



BENG INDIVIDUAL PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Distributed Multi-Robot Trajectory Planning in a Warehouse Environment

Author:
Tian Yi Lim

Supervisor:
Prof. Andrew Davison

June 21, 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 navigation system was implemented, making use of sampling-based global planners coupled with a local planner based on the Dynamic Window Approach (DWA). Extensions to the DWA planner are used to allow multiple robots to navigate around the environment with higher efficiency. In the test scenario created, this is shown to increase the throughput of a group of robots by up to 38.0%.

Acknowledgements

I would like to thank my supervisor Professor 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

1	Introduction	7
1.1	Objectives	7
1.2	Contributions	8
1.3	Project Evolution	8
2	Technical Background	9
2.1	Trajectory Planning	9
2.1.1	Global Planners	9
2.1.2	Local Planners	13
2.2	Multi-Robot Path Planning	15
2.2.1	Centralized Approaches	15
2.2.2	Distributed Approaches	17
2.2.3	Planner Selection Criteria	17
2.3	Development Environment	18
2.3.1	Robot Operating System	18
2.3.2	Gazebo Simulator	18
3	Implementation	20
3.1	Simulation Environment	20
3.1.1	Robot Definition	20
3.1.2	Simulating multiple robots	21
3.1.3	Test Scenario Description	22
3.1.4	Goal Assignment	23
3.1.5	Overall System Architecture	23
3.2	Global Planner	24
3.2.1	RRT Implementation	24
3.2.2	RRT* Implementation	27
3.2.3	Experimental comparison between RRT and RRT*	28
3.2.4	Tuneable Parameters	28
3.3	Local Planner	29
3.3.1	Projecting Future Poses	29
3.3.2	Ranking Future Poses	30
3.3.3	Avoiding Local Minima	31

3.3.4	Parameter Tuning and Algorithm Evaluation	33
3.3.5	Dynamic Window Approach with Replanning	35
3.3.6	Collaborative Dynamic Window Approach	37
4	Evaluation	41
4.1	Test Metrics	41
4.2	Test Results	41
4.2.1	Testing Limitations	43
4.3	Global Planner Evaluation	43
4.3.1	Future Work	44
4.4	Local Planner Evaluation	45
4.4.1	Future Work	45
5	Conclusion	47
5.1	Future Work	47
5.2	Ethical Discussion	47
A	Appendix	48
A.1	DWA parameters	48
A.2	RRT and RRT* parameters	49

List of Figures

1.1	Example of a automated warehouse [1] showing how shelves are aligned in linear corridors	7
1.2	Simulated warehouse environment in the Gazebo simulator as used in the project .	7
1.3	The Amazon Pegasus robot used as a real-world reference, from [2]	7
1.4	Simulated robot in the Gazebo simulator as used in the project	7
2.1	The simulation environment, first presented in Figure 1.2	10
2.2	An occupancy grid representation of the simulation environment	10
2.3	Steps of the RRT algorithm	11
2.4	The difference between RRT (left) and RRT* (right)	13
2.5	Diagram describing the DWA algorithm	14
2.6	Example CBS map	15
2.7	Example CBS configuration	15
2.8	Showing SIPP fail to find a path. Pictures from left to right	16
2.9	Showing CBS finding a path. Pictures from left to right	16
2.10	A diagram of the ROS system architecture from [3]	19
2.11	A screenshot of an empty simulation world loaded in Gazebo's GUI	19
3.1	Screenshot of simulation environment with many robots spawned in random poses	20
3.2	Clockwise from top left: Amazon Pegasus Warehouse Robot from [2], top, side and perspective views of robot model	21
3.3	Illustration of <code>odom_distribution</code> node	21
3.4	Illustration of possible goal locations	22
3.5	Single Robot Scenario	22
3.6	Robot Swap Scenario	22
3.7	Planner System Architecture	23
3.8	Distance to AABB algorithm	25
3.9	How an extension \mathbf{x}_{ext} is found	25
3.10	Why simple inflation with a square is suboptimal	26
3.11	Illustration of forward projection of position	29
3.12	Diagram showing the two-part scoring function	30
3.13	Alternative goals near obstacle	32
3.14	Alternative goals near other robots	32
3.15	Differential Drive Kinematics to get to Goal	33

3.16 Robot moving through single-robot case, left to right	33
3.17 Robot control input selection	34
3.18 Regular DWA planner failing the swap scenario	35
3.19 Robot showing replanning behaviour	35
3.20 DWA-R State Machine	36
3.21 DWA Replan Condition	36
3.22 DWA-M State Machine	37
3.23 Goal Arbitration Motivation	40
4.1 Total goals per run with increasing robots	42
4.2 Average number of goals per robot per run with increasing robots	42
4.3 Average time taken per waypoint per robot per run with increasing robots	43
4.4 Path choices possible due to the random nature of the RRT planner	43
4.5 Robots in gridlock	44
4.6 Path choice difference between grid-based and sampling-based planners	44
4.7 Robots unable to proceed due to reciprocity	45
4.8 Local Minima in DWA algorithm	46

List of Tables

3.1	Results of tests on RRT family of algorithms	28
3.2	Parameter search for tuning values	34
3.3	Results of parameter tuning grid search	34
4.1	Ablation study on initial command values for DWA planner	45

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

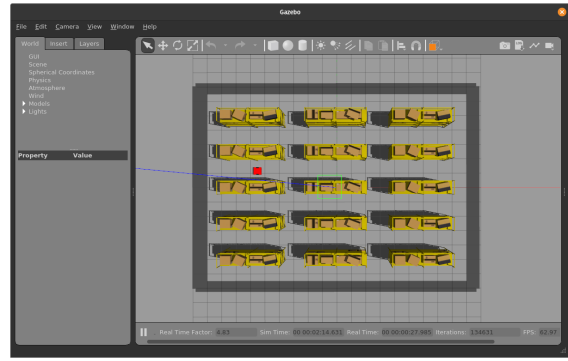


Figure 1.2: Simulated warehouse environment in the Gazebo simulator as used in the project

The simulated robots also follow similar dimensions to industrial logistics robots, for example the Amazon Pegasus robot [2]. The Pegasus robot has dimensions of 75x60x16cm [7], and the simulated robots have similar dimensions of 50x50x20cm.



Figure 1.3: The Amazon Pegasus robot used as a real-world reference, from [2]

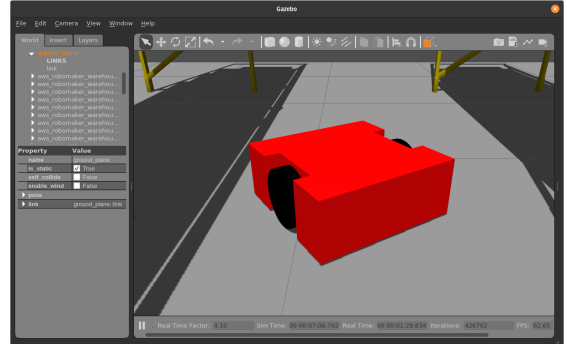


Figure 1.4: Simulated robot in the Gazebo simulator as used in the project

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) [3] 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, this was deemed infeasible midway through the project duration, owing to the lack of reliable hardware to compete. Further time constraints were faced as an EIE student, as the schedule for coursework and examinations was different from the Computing Department. The project therefore focuses on how multiple robots can plan paths for themselves in the presence of other robots in a warehouse 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 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

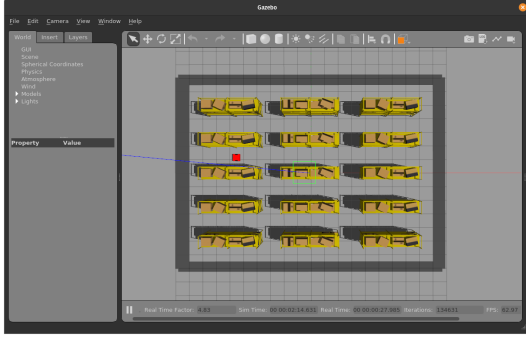


Figure 2.1: The simulation environment, first presented in Figure 1.2

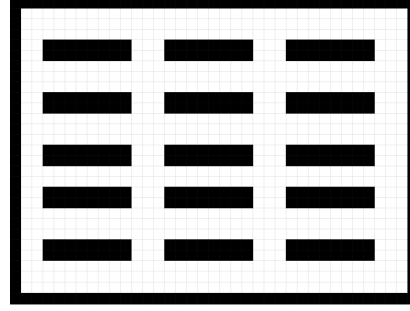


Figure 2.2: An occupancy grid representation of the simulation environment

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. On every 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, n_{near} , to \mathbf{x}_{new} in the node list N is obtained (`getNearestNeighbor`).

Each node n stores its current position $n.pos$ and its parent node, $n.parent$. A point \mathbf{x}_{ext} a set 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`). If a path cannot be found after

MAX_RRT_ITERATIONS, the algorithm terminates in failure.

Algorithm 1 The RRT algorithm

```

1: Initialize node list  $N = \{n_{start}\}$ 
2: for  $k=1$  to MAX_RRT_ITERATIONS do
3:    $x_{new} = \text{randomlySampleAdmissiblePoint}()$ 
4:    $n_{near} = \text{getNearestNeighbor}(N, x_{new})$ 
5:    $x_{ext} = \text{getPointOnExtensionLine}(x_{new}, n_{near}.pos)$ 
6:   if  $\text{lineIsCollisionFree}(x_{ext}, n_{near}.pos)$  then
7:      $n_{new} = \text{createNewNode}(x_{ext}, \text{parent} = n_{near})$ 
8:      $N.append(n_{new})$ 
9:     if  $\text{closeToGoal}(n_{near})$  then
10:      return  $\text{returnPathFromNodeList}(N)$ 
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.

RRT is not complete, as it may fail to find a solution even if one is available, if the solution has not been found by the time the iteration limit is reached. In addition, 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:    $x_{new} = \text{randomlySampleAdmissiblePoint}()$ 
4:    $n_{near} = \text{getNearestNeighbor}(N, x_{new})$ 
5:    $x_{ext} = \text{getPointOnExtensionLine}(x_{new}, n_{near}.pos)$ 
6:   if  $\text{lineIsCollisionFree}(x_{ext}, n_{near}.pos)$  then
7:      $N_{near} = \text{getNearNodes}(N, x_{ext})$ 
8:      $n_{bestParent} = \text{getBestParent}(N_{near}, x_{ext})$ 
9:      $n_{new} = \text{createNewNode}(x_{ext}, \text{parent} = n_{bestParent})$ 
10:     $N.append(n_{new})$ 
11:     $\text{rewire}(n_{new}, N_{near})$ 
12:    if  $\text{closeToGoal}(n_{near})$  then
13:       $\text{goalPathFound} = \text{True}$ 
14:    end if
15:  end if
16:  if  $k > \text{MIN\_RRT\_ITERATIONS}$  and  $\text{goalPathFound}$  then
17:    return  $\text{returnPathFromNodeList}(N)$ 
18:  end if
19: end for
20: return  $\text{False}$ , No path found

```

Algorithm 3 Implementation of the `rewire` function

```

1: for  $n_i$  in  $N_{near}$  do
2:    $c_{prop} = \text{dist}(n_{new}, n_i) + n_{new}.cost$ 
3:   if  $c_{prop} < n_i.cost$  then
4:      $n_i.parent = n_{new}$ 
5:      $n_i.cost = c_{prop}$ 
6:      $\text{propagateCostToChildren}(n_i)$ 
7:   end if
8: end for

```

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

Rewiring, detailed in Algorithm 3 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 proposed cost c_{prop} would be $n_{new}.cost + \text{dist}(n_{new}, n_i)$. If $n_i.cost$ is higher than c_{prop} , 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 (`propagateCostToChildren`).

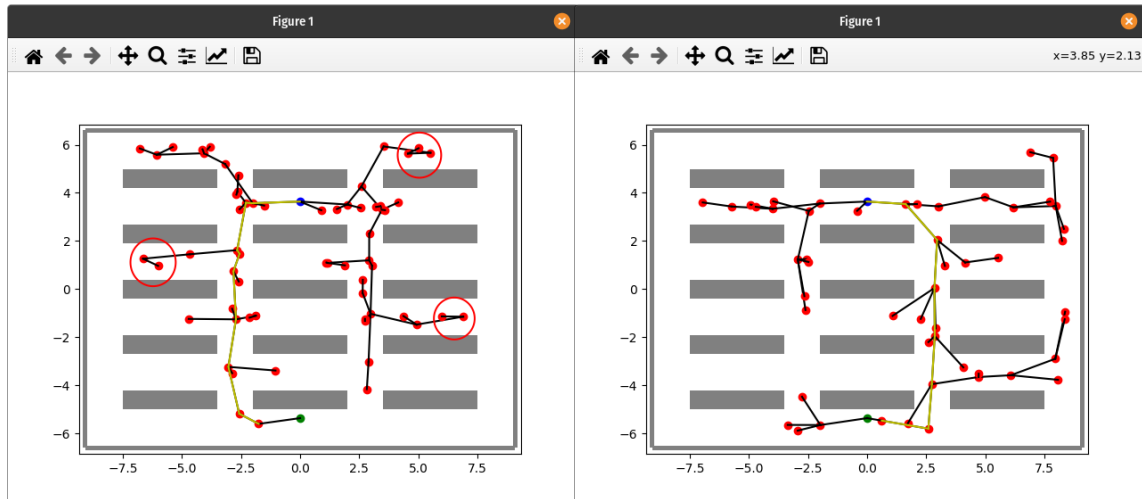


Figure 2.4: The difference between RRT (left) and RRT* (right)

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 (`MIN_RRT_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 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. These are circled in red on the left and right extremes of the image.

2.1.2 Local Planners

Two single-robot local planner approaches were considered and elaborated on below. Typically, these approaches require tuning a cost function. The cost function 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

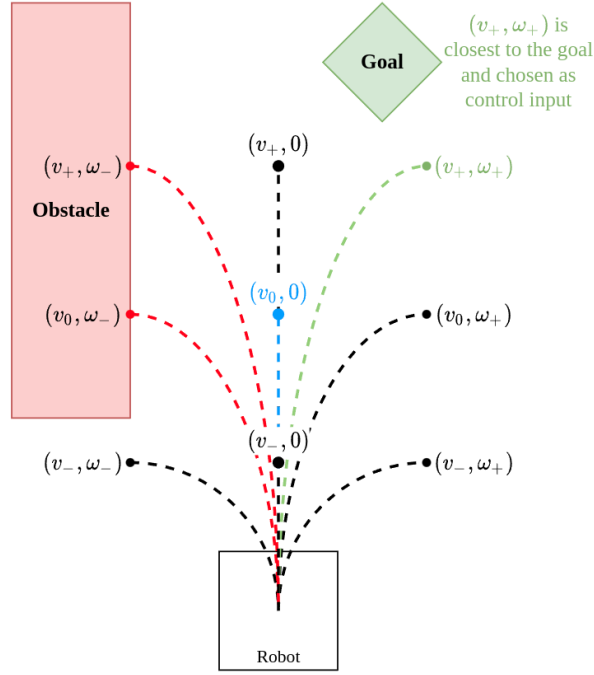


Figure 2.5: Diagram describing the DWA algorithm

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 $\sigma(\alpha \text{ heading}(v, \omega) + \beta \text{ distance}(v, \omega) + \gamma \text{ velocity}(v, \omega))$, with $\sigma, \alpha, \beta, \gamma$ as tuneable parameters. 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. It also keeps distance from obstacles while minimising the overall path length.

A "timed elastic band" is described by a series of n intermediate poses $Q = \{\mathbf{x}_i\}_{i=0 \dots n}$ and $n - 1$ time differences $\tau = \{\Delta T_i\}_{i=0 \dots n-1}$, 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, possible optimization objectives

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. Finding an optimal solution to the MAPF problem has 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.

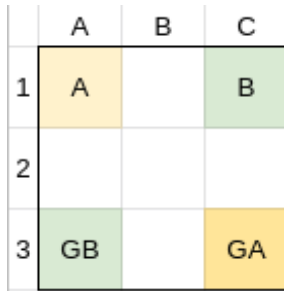


Figure 2.6: Example CBS map

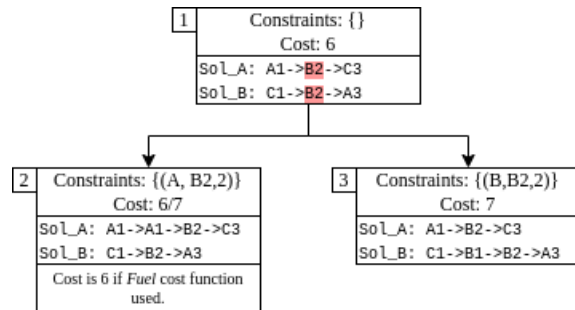


Figure 2.7: Example CBS configuration

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.

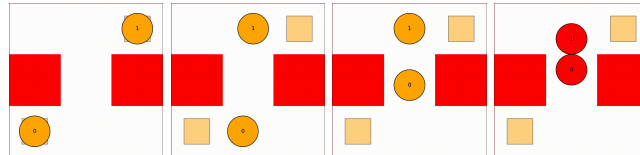


Figure 2.8: Showing SIPP fail to find a path. Pictures from left to right

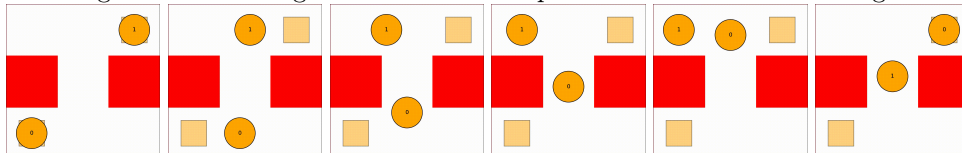


Figure 2.9: Showing CBS finding a path. Pictures from left to right

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 unsuitable.

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 be significant.

2.3 Development Environment

2.3.1 Robot Operating System

The Robot Operating System (ROS) [3] 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

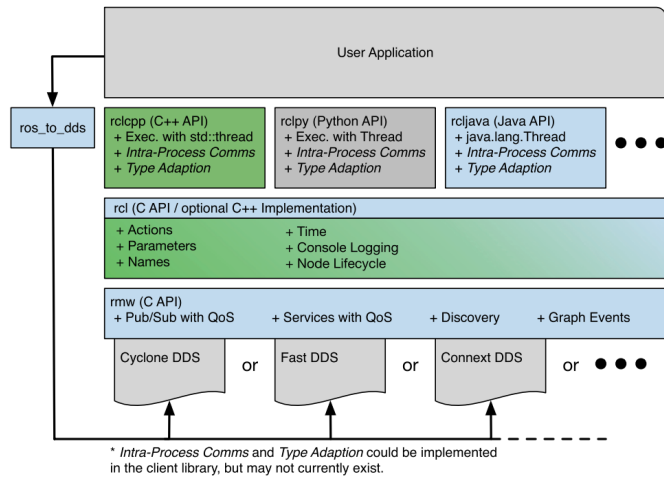


Figure 2.10: A diagram of the ROS system architecture from [3]

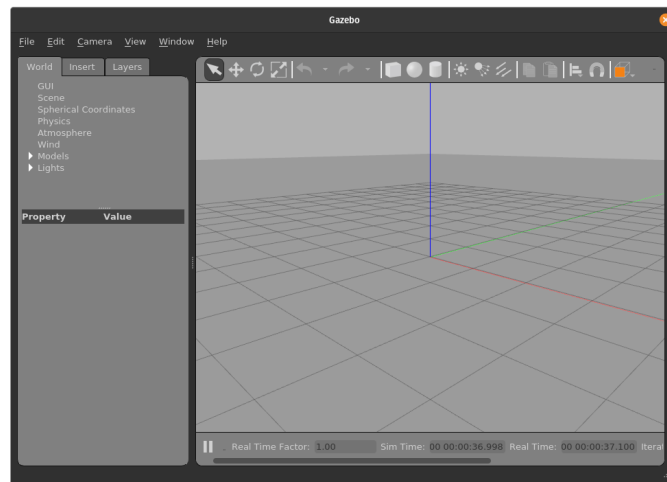


Figure 2.11: A screenshot of an empty simulation world loaded in Gazebo's GUI

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.

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 50x50cm 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.

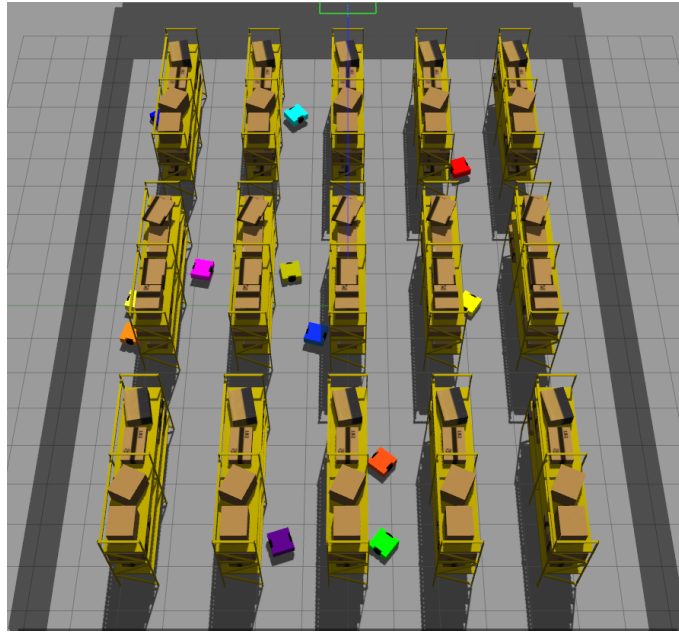


Figure 3.1: Screenshot of simulation environment with many robots spawned in random poses

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 [7]. 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.

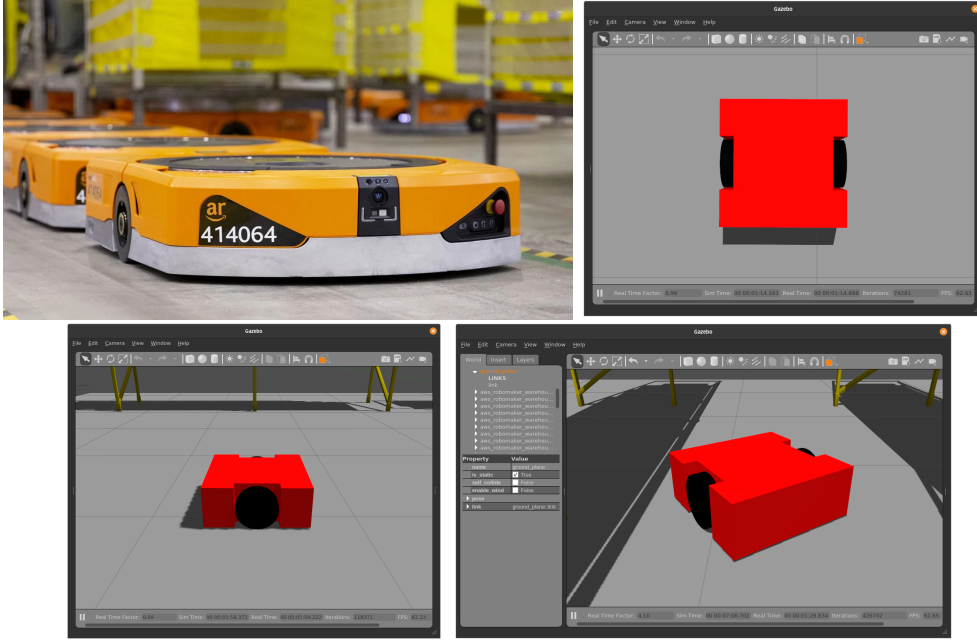


Figure 3.2: Clockwise from top left: Amazon Pegasus Warehouse Robot from [2], top, side and perspective views of robot model

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.

To facilitate a collaborative model, 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.

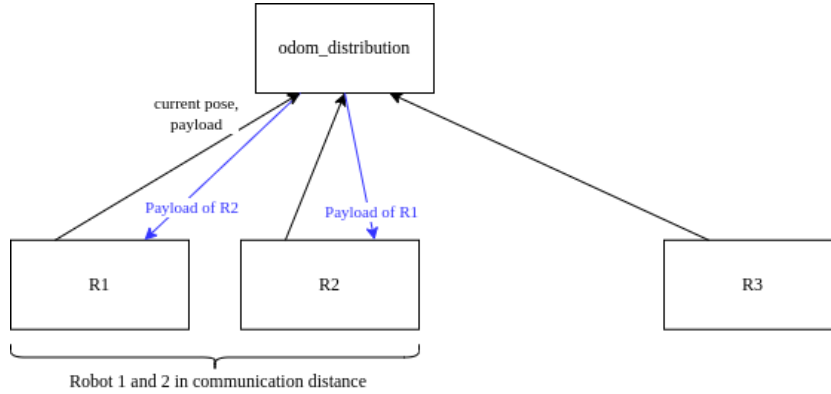


Figure 3.3: Illustration of `odom_distribution` node

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 $R1$ or $R2$, 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 to be 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 at that location.

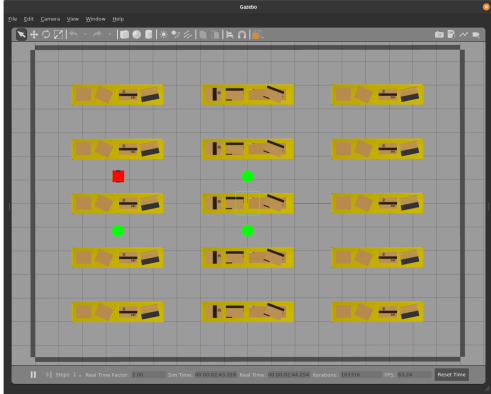


Figure 3.5: Single Robot Scenario

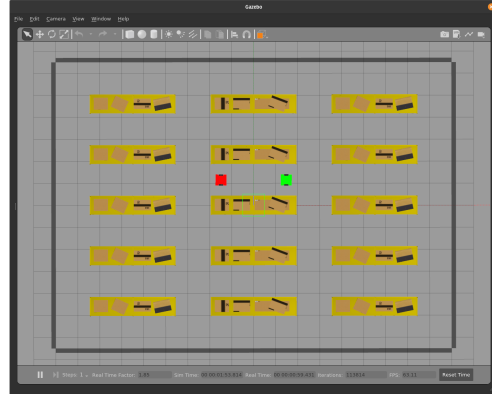


Figure 3.6: Robot Swap Scenario

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.

3.1.5 Overall System Architecture

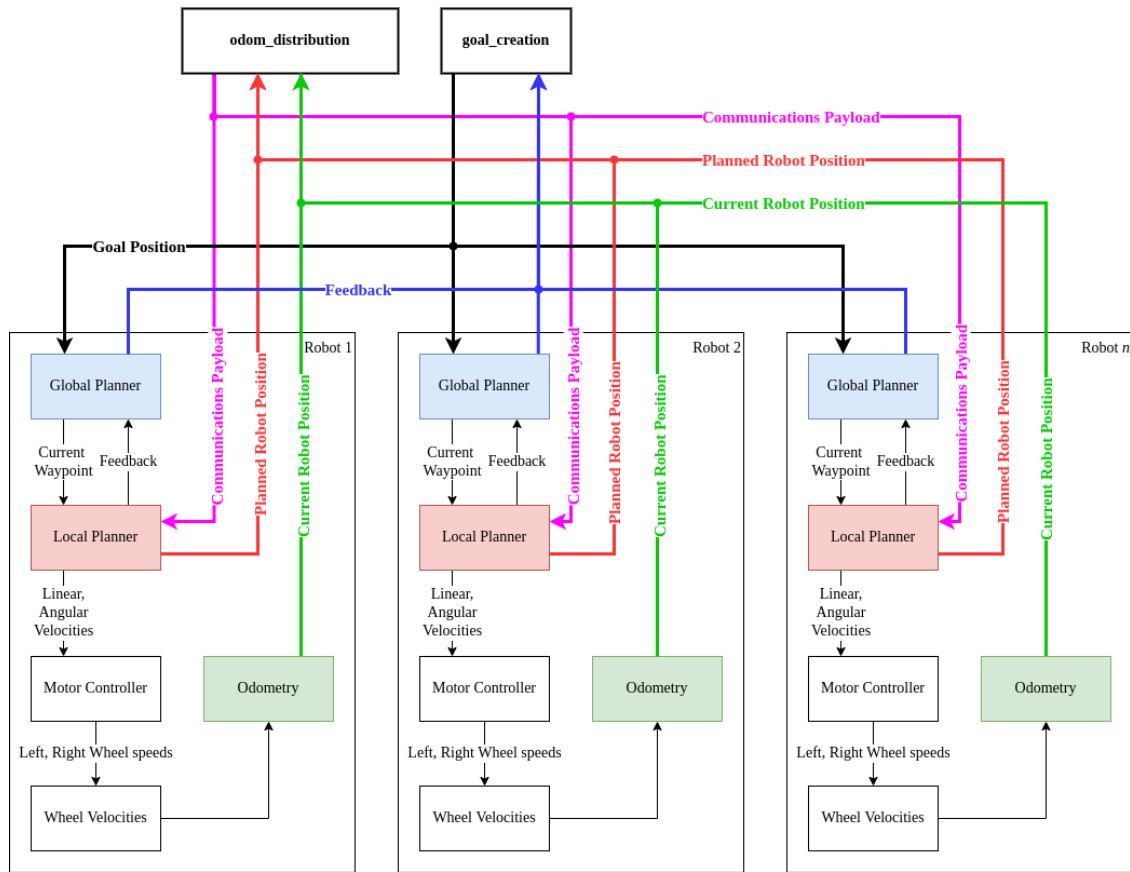


Figure 3.7: Planner System Architecture

Figure 3.7 shows the overall system architecture of the multi-robot simulation. Each robot has a global planner and local planner implementation, and an arbitrary number of robots can be instantiated. The `goal_creation` node assigns new goals to each robot, and keeps track of when each robot has finished each goal, as detailed in 3.1.4. The `odom_distribution` node takes in the current and predicted positions of each robot based on the local planner and odometry nodes in each robot, and passes on the relevant communications payload to robots close to each other, as detailed in 3.1.2.

When given a new goal, each robot's global planner will plan a path from the robot's current position to the goal through a series of waypoints. The robot then navigates towards each waypoint using the local planner. In this implementation, the local planner communicates with a Motor Controller in simulation, which then controls the individual wheel velocities of the robot.

3.2 Global Planner

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*.

The RRT family of sampling-based algorithms was chosen as the GP implementation for this project. 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 sampled 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 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 to 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 [25].

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.

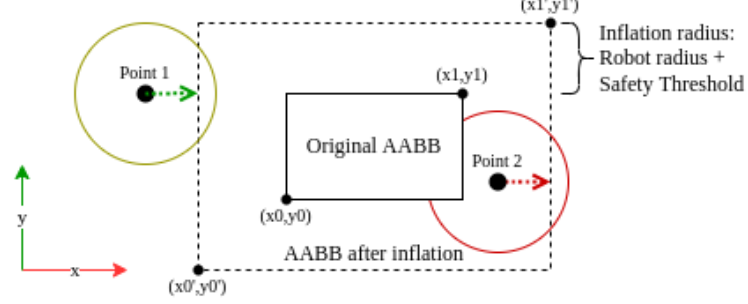


Figure 3.8: Distance to AABB algorithm

Figure 3.8 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.

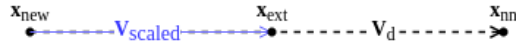


Figure 3.9: How an extension \mathbf{x}_{ext} is found

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 3.9 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. [26] 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.

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

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

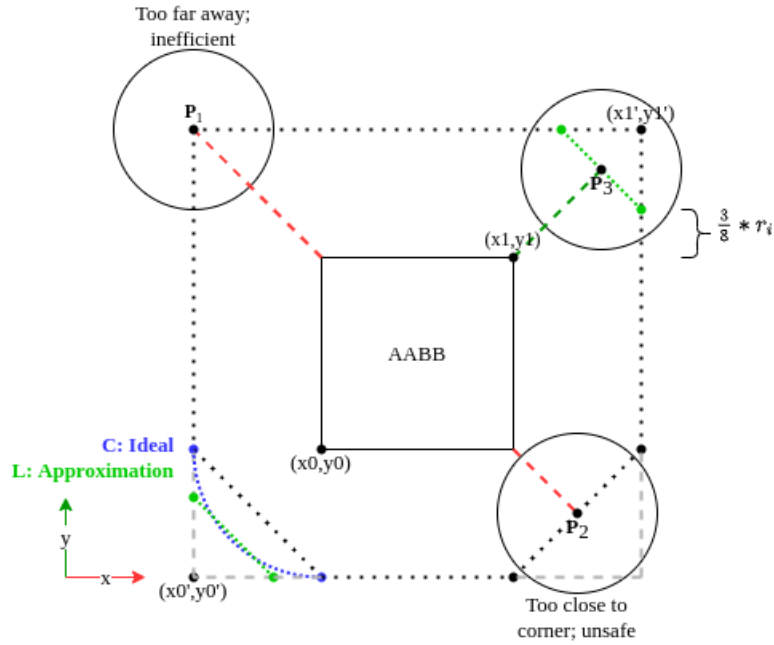


Figure 3.10: Why simple inflation with a square is suboptimal

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 3.10. 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 **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, an approximation showed with the green dotted line **L**. Connecting the edges at (x_1, y'_1) and (x'_1, y_1) as in \mathbf{P}_2 is not safe as the robot would be too close to the AABB. Instead, the octagon's bounding lines are drawn at $(x_1 + \frac{3r_i}{8}, y'_1)$ and $(x'_1, y_1 + \frac{3r_i}{8})$, as in \mathbf{P}_3 . In doing so, the line from P_s to P_{ext} would be checked for an intersection with 8 line segments instead of 4.

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 $\text{len}(N)$. This is given by $r_{thresh} = \sqrt{\frac{\ln \text{len}(N)}{\text{len}(N)}}$.

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}

Algorithm 4 Algorithm for choosing the best parent for a new node

```

1:  $n_{best} = \text{None}$ 
2:  $c_{best} = \infty$ 
3: for  $n_i$  in  $N_{near}$  do
4:   if  $\text{lineIsCollisionFree}(n_{new}.pos, n_i.pos)$  then
5:      $c_{prop} = \text{dist}(n_{new}, n_i) + n_i.cost$ 
6:     if  $c_{prop} < c_{best}$  then
7:        $c_{best} = c_{prop}$ 
8:        $n_{best} = n_i$ 
9:     end if
10:  end if
11: end for
12:  $n_{new}.parent = n_{best}$ 
13:  $n_{new}.cost = c_{best}$ 

```

n_{new} is then connected to the node in N_{near} with the lowest cost, detailed in Algorithm 4. 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_i is checked if it intersects with any obstacles as done earlier in 3.2.1. If the connection is admissible, the cost of n_i is added to the Manhattan distance between the positions of n_{new} and n_i to give the proposed cost of the extension. The n_i 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, and 200 nodes were tested. All other parameters were kept constant.

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

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.

As the number of nodes in the RRT* graph increases, the search space becomes more densely packed with nodes. To find near neighbors of a newly-expanded node, the current implementation iterates through the entire list of nodes. Furthermore, the rewiring function iterates through each near neighbor. If a node is rewired, the implementation of propagating the updated cost to its leaves is implemented by iterating over the list of nodes to find the nodes with the current node as its parent. This leaves a worst-case complexity of $O(n^2)$ with the number of expanded nodes.

With these results, it was deemed that a minimum number of 50 nodes for the RRT* implementation was a good middle ground between cutting down on the overall execution time while obtaining a shorter path for the robots to follow.

3.2.4 Tuneable Parameters

A list of the tuneable parameters of the RRT and RRT* planners is given in Appendix A.2.

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 planner 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 planner 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 [27] 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 rad/s will be considered. If a proposed velocity would exceed the bounds, it is not considered. For example, 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.

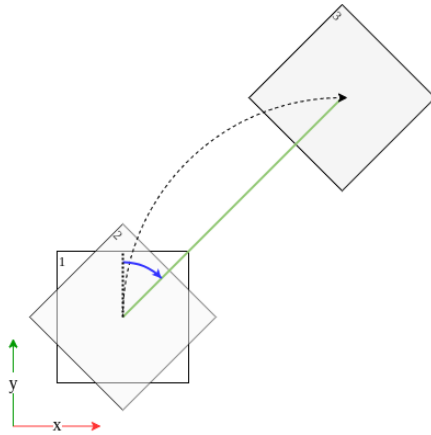


Figure 3.11: Illustration of forward projection of position

Figure 3.11 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 is (x', y', θ') .

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.

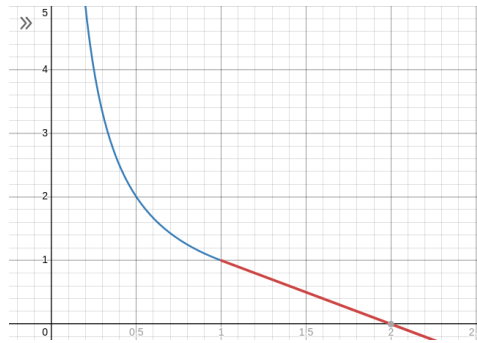


Figure 3.12: Diagram showing the two-part scoring function

Figure 3.12 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

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

d_{θ} is clamped between tuneable maximum and minimum values, $d_{\theta_{max}}$ and $d_{\theta_{min}}$. m_{hdg} and c_{hdg} are set such that $a_{hdg} = 0$ when $d_{\theta} = d_{\theta_{min}}$ and a_{hdg} is maximum when $d_{\theta} = d_{\theta_{max}}$.

Adjusting $d_{\theta_{min}}$ allows 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.

Safety

There are occasions when no valid paths are obtained (all path scores are $-\infty$). In that case, the robot automatically stops (sets v_{lin} and v_{ang} to 0). Setting the velocities to 0 abruptly may allow the robot to find safe velocities in the neighborhood of 0. In the worst case, it will prevent the robot from colliding with other objects at high speed.

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 at least 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.

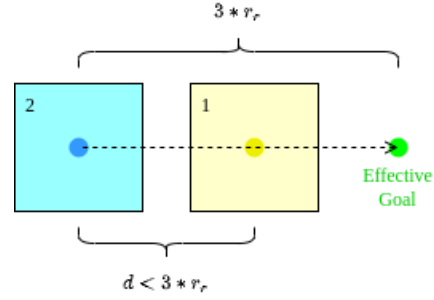
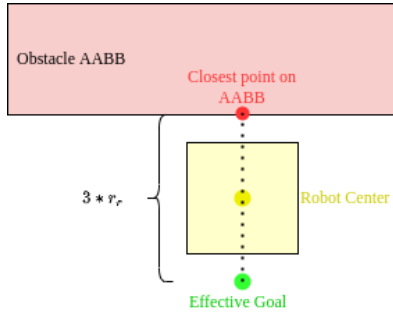


Figure 3.13: Alternative goals near obstacle Figure 3.14: Alternative goals near other robots

Figure 3.13 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 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 safer point.

A similar process is done if another robot is in close proximity. Figure 3.14 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.

An example of this process is shown in Figure 3.15. 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

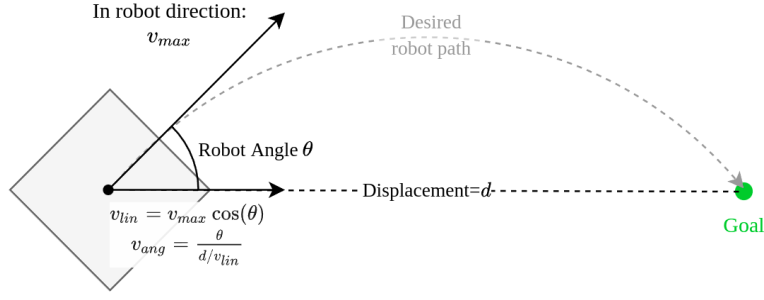


Figure 3.15: Differential Drive Kinematics to get to Goal

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.

However, this approach for calculating v_{ang} did not work well in practice. Some occasions, the discretization of the angular velocity lead to a value of zero angular velocity being proposed. Therefore, another approximation was used where $v_{ang} = v_{max} \sin(\theta)$. Even if the value of v_{ang} was an over-estimate, the negative feedback provided by the DWA algorithm allowed the robots to converge to the correct trajectory eventually.

This method was also useful to give robots an initial hint of the required linear and angular velocities in order to reach a new waypoint. When a new waypoint was to be navigated to, the linear and angular velocities are thus initialized to the values from this method.

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.

Testing for the single-robot case

Firstly, the robot's performance was verified 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.

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.

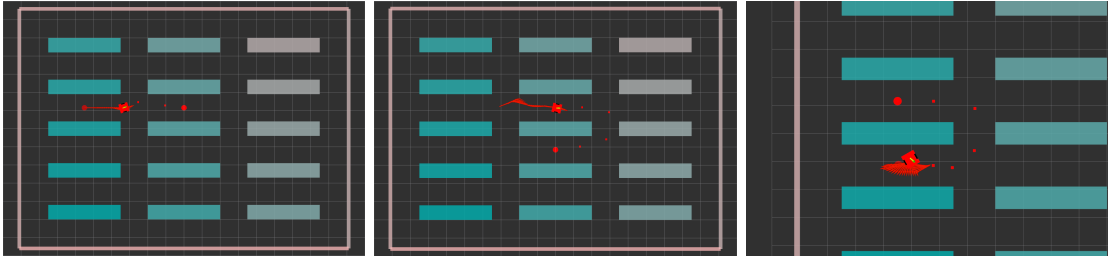


Figure 3.16: Robot moving through single-robot case, left to right

Figure 3.16 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.

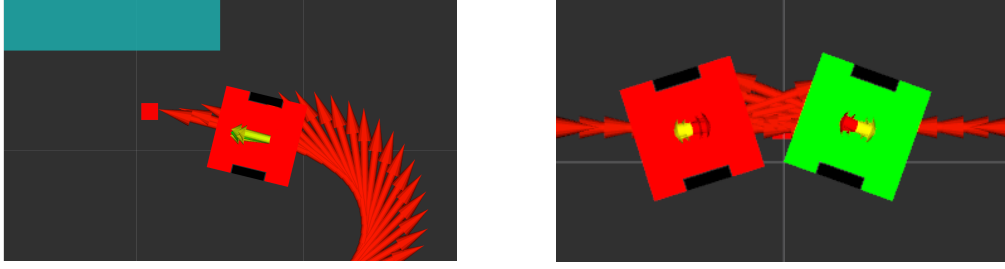


Figure 3.17: Robot control input selection

Figure 3.17 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.

Tuning DWA Parameters

The variables tuned for were the safety radius r_s , simulation duration, and action duration of the DWA planner. To obtain initial values, the simulation was hand-run a few times. Table 3.2 shows the values for the grid search, with the initial values in bold. A list of the tuneable DWA parameters can be found in Appendix A.1.

Safety Radius	0.25	0.30	0.35
Simulation Duration	0.25	0.40	0.55
Action Duration	0.01	0.03	0.05

Table 3.2: Parameter search for tuning values

A grid search was run over the possible values, for 27 combinations in total. Each combination was run 10 times to offset the random nature of the RRT planner.

Safety Radius r_s (m)	Simulation Duration(s)	Action Duration (s)	Goals Completed	Move Time (s)	Distance Moved (m)
0.25	0.25	0.03	3.9	20.301	10.11
0.25	0.55	0.01	3.9	20.555	10.11
0.35	0.4	0.05	3.9	22.193	11.07
0.35	0.4	0.03	3.9	22.431	11.21
0.35	0.25	0.05	3.9	24.726	11.35
0.25	0.25	0.05	3.8	21.283	10.52
0.25	0.25	0.01	3.8	22.888	11.58
0.35	0.25	0.01	3.8	25.430	12.18
0.3	0.25	0.01	3.7	21.793	10.86
0.3	0.55	0.03	3.7	22.026	10.91

Table 3.3: Results of parameter tuning grid search

Table 3.3 shows the results of the grid search. The combinations are sorted by the average goals completed, then by the movement duration, and the top 10 combinations are shown. A lower safety radius results in a shorter moving time and lower distance moved, as robots would be able to cut around corners closer to obstacles instead of making square turns.

A shorter action duration means that robots will be more responsive to changes in the environment, as their control loop runs more frequently. However, it also means that there is higher computational load on the processor. A longer simulation duration would mean the robot projects its position further into the future. This may allow the robot to anticipate the effects of its actions better, but this is contingent on an accurate motion model of the robot. As the model in this

project does not take dynamic considerations into account, a longer simulation duration might lead to inaccuracies in the predicted position of the robot, and be counterproductive in general.

In the end, the highest-scoring values for the three parameters were taken and used for future development.

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 3.18 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.

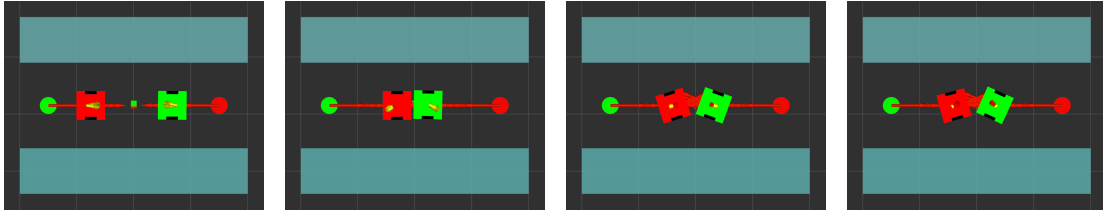


Figure 3.18: Regular DWA planner failing the swap scenario

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 3.18. This variant of DWA is called DWA-Replan, or DWA-R for short.

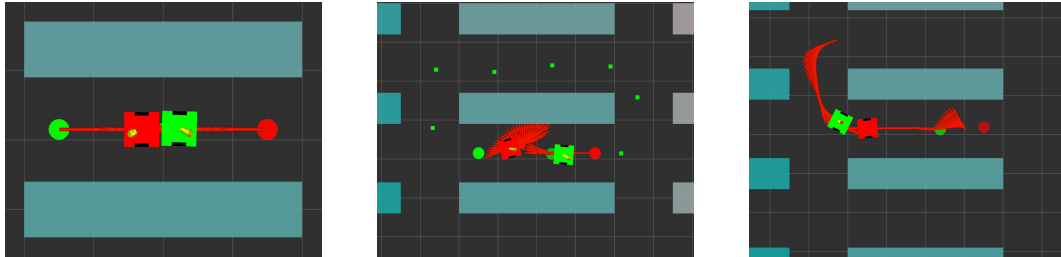


Figure 3.19: Robot showing replanning behaviour

Figure 3.19 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 planner 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.

State Machine

Figure 3.20 shows the DWA-R State machine. The regular DWA planner has two states, **Idle** and **Exec**. The **Idle** state transitions into **Exec** if there is a new waypoint to be followed, and when the waypoint is reached, it transitions back to the **Idle** state. The DWA-R planner extends this by adding another state transition from **Exec** to **Idle** when the replanning criteria are met. This also calls a ROS service to request the Global Planner node to replan its path.

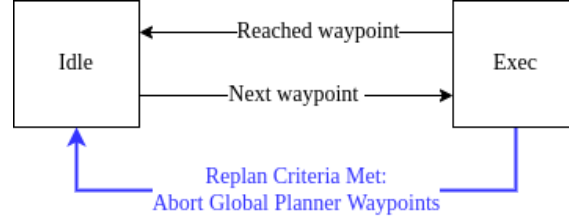


Figure 3.20: DWA-R State Machine

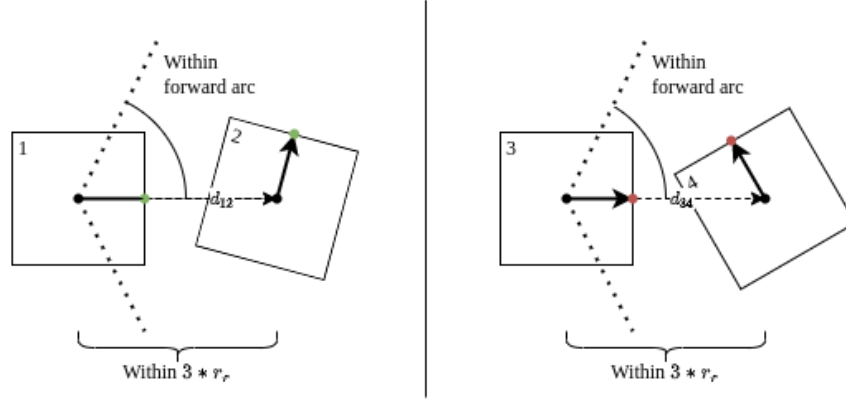


Figure 3.21: DWA Replan Condition

Replanning Criteria

Figure 3.21 illustrates the conditions required for a robot to check if it needs to replan. The DWA-R planner 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 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:

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

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

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 occurs.

Replanning priority

There is the possibility that both robots are facing each other and therefore both meet the replanning 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.

To cover the case when the replan criteria are reached while another robot has already reached its goal or is currently replanning, two special cases are added. If the other robot is already at its goal, it returns $-\infty$ as its remaining Manhattan distance. This ensures that robots that are already at their goals will never be forced to replan. For all intents and purposes, they can be considered static obstacles.

On the other hand, if the other robot is planning, it returns ∞ as its Manhattan distance. To take an example from Figure 3.21, if Robot 3 has higher priority than Robot 4, Robot 4 will replan. However, Robot 3 should wait for Robot 4 to finish replanning, as it will eventually move out of Robot 3's path. To ensure that the replanning robot is always at a lower priority, it returns ∞ as its Manhattan distance.

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.

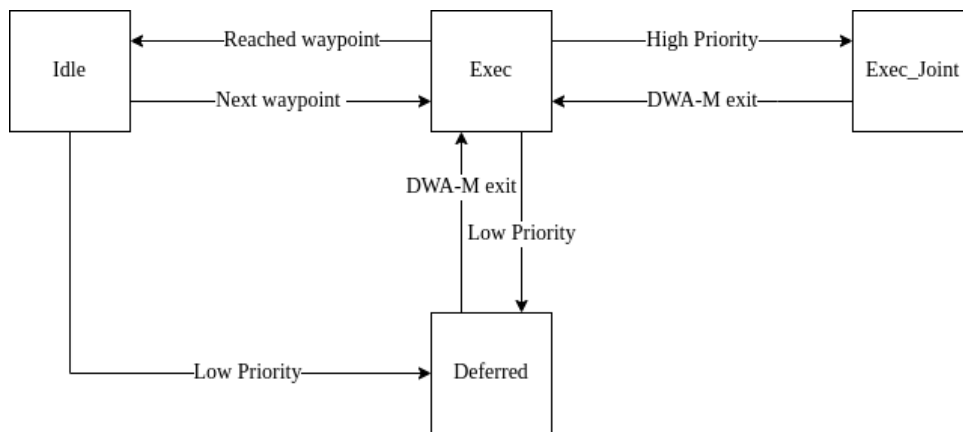


Figure 3.22: DWA-M State Machine

To enable this behaviour, a state machine was implemented, shown in Figure 3.22. This extends the state machine in 3.20 by adding `Exec_Joint` and `Deferred` states. `Exec_Joint` is the state where one robot assumes local centralized control of both robots, and `Deferred` is the corresponding state where a robot allows another robot to take control of it.

From `Exec`, there are additional "high" and "low" priority transitions. A "high priority" transition transits robots to `Exec_Joint`, representing when it is decided that a robot should take over the joint planning. A "low priority" transition is the corresponding transition where a robot defers planning to another robot, and transitions to the `Deferred` state.

In addition, there is also the possibility for robots to enter the **Deferred** state from the **Idle** state. This allows stationary robots to move out of the way of moving robots if needed, for example if they are blocking corridors.

The "DWA-M exit" transition is when the condition for joint planning has ended. Robots will always go back into the **Exec** state to travel towards their original waypoint. This would be important for the **Idle** → **Deferred** transition, as the robot would not be in its original position, and would need to move back to it. For the **Exec** → **Deferred** and **Exec** → **Exec_Joint** transitions, transitioning back to **Exec** allows the robot to continue along their original path.

Waypoint Skipping and Replanning

When robots undergo the "DWA-M exit" transition, they will not be in their original positions. There are two cases, with actions done at the global planner level. These cases would be checked every time a waypoint is reached, or when called via a ROS service.

Firstly, the robot may transit back to **Exec** further along the path than before. In that case, it should skip to its closest waypoint, instead of back-tracking to its original waypoint. This was called *waypoint skipping*. To check if a waypoint could be skipped, the list of waypoints was iterated through. If there is a waypoint within 1.5m of the robot that is further down along the waypoint list, and the line connecting the robot's position to that waypoint is not obstructed by obstacles (line of sight), then that waypoint will be set as the currently-selected waypoint for the DWA planner.

Secondly, the robot may transit back to **Exec** without line of sight to the current waypoint. As the DWA planner assigns scores based on the Manhattan distance to the waypoint, it tends to track a straight-line path towards the waypoint. This means that it may be unable to find the waypoint and continue towards its goal. To prevent the robot from becoming stuck in such a scenario, it should replan. This was called *waypoint replanning*.

State Transition Criteria

When two robots are within a distance threshold from each other, they then consider their priority. This is done by comparing each others' DWA-M states, as well as their remaining Manhattan distances to the goal, as in the DWA-R case.

If one robot is **Idle**, it automatically goes to the **Deferred** state. Conversely, if a robot is in **Exec** and the other robot is in **Idle**, it automatically goes to **Exec_Joint**. However, if both robots are in **Exec**, a tie-breaking is performed based on the remaining Manhattan distance to their respective goals, with the robot with the lower Manhattan distance given the higher priority.

When the robots are no longer close by to each other, the "DWA-M" exit transition then occurs. The robots then return to the **Exec** state, as explained in [3.3.6](#).

Goal Arbitration

To encourage robots to move around each other, their intermediate waypoints would be adjusted using a process called *goal arbitration*. The process is described in [Figure 3.23](#).

Initially, Robots 1 and 2 are in a corridor with waypoints close to one another, shown by the waypoints pointed to with their intended paths. However, this will lead to a collision as both robots cannot reach their respective waypoints at the same time. To allow the robots to move around each other, their waypoints are shifted. Robot 2's waypoint has been shifted such that both robots can now pass each other side to side.

To begin, the robot in **Exec_Joint** that is performing joint planning is called the *current* robot, while the robot in **Deferred** is called the *target* robot. The goal arbitration algorithm checks the currently selected waypoints of the current and target robot in relation to their list of waypoints. If the current or target waypoint are the last waypoint (i.e. the goal), then the other robot's waypoints will be modified to be collision-free. However, if both the current and target waypoints

are the last waypoint, then the target's waypoint will temporarily be set to its previous waypoint on its path.

To modify a waypoint to be collision-free, the distance between the current waypoint and the next waypoint is checked. If the current and next waypoint are within a set radius, the planner skips to the next waypoint. However, if the next waypoint is too far away, a point is found along the connecting line between them.

As waypoints are a concept at the global planner level, the DWA-M planner makes use of several ROS services to get and set the global planner's state:

- Getter services:
 - List of waypoints
 - Current waypoint index
 - Remaining Manhattan distance
 - Global Planner State (Idle, Planning, etc.)
- Setter services:
 - Current waypoint coordinates
 - Current waypoint index

Jointly Ranking Future Poses

After waypoints are modified such that the robots are able to pass by each other, the robot in `Exec_Joint` would need to iterate through the possible control inputs of both the current and target robots, project their eventual positions, and rank the top scores for both.

The scoring metrics would likely be similar as the single robot case detailed in [3.3.2](#), but extended to better handle two-robot scenarios. However due to time constraints, the Joint Ranking section of the DWA-R code was not fleshed out sufficiently, and therefore not implemented to be compared to the DWA-M and base DWA planners. However, some aspects of the development were used to improve the implementation of the planner architecture. For instance, the waypoint skipping approach originally intended for DWA-R was left into the global planner, as it helped to smooth out bad RRT paths where waypoints might be bunched up, allowing the robots to move in straight lines.

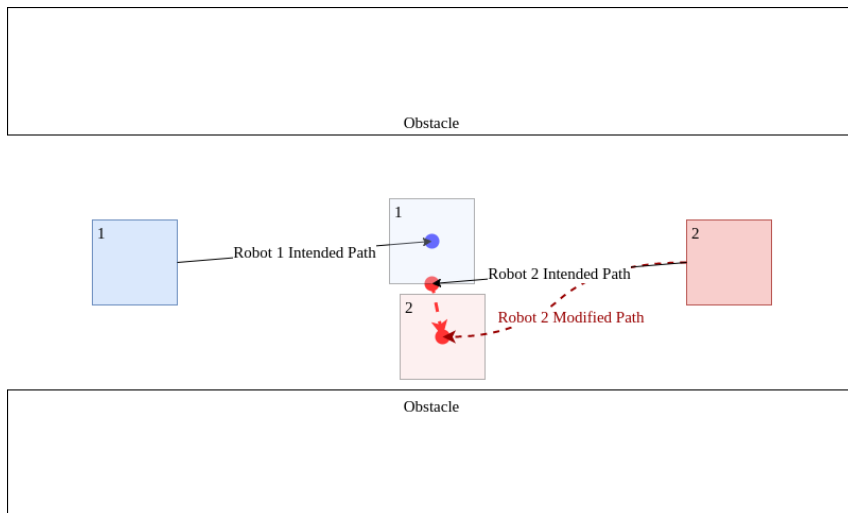


Figure 3.23: Goal Arbitration Motivation

Chapter 4 Evaluation

4.1 Test Metrics

There has been some work on test metrics for mobile robot local planners in [28]. Inspiration was taken from this to come up with the metrics of interest.

To evaluate the performance of the two DWA planner approaches in a generalized case, the test infrastructure detailed in 3.3.4 was 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 in simulated time. 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 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 throughput 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.

4.2 Test Results

Figure 4.1 shows how the total number of goals completed per simulation run is affected by the number of robots in the simulation, and Figure 4.2 shows the number of goals completed per robot per simulation run.

Up to about 5 robots in the environment, the DWA-R approach leads to a larger overall number of goals completed than the base DWA approach. Its best score is at 5 robots, where an average total of 12 goals is attained for the DWA-R planner, compared to the 8.7 goals for the DWA planner. This is an increase of 38.0%. However, it can be seen that the increase is not linear. For each

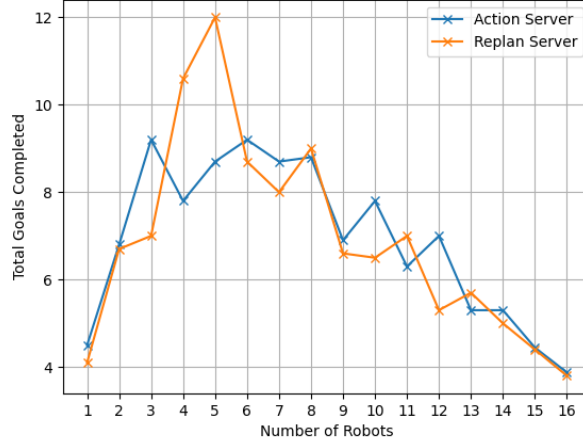


Figure 4.1: Total goals per run with increasing robots

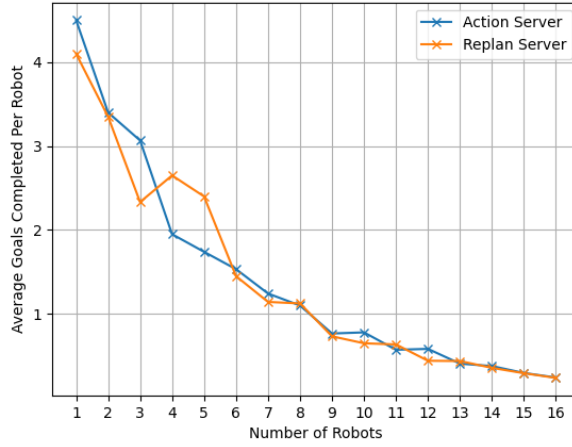


Figure 4.2: Average number of goals per robot per run with increasing robots

robot, the number of goals completed falls overall as the number of robots in the environment increases, as time has to be taken for robots to avoid each other and to replan around other robots. Eventually, as the number of robots in the environment becomes too large, performance of both approaches degrades.

Figure 4.3 shows the time taken per waypoint per robot per simulation run. As expected, the average time taken for each robot to complete a run increases as the number of robots increases.

In terms of the objectives laid out in 1.1, namely, to 1. **Maximise throughput**, 2. **Scale with number of robots**, 3. be **Computationally efficient**, and 4. be **Configurable**, the project partially succeeds.

While able to increase throughput for each robot up to 5 robots in the environment, the proposed DWA-R approach does not scale well with increasing robots.

Run on a single robot, parameters for the the RRT*+DWA-R planner have been chosen to be computationally efficient, as they represent a balance between optimality and runtime. In addition, the implementation has been set up to be highly configurable, as most tuneable parameters can be configured upon instantiating the planner ROS nodes. Furthermore, extensions to the implementations of the global and local planners can easily be done through class-based inheritance in Python.

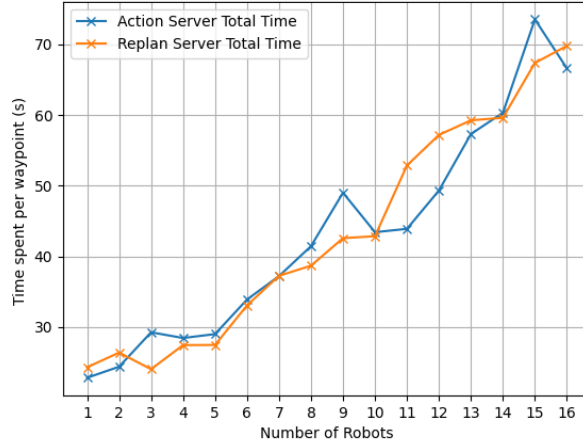


Figure 4.3: Average time taken per waypoint per robot per run with increasing robots

4.2.1 Testing Limitations

Although the computation of each simulated robot was separate from the others, the calculations still all run on the same CPU during testing. This meant that as the number of robots in the simulation increased, the computational burden on the CPU also increased.

Although the simulation runs off a simulated clock time which slows down to account for the additional physics simulations required to support multiple robots, the clock does not take the additional processing time of ROS nodes into account. Thus, it can take a long time for heavy calculations such as RRT path finding to complete when there are many simulated robots. This would affect the accuracy of the timing metric used for testing, as the robots take more time than they usually would to (re)plan paths around obstacles. Therefore, the relative inefficiency of the DWA-R approach could be taken with a grain of salt.

4.3 Global Planner Evaluation

On average, the RRT* planner used was suitable for the multi-agent path planning task. It was able to return the shortest path through the shelves in the environment most of the time. However, on occasion, it would return a longer path, owing to the random nature of the planner, shown in Figure 4.4.

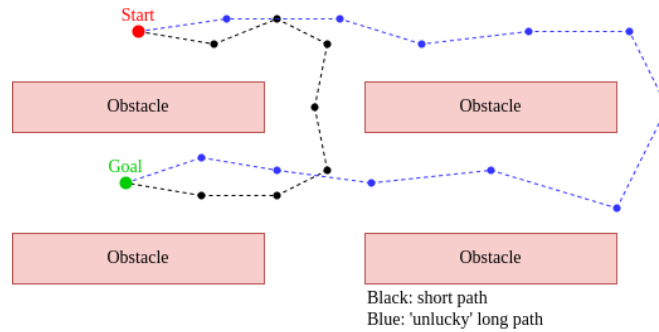


Figure 4.4: Path choices possible due to the random nature of the RRT planner

When required to plan around other robots, the planner was usually able to find an alternative path quickly. However, on occasion, the planner would take unduly long to return a path, sometimes even being unable to find one. This might be due to the limitations of the simulation methodology, as noted in the earlier section. However, this behaviour sometimes led to "traffic jams" happening

at intersections, where multiple robots with higher priority would be waiting for a robot to move out of the way. This was due to the assumption that global replanning would be done fairly quickly.

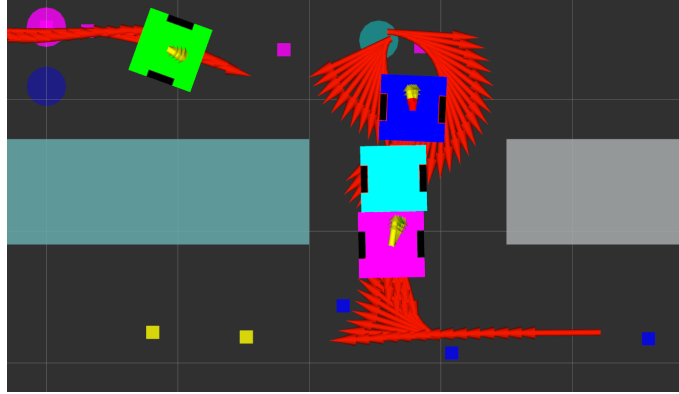


Figure 4.5: Robots in gridlock

Figure 4.5 shows the cyan robot is unable to plan around both the blue and purple robots that are sandwiching it. This situation is impossible to recover from because both the purple and blue robots will have higher priority than the currently planning cyan robot, but the cyan robot will not be able to plan a path to escape either blue nor purple robots.

To counteract this, the logic for replanning could be tweaked such that robots with lower priority move out of the way first before replanning. This could be achieved by editing the position of the waypoints based on the machinery developed as part of the DWA-M planner.

4.3.1 Future Work

In the future, a grid-based search method could be implemented for comparison with the sampling-based methods implemented in this project. A grid-based method would return the shortest path to the goal, but it is uncertain how long the search time would be, especially if the start and goal locations are far apart. On the other hand, the path generated by the grid-based method would probably require post-processing, as it would not be as sparse as the ones generated by the sampling planners. This is illustrated in Figure 4.6.

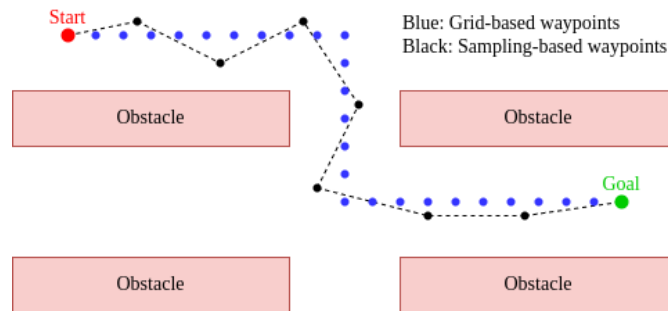


Figure 4.6: Path choice difference between grid-based and sampling-based planners

The implementation of the RRT and RRT* algorithms could also be improved. Notably, another data structure that enables more efficient traversal of nodes' children would allow for shorter execution time. This would be used when updating the costs of rewired nodes' children in RRT*.

In addition, RRT* uses many nearest-neighbor queries. A KD-Tree could be used to split up the list of nodes for faster nearest-neighbor search as the number of nodes increases.

4.4 Local Planner Evaluation

The DWA planner works well in the single-robot case. Particularly, after being given good initial values from differential drive kinematics described in Figure 3.15, the planner is able to move through the waypoints smoothly.

With hint?	Movement Time (s)	Total Time (s)	Movement Distance (m)	Goals Completed
Yes	5.42	6.52	12.49	4
No	6.27	7.77	12.61	4

Table 4.1: Ablation study on initial command values for DWA planner

Table 4.1 shows the importance of good initial values. The test was conducted on the Single Robot test in a similar fashion as in 3.3.4, with 10 runs for each case. It can be seen that using an initial hint leads to a 13.6% drop in the movement time. Without these initial values the DWA planner has to find the best values to maximise the cost function. This process can take some time, especially if the next waypoint is at a sharp angle.

The DWA planner was chosen for its ability to handle dynamic obstacles. The collision avoidance mechanism implemented in the DWA planner, where each robot reports its planned position to other robots, is similar to the Velocity Obstacle method. However, there is no concept of reciprocity, leading to worse than expected results. When two robots face each other, there is oscillation in the robots' trajectories. Furthermore, there is no concept of orientation, as robots simply avoid the present location of other robots, not considering their motion. This works well for sparse environments, but in the more cramped setting of the simulation environment, it leads to deadlock most of the time without the DWA-R or DWA-M implementations. This is shown in Figure 4.7.

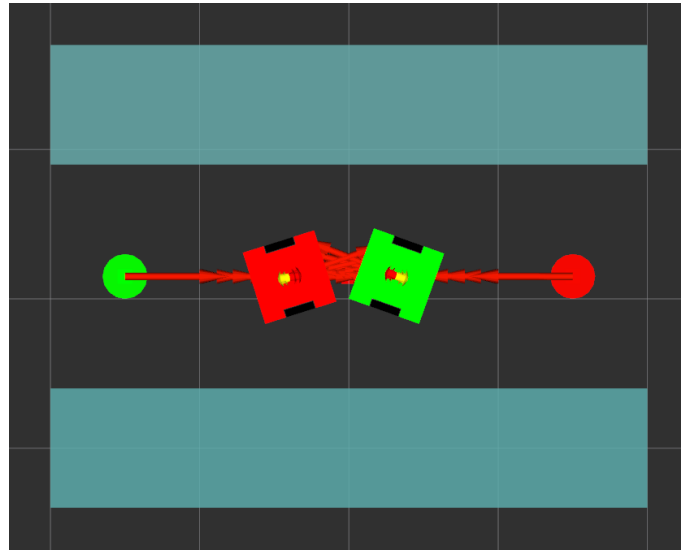


Figure 4.7: Robots unable to proceed due to reciprocity

Due to time constraints, it was not possible to come up with a suitable cost function for DWA-M. However, some of the methods originally developed for DWA-M were of use in other aspects of the project.

4.4.1 Future Work

Improving the DWA implementation

In the future, the Reciprocal (RVO) or Hybrid Reciprocal Velocity Obstacle (HRVO) method could be implemented as a collision avoidance mechanism with respect to other robots. This might remove the need for a joint planning system like in DWA-M. Particularly, HRVO defines

conventions for robots to cooperate. For instance, it could be assumed that robots will yield to other robots on the left side. This is particularly attractive as it removes the need for a complicated state machine.

Alternatively, Predictive DWA controllers like in [29] could be implemented. This approach breaks moving obstacles into moving polygonal segments, which can be checked for collision with the arc of the robot produced by its differential drive movement.

Removal of the global planner

Another interesting extension would be the removal of the global planner. The implemented DWA planners seek to minimize the straight-line distance to each waypoint, which falls apart if there is no line of sight to the waypoint.

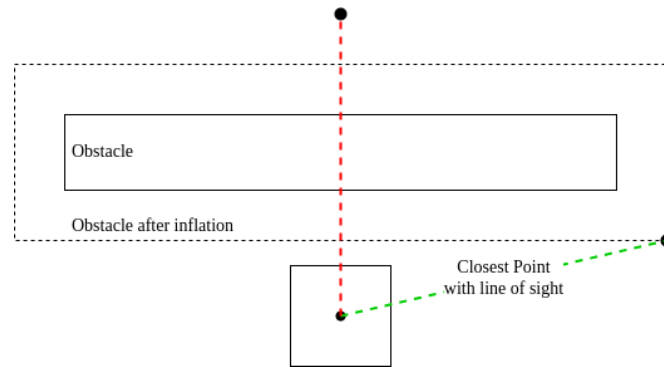


Figure 4.8: Local Minima in DWA algorithm

The red dotted line in Figure 4.8 shows an example of this. However, the distance function could be set to the closest point within line of sight to the current waypoint, shown with the green dotted line. This could remove the need for a global planner, removing a significant part of the computational burden.

Accounting for Dynamics

The current implementation of DWA does not consider the dynamics of the robot. By taking into account the acceleration of the robot, more accurate final poses would be given by each DWA control input. This would allow for larger forward simulation time durations.

Chapter 5 Conclusion

This project has successfully instantiated a multi-robot simulation environment in the Gazebo simulation environment. Two variants of the DWA local planner were created, along with a RRT and RRT* global planners, which were tied together using the ROS programming framework. Lastly, automated testing was created to find optimal parameters for the planners as well as evaluate the approaches used.

When the DWA-R planner was compared against the base DWA implementation, the performance improvement was found not to scale well with increased robots, as the increased traffic in the environment resulted in gridlock. This points to collaborative approaches being better suited to implementation at the local planner level, rather than attempting to interleave the global and local planners as done with DWA-R.

5.1 Future Work

This project focused mainly on distributed planning, assuming that centralized planning was inconvenient in the given use case due to the unsynchronized nature of robots starting and completing their goals. However, lifelong MAPF planners like [30] exist and could be investigated. In smaller settings, these could offer better performance than the distributed approaches that took center stage in this project.

The focus of the project was to develop a multi-robot trajectory planner, so ground truths were given in all cases. To bring this work closer to real-world applications, it could be made more robust under uncertainty and also have integration with perception algorithms.

5.2 Ethical Discussion

Developed further, the multi-robot planning methods created in this project could be applied in situations where there are clear tasks and a well-defined map for each robot. In addition to the warehouse scenario in this project, one could imagine multi-robot systems used to achieve higher levels of automation in factories, farms, and for security coverage, to name a few possible applications.

As with any form of disruptive technology, if MRS becomes widespread, people previously working in these sectors may be displaced. While this would remove these individuals from needing to do menial, repetitive tasks more suited to robots, it will have an impact on their livelihoods. A study in 2017 [31] found that an additional robot per thousand workers reduces the employment to population ratio by about 0.18-0.34 percentage points and wages by 0.25-0.5 percent. On the other hand, additional automation improves the productivity of industries.

This points to the need for a holistic view on the further introduction of robots into industry. While robots may not bring an increase in economic standard of living by themselves, if combined with other policies that create jobs or enable displaced workers to re-skill, it is hoped that this will improve the quality of life for all overall.

Chapter A Appendix

A.1 DWA parameters

Parameter Name	Default Value	Description
robot_radius	0.35	The value of r_r , furthest edge of robot from the center.
safety_thresh	0.3	Safety distance r_s
simulate_duration	0.4	Seconds to forward simulate a set of control inputs
action_duration	0.03	Seconds to carry out a set of control inputs
linear_speed_limit	0.6	Max value of v_{lin} (m/s)
angular_speed_limit	1.5	Max value of v_{ang} (rad/s)
linear_step	0.1	Step size for v_{lin}
angular_step	0.2	Step size for v_{ang}
dist_thresh_hi	0.3	Distance threshold from robot to goal to consider to have reached a waypoint
dist_thresh_lo	0.05	Distance threshold from robot to goal to consider to have reached a goal
dist_method	"L2"	Method of calculating the distance to any point. Currently the only implemented method is the Manhattan distance, or L2 norm.
inter_robot_dist	3.0	Multiples of {robot_radius} for robots to keep away from other robots.
orientation_ub_deg	180.0	Max orientation difference to be considered in scoring function for heading
orientation_lb_deg	20.0	Min orientation difference to be considered in scoring function for heading
angular_K	1.0	Max proportion of s_{goal} that would be deducted from the DWA score based on the orientation of the robot
goal_K	10.0	Score for distance to goal
obstacle_K	1.0	Cost for distance to obstacles
stall_det_period	1.0	Time window to check for stall
stall_dist_thresh	0.1	Lower distance (m) bound for stall detection
replan_duration	5.0	How long to wait (s) to replan when repeated replan requests are made
move_towards_goal_hint	True	Whether or not to use the stall detection method to give the robot initial values for command input

A.2 RRT and RRT* parameters

Parameter Name	Default Value	Description
rrt_path_bias	0.1	Bias rate for RRT planner to extend directly towards goal
rrt_it_lim	500	Maximum number of RRT iterations
rrt_it_min	50	Minimum number of RRT* iterations
rrt_max_extend_length	1.5	Distance (m) to extend a new point away from the original point towards a new one
rrt_connect_circle_dist	1.5	Distance (m) to find near nodes in the RRT* algorithm
rrt_debug_plot	False	Whether or not to plot the pathfinding process to matplotlib
waypoint_skip	True	Whether or not to skip redundant waypoints
waypoint_replan	False	Whether or not to replan if waypoints are too far away
local_planner	"dwa_replan_server"	Choice between DWA-R (default) and base DWA ("dwa_action_server") implementations

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] “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>
- [3] 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>
- [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] “ALL the Amazon Warehouse Robots: From Fulfillment to Last-Mile Delivery.” [Online]. Available: <https://www.agvnetwork.com/robots-amazon>
- [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] “LearnOpenGL - Collision detection.” [Online]. Available: <https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection>
- [26] D. Kirk, *Graphics Gems III*. Academic Press, Inc, 1992.
- [27] “Gazebo ROS Plugins.” [Online]. Available: https://github.com/ros-simulation/gazebo_ros_pkgs
- [28] 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>
- [29] 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.
- [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] D. Acemoglu and P. Restrepo, “Robots and Jobs: Evidence from US Labor Markets,” National Bureau of Economic Research, Cambridge, MA, Tech. Rep. w23285, Mar. 2017. [Online]. Available: <http://www.nber.org/papers/w23285.pdf>