm Benchmarks

Each algorithm was run 10 times on the Depot (for 800 s) and Warehouse (for 1200 s) maps with 1, 2, 4, 8 robots using `simple_sim`. The warehouse map was also run with 16 robots. The relative performance between the behaviors seems stable regardless of robot count. Figure 21 shows a representative result with 4 robots. All results can be found in Appendix G.



(a) 800 s in simple map (Depot)

(b) 1200 s in complex map (Warehouse)

Figure 21: Coverage performance of search algorithms over 10 runs with 4 robots starting in random positions in the same environment. Mean and standard deviation of coverage are shown.

The results clearly show that the Pure Pathing outperforms the other algorithms followed by the Hybrid algorithm. The Gradient algorithm is significantly worse than both the Pure Pathing and Hybrid algorithms. The Roomba algorithm is generally better performing in the beginning of the run, but has a hard time getting full coverage of the map, and is overtaken by the Gradient algorithm which has the search gradient to guide it towards unexplored areas.

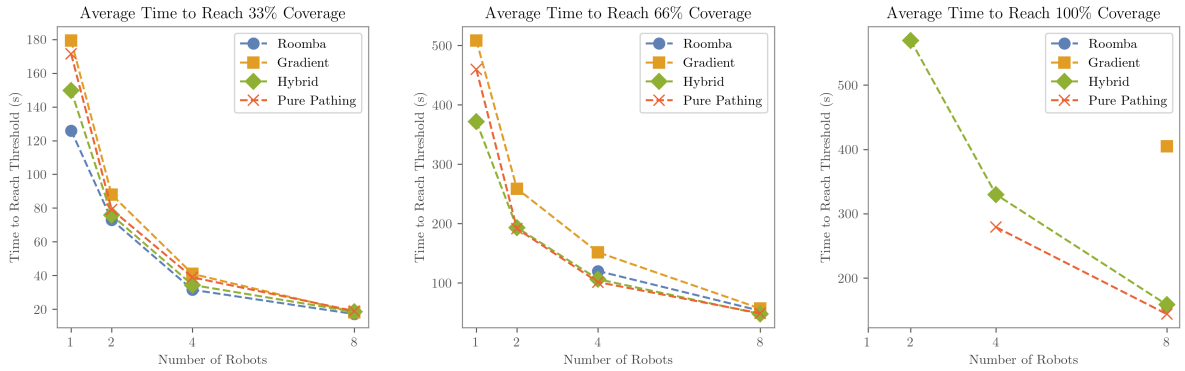


Figure 22: Average time it takes the swarm to cover 33%, 66% and 100% of the map in the Depot environment over 10 runs. Points are left out if the swarm did not reach these thresholds on average.

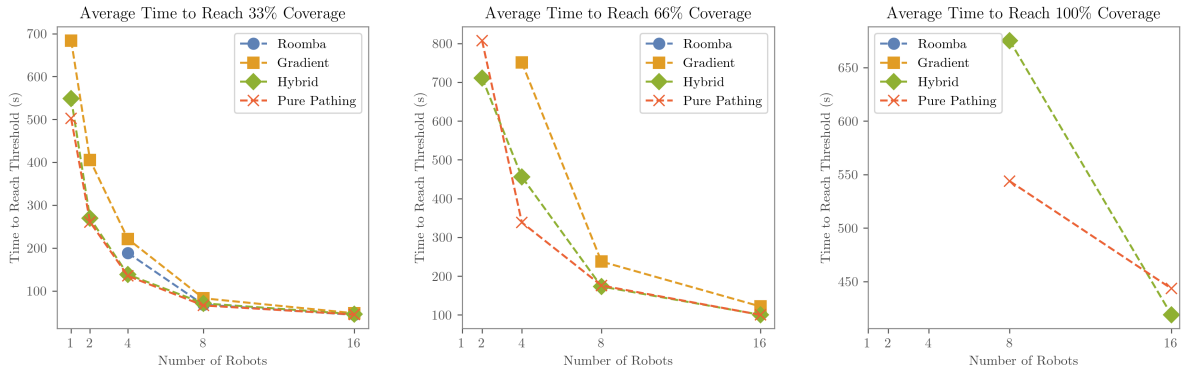


Figure 23: Average time it takes the swarm to cover 33%, 66% and 100% of the map in the Warehouse environment over 10 runs. Points are left out if the swarm did not reach these thresholds on average.

Figure 22 and fig. 23 show the average time required for the swarm to reach various coverage thresholds in the Depot and Warehouse environments, respectively, using different numbers of robots.
The results clearly indicate that the Pure Pathing and Hybrid behaviors are the most efficient in achieving higher coverage levels. This highlights the benefit of incorporating global map awareness and structured path planning into the search strategy.
In contrast, the Roomba algorithm demonstrates significantly reduced performance in the more complex Warehouse environment. Notably, it fails to surpass 33% coverage within the trial duration, regardless of the number of robots. This suggests that simple obstacle-avoiding behaviors without global coordination are insufficient in environments with dense obstacle layouts.

9.4 Computational Performance

A critical aspect of the search algorithm's performance is computational requirement for running the behaviors. Figure 24 shows the distribution of computation times per time step for each search algorithm, measured over six runs in the same environment. The data highlights key differences in computational cost and stability across behaviors.

As expected, the Roomba algorithm demonstrates the lowest and most consistent computation time due to its simplicity and lack of map processing. In contrast, the Pure Pathing algorithm performs efficiently on average but displays noticeable outliers. These spikes are caused by expensive operations — such as

costmap updates, frontier evaluation, and A* path planning — which are not executed at every time step. This variability suggests a degree of computational instability that could pose challenges when deploying on resource-constrained platforms like microcontrollers.

The Gradient method, while computationally stable, exhibits the highest average computation time. This is due to the cost of recalculating the gradient field at every step, regardless of environmental changes. The Hybrid algorithm was expected to fall between Pure Pathing and Gradient in terms of computational load, which is consistent with the observed data. Its ability to switch between strategies allows it to balance responsiveness and efficiency without incurring the full cost of either method.

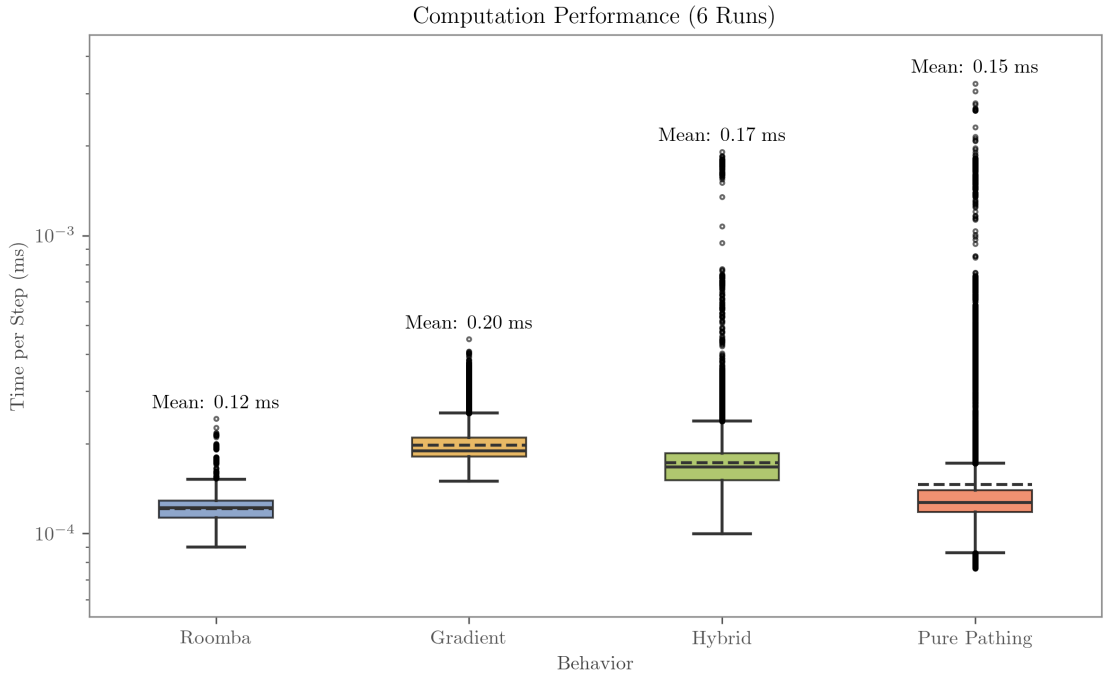


Figure 24: Computation time for each search algorithm in the same map environment. Log scale is used for better visualization.

9.5 Communication Load

The messages transmitted during operation are updates to the search grid, which are sent whenever a robot's internal search grid is modified. From these updates, other robots can synchronize their local search grids and infer the position of the robot that originated the message. Each message has a fixed size of 26 bytes. The `botbrain` library includes a configurable parameter that controls the broadcast frequency. The frequency is set to 10 Hz by default, which is one message every 100 ms, resulting in approximately 260 bytes/sec of outbound communication per robot.

A bandwidth of 260 bytes/sec negligible for most communicatino technologies considered for real-world deployment (see section 12.1.1), including Wi-Fi, Zigbee, and cellular. However, it may exceed practical limits for LoRa, particularly as the number of robots increases or in scenarios requiring higher message frequencies. If LoRa is to be considered for deployment, further message compression, reduced transmission frequency, or more aggressive filtering strategies would be necessary to meet the bandwidth constraints.

9.6 Simulator Performance

The performance and rapid iteration cycle offered by the `simple_sim` simulator were key motivations for its development in this project. To quantify its efficiency, a simple benchmark was conducted by running both simulators headless (without a GUI) for 600 simulated seconds using the Roomba algorithm. Over 5 runs, Gazebo required 628 real-time seconds on average to complete the simulation, whereas `simple_sim` completed the same task in an average of just 18 seconds. This substantial difference highlights the lightweight simulator's superior computational performance.

The ability to run simulations significantly faster than real time is highly beneficial for development. It enables rapid experimentation, easier debugging, and efficient benchmarking of algorithmic changes—making `simple_sim` a preferable tool during the design and testing phases.

9.6.1 Gazebo Stability

Benchmark data was collected with Python scripts using `botplot` to run both `simple_sim` and Gazebo on the same environments automatically for a set of runs. Gazebo was, however, quite unreliable in its initialization which meant that tests had to be supervised during benchmarking. This made it infeasible to run long trials multiple times with multiple behaviors.

10 Discussion

Hypothesis 1 As shown in section 9.3, the gradient-based algorithm does not initially outperform the baseline obstacle-avoidance algorithm. This is due to the baseline algorithm covering a large area quickly at the start of the simulation. However, it is limited by its lack of map awareness and decision-making in later stages. In contrast, the gradient-based method, although slower initially, is guided by local coverage information and ultimately surpasses the baseline in terms of total area covered.

This improvement comes at the cost of increased computational expense. As illustrated in fig. 24, the average computation time of the gradient-based algorithm is approximately 66% higher than that of the baseline algorithm, which confirms hypothesis 1. The gradient-based algorithm outperforms the obstacle-avoidance algorithm in terms of coverage efficiency, albeit with a higher computational cost.

Hypothesis 2 Section 9.3 shows that the hybrid algorithm outperforms the purely gradient-based method in terms of coverage efficiency. Surprisingly, it does so with a lower average computational cost, as shown in fig. 24. This contradicts the initial expectation that the hybrid approach, combining gradient-based navigation and high-level path planning, would be more computationally demanding.

This counterintuitive result may arise because the path planning component only incurs significant computational cost during frontier exploration and path generation. For most time steps, the robot is simply following a previously computed path, which is computationally cheap. To explore this further, a pure path-planning algorithm was implemented. This algorithm outperforms all others in coverage efficiency and is only slightly more computationally demanding than the baseline on average, demonstrating the effectiveness of high-level planning for search tasks. The pure path-planning algorithm has the lower average computation time but with spikes 10 times higher than the gradient-based algorithm. This highlights the high variability in computational performance of path-planning. A behavior which uses path-planning for main navigation and gradient-based movement as a backup could be explored to ensure reliability in corner cases where path planning may become infeasible.
Hypothesis 2 is partially confirmed. While the hybrid algorithm provides better coverage than the baseline and gradient-only methods, it does not incur a higher average computational cost, contrary to the original assumption.

Hypothesis 3 In section 9.2, search algorithms were evaluated across the two simulators to assess whether trends in `simple_sim` could be reproduced in Gazebo. The coverage progression curves for all algorithms in both simulators differ by less than 15 coverage percentage points coverage, as shown in fig. 19. Basic movement experiments (fig. 18) revealed that Gazebo introduces slight deviations due to realistic physical effects such as wheel slippage, friction, and uneven terrain. In contrast, `simple_sim` assumes ideal motion without loss, resulting in more consistent trajectories.

An outlier in the comparison was the Roomba algorithm, which performed better in Gazebo. This may be due to Gazebo's higher fidelity LiDAR emulation with more rays, leading to smoother obstacle avoidance. Overall, the performance of the Gradient, Pure Pathing, and Hybrid algorithms is consistent between simulators. However, some behaviors exhibit non-trivial differences, and `simple_sim` should not be relied upon as the sole performance benchmark for all algorithms.

Despite these limitations, `simple_sim` is useful as it runs significantly faster than Gazebo, up to 35 times faster than Gazebo, as shown in section 9.6, making it a valuable tool for rapid development and testing.

Hypothesis 3 is confirmed. The lightweight simulator approximates the key performance characteristics observed in Gazebo while offering dramatically reduced computational requirements.

11 Conclusion

This project presents a modular and extensible framework for coordinated robotic search, evaluated using both a custom lightweight 2D simulator and a high-fidelity 3D Gazebo environment. Several search strategies were implemented and tested, including a simple obstacle-avoiding baseline, a gradient-based method, a pure pathing method, and a hybrid algorithm algorithm combining gradient and pathing behaviors. A Deep Q-Network based learning approach was also explored. The goal was to assess their performance in terms of coverage efficiency, computational cost, and cross-simulator consistency. The experimental results support several of the initial hypotheses:

- The gradient-based algorithm significantly outperformed the baseline in terms of coverage efficiency, due to its local map knowledge, confirming hypothesis 1. However, as expected it introduced increased computational cost due to continuous gradient computation.
- The hybrid approach achieved higher coverage efficiency than the gradient-based method, as expected, partially confirming hypothesis 2. Interestingly, the algorithm's average computational cost was lower than the gradient algorithm. This result challenges the original assumption but led to the insight that pure path planning is a more performant alternative to gradient-based navigation in all tested scenarios.
- The comparison between `simple_sim` and Gazebo showed strong consistency in coverage trends, even accounting for differences in physical modeling and sensor noise. This was achieved with up to 35 times faster simulation speeds. This supports hypothesis 3 and validates the use of the lightweight simulator for rapid development and evaluation.

Overall, the results highlight that distributed, modular behavior can be developed and tested in using a simulator-to-simulator approach. Of the four behavioral algorithms, global path planning using frontier exploration was found to be most effective, but yielded less reliable computation times than gradient based methods. The stable nature of gradient-based behaviors may serve as a good backup in cases where global path planning becomes too computationally expensive.

12 Further Development

While this project demonstrates a complete simulation-based framework for coordinated robotic search, several opportunities exist for real-world deployment and further algorithmic enhancements. This section outlines key considerations for transitioning to physical systems and explores future directions in algorithmic development.

12.1 Real World Implementation

Although this project has not been tested on physical hardware, the software stack has been developed with real-world deployment in mind. Both the simulation architecture and behavior algorithms are modular, portable, and compatible with widely available robotics platforms and middleware, due to only requirements being: LiDAR, camera, odometry, and communication.

To ensure reliable performance in real-world conditions, the following aspects should be evaluated and potentially refined:

- **Sensor calibration and frame alignment**: Accurate transformations between the LiDAR, camera, and base frames are essential for sensor fusion and localization.
- **Robust object detection**: The current color-based detection may perform poorly under varying lighting conditions. More advanced detection techniques (e.g., learned models or depth-based filtering) could improve reliability.
- **Communication robustness**: Real-world wireless communication introduces latency, packet loss, and range limitations. Strategies such as message batching, compression, or redundancy may be necessary.
- **Real-time performance**: The behavior logic and path planning algorithms should be profiled on embedded hardware to ensure real-time feasibility.
- **ROS 2 control integration**: Currently, differential drive is simulated via a Gazebo plugin. Transitioning to `ros2_control` [25] would enable consistent controller behavior, as it offers modular hardware interfaces and is compatible with both real and simulated environments via further Gazebo plugins.

A practical next step would be to implement a small-scale multi-robot testbed using TurtleBot 4 platforms. This would facilitate hardware-in-the-loop testing and help uncover system-level challenges not observable in simulation.

12.1.1 Communication Considerations

In simulation, communication is assumed to be instantaneous and lossless. However, in real-world deployments, communication introduces latency, potential packet loss, and limited bandwidth. Selecting an appropriate communication technology is critical for maintaining coordinated behavior among robots. The following technologies may be considered:

- **Wi-Fi** [26]: Offers high bandwidth, typically between 50 Mbps and 600 Mbps under ideal conditions. It is widely supported by commercial robotics platforms but is susceptible to interference and has a limited effective range (up to 100 m indoors, 300 m outdoors).

- **Cellular (4G/5G)** [27]: Provides broad coverage and high bandwidth. 4G networks offer download speeds up to 100 Mbps, while 5G can exceed 1 Gbps in ideal conditions. Cellular networks are suitable for large-scale outdoor deployments but may introduce variable latency and incur data costs.
- **Zigbee** [28]: Designed for low-power, low-data-rate mesh networking. Typical throughput is around 250 kbps, with an effective range of up to 100 m line-of-sight. Zigbee is well-suited for structured indoor environments with modest communication demands.
- **LoRa (Long Range)** [29]: Offers long-range communication (up to 10 km in rural settings) with extremely low data rates, typically between 0.3 kbps and 50 kbps. It is appropriate for sparse, low-frequency messaging where continuous data exchange is not required.

Future work may include implementing adaptive communication policies, allowing robots to dynamically adjust their broadcast frequency and data payload based on network conditions and mission priorities.

12.2 Advancing Deep Reinforcement Learning

A basic model has been developed as a proof of concept. More conplex models are currently limited by the hardware available but with better GPU's more advanced models can be trained. The deep reinforcement learning can be extended and tuned in multiple directions to improve both learning performance and real-world applicability:

- **Network architecture tuning**: Adjusting the number of layers, neurons, and activation functions to improve learning stability and performance.
- **Network input layers**: Add input which allows for more information like global knowledge, e.g. from frontiers or the entire costmap.
- **Hyperparameter optimization**: Fine-tuning learning rate, discount factor, batch size, and exploration parameters for better convergence.
- **Reward function design**: Constructing effective reward signals that balance exploration, efficiency, and safety. Incorporating additional reward terms, such as for maintaining communication, minimizing energy use, or avoiding collisions.
- **Transfer learning**: Adapting a pretrained simulation policy to real-world conditions via fine-tuning on physical robots.

Incorporating these techniques could help develop policies that are not only more effective but also more transferable and robust under real-world deployment conditions.

12.2.1 Neuroevolution of Augmenting Topologies (NEAT)

Designing an optimal neural network architecture for a DQN agent remains a non-trivial task. The architecture can greatly affect both convergence speed and final performance.

Neuroevolution of Augmenting Topologies [30] is a genetic algorithm designed to evolve both the topology and weights of neural networks. Each candidate solution, or genome, is represented as a graph of neurons and connections. Networks are evaluated based on a fitness function, and mutation/crossover operations allow the population to explore a diverse architecture space.

NEAT offers the advantage of automatic architecture discovery, potentially outperforming manually designed networks. However, it is computationally intensive and typically requires more training cycles than DQN.
A hybrid approach could involve using NEAT to discover promising network architectures, which are then fine-tuned using standard DQN training. This could accelerate learning and yield more effective policies.

12.2.2 Proximal Policy Optimization (PPO)

Another promising direction for advancing the reinforcement learning is the use of Proximal Policy Optimization [31]. PPO is unlike DQN an on-policy reinforcement learning algorithm that has demonstrated strong empirical performance across a range of robotic control tasks.

Unlike DQN, which is value-based and limited to discrete action spaces, PPO belongs to the family of policy gradient methods and naturally supports continuous action spaces. This makes it particularly well-suited for real-world robotics applications, where actions such as velocity commands are typically continuous.

PPO improves training stability by preventing large updates to the policy. It achieves this by clipping the policy objective function, ensuring that new policies do not deviate too far from the current one. This balances the trade-off between exploration and exploitation while maintaining sample efficiency.

Integrating PPO into the current framework would enable:

- Learning of smoother, continuous control policies.
- Better handling of stochastic environments and sensor noise.

Overall, PPO offers a robust and scalable alternative to DQN for learning complex behaviors in dynamic environments. Exploring this method could enhance policy quality and ease the transition to real-world deployment, especially in continuous control domains.

13 References

[1] Sahar Trigui et al. “A distributed market-based algorithm for the multi-robot assignment problem”. In: *Procedia Computer Science* 32 (2014), pp. 1108–1114.

[2] Daichi Kato, Kosuke Sekiyama, and Toshio Fukuda. “Autonomous cooperation planning for heterogeneous multi-robot”. In: *2011 IEEE Workshop on Robotic Intelligence In Informationally Structured Space*. IEEE. 2011, pp. 63–68.

[3] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. “Deep reinforcement learning for swarm systems”. In: *Journal of Machine Learning Research* 20.54 (2019), pp. 1–31.

[4] Geoffrey Hollinger et al. “Efficient Multi-robot Search for a Moving Target”. In: *The international journal of robotics research.* 28.2 (2009-02). ISSN: 0278-3649.

[5] C Ronald Kube and Eric Bonabeau. “Cooperative transport by ants and robots”. In: *Robotics and autonomous systems* 30.1-2 (2000), pp. 85–101.

[6] Clearpath Robotics by Rockwell Automation. *TurtleBot 4*. `https://clearpathrobotics.com/turtlebot-4/`. Accessed: 23/03-2025.

[7] Martin Dahl. *R2R*. `https://crates.io/crates/r2r`. Accessed: 01/02-2025.

[8] Open Robotics. *rcl*. `https://github.com/ros2/rcl`. 2025.

[9] Using colcon to build packages. *WebGPU*. `https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Colcon-Tutorial.html`. 2025.

[10] Martin Dahl. *Minimal ros2 node using r2r*. `https://github.com/m-dahl/r2r_minimal_node/`. 2025.

[11] OpenCV team. *OpenCV*. `https://opencv.org/`. 2025.

[12] twistedfall. *Rust OpenCV bindings*. `https://github.com/twistedfall/opencv-rust`. 2025.

[13] Open Robotics. *Gazebo Tutorials/Moving the robot*. `https://gazebosim.org/docs/harmonic/moving_robot/`. Accessed: 02/03-2025.

[14] S. Macenski et al. “From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2”. In: *Robotics and Autonomous Systems* (2023).

[15] Open Navigation LLC. *Navigation 2 AMCL*. `https://docs.nav2.org/configuration/packages/configuring-amcl.html`. Accessed: 20/03-2025.

[16] Steve Macenski et al. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020. URL: `https://github.com/ros-planning/navigation2`.

[17] Open Navigation LLC. *Navigation 2*. `https://docs.nav2.org/index.html`. Accessed: 01/03-2025.

[18] Apple. *MacBook Pro (16-inch, 2021) - Technical Specifications*. `https://support.apple.com/en-us/111901`. 2021.

[19] Anirudh Topiwala, Pranav Inani, and Abhishek Kathpal. “Frontier Based Exploration for Autonomous Robot”. In: 2018.

[20] Zhi Lin et al. “An Efficient and Accurate A-Star Algorithm for Autonomous Vehicle Path Planning”. In: *IEEE Transactions on Vehicular Technology* 73.6 (2024), pp. 9003–9008. DOI: `10.1109/TVT.2023.3348140`.

[21] tracel-ai. *burn*. `https://burn.dev/`. 2025.

[22] Rust Graphics Mages. *wgpu*. `https://wgpu.rs/`. 2025.

[23] Yun-Jhong Wu. *burn-rl-examples*. https://github.com/yunjhongwu/burn-rl-examples. 2025.

[24] Open Navigation LLC. *nav2_minimal_turtlebot_simulation*. https://github.com/ros-navigation/nav2_minimal_turtlebot_simulation/tree/main/nav2_minimal_tb4_sim/worlds. 2025.

[25] ros2_control Development Team. *ros2_control*. https://control.ros.org/index.html. Accessed: 18/05-2025.

[26] Bradley Mitchell. *802.11 Standards Explained: 802.11be, 802.11ax, 802.11ac, 802.11b/g/n , 802.11a*. https://www.lifewire.com/wireless-standards-802-11a-802-11b-g-n-and-802-11ac-816553. 2024.

[27] Deanna Darah Michaela Goss. *5G vs. 4G: Learn the key differences between them*. https://www.techtarget.com/searchnetworking/feature/A-deep-dive-into-the-differences-between-4G-and-5G-networks. 2025.

[28] Norah Huang. *Bluetooth VS WiFi VS Zigbee: Which Wireless Technology is Better*. https://www.mokosmart.com/bluetooth-vs-wifi-vs-zigbee-which-is-better/. 2023.

[29] George Hardesty. *LoRaWAN Compared to NB-IoT: Low Power WAN Technologies LPWAN*. https://www.data-alliance.net/blog/lorawan-compared-to-nb-iot-low-power-wan-technologies-lpwan. 2024.

[30] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (2002). Accessed: 20/05-2025, pp. 99–127. DOI: 10.1162/106365602320169811.

[31] John Schulman et al. *Proximal Policy Optimization Algorithms*. arXiv:1707.06347. 2017. arXiv: 1707.06347 [cs.CL].

Appendix

Contents

A Online Version of This Report 44
B `botbrain` Documentation 44
C GitHub Repository 44
D Repository README 45
E Turtlebot 4 Image 49
F `botbrain` Usage Example 50
G More Plots from `simple_sim` on Depot and Warehouse Maps 51
H AI Disclaimer 53

A Online Version of This Report

An online version of this report can be found at the link:
`https://mathiasballing.github.io/coordinated-robot-search/report.pdf`

B `botbrain` Documentation

`botbrain` is documented using Cargo's build in documentation system. The documentation can be found at the following link:
`https://mathiasballing.github.io/coordinated-robot-search/docs/botbrain/botbrain/`

C GitHub Repository

All Source code developed during this project is available on GitHub:
`https://github.com/MathiasBalling/coordinated-robot-search`.

Repository README

Coordinated Robot Search

Bachelor Project 2025

- Thesis Report
- botbrain documentation

`botbrain`

Rust library crate containing the core logic of the project including the search algorithms.

Documentation can be found here.

`simple_sim`

A simple simulator for running behaviors defined in `botbrain`.

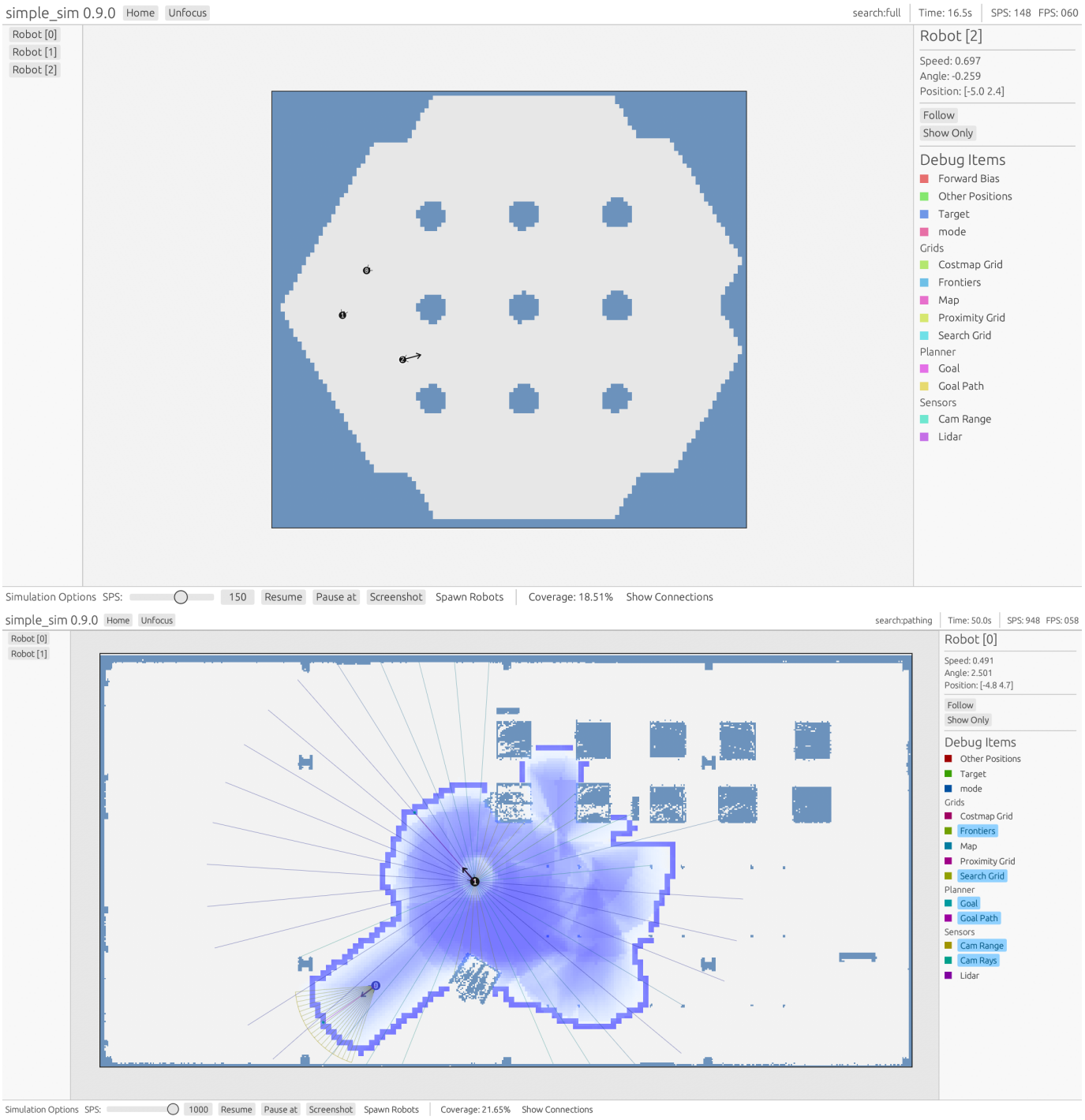
Start the simulation with:

```
cd simple_sim
cargo run --release -- run ../worlds/bitmap/depot/depot.yaml search:pathing
```

When the simulator opens, click `Spawn Robots` and click on the map to spawn robots. The speed of the simulation can be adjusted by dragging the slider in the bottom left. Information in the `Debug Soup` can be visualized by clicking on the robot and choosing a `Debug Item` in the right panel.

More launch options can be found with

```
cargo run --release -- --help
```



ROS 2

The `multi_robot_control` ROS 2 package, contains the nodes needed to run the robot behaviors in ROS 2. `ros_agent` is the main behavior node which manages the `botbrain` robot state.

The ROS 2 packages can be built with:

```
source /opt/ros/jazzy/setup.bash
cd ros_ws
colcon build --symlink-install
source install/setup.bash
```

And run with:

```
ros2 launch multi_robot_control multi_robot.launch.py behavior:=search robots:=0,0,0:2,0,1 map:=../w
```

This will launch Gazebo and Rviz2 windows, where the simulation can be observed.

See ROS 2 Agent for more information.

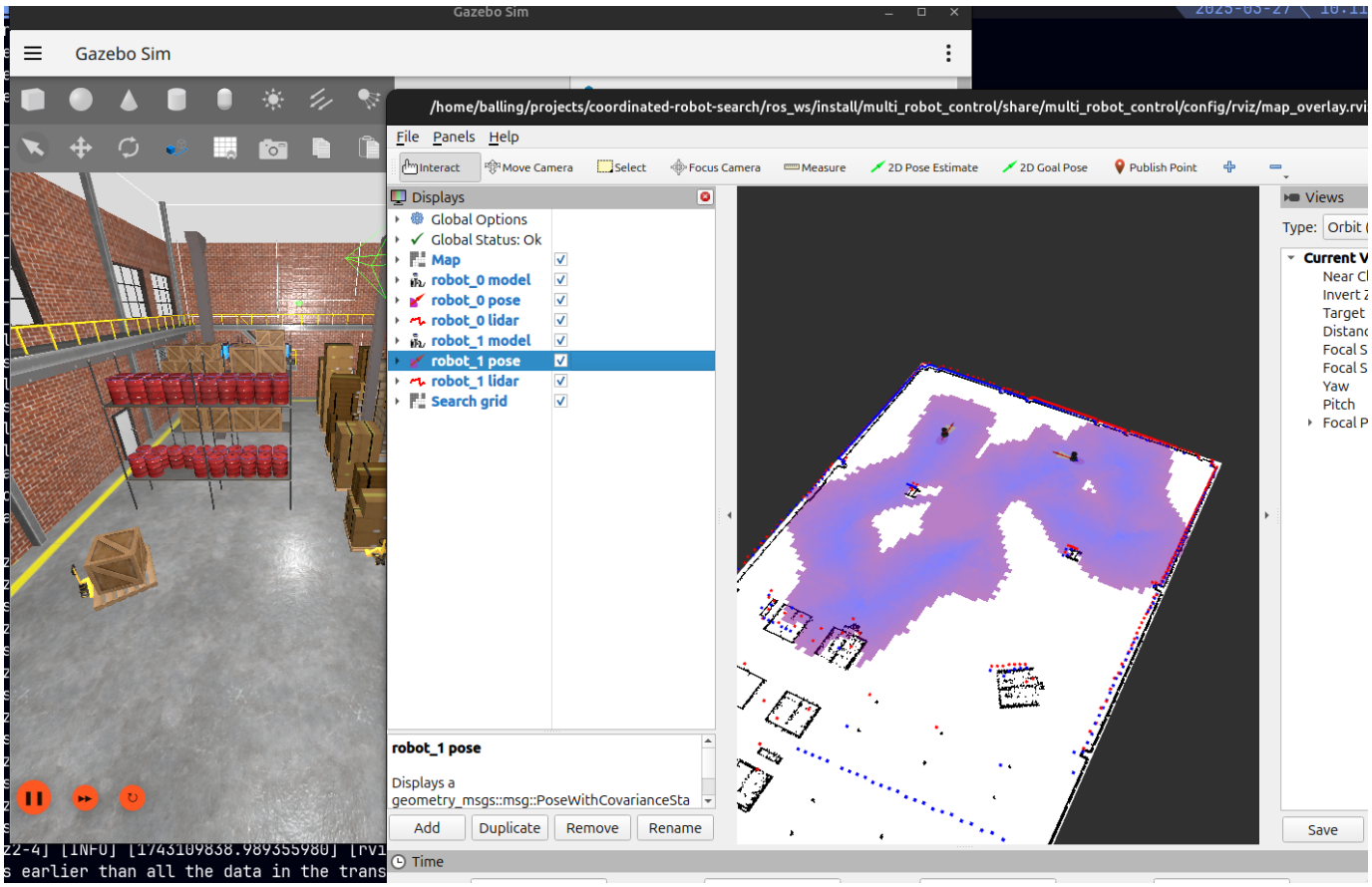


Figure 1: ROS 2 environment

Dockerfile

A Dockerfile is provided to run ROS 2 and Gazebo. The provided `enter-docker.sh` can be used to build and enter the container.

```
./enter-docker.sh
```

To rebuild the container before running run

```
./enter-docker.sh --rebuild
```

Trainer

The `trainer` crate can be used to train models defined in `botbrain`. It defines the training loop and implements deep reinforcement learning.

See the trainer README for more information.

Plotting

The `botplot` python library is used to run `simple_sim` and Gazebo in an automated manner to collect data, which can be plotted with the provided functions. Simulator runs are cached to avoid rerunning scenarios. The `plotting` directory contains scripts which use `botplot` to run the simulators and create plots.

Using `botplot`

Install the package

```
python3 -m venv venv       # Create virtual enviornment
source venv/bin/activate   # Activate the enviornment
pip install -e ./botplot   # Install the `botplot` package
```

Run a plotting script

```
python3 ./plotting/report/sim_consistency.py
```

Compiling the LaTeX Report

```
cd report
latexmk -pdf -bibtex-cond -shell-escape -interaction=nonstopmode main.tex
```

Nix Integration

This project contains a `flake.nix` which allows self-contained build recipes and development shells. All dependencies and build tools are automatically downloaded and validated.

Build the Project Build the entire project which includes:

- Executables: `simple_sim` and `trainer`
- Documentation: `botbrain` and `simple_sim`
- LaTeX Report

```
nix build .
```

The output is located in the `result` directory.

Sub-packages can be listed with the following

```
nix flake show
```

Development Shell Use the following command to install dependencies and start the development shell. ROS 2 is not supported with this shell, the `Dockerfile` can be used instead.

```
nix develop
```

E Turtlebot 4 Image

Below is an image of the Turtbebot 4 platform used in the simulation. More information can be found at the link:
`https://clearpathrobotics.com/turtlebot-4/`



F botbrain Usage Example

This example is also included in the documentation of botbrain.

```
use botbrain::{
    behaviors::{Behavior, RobotKind},
    Map, RobotId,
};

let robot_kind = RobotKind::Dumb;                    // Choose a robot
let behavior = robot_kind.get_behavior_fn("circle"); // Select a behavior function
let mut robot = robot_kind.create_fn()();            // Create a new robot

// Initialize the robot
robot.set_id(RobotId::new(0));
robot.set_map(Map::empty()); // <- Make sure to set an actual map

// Control loop
loop {

    // Get sensor data (implemented by user)
    let pose = get_pose();
    let cam = get_cam();
    let lidar = get_lidar();

    // Get incomming messages (implemented by user)
    let input_messages = recv_messages();

    // Input the sensor data and messages into the robot
    robot.input_pose(pose);
    robot.input_cam(cam);
    robot.input_lidar(lidar);
    robot.input_msgs(input_messages);

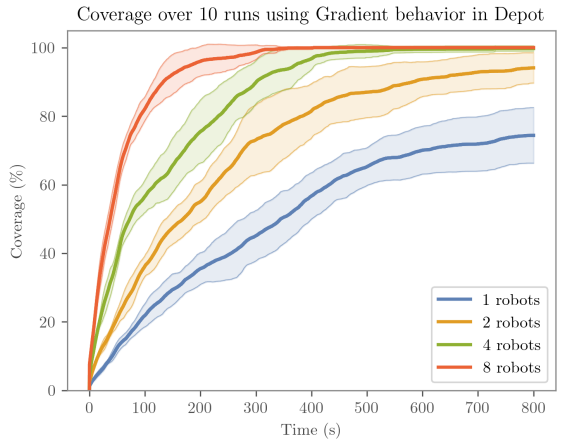
    // Run the behavior function
    let time = get_time();
    let (control, output_messages) = behavior(&mut robot, time);

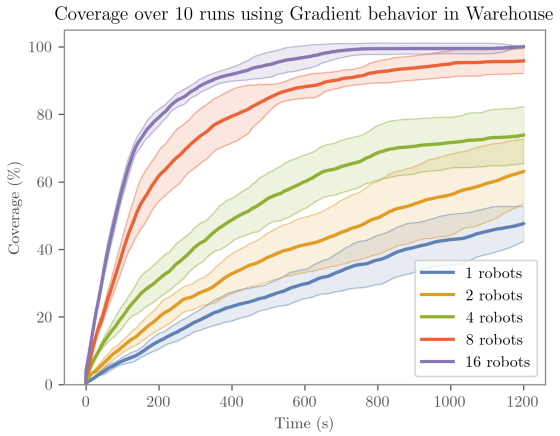
    // Send the control signal to the robot motor controller (implemented by user)
    do_control(control);

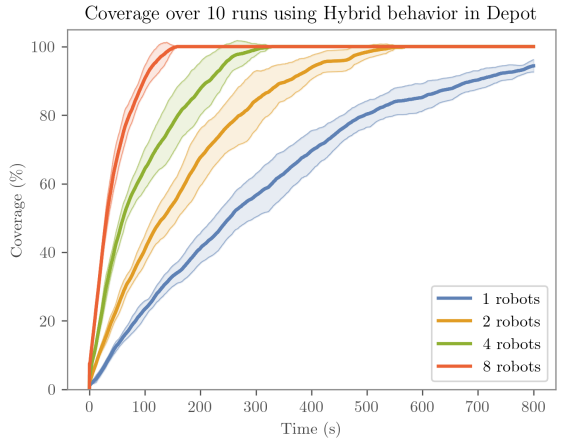
    // Send messages to other robots (implemented by user)
    send_messages(output_messages);

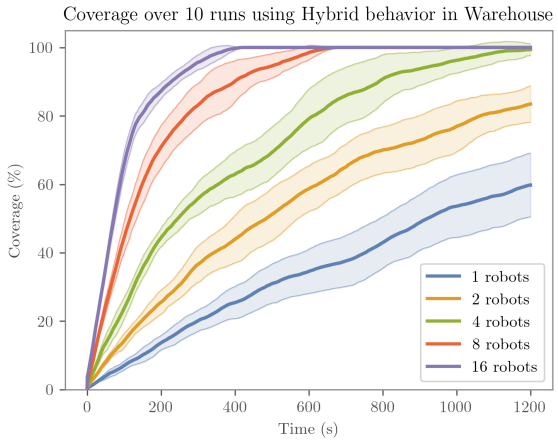
}
```

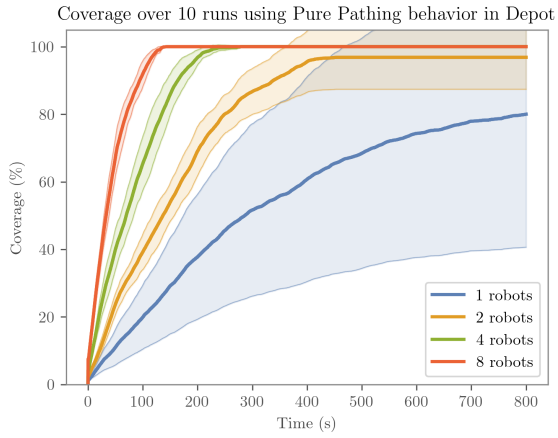
G More Plots from `simple_sim` on Depot and Warehouse Maps

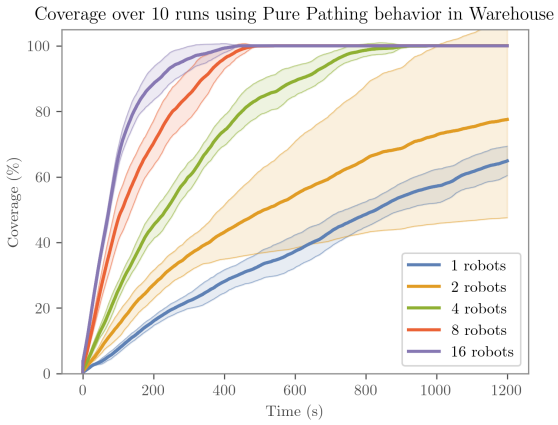


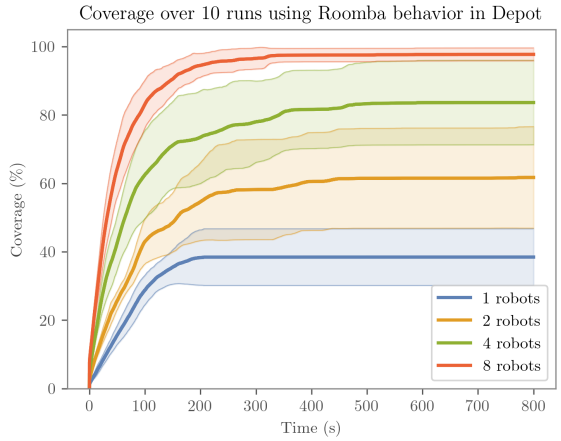


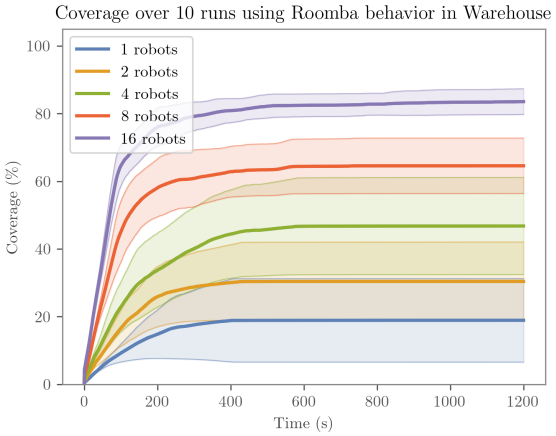


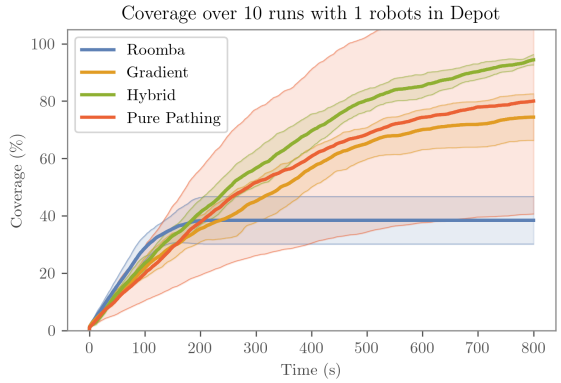


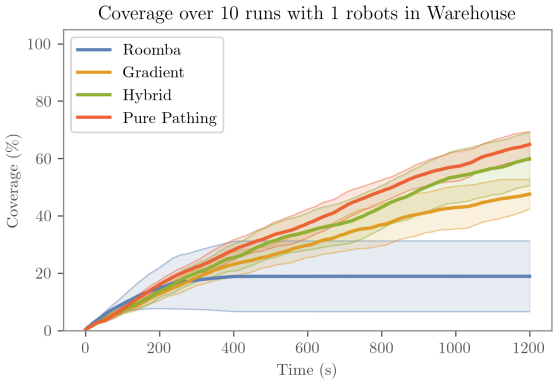


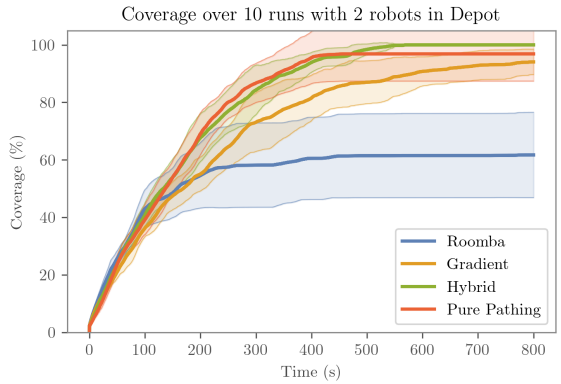


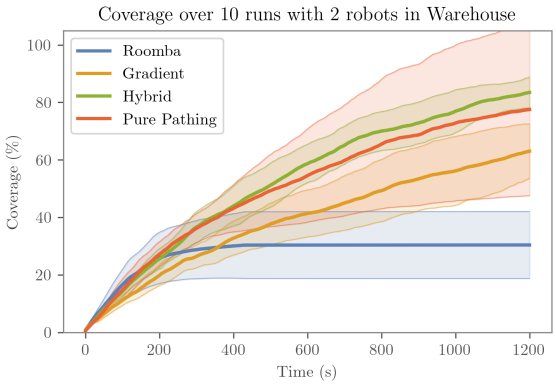

Coverage over 10 runs using Roomba behavior in Depot
Coverage (%)
Time (s)
1 robots
2 robots
4 robots
8 robots

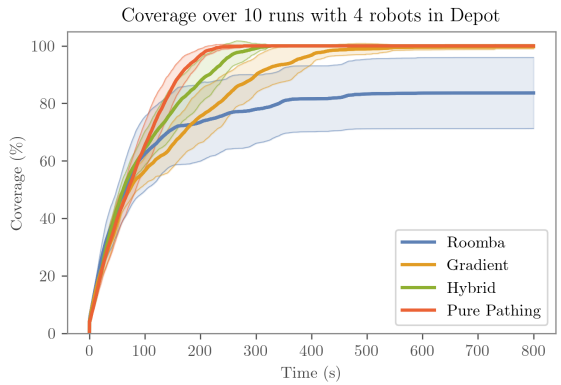

Coverage over 10 runs using Roomba behavior in Warehouse
Coverage (%)
Time (s)
1 robots
2 robots
4 robots
8 robots
16 robots

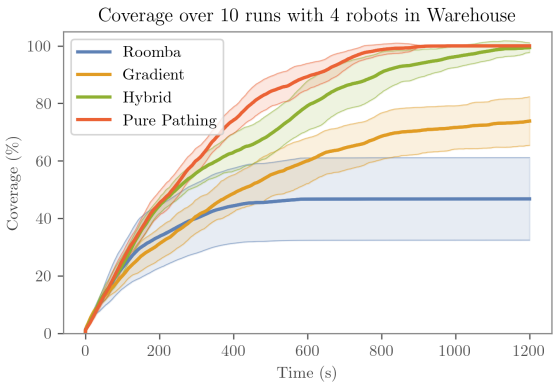

Coverage over 10 runs with 1 robots in Depot
Coverage (%)
Time (s)
Roomba
Gradient
Hybrid
Pure Pathing

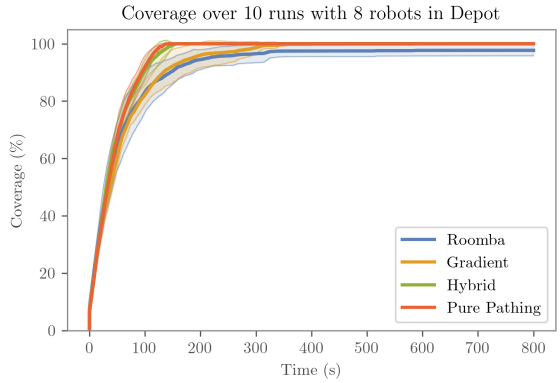

Coverage over 10 runs with 1 robots in Warehouse
Coverage (%)
Time (s)
Roomba
Gradient
Hybrid
Pure Pathing

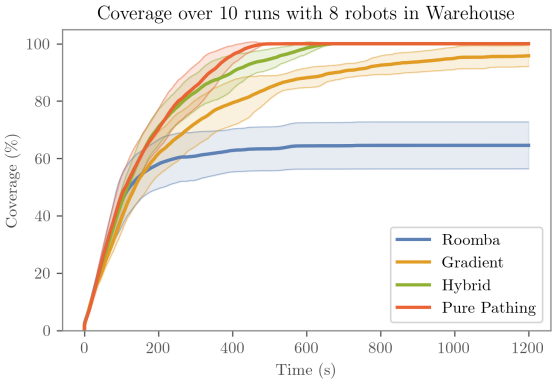

Coverage over 10 runs with 2 robots in Depot
Coverage (%)
Time (s)
Roomba
Gradient
Hybrid
Pure Pathing

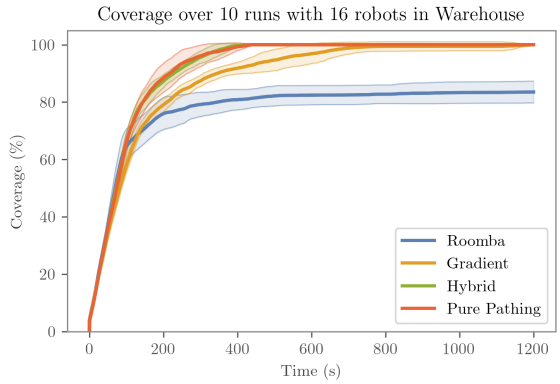

Coverage over 10 runs with 2 robots in Warehouse
Coverage (%)
Time (s)
Roomba
Gradient
Hybrid
Pure Pathing


Coverage over 10 runs with 4 robots in Depot
Coverage (%)
Time (s)
Roomba
Gradient
Hybrid
Pure Pathing


Coverage over 10 runs with 4 robots in Warehouse
Coverage (%)
Time (s)
Roomba
Gradient
Hybrid
Pure Pathing







H AI Disclaimer

AI was used during the development of this project and report. ChatGPT was used to proof-read and generate alternative passages when writing the report, while GitHub Copilot was used to generate code completions while programming.