Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Distributed Multi-Robot Trajectory Planning in a Factory Environment

Author:
Tian Yi Lim

Supervisor:
Prof. Andrew Davison

Second Marker:
Dr. Adam Smith

June 17, 2022

Abstract

Multi-robot systems (MRS) are commonplace in industry and seen as key to achieving Industry 4.0 standards. Example applications include material handling in warehouses and transport. MRS allow for tasks to be performed with greater robustness and efficiency than single robot systems. However, introducing additional robots to the workspace means that additional problems need to be resolved regarding coordination between robots. One such problem is trajectory planning, where robots need to move to their respective goal positions without colliding with obstacles or with each other.

In this work, a simulation environment is created in the Gazebo simulator using the Robot Operating System framework. A collaborative extension of the Dynamic Window Approach is then used to allow multiple robots to navigate around the environment.

Acknowledgements

I would like to thank my supervisor Dr Andrew Davison for agreeing to supervise my self-proposed project, providing some much-needed direction when the initial ideas fell through, and encouraging me to develop my understanding of the source material by adopting a 'from-scratch' approach.

I would also like to thank Dr Thomas Lancaster and Dr Thomas Clarke from the Computing and EE departments respectively for facilitating the administrative side of this project.

Furthermore, I thank my family and friends who provided some much-needed encouragement and inspiration.

Last but not least, my thanks goes to David Cormier, who helped me realise doing a project about my passion was possible.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Multi-robot systems (MRS) are already commonplace in industry and seen as key to achieving Industry 4.0 standards [4]. Example applications include material handling in warehouses [5] and transport [6]. MRS allow for tasks to be performed with greater robustness and efficiency than single robot systems. However, introducing more robots to the workspace means that additional problems need to be resolved regarding coordination between robots. One such problem is trajectory planning, where robots need to move to their respective goal positions without colliding with obstacles or with each other.

1.1 Objectives

The objective of this project is to develop a path-planning solution that allows a group of robots to navigate around a simulated warehouse environment. This environment is made of corridors formed of shelves, a common configuration in a logistics setting.

Figure 1.1: Example of a automated warehouse [1] showing how shelves are aligned in linear corridors

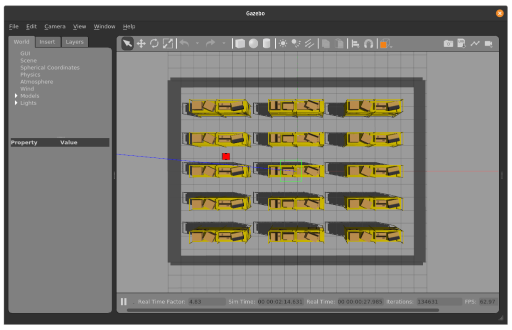

Figure 1.2: Simulated warehouse environment in the Gazebo simulator as used in the project

The simulated robots also follow similar dimensions to industrial logistics robots, for example the Neobotix MP-400 robot [7]. The MP-400 robot has a footprint of 590 by 559mm, and the simulated robots have similar dimensions of 500 by 500mm.

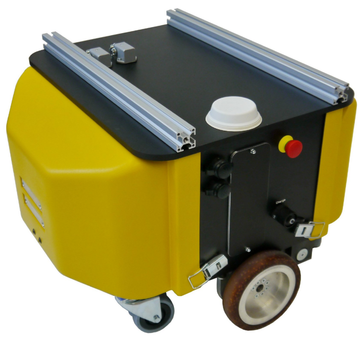

Figure 1.3: The MP-400 robot used as a real-world reference for the simulated robots in this project

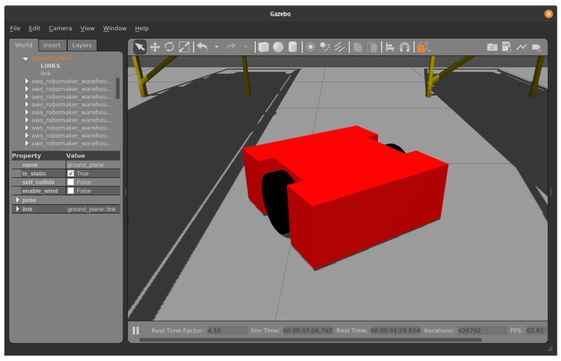

Figure 1.4: Simulated robot in the Gazebo simulator as used in the project

To be effective, the path planner should have the following characteristics:

- **Maximise throughput**: The navigation system should maximise the number of goals completed per unit time.
- **Scalable with number of robots**: As the number of robots increases, the throughput of the multi-robot system should increase.
- **Computationally efficient**: The robots should not spend excessive time stationary while planning out a path to the next goal.
- **Configurable**: The navigation algorithms used should be easy to tune. They should also be easy to swap out if better algorithms are developed.

1.2 Contributions

This project develops a path-planning system that meets the conditions listed in the problem statement. It allows a robot to navigate in a known environment in the presence of other such robots in a distributed manner. The system is flexible in that it exposes key parameters for convenient tuning. Furthermore, the system is designed such that different implementations of planning algorithms can be substituted in and out.

The system was programmed using the Robot Operating System (ROS) [2] framework and tested in the Gazebo physics simulator [8].

Although the system focuses only on path planning and assumes the robot has perfect localization of itself and other robots in relation to a global frame, other programs that handle localization, odometry, and communication between robots can be added in to create a full navigation stack using the ROS framework.

1.3 Project Evolution

Initially, this project started off as an exploration into the collaborative path-planning aspect of robots in the Eurobot [9] competition. However, it was deemed infeasible owing to the lack of reliable hardware to execute the project at the competition. Furthermore, as an EIE student, the schedule for coursework and examinations was different from the Computing Department. The project therefore focuses more closely on how multiple robots can plan paths for themselves in the presence of other robots in a factory environment.

Chapter 2 Technical Background

This chapter discusses the technical concepts needed as part of the project.

2.1 Trajectory Planning

Although this project is mainly about multi-robot planning, it is still important to discuss the single-robot case to better understand how multi-robot algorithms build upon it.

Trajectory planning can be broken down into a two-step process: *global* planning and *local* planning. Given an initial position and goal position, global planners devise a collision-free path through the robot's environment. As global planners must generate paths through an environment while minimising metrics such as distance travelled or overall acceleration, they require a high-level view of the world. Hence, global planners are *deliberative* in nature, where planning relies on an explicitly represented, symbolic model of the world. Deliberative behaviour is contrasted with *reactive* behaviour, which merely reacts using pre-set behaviours according to external stimuli.

Deliberative methods are computationally expensive compared to reactive methods due to the complexity of the environment they must consider. It is thus infeasible to for them to run at high frequency, and they are hence unsuitable to react to unexpected obstacles or environmental disturbances. Thus, local planners are also necessary for robots to navigate in previously unknown or dynamic environments.

To summarize, an analogy of why both global and local planners are needed would be driving to an unfamiliar destination. A global planner would be a route calculated using a GPS map, which provides an overarching path the driver must follow. Local planning would be the driver reacting to other road users and traffic signs, while following the navigation instructions.

2.1.1 Global Planners

Global planners allow a robot to navigate through a complicated environment, where local planners may fail due to local minima. To do so, they require an overall representation of the workspace. These planners can broadly be split into *occupancy grid* and *sampling* based approaches, which are covered below.

Occupancy Grid based approaches

Occupancy grids represent the robot's environment as a discrete grid. Grid coordinates represent locations in the real world. Each grid coordinate contains values that can represent the occupancy state of the grid. Simple approaches may use a binary occupancy state (occupied/free), while probabilistic approaches represent the occupancy of the grid as a probability of occupancy between 1 and 0.

An example of how an environment can be discretized into an occupancy grid is shown in Figure 2.2. The top-down projection of the simulation environment in Figure 1.2 is represented in the figure, where black cells represent occupied cells while white cells are free.

This approach also illustrates potential issues with the discretization of occupancy grids. A resolution of 2 grid cells per meter was used in this grid. However, this leads to rounding errors in the exact dimensions of the grid. Even though the shelves are spaced evenly within the environment, the conversion of continuously-varying coordinates to discrete grid indices leads to conversion errors. Choosing a higher grid resolution reduces the error introduced in discretization at the cost of higher memory consumption due to a larger grid size, as well a larger search space.

Occupancy-grid search approaches exploit the spatial connectivity of the occupancy grid and use graph search algorithms to find intermediate grid coordinates between the start and goal coordinates in order to find a path between them. Examples of such approaches are the A* graph search

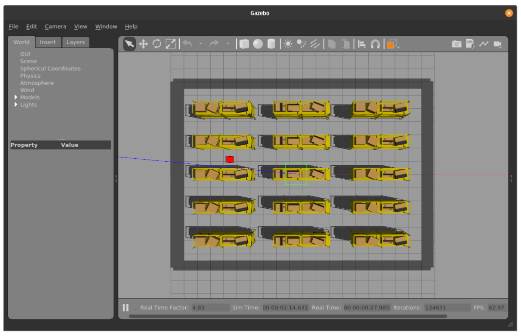

Figure 2.1: The simulation environment, first presented in Figure 1.2

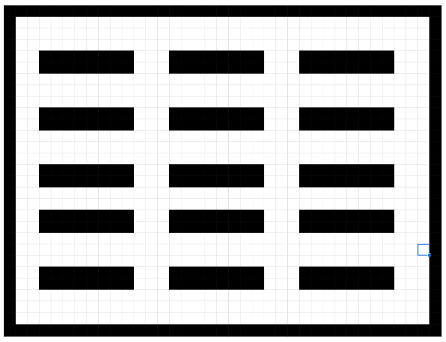

Figure 2.2: An occupancy grid representation of the simulation environment

algorithm [10] and D* reactive search algorithm [11], which modifies A* to take into account changes in the weights of the graph edges that represent the environment.

The A* algorithm is a best-first graph search algorithm which aims to find the path to the given goal node with the smallest cost. It does so by maintaining a tree of possible paths from the start node. At iteration of A*, the node with the lowest overall cost while being admissible is expanded. Cost is formulated as $f(n) = g(n) + h(n)$, where n is the next node to be expanded, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost from n to the goal node. As long as the heuristic $h(n)$ satisfies $h(x) \leq d(x, y) + h(y)$ where $d(x, y)$ is the edge cost between x and y, A* is guaranteed to find the path with the lowest cost.

The D* algorithm expands on A* by repeatedly determining the shortest paths between the node representing the current position of the robot and the goal node as the edge costs of the graph change while the robot moves around. The edge costs of the graph may change due to the structure of the map changing due to inadequate initial mapping, or due to moving obstacles in the environment. By not discarding the set of paths expanded as part of the A* search, D* is orders of magnitude faster than repeatedly using A* searches.

As A* and D* approaches return the paths with the lowest cost, they are *optimal*. They are also *complete*, meaning they will find a solution if it exists. However, graph search methods scale poorly with increased grid cell count (either with increased resolution or environment size). They have a complexity of $O(b^d)$, where b is the branching factor (number of successors per node) and d is the depth of the solution. Furthermore, as the entirety of the robot's environment needs to be represented as a grid, these methods scale poorly in terms of memory usage, with the same $O(b^d)$ memory usage, as all expanded nodes are kept in memory.

Sampling-based approaches

Sampling based approaches do not use an explicit representation of the obstacles in an environment like an occupancy grid. Therefore, they do not need to contain the entire environment in memory. Rather, they sample randomly-chosen points in the environment and check these points for validity. Two examples of sampling-based approaches are Rapidly-Exploring Random Trees (RRT) [12] and one of its extensions, RRT* [13].

RRT, detailed in Algorithm 1, returns a path P as a series of waypoints. RRT randomly samples coordinates in the robot's workspace and obtains an admissible (collision-free) point $\mathbf{x}_{new}$ (`randomlySampleAdmissiblePoint`). The nearest node to $\mathbf{x}_{new}$ in the node list N, n_{near} is obtained (`getNearestNeighbor`). A point $\mathbf{x}_{ext}$ moving an incremental distance away from n_{near} in the direction of $\mathbf{x}_{new}$ is obtained (`getPointOnExtensionLine`), and the line connecting $\mathbf{x}_{ext}$ to n_{near} is checked for collision with obstacles (`lineIsCollisionFree`). If so, a new node n_new is created at $\mathbf{x}_{ext}$ with n_{near} as a parent and added to N. If n_{new} is close enough to the goal coordinates, a path is created from N by traversing each node's parent until the start node is reached (`returnPathFromNodeList`).

Algorithm 1 The RRT algorithm

```
1:  Initialize node list $N = \{n_{start}\}$
2:  for $k$=1 to MAX_RRT_ITERATIONS do
3:      $\mathbf{x}_{new}$ = randomlySampleAdmissiblePoint()
4:      $n_{near}$ = getNearestNeighbor($N$, $\mathbf{x}_{new}$)
5:      $\mathbf{x}_{ext}$ = getPointOnExtensionLine($\mathbf{x}_{new}$, $n_{near}.pos$)
6:      if lineIsCollisionFree($\mathbf{x}_{ext}$, $n_{near}.pos$) then
7:          $n_{new}$ = createNewNode($\mathbf{x}_{ext}$, parent= $n_{near}$)
8:          $N$.append($n_{new}$)
9:          if closeToGoal($n_{near}$) then
10:             return returnPathFromNodeList($N$)
11:         end if
12:     end if
13: end for
14: return False, No path found
```

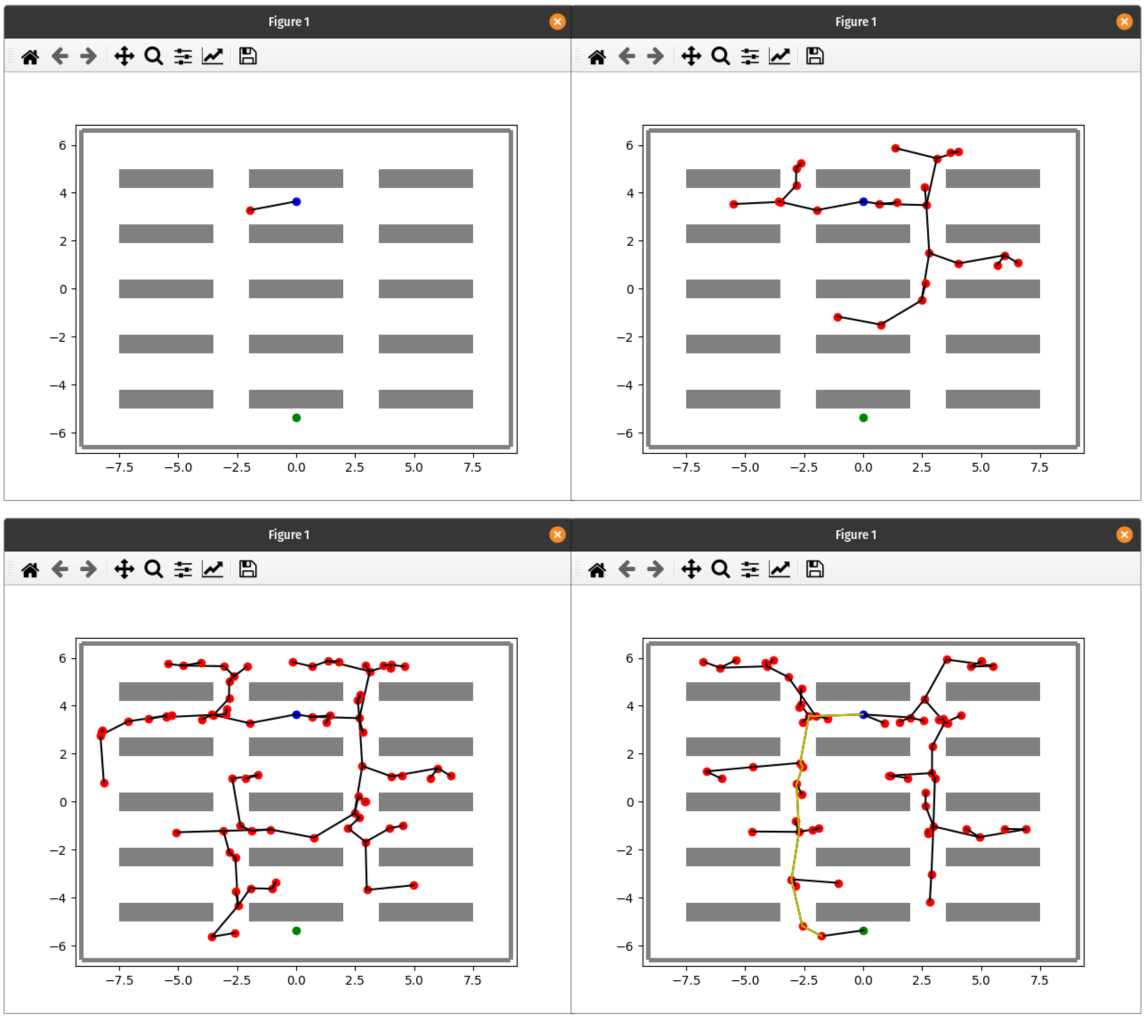

Figure 2.3: Steps of the RRT algorithm

Figure 2.3 illustrates a typical call of the RRT algorithm.

1. *Top Left*: RRT start node is in blue, goal node is in green. A randomly sampled node in red has been connected to the start node with a black line.
2. *Top Right*: RRT can explore unknown environments quickly.
3. *Bottom Left*: After several iterations of the RRT algorithm, a large proportion of the environment has already been explored.
4. *Bottom Right*: The RRT algorithm has found a path to the goal, shown in yellow.

However, RRT does not converge to an optimal solution, as it makes no effort to connect nodes that offer a lower path cost. RRT* builds upon RRT by adding asymptotic optimality guarantees at the cost of increased computation per sampled point. RRT* is similar in principle to RRT, but each node also tracks the path cost from the start node to itself. When new nodes are expanded, the RRT* algorithm rewires the tree as it discovers new lower-cost paths reaching the nodes already in the tree. This means that as the number of expanded nodes approaches infinity, the path from the start to goal node will approach the optimal path.

Algorithm 2 The RRT* algorithm

```
1: Initialize node list $N = \{n_{start}\}$
2: for $k$=1 to MAX_RRT_ITERATIONS do
3:   $\mathbf{x}_{new}$ = randomlySampleAdmissiblePoint()
4:   $n_{near}$ = getNearestNeighbor($N, \mathbf{x}_{new}$)
5:   $\mathbf{x}_{ext}$ = getPointOnExtensionLine($\mathbf{x}_{new}, n_{near}.pos$)
6:   if lineIsCollisionFree($\mathbf{x}_{ext}, n_{near}.pos$) then
7:     $N_{near}$ = getNearNodes($N, \mathbf{x}_{ext}$)
8:     $n_{bestParent}$ = getBestParent($N_{near}, \mathbf{x}_{ext}$)
9:     $n_{new}$ = createNewNode($\mathbf{x}_{ext}$, parent= $n_{bestParent}$)
10:    $N$.append($n_{new}$)
11:    rewire($n_{new}, N_{near}$)
12:    if closeToGoal($n_{near}$) then
13:      goalPathFound = True
14:    end if
15:    if minIterationsPassed and goalPathFound then
16:      return returnPathFromNodeList($N$)
17:    end if
18:  end if
19: end for
20: return False, No path found
```

RRT*, detailed in Algorithm 2, modifies the RRT algorithm by looking at the nodes N_{near} near the newly-generated node n_{new}. n_{new} is connected to the node in N_{near} with the lowest path cost to n_{new}. This node is named $n_{bestParent}$. After this, the nodes in N_{near} are rewired (`rewire`).

Rewiring is done by going through all nodes n_i in N_{near} and checking if changing their current parent to n_{new} would not cause a collision and reduce their overall cost. The hypothetical cost would be $n_{new}.cost$+`dist`(n_{new}, n_i). If $n_i.cost$ is higher than this hypothetical cost, then the parent of n_i is changed to n_{new}. The modified costs of n_i are then recursively propagated to the children of n_i.

The exit condition of RRT* is also different from RRT. While RRT exits as soon as a goal is found, RRT* may choose to only terminate after a minimum number of iterations has passed. This gives the algorithm a chance to find a more optimal path than the first one found through random search.

Figure 2.4 shows the difference implementing RRT* makes. The left image is from RRT while the right image is from RRT*. RRT results in a less direct path, especially down the vertical corridor. In contrast, RRT* results in a remarkably straight path down the vertical corridor even though

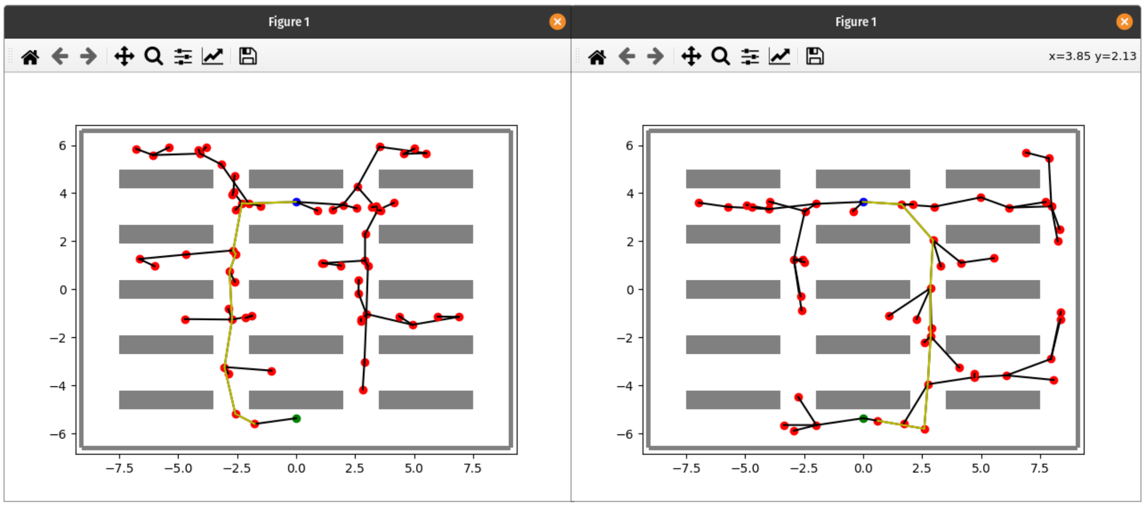

Figure 2.4: The difference between RRT (left) and RRT* (right)

there are a similar number of expanded nodes. Furthermore, it can be seen that no path in the RRT* example loops back on itself due to rewiring. However, this is very evident on the RRT example, especially on the left and right extremes of the image.

2.1.2 Local Planners

Some single-robot local planner approaches were considered, and elaborated on below. Typically, these approaches require tuning a cost function, which takes various factors such as distance to obstacles and the goal into account. This emphasis of tuning can be cumbersome, but also introduces the opportunity for extensibility if the cost function can be tweaked to account for new metrics.

Dynamic Window Approach

The Dynamic Window Approach (DWA) [14] enumerates the dynamically feasible control inputs to a robot and projects them forward in time to come up with predicted positions. It then scores the predicted positions using a ranking function and chooses the control input corresponding to the highest-scoring predicted position. This control input is then executed for a short duration before the process of enumerating and scoring possible trajectories is repeated.

Figure 2.5 depicts a typical time-step of DWA's operation for a differential drive robot, represented by the white square. The robot currently is moving with a given linear velocity $v = v_0$ and zero angular velocity $\omega = 0$, represented by the blue dotted line.

DWA first enumerates the *dynamically feasible* velocity commands based on the acceleration bounds of the robot. These are obtained by adding and subtracting from the current values of v and ω. In this case, linear velocities of v_-, v_0 and v_+ are obtained, while angular velocities of ω_-, 0 and ω_+ are obtained, resulting in a total of 9 possible velocities. For each modified value of (v, ω), DWA assumes that the robot continuously travels at these velocities for a constant forward simulation duration and obtains simulated end poses for each possible velocity. The simulated end poses of each velocity are marked out on the diagram with the dotted lines. For instance, the control input with (v_-, ω_+) will result in the trajectory on the bottom right, with the robot turning right but not moving forward very much.

These eventual end poses are filtered for *admissibility*. In the example, control inputs (v_0, ω_-) and (v_+, ω_-) are inadmissible because they will collide with the obstacle, represented by the red rectangle. They are hence colored red.

The list of admissible end poses are then scored using a heuristic. In the original DWA implementation, the scoring heuristic was $\sigma(\alpha\,\text{heading}(v, \omega) + \beta\,\text{distance}(v, \omega) + \gamma\,\text{velocity}(v, \omega)$ where

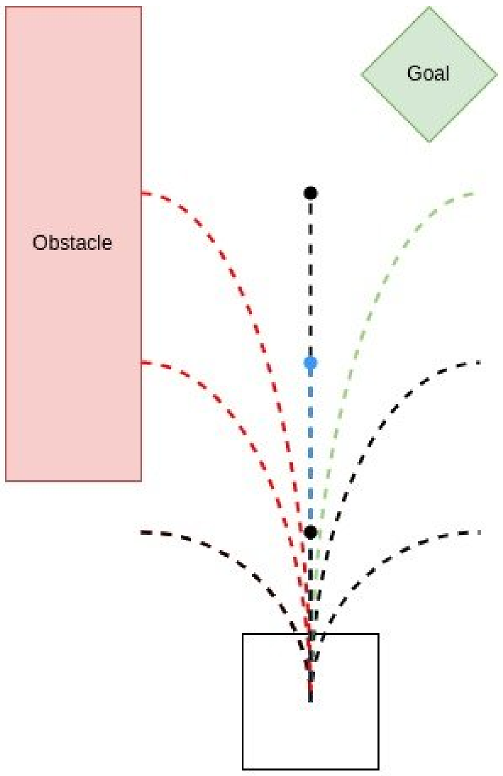

Figure 2.5: Diagram describing the DWA algorithm

heading and distance are with respect to the goal. In the example, control input (v_+, ω_+) comes closest to the goal, and is likely to score the highest. It is hence colored green.

DWA then applies the green control input for a short time period before repeating the process of finding dynamically feasible inputs and filtering them for admissibility before scoring them again. In this way, DWA enables a robot to react to changing local surroundings while moving towards its defined goal. This makes DWA an attractive starting point for the local planner of a multi-robot system, as it gives each robot the ability to react to the position of other robots and potentially move around them.

Timed Elastic Band Approach

The Timed Elastic Band (TEB) approach [15] converts a series of waypoints from a global planner into a trajectory which depends explicitly on time. The path between waypoints is found using a weighted multi-objective optimization framework. This framework takes into account the dynamic constraints of the robot, such as maximum velocities and accelerations, in addition to keeping distance from obstacles while minimising the overall path length.

A "timed elastic band" is described by a series of n intermediate poses $Q = \{\mathbf{x}_i\}_{i=0\ldots n}$ and $n-1$ time differences $\tau = \{\Delta T_i\}_{i=0\ldots n-1}$, in which each time difference denotes the time that the robot needs to transit from one pose to the next. The TEB is defined as tuple of both sequences $B := (Q, \tau)$.

The TEB approach then optimises B in terms of both intermediate poses and time intervals using weighted multi-objective optimization in real time. For instance, objectives to be optimized for could be minimizing the distance to waypoints of the original path, and maximising the distance to static and dynamic obstacles, while respecting the dynamic constraints of the robotic platform.

The TEB approach is similar to DWA in that its core idea is receding horizon control, but is implemented in a more sophisticated manner than DWA. Notably, TEB tries to find the time-optimal solution, and considers the entire set of waypoints given by the global planner to the eventual goal, rather than the more short-sighted view of DWA, which only considers the next waypoint in the sequence. However, as TEB applies optimization to many more parameters than DWA, it is necessarily more complex, both in terms of implementation and in computation.

2.2 Multi-Robot Path Planning

Although not implemented for this project, a review of multi-robot planning approaches was conducted. These problems may also be called multi-agent path finding (MAPF) problems. Solutions to the MAPF problem can be broken down into *centralized* and *distributed* approaches.

In centralized approaches, a single computer needs to find a solution for all agents. Thus, these approaches have full knowledge of the search environment for all agents, and therefore can find an optimal solution. However, they scale poorly with an increasing number of agents, having been shown to be NP-Hard [16].

In contrast, distributed approaches, do not take all agents into consideration. Each agent may find a solution for itself using its own computing power, and agents may be allowed to communicate with each other to resolve conflicts. While this reduces the search space, there is no guarantee that the produced trajectories are free of collisions.

2.2.1 Centralized Approaches

Conflict-Based Search

Conflict-Based Search (CBS) [17] is an optimal, complete search algorithm for graph-based worlds. CBS searches for paths that respect *constraints* between agents. A constraint is defined as an agent a not being allowed at a graph vertex v at a time step t. A constraint can thus be denoted as the tuple (a, v, t).

The CBS algorithm is a two-level search. At the low level, CBS searches for the shortest path for each agent that respects the given constraints, where agents may either *move* or *wait* at each time step. Each agent's paths are compared, and if there are any conflicts, constraints are added to a *Conflict Tree*, a binary tree whose nodes keep track of the set of constraints and a set of solutions consistent with the constraints. At the high level, a priority queue is implemented to expand nodes with the lowest cost.

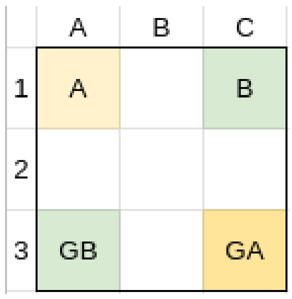

Figure 2.6: Example CBS map

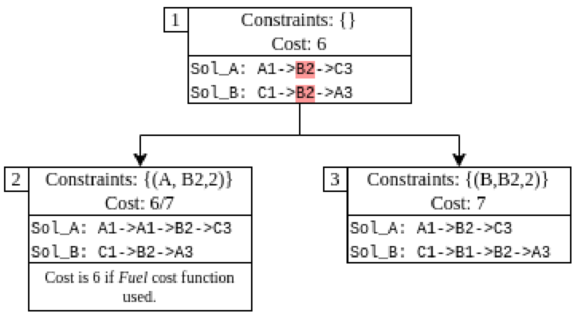

Figure 2.7: Example CBS configuration

A toy example of CBS is shown above. A two-agent scenario is shown in Figure 2.6. In a 3x3 grid world, agents A and B start on opposite edges of the top side of the map and have goals on the bottom side of the map, G_A and G_B respectively. The solution is shown in Figure 2.7. Initially, there are no constraints in the Conflict Tree (CT), and the low-level search simply finds the straight-line path to the goal for each agent, as seen in Node 1 of the CT. However, a conflict is found at $t = 2$, where both agents occupy `B2`. The CT then branches with constraints for either A or B, and low-level search is run again. Depending on the cost for nodes 2 and 3, the one with the lower cost is expanded first (checked for conflicts).

Depending on the cost function, there are two optimal scores for this case. If a "*Fuel*" cost function is used, where the total distance travelled by all agents is used, then Node 2 represents an optimal solution as A waits at `A1` at $t = 2$. In Node 3, B moves to an empty space, `B1`, while A occupies `B2`. However, if the cost is the total time steps taken for agents to reach their goals, then both nodes 2 and 3 have the same cost of 7.

Due to its two-level search, CBS is largely more efficient than pre-existing optimal MAPF approaches. As it also searches all possible configurations of agent paths, it is also complete. However, as CBS branches on each conflict, its runtime is exponential with the number of conflicts. Thus, CBS performs poorly in crowded scenarios.

Prioritized Safe-Interval Path Planning

Safe-Interval Path Planning (SIPP) [18] introduces the concept of *safe intervals* for robots to perform path planning in dynamic environments. While there may be many safe time steps for any given state, SIPP exploits the observation that the number of safe intervals is generally much smaller than the number of time steps that make up these intervals. A safe interval is defined as a time period where there are no collisions, and if it was extended one time period forward or backward, there would be a collision.

SIPP searches through states defined by configuration (robot pose) and safe interval. By using safe intervals instead of time periods to define valid states, the dimensionality of the search space is greatly reduced. For instance, if the configuration $A = (x = 0, y = 0)$ has safe intervals from $[0, 2)$ and $[3 - 5)$, then there are considered to be two states at A. This is reduced from the four states $t = 0, 1, 3, 4$ if time periods were used, and obviously becomes even better as the duration of the safe interval increases.

The SIPP algorithm is also a graph search algorithm. The graph is constructed by creating a timeline of safe intervals for each configuration using predicted dynamic obstacle trajectories. A variant of A* search is then run on the graph, where transitions between states are governed by the time taken to move between configurations, and if the robot will be able to reach the next state within the state's safe interval. The heuristic used for the A* search is the duration of movement, and therefore SIPP returns a collision-free path with the shortest duration.

SIPP can then be extended to *Prioritized* SIPP. Robots are assigned unique priorities, and paths are planned one by one. Subsequent robots see preceding robots as dynamic obstacles. This can be seen as a hybrid between centralized and distributed approaches, as preceding robots' trajectories should be passed to succeeding robots for planning, but each robot's trajectory is decided independently of others. This allows for collision-free paths to be found for all robots, but is not complete like CBS.

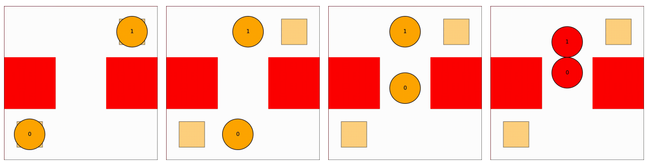
Figure 2.8: Showing SIPP fail to find a path. Pictures from left to right

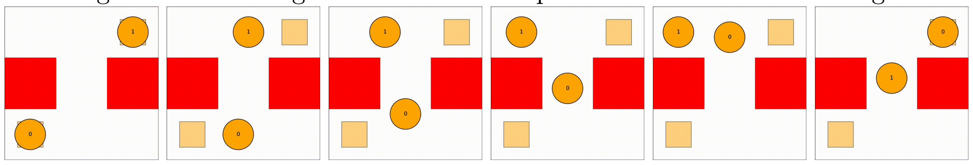
Figure 2.9: Showing CBS finding a path. Pictures from left to right

An example of how SIPP fails to find a path is in Figures 2.8 and 2.9, taken from graphics in [19]. These depict a "swap" scenario where robots 0 and 1 need to swap position through a narrow corridor. SIPP is unable to modify the path of robots with higher priority, and therefore fails to find a path. On the other hand, CBS searches the possible paths of 0 and 1 and is able to modify the path of 1 such that it avoids 0 as 0 comes up the corridor. Depending on the environment, this may mean that SIPP is an unsuitable algorithm.

2.2.2 Distributed Approaches

Velocity Obstacles

When navigating in dynamic environments, Velocity Obstacles (VO) [20] are the set of velocities of a robot that will result in a collision with a dynamic obstacle. The velocity obstacle for a robot A induced by a moving obstacle B can be written as:

$$VO_{A|B} = \{\mathbf{v} \mid \exists t > 0 : (\mathbf{v} - \mathbf{v_B})t \in D(\mathbf{x_B} - \mathbf{x_A}, r_A + r_B)\}$$

A has position $\mathbf{x_A}$ and radius r_A, and B has position $\mathbf{x_B}$, radius r_B, and velocity $\mathbf{v_B}$. $D(\mathbf{x}, r)$ represents a disc with center $\mathbf{x}$ and radius r.

This creates a cone of velocities that A cannot take if it is to avoid collision with B at some point in the future. This concept can be used for dynamic obstacle avoidance if incorporated into their local planners.

However, this approach must be augmented for navigation with other robots using the same control scheme, as it results in oscillating trajectories. This is akin to two humans walking in opposite directions in a corridor and attempting to avoid each other, but dodging in the same direction instead. To avoid this oscillation, Reciprocal Velocity Obstacles (RVO) and Hybrid Velocity Obstacles (HRVO) [21] were developed as extensions of the original VO concept.

HRVO implements a "convention" for robots using it to follow, like how cars keep to a pre-defined side of the road. This is done by enlarging the VO towards one side of the robot, encouraging them to keep to one side. This approach is more efficient than a classical VO approach because it allows for a form of cooperation between robots, in that both robots attempt to avoid each other while both simultaneously moving toward the direction of their goals without needing to wait for each other.

Model Predictive Control

Model Predictive Control (MPC) finds the optimal control input by forward simulating the effects of the control input over a finite time horizon. The optimization aspect can be seen as similar to the TEB approach, while the forward simulation aspect is akin to DWA. However, while DWA uses an approximation of the system dynamics for forward simulation, MPC requires the system dynamics to be identified or estimated accurately. This is of particular value when the system under consideration includes large time delays or higher-order dynamics.

An example of MPC used for multi-robot control is in [22]. Nonlinear MPC was used in a decentralized manner to control several Micro Aerial Vehicles (MAVs). This allowed MAVs to track a trajectory from a global planner while avoiding collisions in real-time. MPC is useful for controlling MAVs, as they operate in 3D space and have a much larger state space than 2D robots. Furthermore, the dynamics of MAVs are much more complicated than those of robots in 2D. However, in the case of 2D robots travelling at low speeds as in this project, simpler controllers are likely to be sufficient.

2.2.3 Planner Selection Criteria

In general, planners should be chosen according to the following criteria [23]:

1. *Completeness*: A complete planner guarantees that a solution is returned, if it exists.
2. *Optimality*: An optimal solution is defined based on application-specific criteria such as distance, time, or energy use.
3. *Correctness*: A correct planner guarantees that if a solution is returned, it will lead the robot to its goal.
4. *Dealing with kinodynamic constraints*: Robots have kinodynamic (kinematic and dynamic) constraints. Kinematic constraints consider the physical configuration of a robot. For instance, a differential drive robot cannot move sideways. Dynamic constraints consider the maximum velocity and acceleration of the robot.

5. *Robustness against a dynamic environment*: Planners may need to handle moving obstacles.
6. *Robustness against uncertainty*: Planners may need to handle uncertain configurations of the robot, obstacles, and environment.
7. *Computational complexity*: The memory usage and time required to come to a solution.

In the motivating scenario, agents are constantly engaged with new tasks. When a robot is done with a task, it is assigned a new task almost immediately somewhere else in the warehouse. As robots are unlikely to reach their navigation goals simultaneously, it is not feasible for robots to wait for all other robots to finish their motions before starting another task. This precludes the use of the centralized multi-robot planners described earlier in Section 2.2.1. Furthermore, it is desirable for these scenarios to scale up both in size and with increased number of robots. Centralized approaches scale poorly in both dimensions, which gives credence to distributed approaches. Lastly, the optimality and completeness guarantees offered by centralized approaches are not needed in the motivating scenario, as the environment is not maze-like. Robots will not get stuck in scenarios like Figure 2.8. Hence, a distributed approach was chosen.

In terms of distributed approaches, a local global planner is needed for individual robots to obtain an initial road map towards their goal. Given the desire to scale up the environment, grid-based methods may not scale as well as sampling-based planners. The decision was therefore taken to implement a sampling based planner for this project.

Lastly, given the options between local planners, it was decided to implement a version of DWA, owing to its relative simplicity. This would reduce the overall computational load on each robot, which would allow it to run more sophisticated algorithms for other tasks. Alternatively, this would reduce the computational requirements for each robot, which would mean a cheaper processor could be used. In a scenario where there are many tens or even hundreds of such robots, such cost savings could add up to a significant sum.

2.3 Development Environment

2.3.1 Robot Operating System

The Robot Operating System (ROS) [2] is a framework for robotics development. Its programming model splits computation into units called *nodes* which communicate over channels called *topics*. Communication between nodes is handled by ROS middleware. This architecture allows for modularity of code, as different capabilities can be split into different ROS nodes and interchanged as the situation demands. Furthermore, ROS is commonly used in the robotics industry and for scientific research, so any implementation of navigation algorithms in this project can be further re-used by other parties. ROS supports nodes written in Python and C++. For this project, Python was used throughout for faster development. A diagram of the system architecture of ROS is in Figure 2.10.

2.3.2 Gazebo Simulator

While ROS allows for implementation of robotics algorithms, a simulation environment is still needed to validate the robots' operation. The Gazebo simulator [8] is widely used alongside ROS for simulation of robots before they are implemented in hardware. Gazebo allows for physics and sensors to be simulated with high fidelity, reducing the simulation-to-reality gap. Most importantly, Gazebo has well-documented ROS *plugins*, which allow ROS nodes to interact directly with the simulator. A screenshot of the Gazebo simulator is in Figure 2.11.

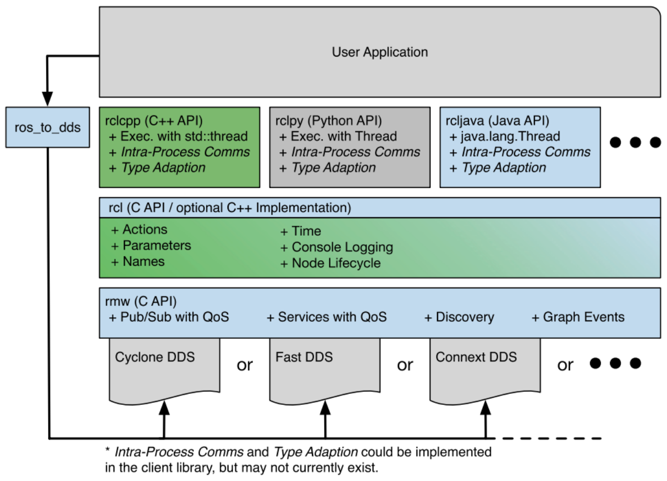

Figure 2.10: A diagram of the ROS system architecture from [2]

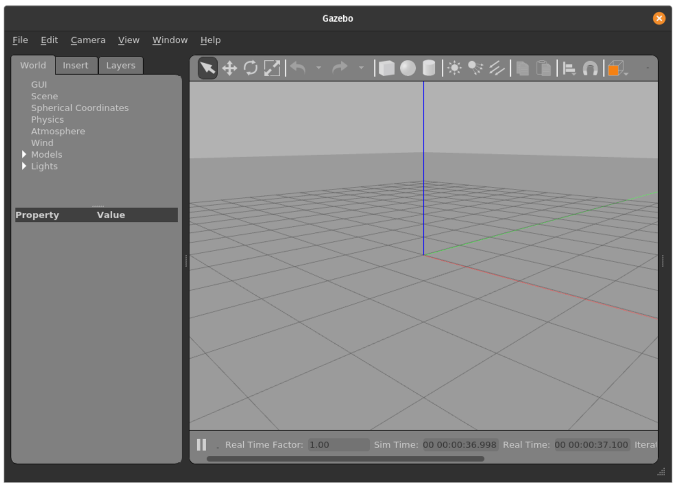

Figure 2.11: A screenshot of an empty simulation world loaded in Gazebo's GUI

Chapter 3 Implementation

3.1 Simulation Environment

In the Gazebo simulator, environments and objects are defined in the Simulation Description Format (`sdf`), an XML format. To mimic an automated warehouse, models of shelves were taken from a sample Amazon Web Services Robomaker world [24] and arranged in a 3x5 grid. Each shelf model is approximately 4m in the x direction and 0.8m in the y direction. The spacing in between the shelves in both the x and y directions was 1.5m to allow two robots to pass through a corridor between shelves simultaneously. To complete the environment, walls were defined 1.5m away from the furthest-out shelf.

The simulation environment would hopefully induce a situation where the number of robots would saturate the environment. This would create a case for a more intelligent multi-robot planner, as well as quantitatively show the performance advantages that a multi-robot planner would offer in comparison to a single-robot planner. A 3x5 environment would hopefully be small enough that a small number (less than 15) of robots would be able to induce such behaviour. A small number of robots would be convenient to simulate with, especially with the author's consumer i7 CPU.

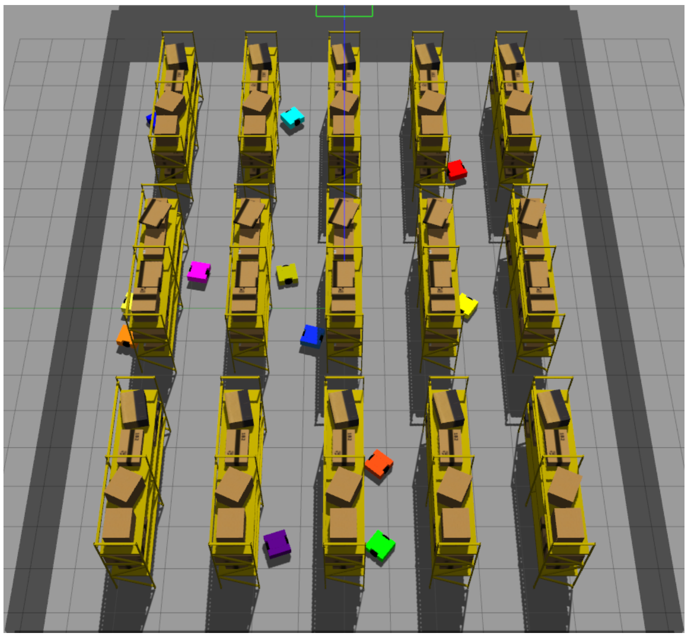

Figure 3.1: Screenshot of simulation environment with many robots spawned in random poses

To allow for programmatic description of the world, `xacro` (XML macro) was used to define the simulation environment. `xacro` allows for *macros*, like reusable functions in other programming languages. This allowed the world to be defined in a parametric way, so the dimension of the grid, as well as the spacing between shelves can be modified conveniently.

3.1.1 Robot Definition

Like the simulation environment, robots are defined as SDF files. A simple box shape was used, and the robots were defined as 50x50x20cm in dimension, which are similar dimensions to Amazon Pegasus warehouse robots at 75x60x16cm [25]. Figure 3.2 shows the simulated robots as well as an Amazon robot for comparison.

An advantage of the XML structure of the robot description is that these descriptions can be modified for each robot instance being spawned in the environment, which allows robots to have different colors. This is illustrated in Figure 3.1.

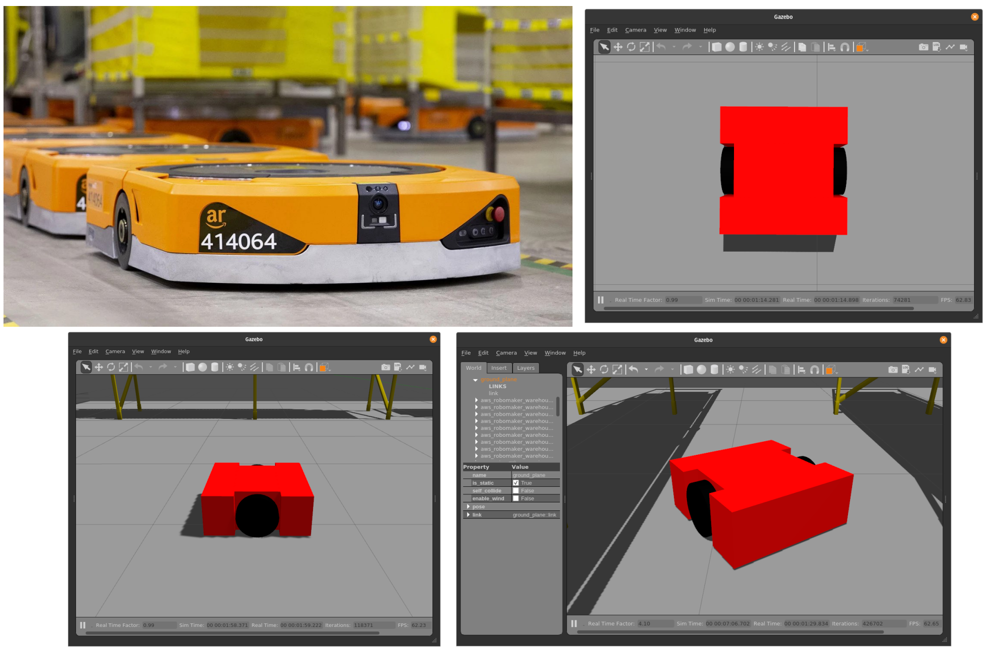
Figure 3.2: Clockwise from top left: Amazon Pegasus Warehouse Robot from [3], top, side and perspective views of robot model

3.1.2 Simulating multiple robots

To simulate a distributed system, the *namespace* functionality of ROS was used. This allows for multiple identical ROS nodes to be instantiated as unique objects. Nodes corresponding to different functionality on each simulated robot could then be isolated from each other.

Since a collaborative model was decided upon, the robots needed to be able to communicate with each other. To reduce computational load on each robot, and to simulate a distance-based communication scheme where robots only are able to communicate with other robots within a certain distance away, a global node called `odom_distribution` was created.

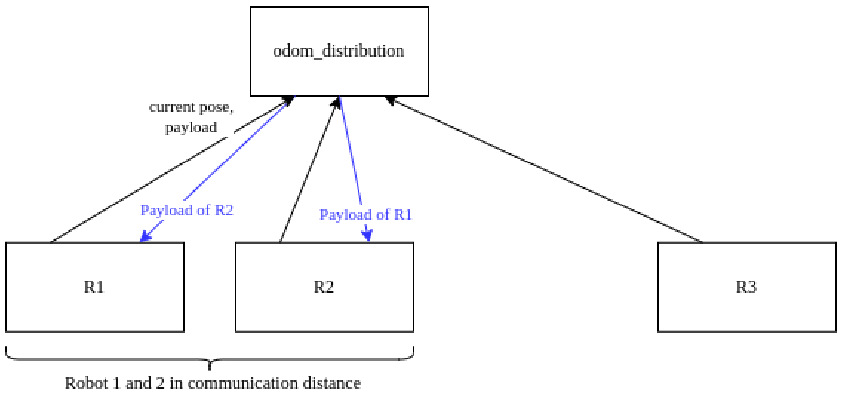

Figure 3.3: Illustration of `odom_distribution` node

All robots send their current locations as well as a communication payload to the node, which then computes the pairwise distances between each robot. If robots are within the set communication distance, then the node forwards the respective communications payloads to each robot. Computing the pairwise distances is implemented by filling up elements on a symmetric matrix.

For example, Figure 3.3 shows an example of the node. $R1$, $R2$ and $R3$ are robots who send their pose and payload to the node. Since R1 and R2 are close to each other, the node sends R2's payload to R1, and vice-versa. R3 is not close to either R1 nor R2, so it does not receive any payload.

3.1.3 Test Scenario Description

It was deemed necessary to come up with a framework that allowed for repeat testing of a specific scenario as well as randomized scenarios to be generated. Being able to repeatedly specify a scenario would help with troubleshooting and debugging during development, as well as for tuning parameters in algorithms and for benchmarking. Randomized scenarios would be useful to test the different algorithms developed in the project in a generalized case.

Scenarios are `YAML` files which minimally specify the number of robots in the simulation. The initial pose of each robot can be specified, as well as an array of goals for each robot. If these are not specified, then the values are randomly chosen from a list of valid locations.

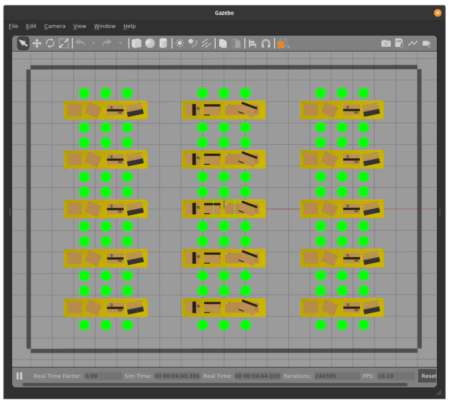

Figure 3.4: Illustration of possible goal locations

Figure 3.4 shows the locations in the environment as green circles. These locations are defined as close to either side of the shelves, as they presumably would be where a robot would stop to pick up items. In the case of randomized initial start locations for each robot, these locations would represent the robot just finishing off a previous goal and starting a new one at that location.

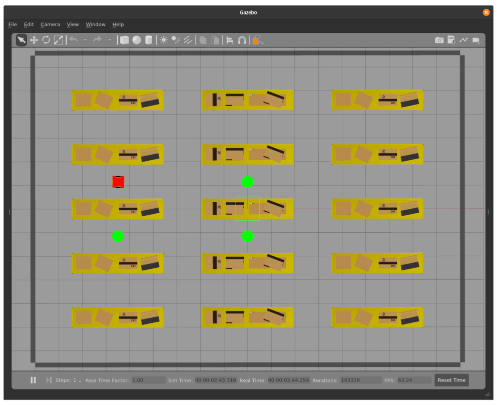

Figure 3.5: Single Robot Scenario

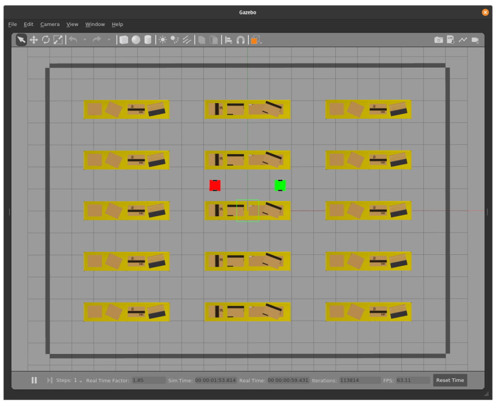

Figure 3.6: Robot Swap Scenario

Two specific scenarios were defined for repeatable testing. The "Single Robot" scenario shown in Figure 3.5 is meant to test the effectiveness of the single-robot trajectory planning algorithms before implementing them in a multi-robot setting. To this end, the robot in red needs to traverse goals in a clockwise fashion starting from the one on the top right till it returns to its original position. The "Robot Swap" scenario shown in Figure 3.6 tests the effectiveness of the multi-robot algorithms by needing the red and green robots to swap positions.

3.1.4 Goal Assignment

In addition to the global `odom_distribution` node described in 3.1.2, another global node was needed to simulate a global task/goal assignment system for each robot. This node was called `goal_creation` and handled the process of assigning goals to each robot, visualising these goals in the simulator, and collating the results of each simulation.

This `goal_creation` node would read the `YAML` scenario files described in 3.1.3, either going down the array of goals per robot or randomly generating goals. The node would exit when all robots finished all goals, or when a timeout was reached. It would then output a result file for further analysis.

3.2 Global Planner

While the RRT family of sampling-based algorithms was chosen as the global planner to be implemented for this project, the implementation of the global planner was separated from the communication with other nodes in ROS. This allows different implementations of global planner to be easily dropped into the system, helping with code modularity. For clarity in this section, the implementation of the global planner will be referred to as the *GP implementation*, while the communication with other nodes will be called the *GP node*.

GP implementations are instantiated as Python classes with minimally an `explore` method. When a new goal is sent to a robot, an instance of the GP implementation is constructed in the GP node, and its `explore` method is called. This returns a set of waypoints, which will then be fed to another node which implements the local planner algorithm.

In this project, RRT and RRT* GP implementations were developed. However, future projects may include other GP implementations by overloading the base GP implementation class provided in the source code.

3.2.1 RRT Implementation

The RRT algorithm was discussed earlier in Section 2.1.1. To reiterate, the algorithm can be broken down into sections:

1. Randomly sampling for points
2. Checking if a point is admissible
3. Connecting to the nearest neighbor
4. Checking if the connection to the neighbor is admissible
5. Returning a path

Randomly sampling for points

To get a new point, x and y coordinates within the bounds of the environment were randomly generated from a uniform distribution. In addition, a *bias* towards the goal was implemented. This biases exploration towards the goal by setting the new point with the coordinates of the goal instead of a randomly-sampled value. The probability of this happening is governed by a bias term.

After a point is chosen, it is then checked for admissibility. To do so, the point is checked for its distance from the obstacles in the environment. If the point does not collide with an obstacle, it is an admissible point and passed to the next stage of the RRT algorithm. If not, a new point is randomly sampled.

Collision Checking

Obstacles are modelled as Axis-Aligned Bounding Boxes (AABBs), rectangles whose sides are parallel the basis vectors. This is an accurate model, as the shelves are arranged in a regular grid, and the world can be oriented such that the axes line up with the orientation of this grid. More importantly, AABBs offer a variety of algorithms to check for collision. In this project, AABBs are defined by the locations of their opposite corners (x_0, y_0) and (x_1, y_1). They are also defined such that $x_0 < x_1$ and $y_0 < y_1$. The robot is modelled as a circular body, which is a convenient approximation because the orientation of the robot is not aligned with the world's basis vectors. A point is admissible if it does not collide with any obstacle.

To check if a randomly sampled point $\mathbf{C}$ collides with an AABB, the AABB is first *inflated* by an inflation radius r_i, defined as the sum of the radius of the robot r_r and the safety threshold r_s. The AABB convention means that $x_0' = x_0 - r_i$ and $x_1' = x_1 + r_i$, and equivalently for y_0 and y_1. The distance to the inflated AABB is then calculated using an algorithm modified from [26].

Firstly, the closest point $\mathbf{P}$ on the AABB to the point $\mathbf{C}$ is calculated. This is done by clamping the vector between the point and the center of the AABB, $\mathbf{D}$, to be within the half-extents of the AABB $\mathbf{E} = (\frac{x_1' - x_0'}{2}, \frac{y_1' - y_0'}{2})$. The `clamp` operation can be implemented as `max(min(`$\mathbf{D}$`, `$\mathbf{E}$`), `$-\mathbf{E}$`)`.

At this point, if $\mathbf{P}$ is within the AABB, $\mathbf{P} < \mathbf{E}$ for both x and y. This indicates a collision. Conveniently, the algorithm also provides a closest point to the edge of the AABB $\mathbf{P}'$, which can be used by other algorithms in the project.

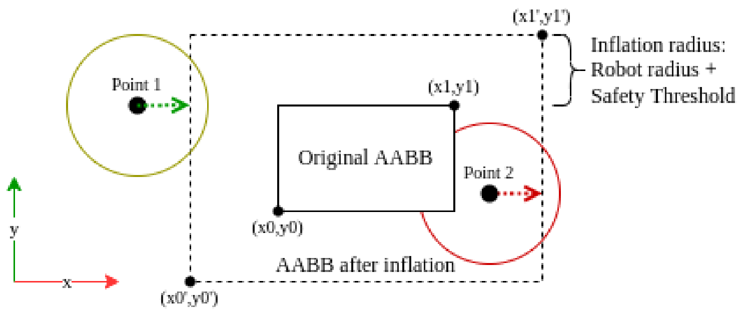

Figure 3.7: Distance to AABB algorithm

Figure 3.7 gives two examples. The original AABB is shown with the inflated AABB. Point 1 lies outside the inflated AABB, as the distance to the center of the AABB along the x axis is greater than the half-width of the inflated AABB. This corresponds to a robot at Point 1 with radius r_r not colliding with the original AABB. In contrast, Point 2 lies within the inflated AABB, with its distance to the center of the AABB being less than the AABB's half-width in the x axis and less than the half-height in the y axis. Correspondingly, a robot at Point 2 would collide with the original AABB. In both cases, the closest point on the outside of the inflated AABB is shown with a dotted arrow.

Checking if the connection to the nearest neighbor is admissible

The nearest neighbor is obtained by iterating through the list of expanded RRT nodes and returning the node with the smallest Euclidean distance.

Once the nearest neighbor coordinates $\mathbf{x}_{nn}$ are found, the point along the line connecting the randomly sampled point $\mathbf{x}_{new}$ within a given distance d_{max} away from $\mathbf{x}_{nn}$ can be found.

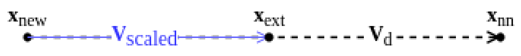

Figure 3.8: How an extension $\mathbf{x}_{ext}$ is found

First, the vector $\mathbf{V}_d$ from $\mathbf{x}_{nn}$ to $\mathbf{x}_{new}$ is found as $\mathbf{V}_d = \mathbf{x}_{nn} - \mathbf{x}_{new}$. The vector is then scaled such that it has a maximum length of d_{max}: $\mathbf{V}_{scaled} = \mathbf{V}_d \times max(\frac{d_{max}}{|\mathbf{V}_d|}, 1)$. Finally the point $\mathbf{x}_{ext}$ is found as $\mathbf{x}_{ext} = \mathbf{x}_{new} + \mathbf{V}_{scaled}$. Figure 3.8 shows this process.

Subsequently, the line connecting x_{new} and x_{ext} is then checked for collision with any obstacle. This is done by checking for an intersection between the lines that make up the obstacle's corresponding AABB. [27] describes a method to find the intersection between two line segments. Firstly, both lines are parameterized as $L_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$, $L_2 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} + u \begin{bmatrix} x_4 \\ y_4 \end{bmatrix}$.

The intersection point can be found with the following values of t or u. If both t and u are within the range $[0, 1]$, there exists an intersection between both line segments.

$$t = \frac{\begin{vmatrix} x_1 - x_3 & x_3 - x_4 \\ y_1 - y_3 & y_3 - y_4 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

$$u = \frac{\begin{vmatrix} x_1 - x_3 & x_1 - x_2 \\ y_1 - y_3 & y_1 - y_2 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = \frac{(x_1 - x_3)(y_1 - y_2) - (y_1 - y_3)(x_1 - x_2)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

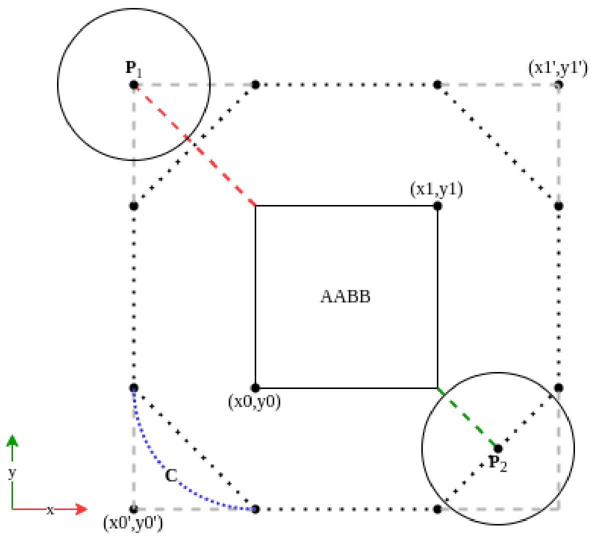

Figure 3.9: Why simple inflation with a square is suboptimal

The AABBs must first be inflated to take the robot's radius into account. However, simply inflating an AABB by adding and subtracting to its corner coordinates like in 3.2.1 is not optimal, as shown in Figure 3.9. As the inflation radius r_i incorporates both the robot radius r_r and a safety radius r_s, at the corners of the inflated AABB (x_0', y_0'), (x_1', y_1') the distance between a robot and the obstacle is overly conservative, as can be seen at point $\mathbf{P_1}$.

Optimally, collision could be checked with a curve of radius r_i centered at the corners of the original AABB and with end-points tangent to the inflated AABB, like the blue curve $\mathbf{C}$ in the bottom-right of the figure. However, it was deemed sufficient to connect the ends of the inflated AABB together in an octagon shape as shown with the dotted lines in the figure. For instance, instead of a single edge at (x_1', y_1'), there are now two edges, one at (x_1, y_1') and one at (x_1', y_1). In doing so, the line from P_s to P_{ext} would be checked for an intersection with 8 line segments instead of 4.

This leads to a more realistic estimate of whether a collision will happen at the corners of an AABB, as seen with $\mathbf{P}_2$. However, care should be taken not to set the safety radius r_s too low with this method, as the effective safety radius at the corners is less than usual due to the approximation.

Returning a path

RRT nodes were implemented as Python objects with *position* and *parent* as attributes. Only the start node has parent attribute set to `None`. To get a path from start to goal node, nodes were traversed from goal node to start node and their positions were appended to a list. After reaching the start node, the list was reversed to be in the right order, giving a list of waypoints for the local planner to follow.

3.2.2 RRT* Implementation

As detailed in Algorithm 2, RRT* improves on RRT by allowing for paths to change if a newly-discovered node results in a shorter path. This process has three parts, which are detailed below. RRT* nodes also have a *cost* attribute, which keeps track of the cost of moving from the start node to the current node. In this project, Manhattan distance was used as the cost function.

Searching for nearby nodes

Firstly, the set of nodes N_{near} that are within a certain radius of a newly-created node n_{new} at point $\mathbf{x}_{ext}$ is computed. This is done by iterating through the node list N and appending all nodes that have Manhattan distance less than a distance threshold. To prevent the size of N_{near} from becoming too large as the density of nodes in the environment increases, this distance threshold r_{thresh} is normalized by a term dependent on the total number of expanded nodes $n(N)$. This is given by $r_{thresh} = \sqrt{\frac{\ln n(N)}{n(N)}}$.

The initial sparsity of nodes in the environment means that there are occasionally no nodes in N_{near}. To prevent this scenario, the distance threshold r_{thresh} is increased by 10% each time $n(N_{near}) = 0$.

Choosing the best parent for n_{new}

n_{new} is then connected to the node in N_{near} with the lowest cost. To choose this best parent for n_{new}, the nodes in N_{near} are iterated over. Firstly, the line connecting n_{new} and the proposed parent n_{prop} is checked if it intersects with any obstacles as done earlier in 3.2.1. If the connection is admissible, the cost of n_{prop} is added to the Manhattan distance between the positions of n_{new} and n_{prop} to give the proposed cost of the extension. The n_{prop} with the lowest proposed cost is then chosen as the parent of n_{new}.

Rewiring nodes and propagating costs

Subsequently, nodes in N_{near} are iterated over and checked if rewiring them to n_{new} would reduce their overall cost. Similar to the previous section, the line connecting n_{new} to a node in N_{near} is first checked for admissibility and then for a reduction in its cost.

If a node n_{child} in N_{near} is rewired with n_{new} as its parent, then the cost of the children of n_{child} (and those nodes' children) are also updated recursively. This is done by searching N for nodes with n_{child} as the parent and updating the cost of such nodes as the cost of n_{child} plus the Manhattan distance of n_{child} to its child.

Termination of RRT*

RRT* may not terminate immediately when a solution is found, as it will return paths with lower cost as the number of nodes expanded increases. Therefore, a lower limit to the number of nodes expanded was added, to increase the chances of a more optimal path being found. If a path to the goal has already been found before the minimum number of nodes have been explored, the path *bias* to the goal node mentioned in 3.2.1 is disabled to encourage exploration of the environment.

3.2.3 Experimental comparison between RRT and RRT*

The RRT and RRT* implementations were tested with driver code written in Python and visualized using Matplotlib. The two methods were compared on the basis of execution time, number of nodes expanded, and the total cost to get to the goal.

During each test run, the methods were given the same randomly-selected start and goal node from the list of valid goals detailed in Figure 3.4. To take into account the random nature of both algorithms, each test run was repeated 10 times, with the average of each run taken. 20 test runs were used in total. Furthermore, to see how increasing the minimum number of nodes in RRT* impacted the optimality of the generated path, minimum values of 50, 100, 200 and 400 nodes were tested. All other values were kept the same.

Algorithm Type	Execution Time (%)	Nodes Expanded	Average Cost (%)
RRT	100.0	68.6	100.0
RRT* (50)	287.0	72.9	89.8
RRT* (100)	451.7	108.7	85.3
RRT* (200)	1155.6	204.5	80.3
RRT* (400)	3248.7	403.0	91.69

Table 3.1: Results of tests on RRT family of algorithms

Table 3.1 shows the result of the experiment. The execution time and path cost of the RRT* algorithms are presented relative to the base RRT node to normalize for the random selection of start and goal locations between test runs. The number in brackets in the "Algorithm Type" column indicates the minimum number of nodes for the RRT* algorithm.

As expected, RRT* takes longer to execute than RRT, but delivers a shorter path. For RRT* from 50 to 200 minimum nodes, the average path cost decreases from 89.9% to 80.3% of the path cost from RRT. This is at the expense of computation time. It is curious to note that the average cost of RRT* with 400 nodes bucks the trend with a higher average cost than with 50 nodes.

<TODO discussion of these results?>

<TODO explain how these values were eventually used elsewhere>

3.3 Local Planner

As covered in 2.2.3, the Dynamic Window Approach (DWA), detailed in 2.1.2, was chosen as a base to build upon owing to its simplicity and robustness of implementation compared to other approaches. Several variants of the DWA method were implemented. Similar to the modular approach used with GP implementations detailed in 3.2, these DWA variants inherited from a base DWA implementation to avoid repeating code.

At its core, the DWA server enumerates possible control inputs, does forward simulation of these control inputs for a simulation duration to obtain an eventual pose, and scores these poses. The best control input is then applied for a control duration, which is shorter than the simulation duration. When the robot is within a distance threshold d_{thresh} to the current waypoint, the DWA server finishes operation.

3.3.1 Projecting Future Poses

Enumerating Control Inputs

Firstly, the set of feasible control inputs needs to be enumerated. In this project, the dynamics of the robot were not considered. In addition, a differential drive controller package that came with Gazebo [28] that mapped linear and angular velocities to control inputs of each wheel was used for ease of interfacing with the simulated robot.

Dynamically feasible control inputs were approximated by considering the two adjacent values of linear and angular velocities to the current linear and angular velocity. For example, if the current angular velocity $v_{ang} = 0.2\text{rad/s}$, the spacing for velocities $v_{spacing} = 0.1$, and the angular velocity is bounded within the range $[-1.0, 1.0]$, then the three angular velocities $0.0, 0.1, 0.2, 0.3, 0.5\text{rad/s}$ will be considered. If a proposed velocity would exceed the bounds, it is not considered. Thus if $v_{ang} = 1.0$, then only velocities $0.8, 0.9, 1.0$ would be considered. The same process is performed for the linear velocities v_{lin}. In this way, up to 25 control inputs are considered by the planner.

Predicting Future Positions

For simplicity, the planner assumes that the velocities commanded are instantly reached and held constant for the simulation duration t_{sim}, an approximation which works satisfactorily in practice.

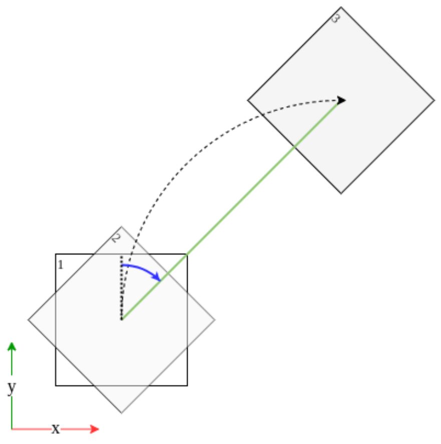

Figure 3.10: Illustration of forward projection of position

Figure 3.10 shows how an arc (black arrow) can be broken down into a rotation (blue arrow) and a translation (green arrow). The linear and angular displacements resulting from the applied linear and angular velocities can be resolved separately as rotations and translations respectively.

For a given initial pose (x, y, θ) and control inputs v_{ang}, v_{lin}, a final pose (x', y', θ') can be calculated. Firstly, the angular displacement $\phi = v_{ang} \times t_{sim}$ is resolved to obtain $\theta' = \theta + \phi$. Subsequently, a 2D rotation matrix $\mathbf{R}$ is used to simplify the effects of translation in the direction of θ'.

$$\mathbf{R} = \begin{bmatrix} \cos(\theta') & -\sin(\theta') \\ \sin(\theta') & \cos(\theta') \end{bmatrix}$$

$\mathbf{R}$ can then be multiplied with a vector with the linear displacement $d_{lin} = v_{lin} \times t_{sim}$ to obtain the displacement (d_x, d_y) of the original point:

$$\begin{bmatrix} d_x \\ d_y \end{bmatrix} = \mathbf{R} \begin{bmatrix} d_{lin} \\ 0 \end{bmatrix}$$

The displacements then be added to the original robot coordinates (x, y) to obtain (x', y'). Thus, the eventual pose of the robot (x', y', θ') is calculated.

3.3.2 Ranking Future Poses

The set of estimated future poses must now be scored. Four sets of scoring criteria were devised. The sum of the scores for each criterion is then used as the overall score for that set of control inputs.

Goal Proximity

A two-part scoring function was used. Here, K_{goal} is a tuneable constant.

$$s_{goal} = \begin{cases} \frac{K_{goal}}{d_{goal}} & \text{if } d_{goal} \leq 1 \\ K_{goal} * d_{goal} * m_{goal} + c_{goal} & \text{otherwise} \end{cases}$$

When the Manhattan distance of the predicted coordinates to the current waypoint d_{goal} is less than 1, the inverse of d_{goal} was used. The inverse function is chosen because coordinates closer to the waypoint will be scored much higher than if a linear function was used.

On the other hand, when d_{goal} is greater than 1, a linear function with gradient m_{goal} and y-intercept c_{goal} that is continuous with the inverse function is used. This is because further away from 1, the inverse function drops off very slowly without going to zero. This is undesirable as moves further away from the waypoint will not be penalized as much, which would hinder the pathfinding of robots towards far-away waypoints.

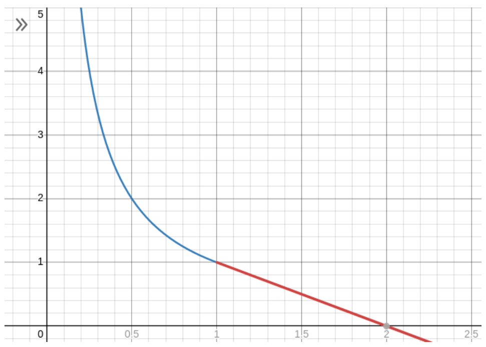

Figure 3.11: Diagram showing the two-part scoring function

Figure 3.11 shows the two-part scoring function. The inverse function for $d_{goal} < 1$ is shown by the blue line, and the linear function for the other case is shown in red. In this case, m_{goal} and c_{goal} have been set such that s_{goal} is 0 at $d_{goal} = 2$, although this is also tuneable.

Goal Orientation

The difference between the heading towards the waypoint and the robot's actual heading is penalized as a_{hdg}. The scoring function is a linear function

$$a_{hdg} = m_{hdg} * d_\theta + a_{hdg}$$

d_θ is clamped between tuneable maximum and minimum values, $d_{\theta max}$ and $d_{\theta min}$. Particularly, adjusting the minimum values allow the responsiveness of the DWA planner to the angle difference to be changed. For instance, setting $d_{\theta min}$ to $20°$ will mean that any angle error from 0 to $20°$ will incur no penalty, thus meaning the robot is less likely to fixate on having exactly the right heading.

s_{goal} is still the most important metric as it guides the robot towards the waypoint. Therefore, a_{hdg} is implemented as a percentage cost to s_{goal}. If it were implemented as a flat cost, it might be overwhelmed by the large values of s_{goal} close to the waypoint, or in turn overwhelm s_{goal} when the robot is far away from the waypoint. Hence, after a_{hdg} is calculated, the score is $s_{goal} * (1 - a_{hdg})$.

Obstacle Proximity

The distance to each obstacle is calculated using the distance of (x', y') to an un-inflated AABB representing the obstacle, as described in 3.2.1. If the predicted position is less than a robot radius away from an AABB, then a collision is assumed, and the proposed control input is marked as invalid with a score of $-\infty$.

If the distance is between a robot radius r_r and the inflation radius $r_i = r_r + r_s$ away, then a linear function $a_{obs} = m_{obs} * d_{obs} + c_{obs}$ is used to obtain a penalty a_{obs} for that particular obstacle.

Proximity to other robots

Basic collision avoidance is implemented by allowing robots to send their predicted positions to other robots in the vicinity through the `odom_distribution` node detailed in 3.1.2. These positions are then treated similarly to obstacles, except that the inter-robot distance d_{ir} between the robot's predicted position (x', y') and another robot's predicted position can be used. If the distance is less than $2 * r_r$ then a collision is assumed, and the proposed control input is marked as invalid with a score of $-\infty$. Else, an inverse function $a_{ir} = K_{ir}/d_{ir}$ is used.

Movement Speed

To encourage movement, a cost equal to $(\frac{K_{goal}}{d_{thresh}})/4$ is applied when both v_{lin} and $v_{ang} = 0$. This value is chosen such that when the robot is far away from the waypoint, it is penalized for stopping. However, when the robot is close to the waypoint, stopping may be the best option if all other options cause overshoot. Thus, a fraction of the maximum expected score $(\frac{K_{goal}}{d_{thresh}})$ is used.

3.3.3 Avoiding Local Minima

Despite the methods used to penalize the robot stopping, the robot was still found to stall in progress on occasion as the DWA planner falls into local minima. To avoid this, a stall detection algorithm was implemented.

Stall Detection

Stall detection was implemented by tracking the translation in (x, y) of the robot within a fixed time frame t_{stall}. If the robot does not translate by d_{stall} every t_{stall} seconds, it is considered to have stalled, and actions can be taken to take the robot out of a local minimum.

Goal Locations

A local minimum is reached when the chosen control input is continually a small value, despite the robot being far from a waypoint. To break out of a local minimum, the set of considered control inputs should be forcibly changed to move towards a given goal location.

However, the goal location may not always be directly towards the current waypoint. Blindly moving towards the current waypoint may steer the robot into an obstacle if the robot was already close to an obstacle.

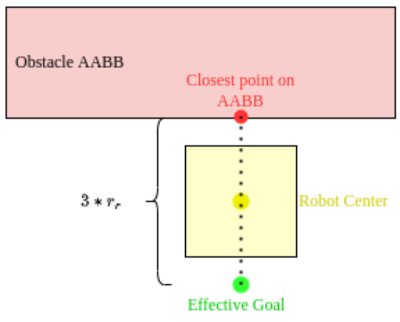

Figure 3.12: Alternative goals near obstacle

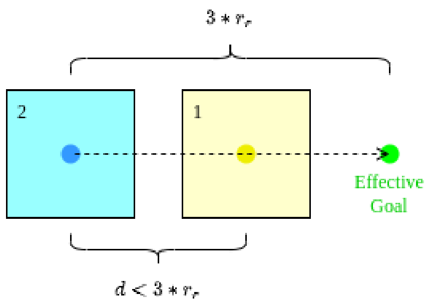

Figure 3.13: Alternative goals near other robots

Figure 3.12 shows what happens if the robot is too close to an obstacle. As the closest point on an AABB is used in getting the distance from an AABB, it is used as an anchor point, shown in red. The robot's center is used as another anchor point, shown in yellow. The point on a the line connecting the two anchor points with distance $3 * r_r$ away from the AABB is used as an alternative goal to move the robot further away from the obstacle to a hopefully closer point.

A similar process is done if another robot is in close proximity. Figure 3.13 shows this. Robot 1, in yellow, is too close to Robot 2, in blue. The (x, y) coordinates of both robots are used as anchor points for a connecting line, and the goal position for Robot 1 is set $3 * r_r$ away from the position of Robot 2, shown in green.

To get a point on a connecting line a specific distance away, the difference vector $\mathbf{v}$ from the start coordinate $\mathbf{x}_s$ to the end coordinate $\mathbf{x}_e$ is computed as $\mathbf{v} = \mathbf{x}_e - \mathbf{x}_s$. It is then scaled by the desired distance d: $\mathbf{v}_{scaled} = \mathbf{v} * \frac{d}{|\mathbf{v}|}$. Finally, the scaled difference vector is added to the start coordinate to get the coordinates a specific distance away: $\mathbf{x}_d = \mathbf{x}_s + \mathbf{v}_{scaled}$.

If there were several alternative goals, for instance if a robot was near both an obstacle and another robot, the mean of all goals was taken so that the robot would move into a space away from both of them.

Translating to linear and angular velocities

Given a goal position, the next task would be to give linear and angular velocities to the controller.

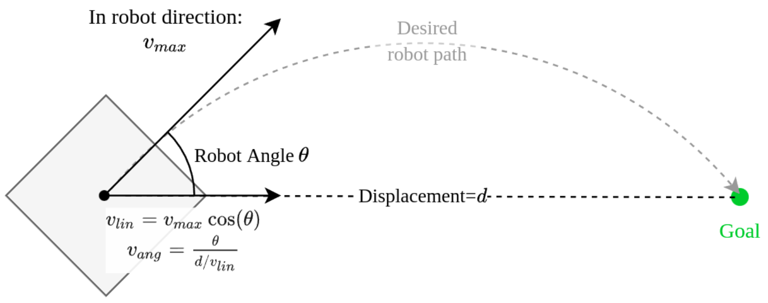
In robot direction: v_{max}
Desired robot path
Robot Angle θ
Displacement=d
Goal
$v_{lin} = v_{max}\cos(\theta)$
$v_{ang} = \frac{\theta}{d/v_{lin}}$

Figure 3.14: Differential Drive Kinematics to get to Goal

An example of this process is shown in Figure 3.14. The robot needs to move to the goal point in green. The process for a differential drive robot would be to follow the path in gray. To do so, it would need to apply a constant linear and angular velocity over some time period t. In 3.3.1 it was seen that such a motion can be broken down to a rotation and a translation.

The required linear velocity v_{lin} is first calculated by taking a tuneable parameter v_{max} scaled by the cosine of the heading towards the goal θ. The time period required for a robot to translate a distance d would then be $t = d/v_{lin}$. For the robot to follow the desired path, the angular velocity v_{ang} would thus be θ/t. Finally, the calculated values of v_{lin} and v_{ang} are then discretized into steps as mentioned in 3.3.1.

3.3.4 Parameter Tuning and Algorithm Evaluation

The DWA planner has many tuneable parameters. In order to find the best values for each of them, a grid search over different possible parameters was used. The test infrastructure consisted of an overall bash script which calls Python files that generate test scenarios and invoke the commands to launch the Gazebo simulator.

Tuning for the single-robot case

Firstly, the robot's performance would be tested using the single-robot scenario illustrated in Figure 3.5. This tests the DWA planner's ability to keep to a trajectory given by the RRT* planner. There has been some work on test metrics for mobile robot local planners in [29]. Inspiration was taken from this to come up with the metrics of interest.

In the single robot test scenario, the robot has to complete a set of 4 goals in a rectangle. The main test metric for this was *moving time*, or the duration the robot took to move to complete all four goals. This decouples the time of execution away from the time taken to find a path through the RRT* planner.

<TODO present values here>

<TODO present results here>

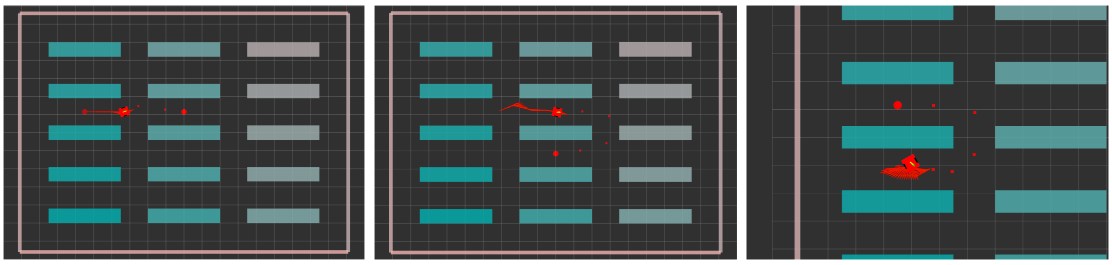

Figure 3.15: Robot moving through single-robot case, left to right

Figure 3.15 shows the robot moving through the single-robot scenario. The visualization is from a visualization tool for ROS, *RViz*, which shows the planned waypoints of the robot as well as the shelves in the environment. As can be seen from left to right, the images show how the robot moves through the goal sequence defined.

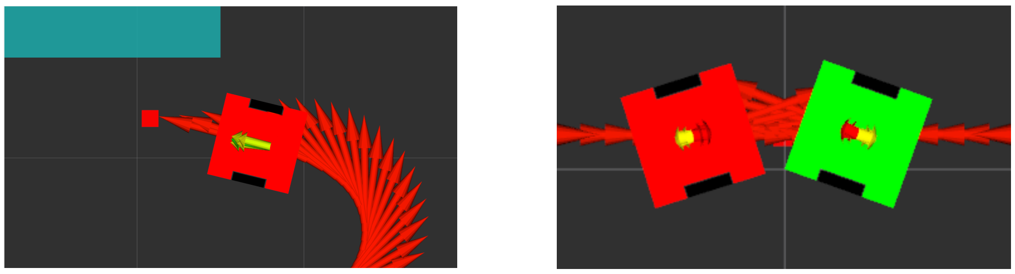

Figure 3.16: Robot control input selection

Figure 3.16 shows the DWA planner selecting control inputs. The possible end-points are shown as arrows centered on the center of the robot, with the highest-scoring trajectory in green. Inadmissible trajectories are coloured red, and all other arrows are yellow. In the picture on the left, the highest-scoring trajectory is the one pointing directly towards the waypoint. The picture on the right illustrates how robots avoid each other by choosing trajectories that would otherwise cause a collision.

Testing for the multi-robot case

To evaluate whether the simple DWA+RRT* planner combination was sufficient, the robot swap case shown in Figure 3.6 was carried out.

Despite having some notion of the position of nearby robots, the standard DWA planner is not sophisticated to give rise to the behaviour where both robots move around each other within the narrow corridor. Instead, the robots get stuck facing each other. Figure 3.17 illustrates this behaviour. This is a motivating case to extend the DWA planner to more intelligently plan around other robots, which will be elaborated upon in the following section.

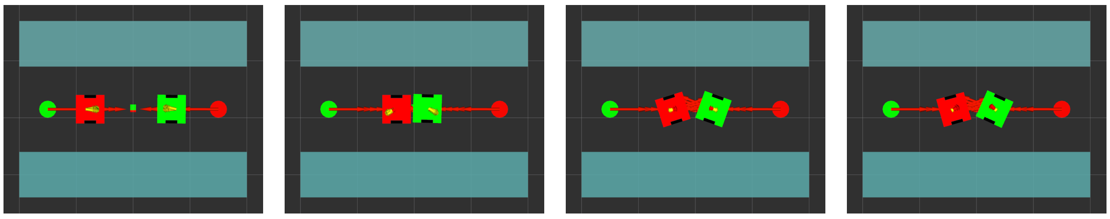
Figure 3.17: Regular DWA planner failing the swap scenario

Testing in the general case

In addition, to evaluate whether the simple DWA+RRT* planner combination in a generalized case, the test infrastructure was also extended to vary the number of robots in the simulation. The specific number of robots in the simulation can be varied, and each robot spawns on one of the goal locations with a random orientation. Each robot has 100 goals and the simulation runs for 120 seconds. It is not possible for the robots to finish 100 goals in 120s, so the actual measurement is the total number of goals completed within a 120s timeframe.

Several such scenarios were created, akin to the testing process for the RRT implementations in 3.2.3. Each scenario was repeated for each permutation of parameters. This meant that each permutation could be compared against the other permutations in a fair manner, as the starting pose and goal locations were the same. The tests would hopefully also be generalizable due to repeated testing on different random scenarios.

The test metrics used were:

1. Average number of goals completed per robot per run
2. Total number of goals completed per run
3. Average distance travelled per robot per run
4. Average time spent planning per robot per run
5. Average time spent moving per robot per run

The average number of goals completed per robot per run (1) indicates the per-robot efficiency of the planning algorithm. It is expected to drop as the number of robots in the environment increases. This can be compared to the total number of goals completed per run (2), which measures the overall efficiency of the planning algorithm. Furthermore, comparing the total number of goals completed as the number of robots increases could be a measure of how much productivity can be extracted from any particular algorithm.

The average distance travelled per robot per run (3) as well as the average moving time per robot per run (5) are measures of the efficiency of the algorithm. All things equal, an algorithm with a lower moving time and shorter travelling distance is better. In addition, these two metrics can be combined to give a measure of average robot speed.

Lastly, average planning time per robot per run (4) measures the time spent by each robot on the global planner. A lower value is better.

3.3.5 Dynamic Window Approach with Replanning

To allow robots to move around each other, a simple method was to allow robots to consider other robots as static obstacles and replan around each other. This would mitigate the case where robots block each other and cause deadlock as in 3.17. This variant of DWA is called DWA-Replan, or DWA-R for short.

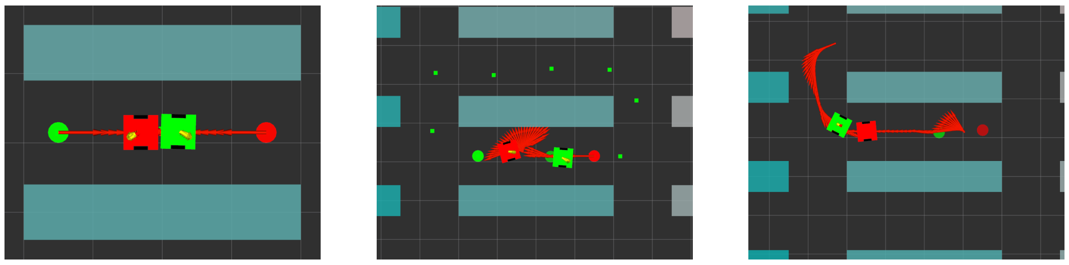

Figure 3.18: Robot showing replanning behaviour

Figure 3.18 shows this behaviour in the robot swap scenario. The green robot yields to the red robot and replans a path around it, eventually reaching its goal on the other side. In order to do so, the DWA-R server must be able to induce the global planner to perform a replan, and also pass in the location of another robot in its vicinity as an additional obstacle to plan around. To this end, ROS *services* were used, which allow nodes to make queries and requests to other nodes.

Replanning Criteria

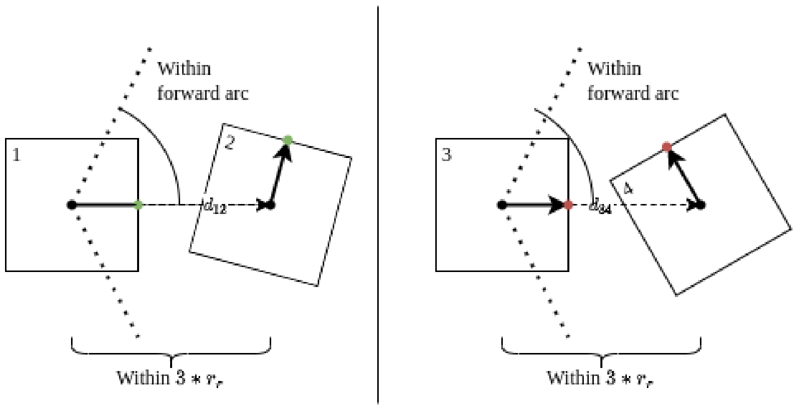

Figure 3.19: DWA Replan Condition

Figure 3.19 illustrates the conditions required for a robot to check if it needs to replan. The DWA-R server periodically checks if another robot is within 3 robot radii in distance and within 70° of the forward arc of the current robot, as illustrated with Robots 1,2 and 3,4. In addition, the orientation of the robots is checked. While Robots 1 and 2 are close to each other and Robot 2 is within Robot 1's forward arc, Robot 2 is heading away from Robot 1, and there is no need for Robot 1 to replan, as deadlock will not occur.

On the other hand, Robots 3 and 4 may collide, as Robot 4 is heading in Robot 3's direction. Robot 3 should therefore consider replanning. This behaviour is implemented by calculating the Manhattan distance of the point on the outline of the robot in the direction it is facing, illustrated

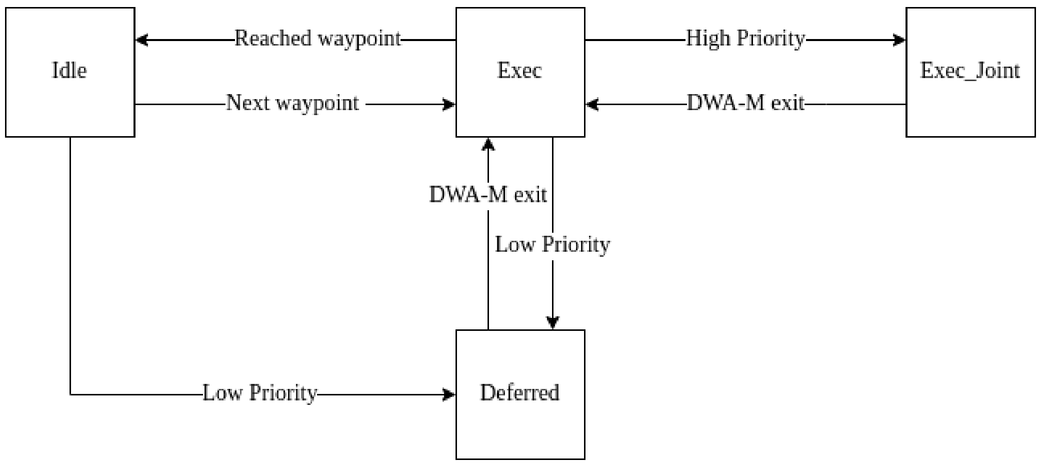

Figure 3.20: DWA-M State Machine

by the green points for Robots 1 and 2 and the red points for Robots 3 and 4. The location of these points $\mathbf{p}_{outer} = (x_{outer}, y_{outer})$ can be calculated as:

$$x_{outer} = x_{robot} + \cos(\theta)$$
$$y_{outer} = y_{robot} + \sin(\theta)$$

If the distance between $\mathbf{p}_{outer}$ and the (x, y) location of the robot is less than the distance between the robot centers, then a replan is considered.

There is the possibility that both robots are facing each other and therefore both meet the re-planning condition. To perform a tie-break, each robot queries both its own and the other robot's global planner. The robot with the shorter distance left to travel to its overall goal is given priority. In this case, the sum of Manhattan distances between waypoints along the path is given.

For example, if Robot 1 and Robot 2 are facing each other, Robot 1 will query both its own global planner and Robot 2's global planner to obtain the remaining distances to each robot's respective goals. Robot 2 will do the same. If Robot 1 still has 10m to travel while Robot 2 has 8m to travel, then Robot 1 will send a replan request to its global planner, while Robot 2 will wait until Robot 1 moves out of the way. By giving priority to robots closer to their goals, it is hoped that the overall number of completed goals will be increased.

<TODO Show how this helps with the situation>

3.3.6 Collaborative Dynamic Window Approach

While the DWA-R implementation is sufficient for multiple robots to avoid deadlock, there is still room for improvement. For instance, when robots are in between shelves, there is enough space for both robots to move around each other. This would mean that neither robot needs to replan and take a long path around the shelves, greatly improving efficiency.

This approach would be called DWA-Multirobot, or DWA-M. Similar to DWA-R, the DWA-M inherits from the base DWA class and extends its operation.

State Machine

The basic idea of the DWA-M planner is to allow for a form of local centralized planning. When a pair of robots are going to collide, one of them will take over planning and find the joint best trajectory for both robots. Such a centralized planner might be able to find trajectories for both robots that might not be the most optimal for either in the base DWA planner, but overall help both robots to avoid each other.

To enable this behaviour, a state machine was implemented, shown in Figure 3.20.

<TODO explain diagram>

Prerequisites

<TODO Waypoint skip> <TODO Waypoint replan>

Jointly Ranking Future Poses

<TODO>

Chapter 4 Evaluation

TODO talk about the metrics of comparison

4.1 Global Planner Evaluation

TODO talk about the limitations of the RRT planner, how A* could be better in some cases

4.2 Local Planner Evaluation

TODO elaborate on these points

- DWA planner implemented is akin to RVO method, albeit less complicated
- DWA planner performance is quite sensitive to parameter values
- unable to make come up with a suitable cost function for DWA-M in time so it was left out

4.3 Overall Results

TODO gather more results up to 16 robots, run more iterations (currently 8/9 robots was only on one iteration)

TODO explain results

Limitations in testing:

- RRT plan time does not run in simulation time
- Simulation was unstable when run with more than 10 robots on author's CPU, RRT takes a very long time to replan

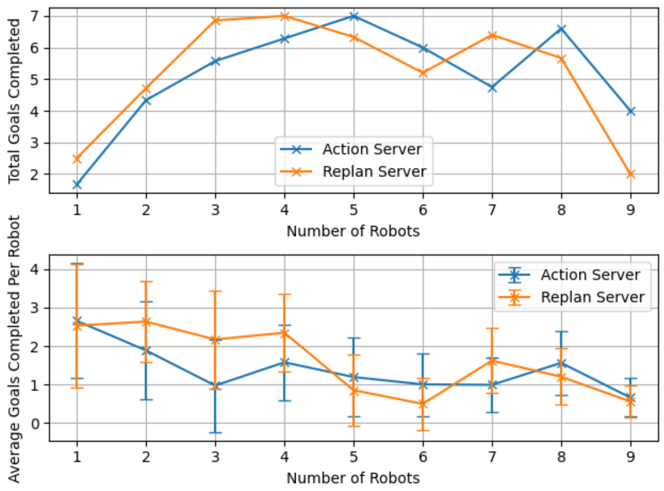

Figure 4.1: Average goals per robot per run against the local planner variant with increasing robots

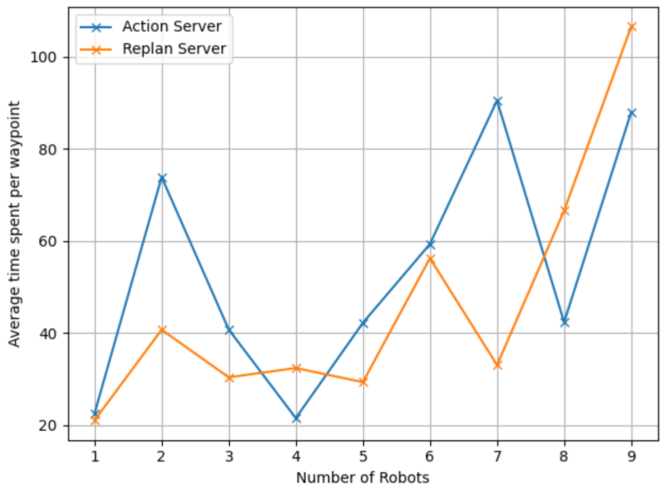

Figure 4.2: Results showing average time taken per waypoint per robot against the local planner variant with increasing robots

Chapter 5 Conclusion

Perhaps can use lifelong MAPF planners like [30]

Perhaps implement Predictive DWA controllers like in [31]

Chapter A First Appendix

Bibliography

[1] "Robots Will Be Working in 50,000 Warehouses by 2025, Report Says," Mar. 2019. [Online]. Available: https://www.roboticsbusinessreview.com/supply-chain/robots-will-be-working-in-50000-warehouses-by-2025-report-says/

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[3] "The story behind Amazon's next generation robot," Mar. 2019. [Online]. Available: https://www.aboutamazon.com/news/innovation-at-amazon/the-story-behind-amazons-next-generation-robot

[4] J. K. Verma and V. Ranga, "Multi-robot coordination analysis, taxonomy, challenges and future scope," *Journal of Intelligent & Robotic Systems*, vol. 102, no. 1, 2021.

[5] I. B. Daily, "Covid-19 Crisis Could Be Time For Warehouse Robots To Shine," Apr. 2020. [Online]. Available: https://www.investors.com/news/technology/industrial-automation-opportunity-seen-coronavirus-crisis/

[6] "Logistics companies turning to robotics and automation as way out of coronavirus crisis," Aug. 2020. [Online]. Available: https://roboticsandautomationnews.com/2020/08/12/logistics-companies-turning-to-robotics-and-automation-as-way-out-of-coronavirus-crisis/35041/

[7] "Neobotix: Mobile Robot MP-400." [Online]. Available: https://www.neobotix-robots.com/products/mobile-robots/mobile-robot-mp-400

[8] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, 2004, pp. 2149–2154 vol.3.

[9] G. Genty, "Home." [Online]. Available: https://www.eurobot.org/

[10] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[11] S. Koenig and M. Likhachev, "D^* lite," *Aaai/iaai*, vol. 15, pp. 476–483, 2002.

[12] S. M. LaValle *et al.*, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[13] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *CoRR*, vol. abs/1105.1186, 2011. [Online]. Available: http://arxiv.org/abs/1105.1186

[14] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23–33, 1997.

[15] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram, "Trajectory modification considering dynamic constraints of autonomous robots," in *ROBOTIK 2012; 7th German Conference on Robotics.* VDE, 2012, pp. 1–6.

[16] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[17] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370214001386

[18] M. Phillips and M. Likhachev, "Sipp: Safe interval path planning for dynamic environments," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 5628–5635.

[19] A. Bose, "Multi-Agent path planning in Python." [Online]. Available: https://atb033.github.io/multi_agent_path_planning/

[20] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *The international journal of robotics research*, vol. 17, no. 7, pp. 760–772, 1998.

[21] J. Snape, J. Van Den Berg, S. J. Guy, and D. Manocha, "The hybrid reciprocal velocity obstacle," *IEEE Transactions on Robotics*, vol. 27, no. 4, pp. 696–706, 2011.

[22] M. Kamel, J. Alonso-Mora, R. Siegwart, and J. I. Nieto, "Nonlinear model predictive control for multi-micro aerial vehicle robust collision avoidance," *CoRR*, vol. abs/1703.01164, 2017. [Online]. Available: http://arxiv.org/abs/1703.01164

[23] J. Lunenburg, S. Coenen, G. Naus, M. van de Molengraft, and M. Steinbuch, "Motion planning for mobile robots: A method for the selection of a combination of motion-planning algorithms," *IEEE Robotics & Automation Magazine*, vol. 23, no. 4, pp. 107–117, 2016.

[24] "GitHub - aws-robotics/aws-robomaker-small-warehouse-world." [Online]. Available: https://github.com/aws-robotics/aws-robomaker-small-warehouse-world

[25] "ALL the Amazon Warehouse Robots: From Fulfillment to Last-Mile Delivery." [Online]. Available: https://www.agvnetwork.com/robots-amazon

[26] "LearnOpenGL - Collision detection." [Online]. Available: https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection

[27] D. Kirk, *Graphics Gems III*. Academic Press, Inc, 1992.

[28] "Gazebo ROS Plugins." [Online]. Available: https://github.com/ros-simulation/gazebo_ros_pkgs

[29] J. Wen, X. Zhang, Q. Bi, Z. Pan, Y. Feng, J. Yuan, and Y. Fang, "MRPB 1.0: A unified benchmark for the evaluation of mobile robot local planning approaches," *CoRR*, vol. abs/2011.00491, 2020. [Online]. Available: https://arxiv.org/abs/2011.00491

[30] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," 2017. [Online]. Available: https://arxiv.org/abs/1705.10868

[31] M. Missura and M. Bennewitz, "Predictive collision avoidance for the dynamic window approach," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 8620–8626.