Distributed Collision Avoidance and Object Sorting for Robot Swarms

by

Mohammed Abdullhak

Supervisor: Dr. Andrew Vardy

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

September 2021

St. John's Newfoundland

Abstract

A robotics swarm is a set of simple distributed agents cooperating to achieve a certain goal. Each robot works independently of the others based on its own local knowledge without any central coordination. These systems are often inspired by social insects which work collaboratively to perform complicated tasks such as sorting their brood or waste in certain patterns through local interactions. Swarm robotics take inspirations from these natural phenomena to design scalable and robust systems.

The work in this thesis can be divided into two main objectives. The first is a general purpose distributed collision avoidance algorithm that enables multiple robots to seamlessly share their environment without colliding or blocking each other's paths. The proposed algorithm is very fast with $\mathcal{O}(n)$ complexity and only requires relative positions of neighboring robots. It also has special mechanisms for early deadlock prediction and recovery to prevent robots from getting stuck.

The second objective is proposing a distributed sorting algorithm. It builds upon the previous algorithm, which guarantees collisions avoidance and minimizes deadlocks while driving the robots to their goals, and incorporates the ability for a robotic swarm system to cooperatively sort a collection of objects from different classes into desired areas for each class.

The design and implementation of this swarm system on a simulation platform and on physical robots will be detailed. A web-based multi-robot simulation platform is developed as a general robotics simulation and will be used to evaluate the different algorithms in our system. We will also showcase and evaluate the proposed swarm system by deploying these algorithms on actual robots.

Acknowledgements

First and foremost I am forever grateful to my supervisor, Prof. Andrew Vardy for giving me the opportunity to pursue this degree by accepting me to the Bio-Inspired Robotics Lab (BOTS) and for his invaluable advice and continuous support throughout this program. I am also deeply grateful to the funding received from the Institute of International Education (IIE) and Memorial University of Newfoundland without which it would have been impossible for me to pursue this degree.

I would also like to express my gratitude to my parents and my whole family for their tremendous encouragement. And finally, I wish to thank my wife for standing by me, for always being extremely patient and supportive, and for her countless sacrifices to help me get here.

The work discussed in this thesis was partially supported by the Natural Sciences and Engineering Research Council of Canada.

Contents

Citations to Published Work

Parts of the work presented in this thesis was published in the following paper [1]:

Abdullhak, Mohammed, and Vardy, Andrew. "Deadlock Prediction and Recovery for Distributed Collision Avoidance with Buffered Voronoi Cells." 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)

List of Figures

Chapter 1

Introduction

Robotic swarms consist of multiple autonomous robots without any centralized control. In this thesis we will present a distributed collision avoidance algorithm that allows multiple robots to seamlessly work in a shared environment without colliding or blocking each other paths. We will then build upon this algorithm to introduce a swarm sorting system where autonomous robots work cooperatively to gather objects into specific locations in the environment. We will also introduce SwarmJS, a web-based multi-robot simulation platform that was developed specifically for implementing and bench marking swarm algorithms.

Swarm systems often use simple robots with limited capacity to sense and act in their local environments, so they must cooperate to achieve a given task [2]. This cooperation can lead to emergent behaviors similar to ones observed in social insects, which was described by Beni [3] as swarm intelligence (SI), the collective behavior of decentralized, self-organized systems, whether natural or artificial.

Swarm robotics was defined by Brambilla [4] as the study of collective robotics

that takes inspiration from the self-organized behaviors of social animals. Brambilla et al. further detail swarm systems as multi-robot systems which have the following characteristics:

- robots are autonomous
- robots are situated in the environment and can act to modify it
- robots' sensing and communication capabilities are local
- robots do not have access to centralized control and/or to global knowledge
- robots cooperate to tackle a given task

These characteristics lead to the main advantages of swarms robots: flexibility, robustness, and scalability [5]. These advantages are similar to those exhibited by the main inspiration for swarm robotics, social animals such as ants and bees.

Robustness is a system's ability to continue normal uninterrupted operation in case of disturbance or loss of individual agents within the system. Swarm systems' redundancy and lack of central control allows robots to perform their tasks regardless of the state of individual robots within the system, since the failure of some members will be compensated by others [6].

Scalability is the property of a system to handle a growing amount of work by adding resources to the system [7]. The applied methods in swarm robotics -such as control algorithms of the robots- scale to any size of the swarm [8]. Such a system "can maintain its function while increasing its size without the need to redefine the way its parts interact" [9]. Thus, swarm systems can handle increasingly bigger problems such as larger areas or more objects to sort simply by adding more robots to the

swarm without the need to make any major modifications to how the system works. This, in part, is supported by the robots' use of local sensing and planning.

Flexibility is the ability to cope with a broad spectrum of different environments and tasks [4]. This is supported by the use of simple agents and behaviors which can be easily modified to handle different environments and tasks using the same basic underlying algorithms. These characteristics are analyzed by Camazine et al. in [10].

Object sorting is an important problem in swarm robotics. It has many possible applications such as sorting recyclable objects and sorting different materials and products in warehouses and factories. Object sorting can also be seen as a generalization of foraging, where multiple classes of objects must be gathered at corresponding depots. As such, our algorithm can also be applied to gather a single class of objects in a desired location such as cleaning a plant floor by gathering all waste into a designated location.

This thesis investigates a sorting problem where a swarm of robots are tasked with grouping randomly distributed objects from one or more classes into separate clusters for each class.

We introduce a fully distributed algorithm for sorting randomly distributed objects (pucks) by a swarm of robots. An algorithm is introduced for target (object) selection and target conflict avoidance (two robots competing to push the same target) without any central coordination or communication between the robots. Global planning -to determine the direction to push targets towards in order to reach their end goals- is performed using goal maps generated from the distance transforms of these goals within the environment. A collision avoidance algorithm based on the concept of Buffered Voronoi Cells (BVC) [11] is also proposed, including dynamic

obstacle avoidance (other robots), static obstacle avoidance (environment obstacles), and new mechanisms for deadlock avoidance and recovery.

The development of SwarmJS, an open-source web-based swarm robot simulation platform will be detailed. The simulation platform is modifiable, modular and easily extensible; allowing it to be used as a general purpose swarm simulation platform. This platform will be used to showcase and evaluate the proposed algorithms before being deployed to actual robots.

The proposed algorithms are also deployed on a swarm of multiple small Pololu 3pi robots and a set of experiments are performed to showcase the proposed algorithms in action.

1.1 Thesis Outline

Chapter 2

The second chapter includes literature review of relevant collision avoidance and swarm research, with discussion of different approaches to solve some of the most important problems within these fields. It also includes a formal definition of the main problems we are trying to solve in this work.

Chapter 3

The third chapter describes the proposed collision avoidance algorithm. It includes some background of the theoretical principles based on which the algorithm is built, such as the Voronoi diagram. A detailed description of the collision avoidance algo-

rithm will be described here, with special emphasis on the proposed deadlock avoidance and recovery algorithms.

Chapter 4

The fourth chapter describes the proposed sorting algorithm with detailed description of each part of the algorithm, including the target selection, target conflict avoidance, control strategy, and global planning for calculating paths from targets' current positions towards their goals.

Chapter 5

The fifth chapter describes the simulation platform. It discusses the general design of the system as well as the different parts in the software, their functions, interactions and inter-dependencies. It also includes a description of how the platform can be modified and extended to implement and evaluate different robotics algorithms.

Chapter 6

The sixth chapter describes the physical swarm system's design. It includes detailed discussion of the hardware design of these robots, the software systems running on the robots, and the setup of the experimental system including the tracking and sensing simulation system.

Chapter 7

The seventh chapter details the experiments used to validate the proposed collision avoidance and object sorting algorithms and a detailed analysis of the results of the various experiments performed.

Chapter 8

The eighth chapter discusses the conclusions of our work and possible future work that can be performed on both algorithms as well as the developed simulation platform and the physical swarm system.

1.2 Contributions

Below is a list of the main contributions in this thesis:

- Distributed Collisions Avoidance Algorithm: This algorithm build on previous research where buffered Voronoi cells were used to guarantee collision avoidance. However, deadlocks have been mostly ignored or handled with simple heuristics such as the right hand rule. Our contributions lie mainly in introducing a deadlock prediction and recovery algorithm consisting of the following steps:
  - Deadlock prediction for preemptively detecting possible deadlocks and starting maneuvering actions to prevent them.
  - Deadlock recovery for choosing an alternative path that provides the most maneuverability to prevent the predicted deadlock.

  - Deadlock recovery success prediction for deciding when to end the recovery maneuver and go back to normal operation.

- Swarm Sorting Algorithm: Previous work in the literature has tackled similar sorting tasks, often using randomized motion that could result in collisions and deadlocks. Our contribution lies in proposing a swarm sorting algorithm that builds upon the proposed collision avoidance algorithm to introduce distributed sorting behaviour with guaranteed collision-free motion and minimal deadlocks. Our algorithm also handles complex environments with obstacles which was not considered in previous work.

- SwarmJS Simulation Platform: Interactivity is key in demonstrating how swarm systems work. Existing robotics simulations often run in their own environments with no way to showcase and share the results on the web. Having a web-based interactive simulation platform can be a valuable tool to many researchers where they can showcase their work in an interactive and easily shareable format. SwarmJS is an interactive web-based multi-robot simulation platform that was developed to rapidly prototype and share multi-robot algorithms as quickly and seamlessly as possible. It runs completely in the browser so it can be run on any system that supports modern web browsers, and can be easily hosted online using any hosting service that supports static websites. It allows developers to define 2D environments and populate them with multiple type of objects (static, passive, dynamic, ... etc) in addition to being open source, modular and extensible allowing researchers to easily add required functionalities as needed.

Chapter 2

Literature Review and Problem Definition

This chapter discusses existing work in the area of multi-agent robotic systems. It focuses on two areas where this thesis's main contributions lie: multi-robot collision avoidance, and object sorting by a swarm of robots. It demonstrates some existing approaches to each of these problems and details how our work fits within the general field of swarm robotics. It concludes by providing a clear definition for the problems we are aiming to solve in this work.

2.1 Multi-Robot Collision Avoidance

Multi-robot collision avoidance is a fundamental problem in autonomous robotics. It enables robots to avoid each other while navigating their local environments towards their goals. Distributed collision avoidance is particularly important for multi-robot system applications where centralized control is not wanted or not possible, such as

environment mapping, natural resource mining, industrial fault diagnosis and repair, and factory floor applications such as cleaning, sorting, and packaging.

Many existing algorithms that solve the collision avoidance problem require centralized processing [12], or extensive information such as the position, velocity, and trajectory details of other robots [13, 14, 15, 16]. This information can either be communicated between the robots or sensed and estimated locally, both of which can lead to delays, and introduce errors.

Deadlocks happen when multiple robots block each others' paths in a way that at least one robot is unable to advance along its planned trajectory to reach its goal. Another phenomenon that often arises in distributed collision avoidance algorithms is livelock, where a robot keeps alternating between a deadlock state and performing a deadlock recovery action to recover from that state.

Multi-robot collision avoidance has been extensively studied and many approaches have been developed. Velocity-based methods, which were proposed in [13], have seen wide success and adoption. They require knowledge of other agent's position, velocity and shape. The main idea behind such approaches is to repeatedly calculate the velocity obstacles, which are the set of velocities that would lead to a collision with a certain obstacle moving at a specific velocity, then generate avoidance maneuvers using feasible velocities that are outside of the velocity obstacles. Reciprocal collision avoidance (RVO) [14] extended this approach by assuming that both agents will work to avoid the collision. In this paper, Jur van den Berg et al introduced a new concept called "Reciprocal Velocity Obstacle" which assumes that the other agents in the environment are using similar collision-avoidance reasoning. This assumption helped resolve common oscillation issues with similar approaches and generate collision-free

and oscillation-free trajectories. This work is continued in [17] where Jur van den Berg et al introduced the concept of acceleration velocity obstacles (AVO) to take the agent's acceleration constraints into account when accelerating towards a new velocity. This guarantees collision avoidance with other moving obstacles by using proportional control when accelerating the agent towards a new velocity, where the acceleration corresponds to the difference between the current velocity and the new one and a new velocity is continually selected outside the values of forbidden velocities which would lead to a collision at one point in the future. The Hybrid Reciprocal Velocity Obstacles (HRVO) approach eliminates oscillations by explicitly considering that other robots sense their surroundings and change their trajectories accordingly [18]. The Optimal Reciprocal Collision Avoidance (ORCA) algorithm increases computation efficiency by reducing the problem to solving a low-dimensional linear program [15]. It assumes that all agents are using the same collision-avoidance algorithm. Each agent then calculates possible values in its velocity space by observing the velocities of other agents in the environment and marks regions (half-planes) within the velocity space that leads to collisions with these agents as 'forbidden'. Then it efficiently selects the optimal velocity from the intersections of the allowed half-planes using linear programming. If no safe velocity is feasible, the safest possible velocity is selected. Many other variations also exist in this class [19, 20, 21, 22] but they all require knowledge of the neighboring agents' velocities in addition to their positions; while the method proposed by this thesis requires knowledge of the neighbors' positions only.

Model predictive control (MPC) approaches have also been successfully used for collision avoidance. D. H. Shim et al [23] adopted ideas from the potential field

method which is another popular approach in path planning [24], [25] and presented a nonlinear model predictive control (NMPC) algorithm that provided a framework to solve discrete control problems for nonlinear systems under state constraints and input saturation. It combined stabilization of vehicle dynamics and operational constraints in trajectory generation with a potential function representing the state of information of an obstacle or another agent to the cost function. Daniel Morgan et al proposed a decentralized model predictive control approach for optimal trajectory planning of multi-agent systems [26]. It built on previous work [27] which found that using j_2-invariant relative orbits with minimal relative drift between agents can drastically reduces collisions and generate collision-free trajectories for many agents. Daniel Morgan et al proposed an optimal real-time control algorithm that changes between multiple j_2-invariant orbits to avoid collision using sequential convex programming to solve approximate path planning problems until the solution converges resulting in decentralized computations and communication between neighboring agents only. H. Zhu and J. Alonso-Mora presented a probabilistic collision avoidance by formulating a chance constrained nonlinear model predictive control problem (CCNMPC) [28].

Many other approaches have also been proposed such as deep reinforcement learning. In [29], Yu Fan Chen et al proposed a method which offloads online computations to an offline learning procedure by developing a network of values representing the time it takes an agent to reach a goal considering its current configuration and the configurations of its neighbors. The network is then used in real-time to find a feasible velocity that guarantees collision-free trajectory with all neighbors. Another approach is using gyroscopic forces and scalar potentials fields [30], and control barrier functions (CBF) [31] among others.

Our approach is based on dividing the environment into non-overlapping regions where agents can locally plan their trajectories while avoiding collisions with other robots. Zhou et al [11] use this approach to propose a multi-agent distributed collision avoidance algorithm in arbitrary dimensions which guarantees collision avoidance between multiple agents by utilizing the Voronoi diagram of the agents to partition the environment. Each agent is assigned a section of the environment (its Voronoi Cell) where it can plan its own trajectory in a receding horizon manner. They also introduce the concept of buffered Voronoi cells (BVC) which takes the robot size into account to guarantee collision avoidance with other agents.

This work was later extended to take uncertainty into account and extend this approach with probabilistic collision avoidance. Wang and Schwager [32] proposed the Probabilistic Buffered Voronoi Cell (PBVC) which aims to take the uncertainty of the on-board sensors' measurements into consideration by introducing the notion of safety levels, where multiple BVCs are calculated with different safety levels corresponding to the probability that this area lies within the robot's actual BVC. Then the PBVC calculates corresponding probabilistically safe trajectories the agent can pursue. Zhu et al [33] describe a similar approach by proposing the Buffered Uncertainty-Aware Voronoi Cells (B-UAVC), which calculates regions in the environment where the agent can travel to while remaining within its B-UAVC, given a set of chance constraints. This method guarantees that the collision probability between the agents remain below a certain threshold.

Another extension to the buffered Voronoi Cell approach was proposed by Pierson et al [34] which introduced the Weighted Buffered Voronoi tessellation to take prioritization between agents into account by using dynamic weights to bias the cells'

boundaries towards agent with lower importance leading agents with higher priority to have bigger relative cells.

All of these buffered Voronoi cell based approaches either do not specify any deadlock avoidance behavior or use simple heuristics such as the right-hand rule, which do not always perform well and tend to fail in densely populated environments. We extend these methods by proposing a deadlock avoidance algorithm with three stages for early deadlock prediction, deadlock recovery, and deadlock recovery success prediction and integrating it into a collision avoidance algorithm based on buffered Voronoi cells.

2.2 Swarm Sorting

Swarm sorting can be considered a part of a more general research area called swarm intelligence (SI) which was defined by Gerardo Beni [35] as "a 'swarm' of agents (biological or artificial) which, without central control, collectively (and only collectively) carry out (unknowingly, and in a somewhat-random way) tasks normally requiring some form of 'intelligence'". Swarm Intelligence was first introduced by Gerardo Beni and Jing Wang in 1989 [36] where they defined the intelligence of robot systems with respect to how improbable their behavior is, and described how non-trivial intelligent behavior can arise in such systems. Since then, the SI term has greatly increased in popularity and seen widespread usage in many different fields such as robotics, artificial intelligence, computation, self-organization, complexity, economics, and sociology [35].

One application of swarm intelligence is robot foraging, which was defined by

Alan F. T. Winfield [37] as "searching for and collecting any objects, then returning those objects to a collection point". Foraging can be performed by a single robot or multiple robots. In the context of swarm intelligence we are interested in foraging tasks performed by multi-robot systems, where each robot should be able to search for the objects scattered in the environment, recognize them, transport them by picking or pushing or some other means to move them towards the desired collection point. Foraging is very important in robotics because it represents a whole class of other problems where search, recognition, navigation, and transport is needed, which includes real-world applications such as cleaning, sorting, collective construction, collective transport, and search and rescue [37].

These problems have their origin in nature. Obvious examples of animals exhibiting foraging behavior are social insects like ants and bees. Ants cooperatively search for food and bring it back to their nest. They where also found to organize their brood in special patterns [38] and cluster their waste in internal nest chambers or external piles[39], all without any central decision making or coordination. This cooperation is one of the reasons behind the ecological success of social insects [40] where they have been estimated to make up approximately 75% of the world's total insect biomass [41].

Robot foraging algorithms can be very different depending on the capabilities of the individual robots within the swarm. Capabilities such as the robots' ability to access maps of their environments, localize within their environments, communicate with a central entity, communicate locally among themselves, and have powerful onboard sensors greatly change the type of algorithms used to solve a foraging task.

Deneubourg et al [42] proposed a very simple yet highly influential distributed sort-

ing algorithm inspired by how ant colonies sort their brood. This algorithm assumes that individual agents have very limited planning (random movements), communication (no communication nor hierarchical organisation), and sensing capabilities (no global mapping nor localization). The main ability of the agents is sensing the existence and type of objects directly in front of them and being able to pickup or drop such objects. Even with such limited capabilities, Deneubourg et al presented a clustering model where the agents were able to sort objects into common clusters of the same class by setting the probability of an agent to pick up or drop a specific object based on the number of objects it has recently encountered from the same class. Beckers et al [43] evaluated this model by developing a multi-robot system to gather randomly placed objects into a single cluster using the same approach. Each robot was designed with the ability to move a few objects and leave them in locations with high local density decided by a simple threshold. Maris and Boeckhort [44] studied the dynamics of this clustering process with respect to the number of objects and robots in the environment and concluded that limited mutual interference is critical for the formation of large clusters.

Object sorting was later studied by many researchers such as Melhuish et al [45], [46] and [47] where simple agents were used to group object from multiple classes so that "each is both clustered and segregated, and each lies outside the boundary of the other" [45].

Previous work within the Bio-Inspired Robotics Lab (BOTS) at Memorial University of Newfoundland have also tackled similar problems. Vardy [48] utilized the agents' vision to sense the size of visible clusters and their homogeneity to devise a guidance strategy towards objects to pick up or drop that can dramatically accelerate

the sorting process. And in another work [49], Vardy et al gave the agents the ability to localize within their environment and remember cluster sizes and locations and compare them to observed clusters, which led to a significant improvement in the rate of convergence.

The next section provides a formal definition of the problem we are investigating in this thesis, and a brief introduction into the proposed algorithms to solve this task.

2.3 Problem Definition

This thesis focuses on two main problems. The first is distributed goal seeking and collision avoidance for mobile robot systems where independent robots share the same work space. The second is sorting as a distributed task, where a swarm of simple robots are tasked to search and collect randomly distributed targets within the environment that belong to one or more different classes, while avoiding collisions with the static obstacles in the environment and avoiding collisions among themselves.

We propose a distributed sorting algorithm with no centralization nor cooperation requirements. The robots are assumed to have plans of their environments, including the desired location for each class of pucks. We also assume the robots can localize and determine the relative positions of nearby robots and pucks. This assumption could be met via an external localization system (e.g. [50]) or by self-localization by reference to sensor cues. The execution of the algorithm on each robot relies solely on the position of the robot itself and the sensed relative positions of neighboring robots and pucks. The algorithm assumes that the robots have circular bodies with the same radius, however, it can be deployed on robots of any shape by using the

radius of the robots bodies' outermost extent to define their circular area.

We also propose an underlying local planning and collision avoidance algorithm by integrating a novel deadlock avoidance algorithm into a base collision avoidance algorithm based on buffered Voronoi cells (BVC), similar to the one proposed by Zhou et al [11]. We do not introduce any additional centralization nor cooperation requirements. The proposed algorithm is fully distributed and the execution of the algorithm on each robot relies only on the relative positions of neighboring robots.

Chapter 3

Collision Avoidance

[1] This chapter provides a detailed description of the proposed collision avoidance algorithm and the basic concepts on which it is built such as the Voronoi diagram and the buffered Voronoi cell.

3.1 Background

3.1.1 Voronoi Diagram

The Voronoi diagram is defined by a set of 'sites' which partition the environment into a set of non-overlapping regions, one for each site. The region for a site s consists of all points closer to s than to any other site [51]. Each region is called a *Voronoi cell* [52]. For a finite number of sites $s_1, \ldots, s_n \ : \ n > 2$ in the Euclidean plane, the

[1] The contents of this chapter have been published as a paper in the 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2021) [1]

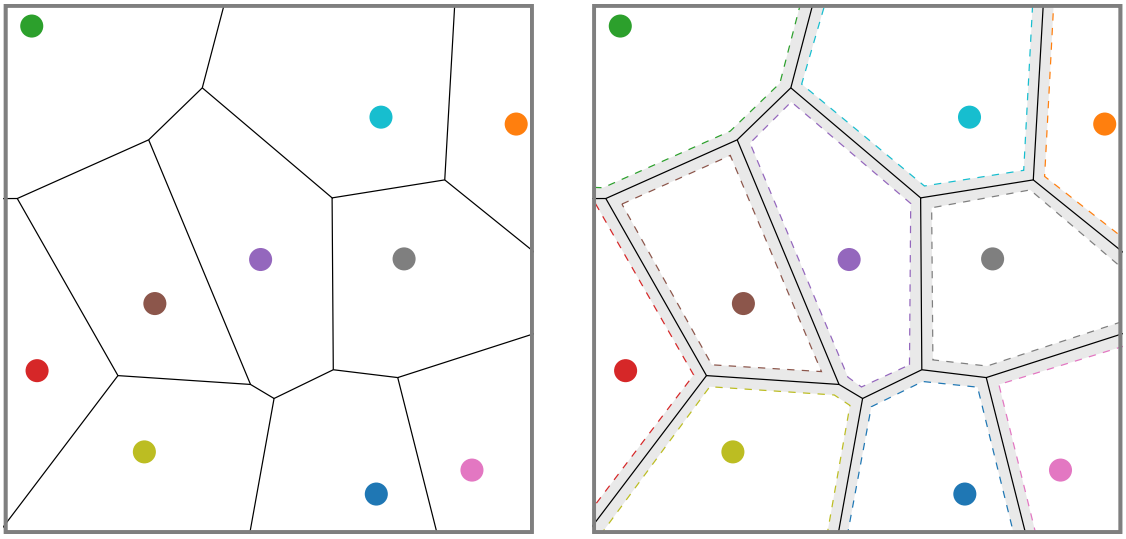


Figure 3.1: Voronoi Diagram and Buffered Voronoi Cells for 10 robots

Voronoi cell of s_i is:

$$V(s_i) = \{p \mid \|p - p_i\| \leq \|p - p_j\| \text{ for } j \neq i \ , \ j \leq n\}$$

where p_i is the location vector of s_i.

In the context of collision avoidance, a Voronoi diagram for the whole environment is generated at each time step with the robots' current positions as the sites. This divides the environment into non-overlapping regions around each robot. If each robot stays completely within its own cell, collisions can be avoided. To guarantee this condition, the Buffered Voronoi Cell (BVC) is introduced.

3.1.2 Buffered Voronoi Cell

Assume we have a set of N disk-shaped robots, with radius R, and center points $p_1, \ldots, p_n$, in a collision-free configuration (the distance between any two robots is larger than $2R$). The buffered Voronoi cell of robot i is its Voronoi cell retracted by

its radius R, so that if the center of the robot is within its buffered Voronoi cell, the entirety of its body is guaranteed to be within its Voronoi cell [11]. Figure 3.1 shows the Voronoi cell and buffered Voronoi cells of a set of 10 robots.

3.1.3 Collision Avoidance with Buffered Voronoi Cells

For a set of disk-shaped robots in a collision-free configuration, the definition of the buffered Voronoi cell guarantees that the BVC of each robot is non-empty and contains its center. If each robot's incremental movements are restricted to lie within their own BVC, then their new positions are guaranteed to be in a collision-free configuration for all future time-steps [11].

Thus, distributed collision avoidance between this group of robots can be achieved by each robot executing the following steps at each time step until they reach their own goals:

1) Calculate the Voronoi diagram and the buffered Voronoi cell based on the positions of the robot and its sensed neighbors.*

2) Calculate a local waypoint within the buffered Voronoi cell.

3) Move towards this local waypoint.

* The computational complexity of the 2D Voronoi diagram is $O(n.log(n))$ so calculating the full Voronoi Diagram for all the robots in the environment would take more time as more robots are added to the environment, possibly becoming too slow for the robots to perform in real time. However, since each robot does not have global knowledge of the positions of all robots in the environment, only a localized

subset of the full Voronoi diagram is calculated by each robot at each time-step using the detected neighboring robots. This greatly limits the possible number of Voronoi sites (robots) used when generating each of the local Voronoi diagrams allowing the algorithm to run in real-time. We were able to perform the algorithm in real-time in both the simulation and the physical robots.

Similar to other distributed collision avoidance algorithms, the lack of coordination between robots can lead to deadlocks, where robots block each others' paths in a way that prevents at least one of them from reaching its final goal. Some algorithms use simple heuristics to solve these deadlocks such as the right-hand rule, where each robot moves to its right when facing another robot [11, 34]. In another technique the robot chooses one of the nearby edges of its current buffered Voronoi cell to detour along when in a deadlock configuration [33, 11].

These heuristics perform well in sparsely populated environments but fail to reliably prevent deadlocks as the population size increases. Figure 3.2 shows comparison between the collision avoidance algorithm introduced in [11] with right-hand heuristics (in blue) and the proposed collision avoidance algorithm (in orange). Each algorithm is used to drive the robots from random starting positions towards random goals within the simulation environment described in chapter 5.

The number of robots in the environment was increased from 10 to 100 by a step of 10 robots, and 100 simulations were performed for each configuration using both algorithms, each lasting 2000 time steps. The long run-time is used to ensure that the robots that did not reach their goals were trapped in deadlocks since even for the highest number of robots used, most robots reach their final state by time step 600 (this can be clearly seen in figure 7.5 for experiment case 4, which uses the same

number of robots and random configuration as these experiments).

Figure 3.2 shows the average total distance between the robots and their respective goals after 2,000 time steps for both algorithms. We can see that when using the right-hand heuristics (in blue) all robots can reach their goals in small population sizes, but as the number of robots increases from 10 (environment occupancy rate of 1.3%) to 100 (environment occupancy rate of 13%), the total distance increases drastically as a result of more and more robots getting stuck in deadlocks. While for the proposed algorithm (in orange), the total distance barely increases as more robots are added to the environment and at 100 robots only reaches a tenth of its corresponding value when using the right-hand heuristics.

3.2 Deadlock Prediction and Recovery Algorithm

In this section, we will first outline the proposed collision avoidance algorithm, then describe each of its stages.

3.2.1 Collision Avoidance Algorithm Overview

We add to the assumptions given in Section 3.1.2 that the set of n disk-shaped robots starts in a collision-free configuration and each robot is assigned a goal g, with the goals also being in a collision-free configuration (when the robots reach them). With these assumptions in mind, We propose the following distributed collision avoidance algorithm.

At time t, each robot senses its own position p_t, and the positions of neighboring robots, forming the set N_t, where N_t is the set of sensed neighbors' positions at

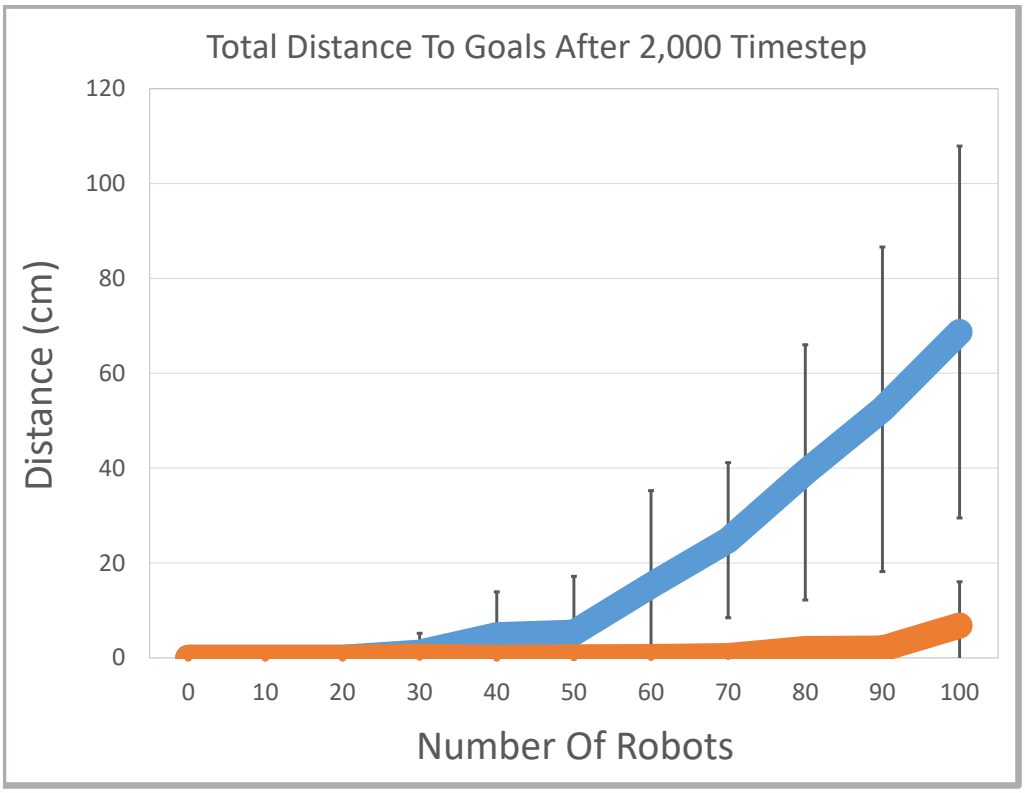


Figure 3.2: Deadlocks With Respect to Environment Density

Mean and standard deviation of remaining total distance to goals after 2,000 time steps for 100 simulations for the right-hand heuristics (blue) and the proposed algorithm (orange).

time t. Then it calculates its current buffered Voronoi cell (BVC) B_t based on these measurements. A local waypoint g_t within the current BVC is then selected. Once g_t is found, the control input u_t is calculated to drive the robot towards it for that time step. This process is repeated until the robot reaches its goal within an acceptable distance ϵ such that

$$\|g - p_t\| < \epsilon.$$

Algorithm 1. Collision Avoidance Algorithm

while $\|g - p_t\| > 0$ **do**

Update p_t (current position)

Update N_t (current neighbors' positions)

Calculate B_t (current buffered Voronoi cell)

Calculate g_t^* (closest point in B_t to goal)

if $g \in B_t$ **then** # Goal is Within BVC

Set $g_t = g$

else if $Recovering from Deadlock = True$ **then**

Perform Deadlock Recovery Step

else if $DeadlockDetected$ or $DeadlockPredicted$ **then**

Initiate Deadlock Recovery

else

Set $g_t = g_t^*$

end if

Calculate u_t to drive robot directly towards g_t

end while

By default, the waypoint at time t, g_t, is the point within the current buffered Voronoi cell B_t that is closest to the goal, we will call this special point g_t^*. However, if a deadlock is detected or if this waypoint is predicted to lead to a deadlock configuration according to the proposed deadlock prediction step, or if the robot is still recovering from previous deadlock, a different waypoint is selected in the deadlock recovery step. An overview of this algorithm is given in Algorithm 1 3.2.1.

3.2.2 Deadlock Prediction

The proposed deadlock avoidance algorithm consists of three stages: deadlock prediction, deadlock recovery, and deadlock recovery success prediction. Early deadlock prediction and recovery has many benefits over sensing and reacting to deadlocks; it helps steer the robots clear of each other and increase the safety distances between them since sensing a deadlock often means the robots are already at the minimum distance possible without causing a collision -as proved by condition (1) in section 3.2.2-. The lack of deadlock prediction can also lead to extreme congestion where many robots get too close to each others leaving the robots in the center stuck while waiting for outside robots to detect and recover from their deadlock configurations. Deadlock prediction can reduce this risk by predicting possible deadlocks caused by congestion and maneuvering towards clearer paths. This difference can be clearly seen in the simulations discussed in section 7.1.1. Deadlock recovery success prediction ends deadlock recovery maneuvers as soon as it predicts that the deadlock has been successfully avoided -as explained in section 3.2.4, which helps decrease the distance and time to reach the goal by reducing the overshoot during these maneuvers.

The proposed deadlock prediction algorithm is based on the position of the waypoint g_t at each time step with respect to the positions of neighboring robots and the goal g.

We know that a deadlock can only occur when the robot is located at g_t^*, the closest point to a robot's goal within its buffered Voronoi cell, otherwise, there would be a closer point to the goal, and according to Algorithm 1 3.2.1, the robot would move towards that point for at least one time step. Furthermore, Zhou et al [11] prove that this point can only be a) A vertex of the BVC or b) On an edge of the BVC such that a line from p_t to g_t^* is perpendicular to this edge, and the center of the neighboring robot who shares the Voronoi edge must also be located on the same line. Case (b) is trivial due to the collinear requirement which can be broken with any deviation to either side by either robot, thus we are only interested in predicting the case (a) where g_t^* is a vertex of the BVC.

Since this deadlock configuration only occurs when the robot is located at g_t^*, and g_t^* is a vertex of the BVC, then by definition of the BVC, the vertex is connected with two edges shared with two neighboring robots n_i, n_j positioned at p_{i_t} and p_{j_t}, the distance between the robot and these neighboring robots must be $2R$:

$$\|g_t^* - p_{i_t}\| = \|g_t^* - p_{j_t}\| = 2R \qquad (1) \tag{3.1}$$

Proposition 1: For condition (1) to be satisfied, the distance between n_i and n_j, must be equal to or less than $4R$:

$$\|p_{i_t} - p_{j_t}\| \leq 4R \qquad (2) \tag{3.2}$$

Proof: Let us assume that $\|p_{i_t} - p_{j_t}\| > 4R$, then for any point $x \in \mathbb{R}^2$ not on the

line segment between p_{i_t} and p_{j_t}:

$$\|x - p_{i_t}\| + \|x - p_{j_t}\| > \|p_{i_t} - p_{j_t}\| \quad \forall\ x \in \mathbb{R}^2$$

then,

$$\|x - p_{i_t}\| + \|x - p_{j_t}\| > 4R \quad \forall\ x \in \mathbb{R}^2$$

If we choose x as a point where,

$$\|x - p_{j_t}\| = 2R$$

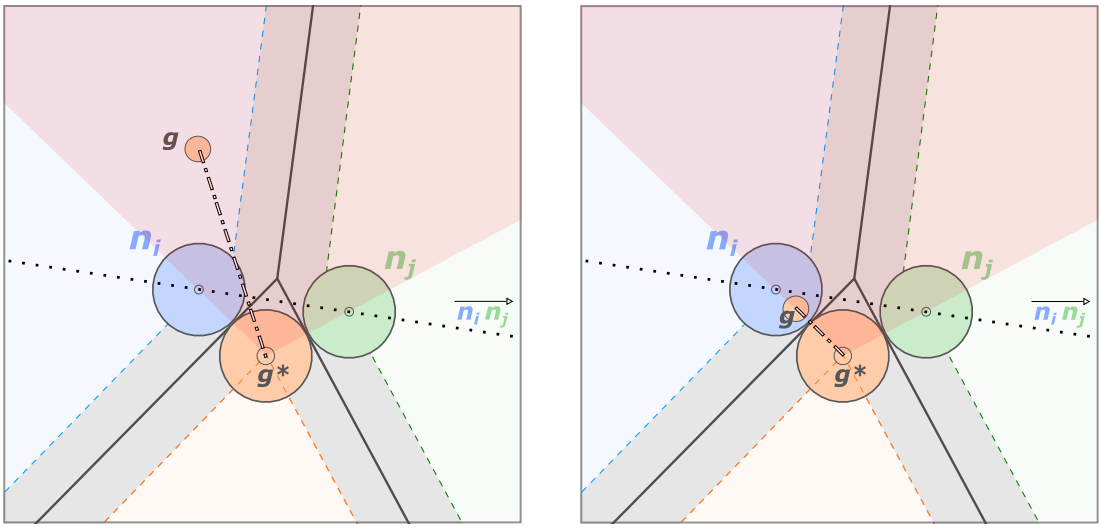
then it must be that,

$$\|x - p_{i_t}\| > 2R$$

which contradicts condition (1).

When condition (1) is satisfied, we are only interested in predicting deadlock configurations where the current waypoint g_t^* and the goal g are on different sides of the line between the neighbors n_i and n_j. Otherwise the deadlock would be a result of one of the neighboring robots being positioned too close to the goal, preventing the robot from reaching it. This means that the robot is already as close to the goal as possible without colliding with other robots and performing a deadlock recovery maneuver would drive the robot further away from the goal, so it would be better for the robot to remain still and wait for the neighboring robot to move away allowing it to continue towards its goal.

Figure 3.3 shows an example of each of these cases. The red sector corresponds to the points where a goal would lead to a deadlock according to condition (1), since the closest point to it within the BVC would be the vertex. Otherwise, the closest



(a) (b)

Figure 3.3: Examples of Deadlock Configurations

Condition (3) is satisfied in (a) and but not satisfied in (b), while condition (1) is satisfied in both cases. It is clear that a detour around one of the neighboring robots in the first case would be helpful, but not in the second case. The red sector corresponds to the points where a goal would lead to a deadlock according to condition (1).

point to the goal would be other than the vertex and the robot can move towards it according to Algorithm 1 3.2.1 without causing a deadlock.

For two robots $n_i(x_i, y_i)$ and $n_j(x_j, y_j)$, we can find on which side of line $\overrightarrow{n_i, n_j}$ a given point $g(x_g, y_g)$ is located by calculating the following term:

$$d = (x_g - x_i)(y_j - y_i) - (y_g - y_i)(x_j - x_i)$$

The sign of d indicates on which side of the line the point lies and $d = 0$ means that it lies exactly on the line.

We can then define condition (3) as: the local waypoint at time t, $g_t^*(x_{g_t^*}, y_{g_t^*})$ and the goal $g(x_g, y_g)$ must be on different sides of the line $\overrightarrow{n_i, n_j}$ connecting the two neighbors $n_{i_t}(x_{i_t}, y_{i_t})$ and $n_{j_t}(x_{j_t}, y_{j_t})$. Then can be formulated as:

$$\begin{aligned} &sign((x_{g_t^*} - x_{i_t})(y_{j_t} - y_{i_t}) - (y_{g_t^*} - y_{i_t})(x_{j_t} - x_{i_t})) \neq \\ &sign((x_{g_t} - x_{i_t})(y_{j_t} - y_{i_t}) - (y_{g_t} - y_{i_t})(x_{j_t} - x_{i_t})) \end{aligned} \tag{3.3}$$

Using conditions (1), (2), and (3) we propose our deadlock prediction algorithm to be performed for g_t^* at each time step as:
Find $N_t{}'$, the set of neighbors within kR distance to g_t^*, this is similar to condition (1), after replacing 2 with constant $k \geq 2$ (we used $k = 3$) to achieve earlier prediction of deadlocks:

$$N_t{}' = \{n_i \in N_t \mid \|g_t^* - p_{n_i}\| < kR\}$$

We then calculate the number of neighbors satisfying this condition as $|N_t{}'|$, the cardinality of set $N_t{}'$. If $|N_t{}'| < 2$, we predict no deadlock will occur. Otherwise, We look for neighbor pairs n_i, n_j in $N_t{}'$, within $4R$ of each other and for each pair we

check whether g_t and g_t^* are on opposite sides of the line $\overrightarrow{n_i n_j}$. If this condition is satisfied for any pair, then a deadlock is predicted.

3.2.3 Deadlock Recovery

We propose a multi-step deadlock recovery algorithm; when a deadlock is predicted or detected, the robot goes into deadlock recovery state, during which multiple maneuvers are performed until deadlock recovery success is predicted or the maximum number of maneuvers M is reached.

During each maneuver, simple heuristics are used to select a detour waypoint within the current buffered Voronoi cell, $g_t \in B_t$ (selection process is detailed below). The robot then moves towards this waypoint at each time step t until one of following condition occurs:

a) The deadlock recovery success prediction algorithm predicts that deadlock recovery has succeeded, in which case the deadlock recovery process is terminated, and the robot goes back to normal operation state.

b) The waypoint g_t is reached: then a new maneuver is re-initiated, and a new detour waypoint g_t is calculated if the maximum number of maneuvers M has not been reached. Otherwise, the deadlock recovery process is terminated, and the robot goes back to normal operation state.

c) The waypoint g_t goes outside of the new buffered Voronoi cell (as a result of the robot moving and its BVC changing according to its new position and the new positions of its neighboring robots): then a new maneuver is re-initiated, and a

new detour waypoint g_t is calculated if the maximum number of maneuvers M has not been reached. Otherwise, the deadlock recovery process is terminated, and the robot goes back to normal operation state.

There are alternate approaches which select the detour waypoint as a point on one of the adjacent edges to the vertex where deadlock was detected [11, 34]. While this works well in some cases, it leads to poor performance in many others. For example, when the adjacent edges, and consequently, the distance between the deadlock position and the detour waypoint are very short, the robot fails to recover from the deadlock and instead goes into a livelock state where it keeps alternating between these two points.

To increase the likelihood of successful deadlock recovery, we propose a detour point that takes into account the shape of the buffered Voronoi cell and the positions of the neighboring robots with respect to the goal. We call this the outermost point o_t and define it as the point within the BVC that is furthest from the line between the current position of the robot and the goal, $\overrightarrow{p_t g}$ as shown in Figure 3.4. We formulate o_t as:

$$o_t = v_i \in V_{BVC} \mid \|v_i - v_i'\| \geq \|v_j - v_j'\| \; \forall \; v_j \in V_{BVC} \; , \; v_i \neq v_j$$

where:

- V_{BVC} is the set of all vertices of the current BVC, B_t, for the first maneuver when a deadlock recovery is initiated. For all other maneuvers, only the vertices of the B_t that are on the same side of $\overrightarrow{p_t g}$ as the initial o_t was in the first corresponding maneuver (to keep the robot heading in one direction for the whole recovery).

- v_i is one of the vertices of V_{BVC}

- v_i' is the projection of v_i on $\overrightarrow{p_t g}$

To decrease overshooting, we can use a point between the robot's current position and o_t as the detour point g_t:

$$g_t = (1 - c)p_t + c \; . \; o_t$$

where c is the proportion of the distance for which the robot moves towards the outermost point, o_t.

3.2.4 Deadlock Recovery Success Prediction

Deadlock recovery success prediction works by using heuristics based on the conditions that predicted the deadlock and the locations of the robot and neighboring robots; the robot tracks the position of neighbors near the location of the last deadlock and detects whether the robot's maneuver has caused a deadlock condition to be broken.

From deadlock conditions (1) and (3) we can deduce that in a deadlock configuration where the robot is located at g_t^*, and g_t^* is a vertex of the BVC; the neighboring robots n_1 and n_2 who share the Voronoi edges connected to g_t^* must be on different sides of the line $\overrightarrow{p_t g}$ connecting the robot position and its goal. We use the failure of these conditions (1) and (3) as prediction of success of this deadlock maneuver.

We propose a deadlock recovery success prediction condition requiring all robots within distance $kR \;\; : \;\; k \geq 2$ (where k is the same constant used in the deadlock prediction algorithm) to be on the same side of the line $\overrightarrow{p_t g}$. Furthermore, the distance between any of these robots and the line $\overrightarrow{p_t g}$ must be larger than the radius of the robot.

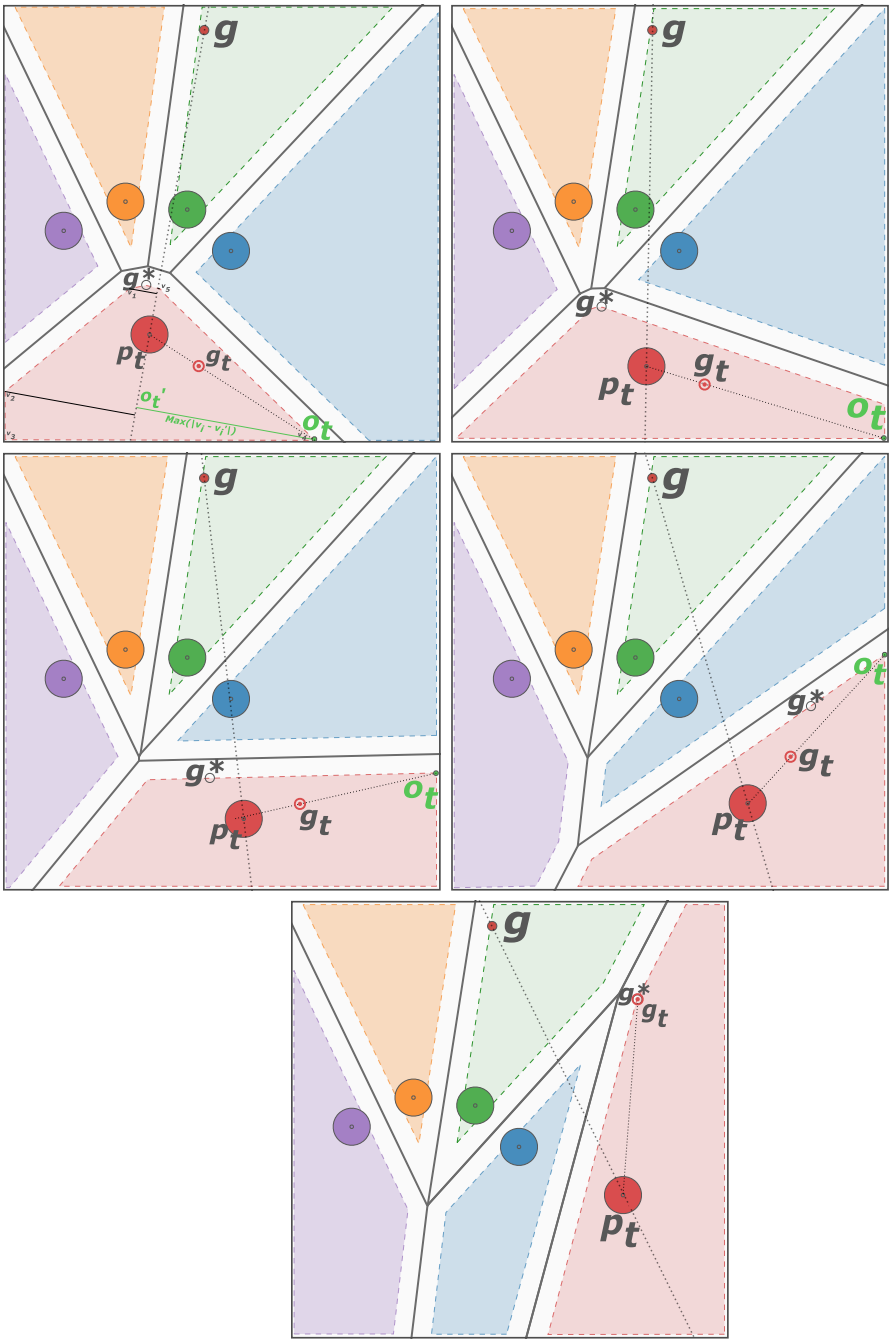


Figure 3.4: Example of Deadlock Prediction and Recovery

Step 1 shows how the outermost point o_t and the detour point g_t were selected after a deadlock configuration was predicted at g_t^*. Steps 2-4 show the following maneuvers until normal operation is resumed on step 5 ($g_t = g_t^*$).

For a robot at position p_t, moving towards goal g, and currently recovering from a deadlock that occurred at position p_d; we refer to the set of all neighbors at time t as N_t, and define $N_t{}^*$ as the subset of neighbors that are located within kR of p_d:

$$N_t{}^* = \{\ n_i \in N_t\ :\ \|p_d - p_{ni}\|\ < kr_s\}$$

We define s_i as the side of line $\overrightarrow{p_t g}$ on which neighbor n_i is located,

$$s_i = sign((x_{ni} - x_{p_t})(y_g - y_{p_t}) - (y_{ni} - y_{p_t})(x_g - x_{p_t}))$$

and the distance between neighbor n_i and line $\overrightarrow{p_t g}$ as,

$$m_i = \|p_{ni} - p_{ni}{}'\|,$$

where $p_{ni}{}'$ is the projection of p_{ni} on the line $\overrightarrow{p_t g}$.

We can then formulate the proposed conditions for deadlock recovery success prediction as:

$$s_i = s_j\ \forall\ i, j\ |\ n_i, n_j \in N_t{}^*\ and\ m_i > R\ \forall i\ |\ n_i \in N_t{}^*$$

Summary

In this chapter we detailed the proposed collision avoidance algorithm including a background look at some of the theoretical concepts it is built upon, such as the Voronoi Diagram; how collision avoidance is guaranteed; and the proposed deadlock prediction and recovery algorithm, describing each of its three stages: deadlock prediction, deadlock recovery, and deadlock recovery success prediction. The proposed collision avoidance algorithm runs slower than the generic algorithm since it adds

more steps for predicting and avoiding deadlocks while the generic algorithm lacks similar steps. However, the measured difference between the two algorithms is minimal, only around 10% according to our bench marking. Chapter 7 provides further details about how this algorithm was implemented and validated in both simulations and real-life experiments.

In the next chapter we propose a distributed sorting algorithm that builds upon this collision avoidance algorithm to enable multiple robots to gather objects from different sets into specific areas in the environment.

Chapter 4

Swarm Sorting Algorithm

In this chapter we will detail the proposed distributed sorting algorithm, with the goal of sorting randomly positioned objects in the environment into specific goal areas for each class of targets. We introduce algorithms for target selection, target conflict avoidance, control strategy, and global planning. The proposed algorithm uses the previously detailed collision avoidance algorithm in section 3.2.1 as the underlying local movement planning algorithm. We assume that the environment will contain static obstacles so at the end of this chapter we propose additional mechanisms for avoiding static obstacles using the same concept of buffered Voronoi cells.

4.1 Sorting Algorithm

In this section we will discuss the sorting algorithm including the following sub-problems:

- Target selection: how the robots choose a suitable target to push.

- Target conflict avoidance: how robots avoid competing for the same target.
- Control strategy: how to control the robot in order to push a target towards its goal.
- Target direction: how robots decide which direction to push each target towards.
- Default behavior when no suitable targets can be found: orbit the environment.

For this sorting problem we are assuming that the environment is static, the robots have a map of the environment, and know the desired final position for each class of targets. We also assume that the robots can localise themselves within their environment and can sense the position of nearby robots, targets, and static obstacles. The algorithm is fully distributed; each robot works independently without any need for communication or central coordination.

4.1.1 Default Behavior - Orbiting the Environment

The default behavior for robots with no suitable targets in range is to circle the environment looking for new targets, this behaviour is inspired by other algorithms such as [53], where the robots orbit the environment and push objects inwards to build enclosures with specific shapes. Similar to that algorithm, our robots move targets (pucks) towards their goals only by pushing them. This can be done optimally when the puck lies between the robot and its goal. With the robot directly behind the puck with respect to the puck's goal, it can push the puck directly towards its goal (from the outside in, with respect to the environment). To achieve this optimal positioning while orbiting the environment looking for pucks, the path length of the orbit around

the environment is continuously expanded, causing the robots to slide further towards the edges as they orbit the environment.

Instead of repeatedly calculating the environment orbit route, we can use the robot's knowledge of the environment to generate a static map for the orbit route. This map will contain, for each position in the environment, the direction towards which the robot should move when located at that position, if no suitable targets are found. This can be represented as a direction vector or as a local goal point. In this section, we will detail how this map is generated using the fast marching method [54], which describes the propagation of a closed surface as a function of time.

The first step is to produce the closed surface that we want to propagate, which is the outermost orbit, where the robot moves directly along the border of environment, we will call it the orbit border. This can be generated directly from the polygon representing the shape of the environment boundary and the static obstacles, after shrinking the resulting polygon by a safety distance that equals the radius of the robot R.

We can define this surface using a discretized grid representation of the environment such as a binary array of the same size as the environment. We initialize all cells in this array with a default value ($V_0 = -1$). We use another value ($V_1 = 1$) for all points within the environment boundary that are at least R distance away from the border of the environment and from any static obstacle. The surface can then be defined as the contour resulting between these two sections of the map. Figure 4.1 shows how the orbit border can be defined for a simple map with a single static obstacle.

We then calculate the distance to this orbit border from every cell in the environ-

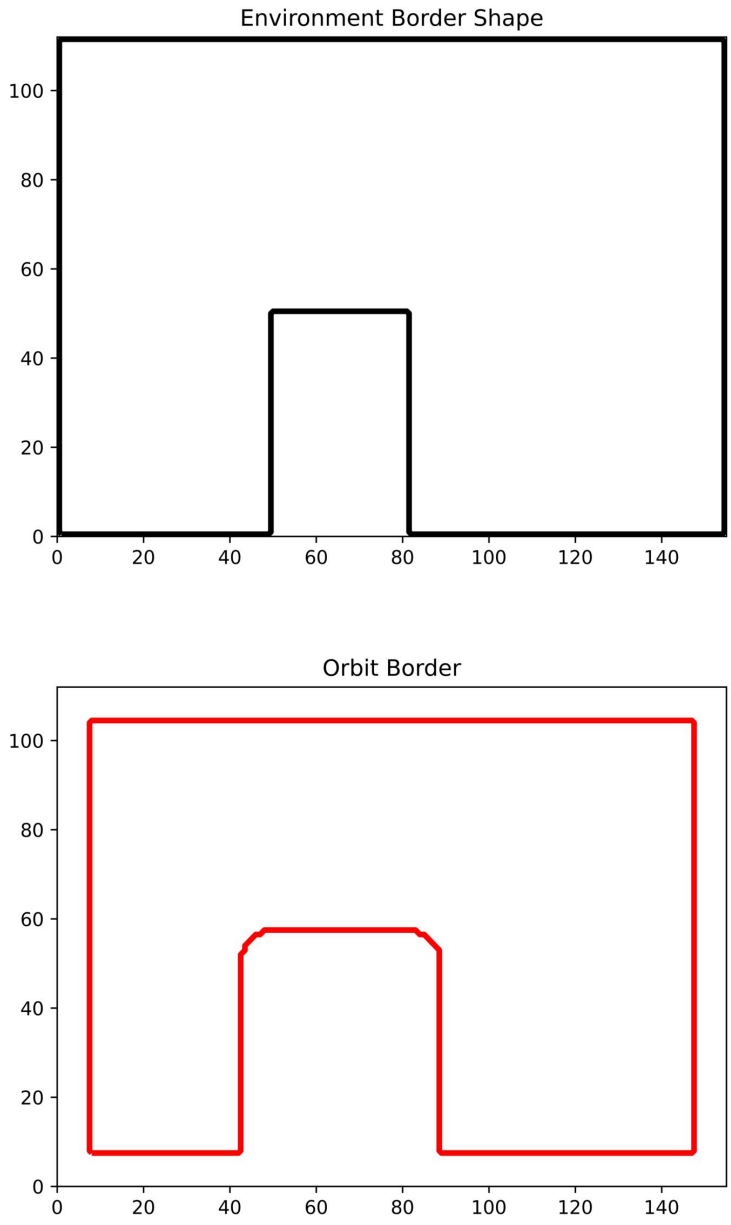


Figure 4.1: Environment Shape and Resulting Orbit Border

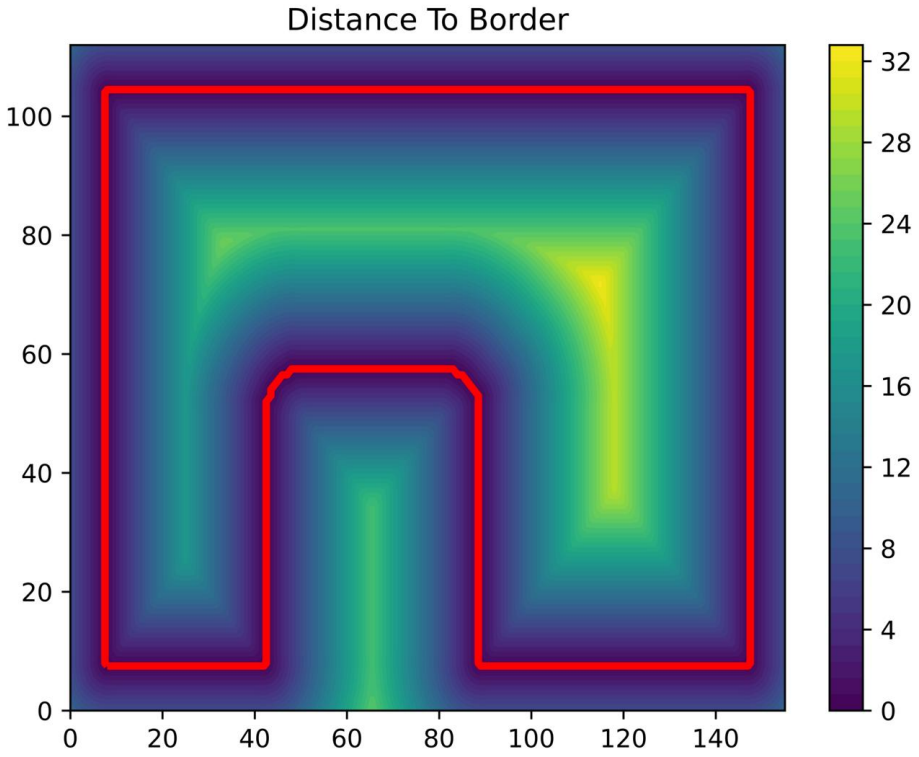
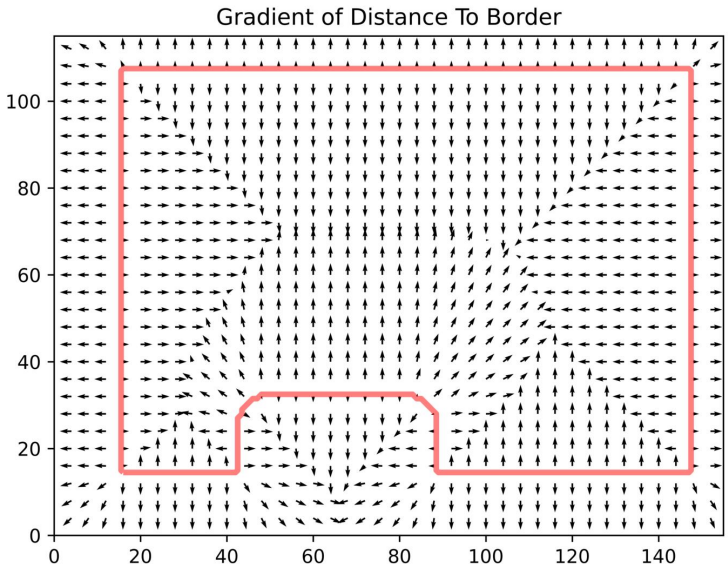


Figure 4.2: Distance to Orbit Border

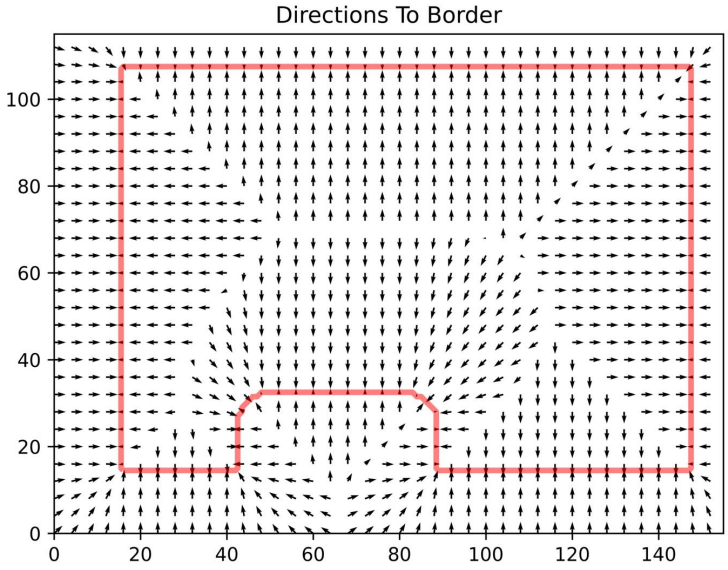
This figure shows the distance to orbit border array calculated using the fast marching method. Distance is denoted with colors where darker colors (blue shades) represent small distances near the orbit border while lighter colors (green and yellow shades) represent longer distances further away from the border.

ment using the fast marching method. This generates an array with the same size as the discretized environment array, where each cell's value is the closest distance from the cell's position to the orbit border. Figure 4.2 shows the distance array generated for the environment shown in figure 4.1.

This array can be used to find the shortest path from any point in the environment



(a) Gradient vectors pointing to increasing distances from orbit border



(b) Vectors pointing to decreasing distances to orbit border

Figure 4.3: Gradients of Distance Function and Direction to Orbit Border Vectors

to the orbit border by walking downhill (following neighbors with minimal distances). First, we calculate the gradient of the distance array on both the X and Y axes resulting in a gradient vector for each point in the environment that describes the direction and rate of change of the fastest local increase in distance. figure 4.3 shows the gradient vectors generated from the distance array shown in figure 4.2.

As we want to find the directions towards the orbit border (towards decreasing distances), we reverse the direction of the gradient vectors (multiply the vectors by -1). This produces an array of vectors pointing along the shortest path to the orbit border from any point in the environment as shown in figure 4.3.

In order to introduce the environment orbiting behavior in the same direction on both sides of the orbit border; we define the orbit direction vector $\overrightarrow{d_r}$ for a vector denoting the direction towards the border $\overrightarrow{b(x,y)}$ by rotating $\overrightarrow{b(x,y)}$ by 90° clockwise for all points within the orbit border:

$$\overrightarrow{d_r} = (y, -1x)$$

and by 90° anti-clockwise for all points outside of the orbit border:

$$\overrightarrow{d_r} = (-1y, x)$$

By adding the two vectors $\overrightarrow{b}$ and $\overrightarrow{d_r}$ for all points in the environment and smoothing the resulting array using a Gaussian filter we reach a smooth orbiting behavior for the entire map where the robot circles the environment on an orbit with an ever increasing radius, moving closer to the boundary until it reaches the orbit border, where it keeps circling the environment on this border along side the edge of the environment until a suitable puck is found. The corresponding vector field for the environment shown in figure 4.1 is shown in figure 4.4.

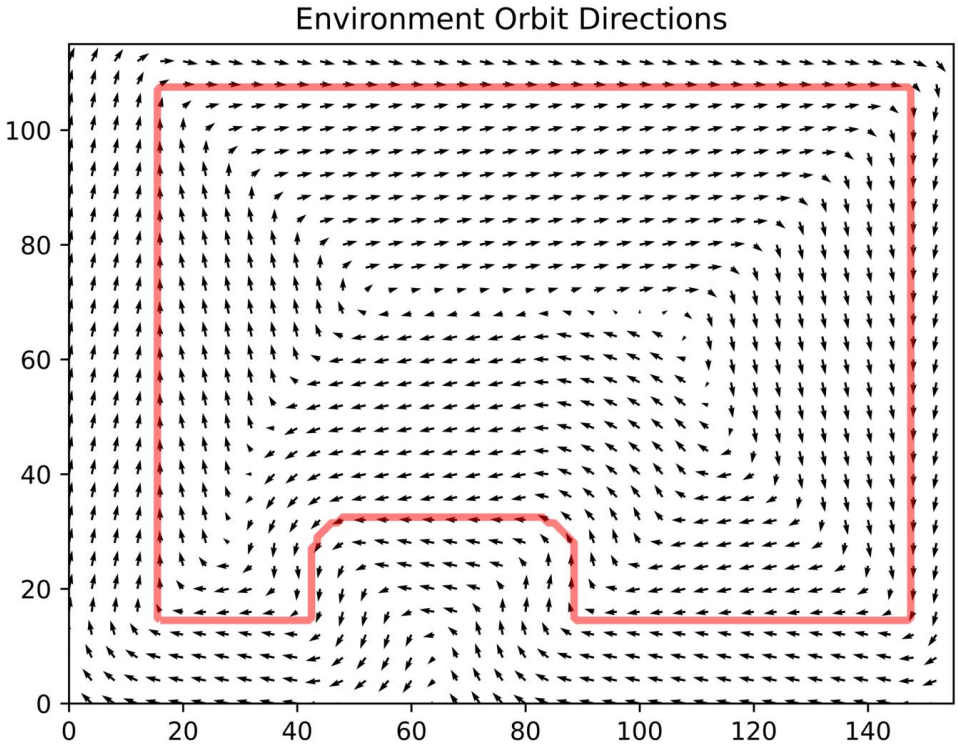


Figure 4.4: Environment Orbits

This figure shows the paths on which a robot orbits the environment using direction vectors pointing along orbits with increasing radii up to the orbit border.

This map can be calculated once either offline or on startup of the robot, and can be dynamically regenerated only when necessary, such as in cases where static obstacles change during operations. The map can be accessed quickly in real-time, providing an efficient default path planning for robots when no suitable targets are detected.

4.1.2 Target Selection and Target Conflict Avoidance

In this section, we will refer to the objects in the environments which the robots need to sort into different areas as targets or pucks. At each time step t, the robot detects all nearby targets within the sensing range, we can refer to the set of all detected nearby targets as Q_t.

In order to avoid target conflicts, where multiple robots compete to push the same target, only targets located within the robot's current Voronoi cell V_t are considered, which prevents multiple robots from pursuing the same target since the voronoi cells do not overlap. We define T_t as the set of detected targets that lie within the current voronoi cell:

$$T_t = \{\ n_i \mid n_i\ \in\ Q_t\ ,\ p_i\ \in\ V_t\ \}$$

Where: p_i is the position of target n_i.

In the case where $T_t = \phi$, when no target in the sensing range lies within the current voronoi cell V_t, the robot falls back to orbiting the environment as described in section 4.1.1.

For each target n_i in T_t, the robot retrieves $\overrightarrow{d_i}$, the direction towards which the target should be pushed -using the algorithm described in section 4.1.4- and calculates

a local goal g_i for this target based on $\overrightarrow{d_i}$ and p_i:

$$g_i = p_i + \overrightarrow{d_i}$$

Then the robot calculates two metrics for each target n_i that will be used to determine which target to pursue: the first is the distance d_i between the robot's current position P and the target's current position p_i. The second is the angle a_i which is calculated from the angle $\angle P p_i g_i$ between the robot's current position P, the puck's current position p_i, and the puck's local goal g_i.

The angle a_i for each target is then converted to the $[0° - 180°]$ range as follows:

$$a_i = |\angle P p_i g_i - 180|$$

The target with the smallest angle a_i is then selected as the best target W_t, since targets with small angles are best positioned to be pushed to their local goal from the robot's current position as explained in 4.1.3. If multiple targets have the same angle, the one closest to the robot is selected as the best target W_t:

$$W_t = n_i \mid (a_i < a_j) \; or \; (a_i = a_j \; and \; d_i \leq d_j) \; \forall \; n_i, n_j \in T_t$$

4.1.3 Control Strategy

After the robot selects its target, it tries to find a trajectory that allows it to push that target towards its goal. We propose two trajectories based on the angle a_i of the target as described in section 4.1.2.

If the angle a_i is very small, smaller than a specific threshold c_1 ($c_1 = 15°$ in our experiments), then the robot is already well positioned to directly push the target

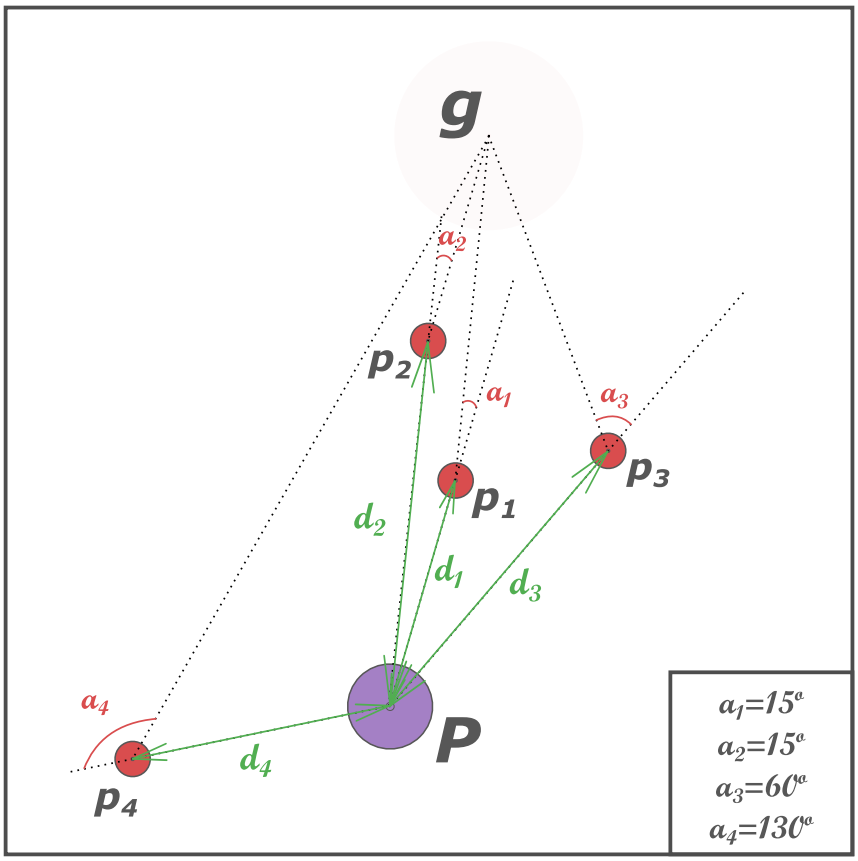


Figure 4.5: Target Selection

Figure shows how two metrics, distance d_i and angle a_i, are used for selecting the best target. In this figure, pucks 1 and 2, located at p_1 and p_2 have the smallest angles ($a_1 = a_2 = 15°$) but since puck 1 is closer to the robot ($d_1 \leq d_2$), puck 1 is chosen as the best target.

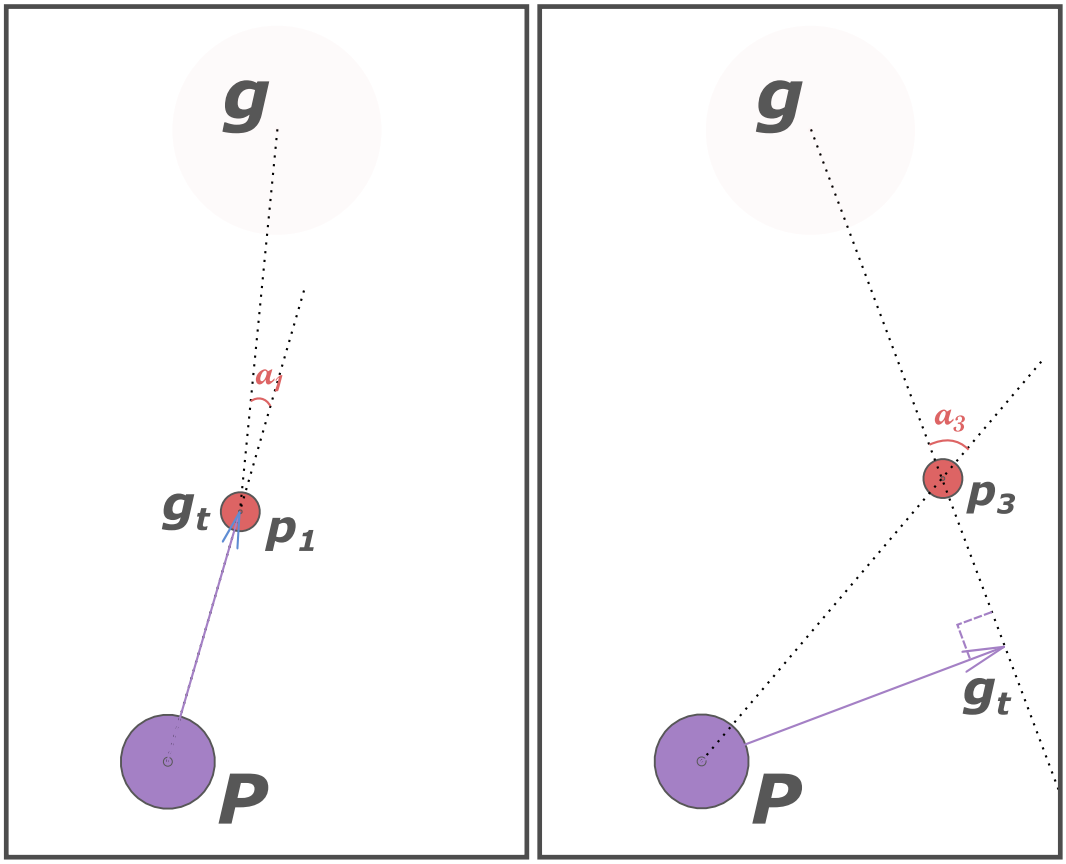


Figure 4.6: Control Strategy

Possible trajectories for a robot depending on the selected target's position:

Left: For target p_1, $a_1 = 15° \leq c_1 = 15°$ so the current position of the puck is set as the robot's local goal g_t.

Right: For target p_3, $c_1 = 15° \leq a_3 = 60° \leq c_2 = 75°$ so the robot's local goal g_t is set as the point closest to the robot's current position P from the line connecting the puck's position p_3 and its goal g.

towards it goal. Thus, the robot moves directly towards the target by setting its local goal g_t as the target's current position p_i.

Otherwise, if the angle a_i is larger than c_1 but is still smaller than another threshold c_2 ($c_2 = 75°$ in our experiments), then the robot needs to perform a maneuver to better position itself to be able to push the target towards its goal. This maneuver can be performed by selecting the local goal g_t as the point g'_i that is closest to the robot's current position P from the line connecting the target's current position and the target's current local goal $\overrightarrow{p_i g_i}$. This causes a_i to decrease as the robot gets closer to the line $\overrightarrow{p_i g_i}$ and the previous threshold c_1 to be eventually crossed when $a_i < c_1$, the robot can then move directly towards the target to push it towards its goal as explained in the previous paragraph. Figure 4.6 shows both of the previous scenarios.

If the angle a_i is larger than c_2 then the current target is not well positioned to be pushed towards its goal from the current robot position, because the robot would need to perform a complicated maneuver to decrease a_i (get behind the target with the respect to the target's goal). An example of this configuration is puck 4 shown in figure 4.5. In this case, the robot ignores this puck and falls back to orbiting the environment. The default environment orbit behaviour described in section 4.1.1 drives the robot towards the environment boundaries; this causes the robot's orbit to expand with time and leads the robot in future encounters with the puck to be better positioned to push it from the outside in towards its goal (if the robot doesn't diverge from its orbit by finding another suitable target to pursue).

4.1.4 Global Planning

The previously discussed behaviors allow the robots to navigate their local environment and select and push suitable targets towards their goals. However, choosing a suitable target assumes the robots know the direction towards which each target should move from its current position to get closer to its goal. This might be trivial in environments with no static obstacles, where the targets can always be pushed directly towards their goals. But with static obstacles present, it requires global planning for finding a suitable route a target can take from its current position towards its final goal. This can be extremely time consuming if the path has to be recalculated for each target at each time step. We will address this issue by generating global goal maps for each target group. This map will contain, for each position in the environment, the direction towards which a target located at that position should move to get closer to its goal. This can be represented as direction vectors or as local goal points. In this section, we will detail how these maps are generated using the fast marching method [54] similar to how the environment orbit map was generated in section 4.1.1, but with the addition of considering the propagation speed within the environment.

The first step is to produce a discretized grid representation of the environment such as an array of cells of the same size. Then we define the closed surface that we want to propagate, which is the goal position in our case. This is achieved by generating a binary array of the environment where all cells representing empty space have the same value $V_0 = -1$, while the cells containing the goal has a different value $V_1 = 1$, the closed surface is thus defined by the contour around the cells with value

V_1. We call this array the goal mask. Figure 4.7 shows a goal mask where the goal position $(75, 65)$ is enclosed by a circle separating it from the rest of the empty space.

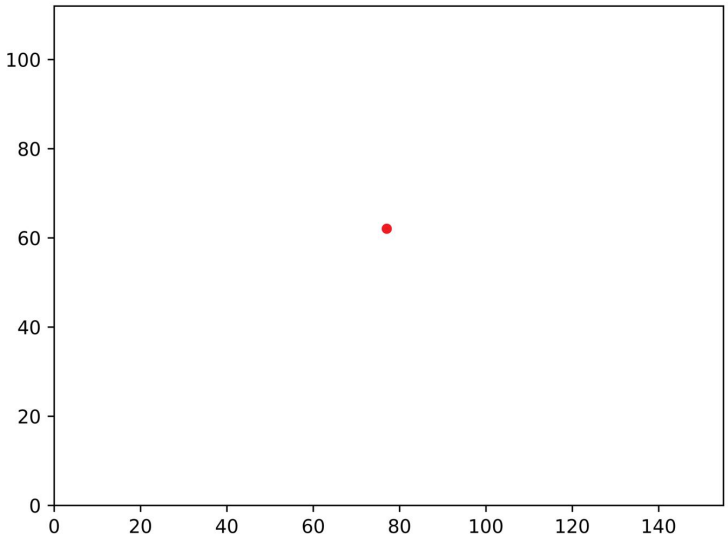


Figure 4.7: Goal Mask

This figure shows the goal mask where the goal cell has a value of $V_1 = 1$, while all other cells representing the empty space have a different value $V_0 = -1$. The red contour separates the goal from the empty space and acts as the closed surface in the fast marching method.

The next step needed for the fast marching algorithm is defining the speed function, which describes the propagation speed at each cell in the environment. Since we want to use the fast marching method to calculate the distance to the goal, we can set that speed to $s = 1$ in empty space, and denote static obstacles in the environment by setting the speed in their corresponding cells to zero. Figure 4.8 shows a speed

mask where the static obstacle cells have a propagation speed of 0 while the rest of the cells have a speed of 1.

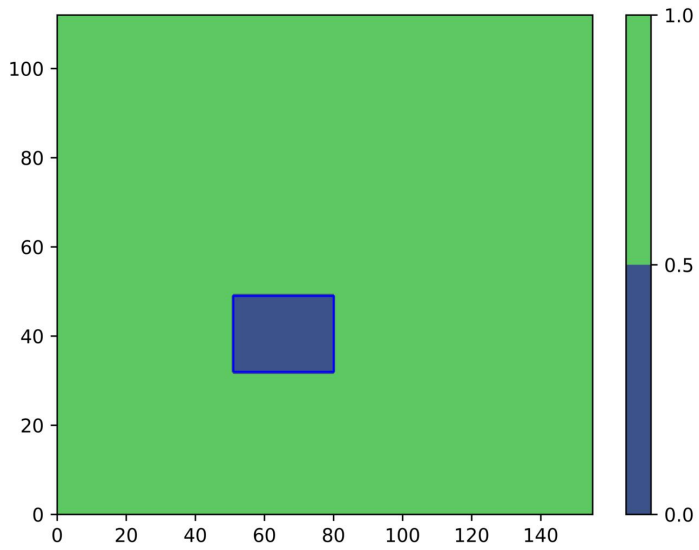


Figure 4.8: Speed Mask

The figure shows the speed mask where the static obstacles' cells (in blue) have a value of 0, while all other cells representing the empty space (in green) have speed value of 1.

We then calculate the distance to goal array using the fast marching method, this generates an array with the same size as the discretized environment array with each cell's value set to the total distance of the shortest path from the cell's position to the goal. Cells that represent static obstacles where the propagation speed is set to 0 and no path to the goal can be calculated are represented with a special value such as *Null*. Figure 4.9 shows this array calculated using the goal and speed masks shown in figures 4.7 and 4.8.

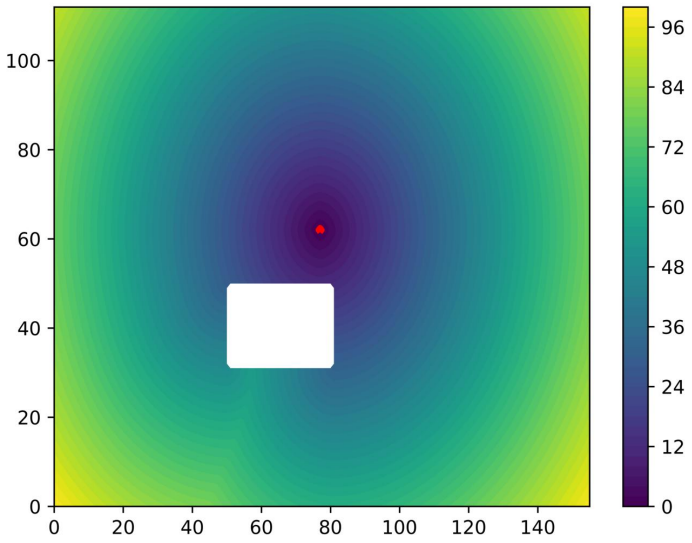


Figure 4.9: Distance to Goal

This figure shows the distance to goal array calculated using the goal and speed masks shown in figures 4.7 and 4.8. Distance is denoted with colors where darker colors (blue shades) represent small distances near the goal while lighter colors (green shades) represent longer distances, and white is used to denote static obstacles where a path to goal cannot be calculated.

This array can be used to find the shortest path from any point to the goal by walking downhill towards the goal. One way to achieve this is to calculate the gradient of the distance array on both the X and Y axes. This results in gradient arrays, which are two arrays that represent the changes in the distance function on both axes. The value of these gradient on X and Y axes at a certain point p in the environment can be interpreted as a gradient vector $\overrightarrow{g_p}$ that describes the direction and rate of change for the fastest local increase in the distance. If these values are non-zero,

then the direction of the gradient vector points towards the direction of the fastest increase, while its magnitude represents the rate of this increase. Figure 4.10 shows the gradient vectors of the distance to the goal array shown in figure 4.9.

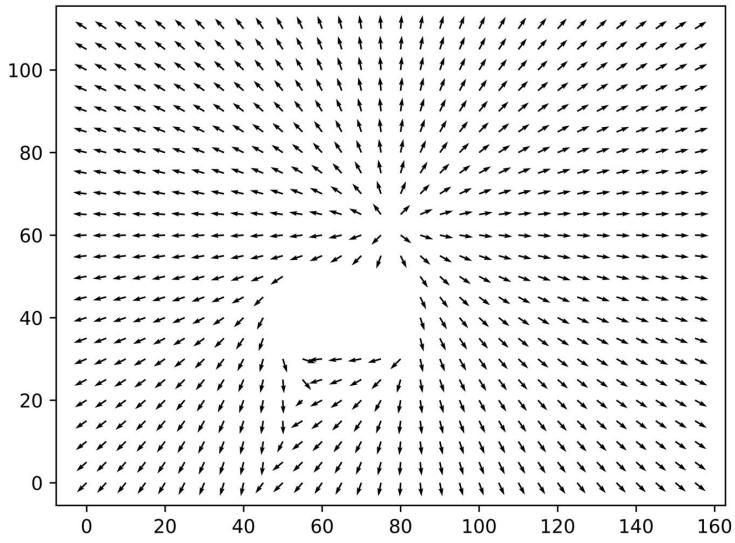


Figure 4.10: Gradient of the Distance Function
This figure shows the gradient vectors pointing towards increasing distances to the goal at each point in the environment.

However, since we want to find the path towards the goal, we are interested in calculating the directions towards decreasing distances to the goal rather than increasing distances. This can be calculated from the gradient vectors as follows:

$$\overrightarrow{d_p} \;=\; -1\;\overrightarrow{g_p}$$

We call $\overrightarrow{d_p}$ the direction to goal vector, which defines the direction along the shortest path to the goal from a target located at point p in the environment.

Using the gradient arrays we can calculate a new array with the same size as the discretized environment array where each cell's value is $\overrightarrow{d_p}$ (the direction to goal vector), we call this array the goal map. Figure 4.11 shows the goal map calculated from the gradient vectors shown in figure 4.10.

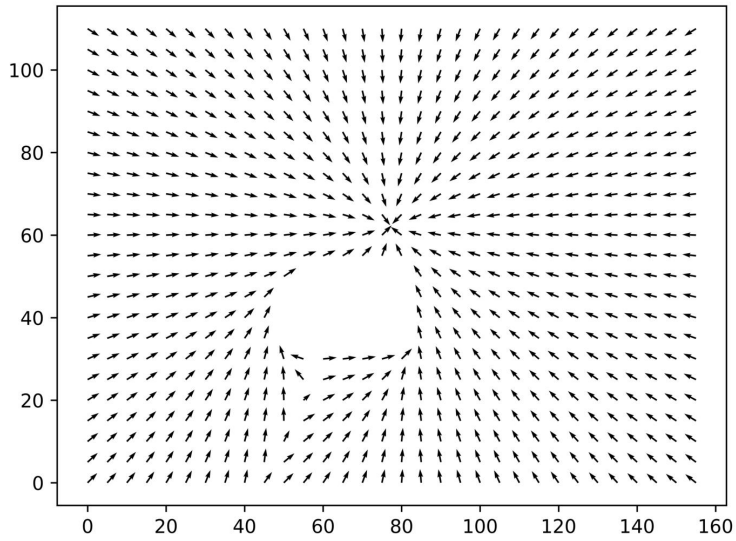


Figure 4.11: Goal Map

This figure shows the direction to goal vectors $\overrightarrow{d_p}$ pointing towards decreasing distances along the shortest path to the goal from each point in the environment.

These maps can be calculated once, using the definition of the environment, static obstacles, and the goal positions for each target group, either offline or on startup of the robot. They provide fast online access to global planning for each target group in real time. At each time step, each robot senses the type and location of neighboring pucks and directly finds each puck's direction using the goal map of that particular

target type with no need for any extra path planning calculations during runtime. It then chooses the best candidate to push according to the target selection algorithm detailed in section 4.1.2.

4.2 Obstacle Avoidance with Voronoi Cells

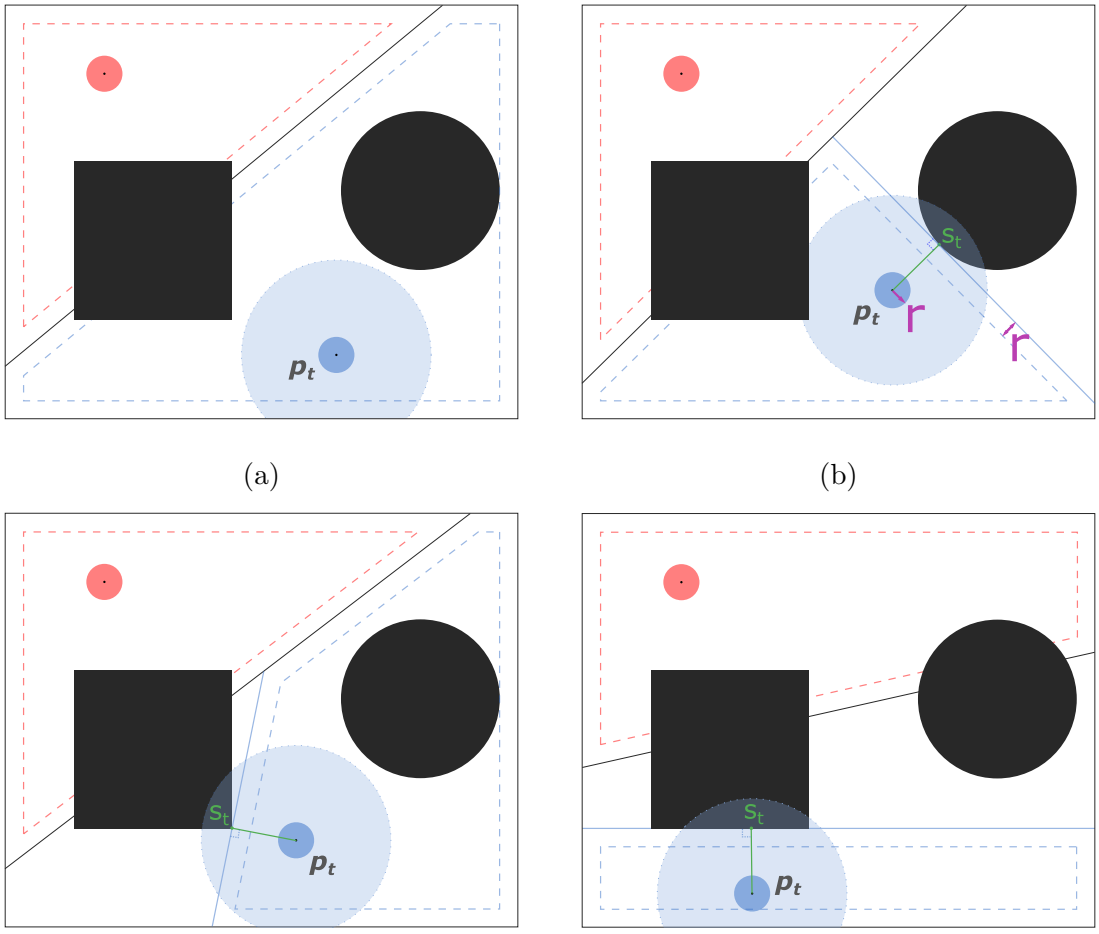
The collision avoidance algorithm described in chapter 3.2.1 handles collision avoidance against other robots but does not deal with other obstacles in the environment. In this section we will propose a mechanism for avoiding static obstacles within the environment based on the same concept of buffered voronoi cells. The algorithm works by finding the closest obstacle to the robot at each time step and modifying the voronoi cell in a way that prevents the robot from colliding with that obstacle.

At each time t, each robot calculates its current Voronoi cell V_t based on its position P and the positions of neighboring robots. The robot then senses the nearby objects within its sensing range forming S_t, the set of all sensed points belonging to nearby static obstacles at time t that lie within V_t. If S_t is empty, meaning no point within the current sensing range is found to be within V_t, then V_t is used to calculate the buffered voronoi cell without modification. Otherwise, a new V_t is calculated as follows.

The closest point to the current robot position P within S_t is selected as s_t:

$$s_t = p_i \mid \|P - p_i\| \leq \|P - p_j\| \; \forall \; j \neq i \; , \; p_i, p_j \in S_t$$

We then calculate the equation of the line that passes through s_t and is perpendicular to the line connecting s_t and the current position of the robot P. We first



(a) (b)

(c) (d)

Figure 4.12: Static Obstacle Avoidance

Examples of dynamically modifying the Voronoi cell to avoid static obstacles. The figures show the robot at position P and its sensing range. In (a) no static obstacle is detected so the Voronoi Cell is not modified. (b, c, and d) show how the Voronoi cell is modified by splitting it -with the blue line- depending on the position of the point s_t, the closest static obstacle detected within the sensing range.

calculate the vector $\overrightarrow{r_t}$ from P to s_t as:

$$\overrightarrow{r_t} = s_t \ - \ P$$

We then find a vector $\overrightarrow{r_t}'$ that is perpendicular to $\overrightarrow{r_t}$ from the fact that their dot product is equal to 0:

$$\overrightarrow{r_t}'.\overrightarrow{r_t} \ = \ 0$$

The equation of the line passing through s_t and perpendicular to $\overrightarrow{r_t}$ can then be formulated as:

$$l_t = s_t + m.\overrightarrow{r_t}' \quad : \quad m \in \mathbb{R}$$

We use this line to split the robot's current Voronoi cell V_t into two polygons. Since all voronoi cells are convex polygons, this split will result into two polygons that are also convex polygons. We then set the polygon that contains the robot's current position as the new voronoi cell V_t, which is used to calculate the current buffered voronoi cell and the navigation process is continued as described in 3.2.1.

As the robot moves closer towards other obstacles in the environment, the closest point to these obstacles s_t is dynamically recalculated along with the line splitting the voronoi cell, limiting the robot's movements within this cell and guaranteeing no collisions can occur with any obstacle.

Summary

In this chapter we described the proposed distributed sorting algorithm, With detailed solutions for solving each of its sub problems; target selection, target conflict avoidance, control strategy, and global planning. We also proposed an extension to

the collision avoidance algorithm described in chapter 3 for avoiding static obstacles using the same concept of buffered Voronoi cells. Chapter 7 provides further details about how this algorithm was implemented and validated in both simulations and real-life experiments.

In the next chapter we will describe the design and development of a web-based robotics simulation platform that will be later used to implement and validate the proposed algorithms.

Chapter 5

Simulation

This chapter describes the design and development of the SwarmJS simulation platform. This platform is open source and available on GitHub (`github.com/m-abdulhak/SwarmJS`). It is written in JavaScript and relies only on open source libraries. It was custom built to be a general simulation for swarm robotics algorithms and was used to develop and benchmark the collision avoidance algorithm (`github.com/m-abdulhak/Buffered-Voronoi-Cell-Deadlock-Avoidance`) and the swarm sorting algorithm (`github.com/m-abdulhak/swarm`).

5.1 Introduction

Computer simulations can be invaluable tools in the development of new robotics platforms and algorithms. They provide a virtual playground to rapidly prototype ideas and collect testing data to validate which approaches work well. Having access to an accurate and easy to use simulation can drastically cut down on the development time and cost, while also providing a safe testing environment [55].

In addition to providing easier diagnosis and much finer control and debugging capabilities. They also provide easier separation of concerns, where we can test a specific part of the system without worrying about the effects other parts of the system might have on the results; such as evaluating a path planning problem without worrying about the accuracy of the localization sensors, or the mechanical limitations of the robot's physical body.

The use of simulations can also be an important factor in avoiding cost and time limitations, as they commonly allow testing configurations that would be infeasible or too expensive to perform otherwise; such as evaluating how a collision avoidance algorithm would perform with hundreds of robots instead of being limited by the number of available physical robots, or how a foraging algorithm would fare with thousands of targets, which might take hours to perform in real life.

5.2 SwarmJS Simulation Platform

The SwarmJS platform is a web-based multi-robot simulation platform that was developed to prototype and benchmark the previously discussed robotic algorithms before deployment to actual robots. While this simulation was built from scratch for this purpose, it was designed to be a general purpose simulation so that it can be used for simulating multi-robot algorithms as quickly and seamlessly as possible.

SwarmJS is written in JavaScript and runs completely in the browser. This makes it very portable as it can be run on any system that supports modern web browsers, and since it runs completely in the browser, it does not need a back end server and can be easily hosted online using any hosting service that supports static websites,

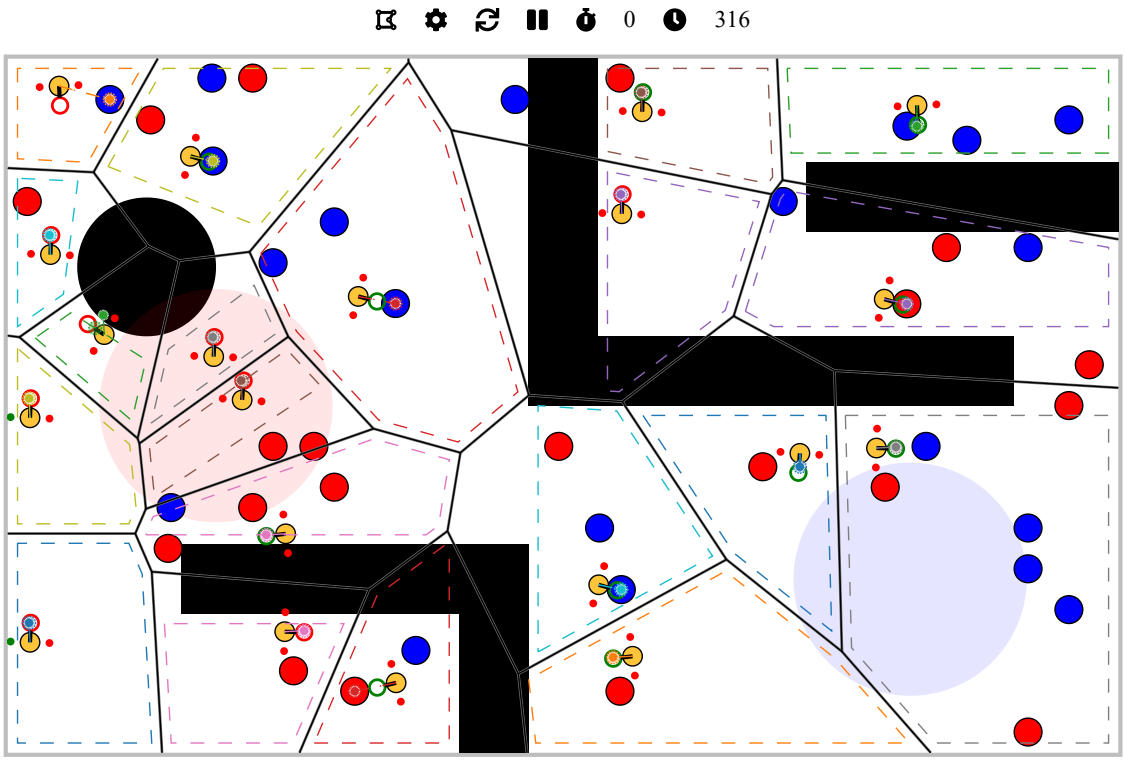


Figure 5.1: Simulation Rendering

Figure shows a simulation scene with static objects shown in black. Two types of pucks and their goal areas are shown in red and blue. Robots are shown in yellow with their actuators, sensors, goals, local waypoints, Voronoi cells, and buffered Voronoi cells.

such as GitHub pages, where it is currently accessible.

SwarmJS allows developers to define 2D environments and populate them with multiple kinds of components including static objects such as obstacles, passive dynamic objects such as targets (pucks), and active dynamic objects such as the robots. It utilizes a realistic physics engine to simulate these elements' interactions within the environment, giving the ability to define the physical properties for each element

such as shape, mass, and density in addition to advanced properties such as friction coefficients, restitution coefficients, collision filters to limit interactions between different types of objects, and gravity forces on each axis of the environment. In short, SwarmJS provides capabilities to simulate the following:

- Virtual environment with static and dynamic objects
- Robot sensing
- Robot planning
- Robot motion
- Interactions between multiple robots
- Interactions between robots and objects in the environment

5.2.1 Quick Actions

A list of quick action buttons can be found above the simulation window as shown in figure 5.1 including:

- Toggle Rendering: Enables / disables the rendering of the simulation. Disabling the simulation can lead to 2x better performance while the simulation is running.
- Toggle UI: Enables / disables a user interface that displays more options and the benchmarking graphs.
- Reset Simulation
- Pause / Resume simulation

- Start / Stop benchmarking
- Benchmark Runs: Number of times the benchmark has finished a simulation run across all provided simulation scenarios.
- Simulation Time

5.2.2 Quick Start

The following commands shown in code listing 5.1 can be run to startup the simulator:

```
> git clone https://github.com/m-abdulhak/SwarmJS.git
> cd SwarmJS
> npm install
> npm run dev
```

Listing 5.1: SwarmJS Quick Start

The App component provides the entry point of the simulator. Two parameters should be passed when initializing a new simulation:

- config: defines the simulation configuration.
- benchSettings: defines the benchmarking configurations.

Two example configurations are provided and can be used to start new simulations as shown in code listing 5.2:

```
import { examples } from './swarmjs-core';

const {simConfig, benchmarkConfig} = examples.voronoiSorting;
// or:
// const {simConfig, benchmarkConfig} = examples.simpleSorting;

ReactDOM.render(
  <App config={simConfig} benchSettings={benchmarkConfig}/>,
  document.getElementById('root')
);
```

Listing 5.2: SwarmJS Simulation Initialization

5.2.3 Configuration

Simulations are generated based on configuration objects passed to the SwarmJS library. Below are the main parameters that can be used:

- env: Defines the environment properties, including:
  - width: width of the simulated environment
  - height: height of the simulated environment
  - speed: relative speed of the simulation
- objects: a list of static object definitions, two types of static objects are supported, circles and rectangles. Each object definition can include the following:
  - type: 'rectangle' or 'circle'

  - center: the center of the object
  - radius: radius of the circle, for circles only.
  - width: width of the rectangle, for rectangles only.
  - height: height of the rectangle, for rectangles only.

- robots: Defines the robots properties, including:
  - count: number of the robots
  - radius: radius of the robots
  - sensors: list of enabled sensors.
  - actuators: list of enabled actuators.
  - useVoronoiDiagram: boolean, if true the Voronoi diagram for the robots is calculated at each timestep.
  - controllers: controllers are either passed directly or along with tuning parameters using the controller and params keys. Both syntax are present in the provided example configurations. Below are the 4 types of controllers that can be defined:
    * goal: sets the goal of the robot at each timestep. Accepts 3 parameters oldGoal, sensors, actuators and returns the new goal.
    * waypoint: provides motion planning for the robots. Accepts 3 parameters goal, sensors, actuators and returns a waypoint for the robot to head towards.
    * velocity: provides the control signals (velocities) that should move

the robot towards the waypoint. Accepts 3 parameters goal, sensors, actuators and returns a vector of velocities linearVel, angularVel .

    * actuator: optional controller to control the actuators. Accepts 2 parameters sensors, actuators and does not return any value.

- pucks: Defines the pucks properties, including:
  - groups: list of puck group definitions, each should contain:
    * id: unique value for each group
    * color: unique color for each group
    * count: number of pucks in the group
    * radius: radius of each puck in the group
    * goal: coordinates of the center of the goal area the pucks should be gathered at (if one exists)
    * goalRadius: radius of the goal area (if one exists)
  - useGlobalPuckMaps: boolean, if true a goal map for each puck group will be calculated at the start of the simulation, for each point in the environment the map provides a corresponding goal point where the puck should go towards to reach the group goal, useful for environments with static obstacles, but has a huge impact on the startup time of the simulation, should be disabled if not used.

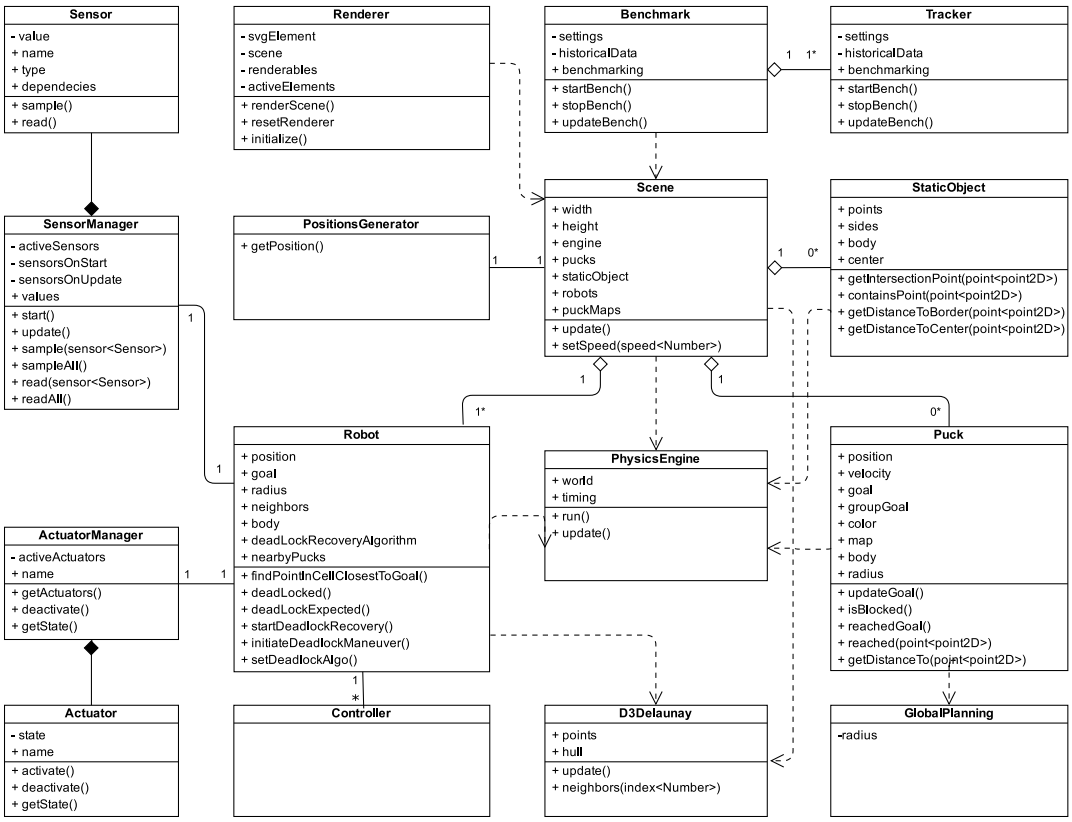


Figure 5.2: SwarmJS Architecture

Major modules in SwarmJS with their most important properties and methods

5.2.4 Software Architecture

There are multiple modules in SwarmJS. Each provides useful functionalities for creating different simulations. Some are vital for running any simulation such as the scene and physics engine, while others might be used or not depending on the specific experiment being simulated such as pucks, globalPlanning, and staticObjects. Figure 5.2 shows the main modules in SwarmJS.

Scene

This module can be considered the main coordinator in the simulation. It handles the simulation update cycle and provides methods and properties to specify different aspects of the simulation such as the shape of the environment, the number of robots and pucks present, and the physical properties of the simulation environment. It is closely dependent on the physics engine, which handles all the physical interactions between the various elements in the scene. Figure 5.3 shows the main steps in the simulation loop, the scene module coordinates these steps and assigns each responsibility to the appropriate module.

Renderer

The Renderer module provides a simple canvas based visualization for the simulation elements using the D3.js library, which is a JavaScript library for manipulating documents based on data. It allows data-based visualization using HTML, SVG, and CSS by binding arbitrary data to a Document Object Model (DOM) and applying data-driven transformations to the document. For example, we can bind the robots

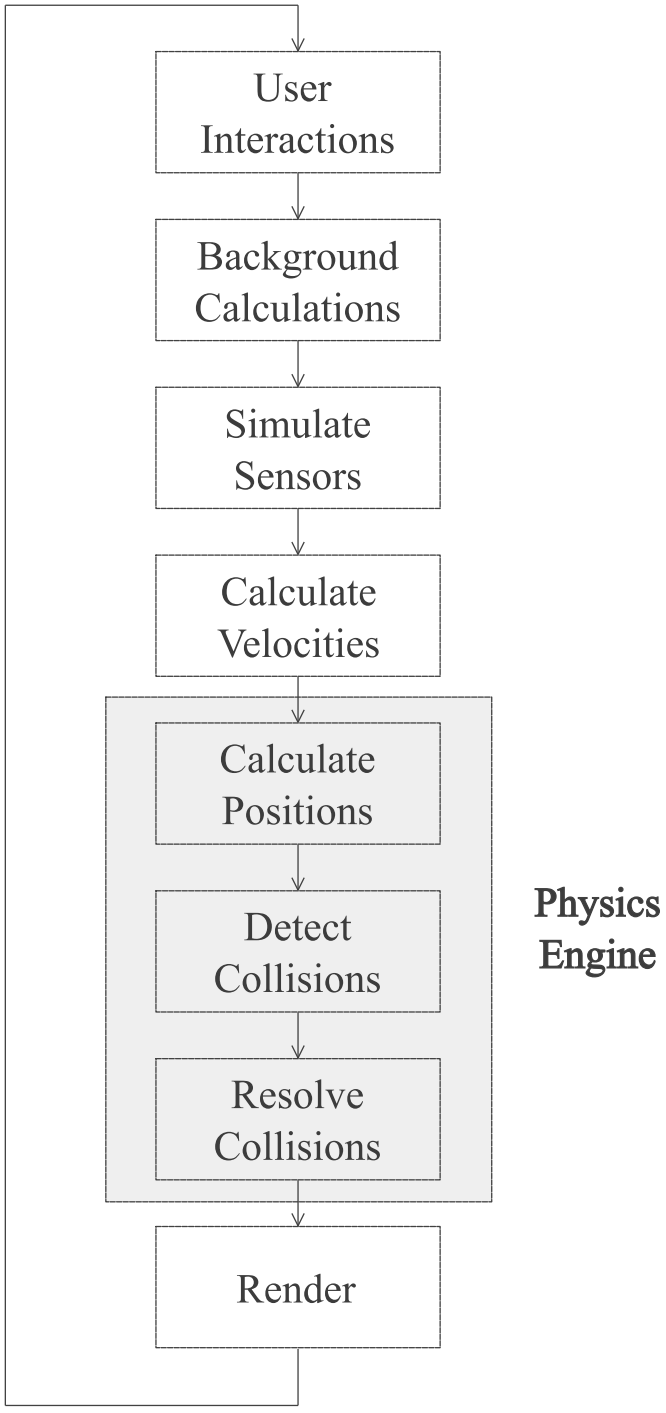


Figure 5.3: Simulation Loop

and pucks positions data to the renderer's canvas to create an interactive SVG-based map showing the static environment and the moving robots and pucks, with smooth transitions and interactions.

D3 is not a monolithic framework that seeks to provide every conceivable feature. Instead, it solves the crux of the problem: efficient manipulation of documents based on data. This avoids proprietary representation and affords extraordinary flexibility, exposing the full capabilities of web standards such as HTML, SVG, and CSS. With minimal overhead, D3 is extremely fast, supporting large data sets and dynamic behaviors for interaction and animation. D3's functional style allows code reuse through a diverse collection of official and community-developed modules.

A configuration-based rendering engine was built on top of D3.js to rapidly define renderable elements in the simulation with minimal code. A simple configuration format is used to define each renderable element, its attributes, and attach it to a specific object in the environment so that it is automatically updated while the simulation is running. Renderables can be defined within any module but they should always be imported and registered into the renderering module.

The Renderer is attached to the scene at startup and a visual element is created for each renderable in the configuration. Examples of renderables are:

- Environment boundaries
- Static obstacles
- Pucks
- Goal area for each puck group

- Robots, with each of the following items visualized independently for each robot:
  - Robot body
  - Voronoi cell
  - Buffered Voronoi cell
  - Goal
  - Local waypoint
  - Sensors
  - Actuators

Figure 5.1 shows a scene being rendered with all of the previous items visible.

Examples of renderable definitions can be found in the Scene, Robot, and Puck modules. Each renderable element can include the following parameters:

- type: mandatory, used for grouping renderables into UI buttons to enable/disable them.
- svgClass: optional, used to add classes to the svg elements.
- dataPoints: optional, defines the data points if the renderable is repeated for multiple objects such as robots, pucks, or static objects. DataPoints are usually defined as a property of the scene with the 'sceneProp' key. If dataPoints are defined, prop key can be used in the following configurations to refer to properties of the datapoint object. Otherwise, only 'sceneProp' can be used throughout the renderable definition.
- shape: mandatory, svg shape to be rendered

- staticAttrs: optional, defines the attributes to be set only once when the element is initialized.
- styles: optional, defines the styling attributes for the element, also only applied once, when the element is initialized.
- dynamicAttrs: optional, defines the attributes to be set on every simulation update.
- drag: optional, defines the draggable behavior for the element through the following properties:
  - prop: the property of the datapoint to be set using the position of the drag event, such as the position of the robot.
  - pause: whether the simulation should be paused while dragging the element.
  - onStart / onEnd: define the actions to be performed when dragging starts and ends:
    - styles: defines the styles to set when dragging starts / ends.
    - log: defines the attributes to be logged to console when dragging starts / ends.
  - onDrag: defines the actions to be performed when dragging:
    - log: defines the attributes to be logged to console when dragging is in progress

Syntax

Any property can be one of the following:

- string or number: the value is set directly.
- { prop, modifier } : the value of prop is parsed as a property of the datapoint, a modifier function can be defined to modify the value after it is parsed.
- { sceneProp, modifier }: the value is parsed as a property of the scene, a modifier function can be defined to modify the value after it is parsed.
- { special } : used for special behaviors, such as setting a color according to the color schema, currently only schemaColor is supported.

At each time step in the simulation, the dynamic attributes (such as location and shape) for each renderable (robots, pucks, Voronoi cells, etc) are recalculated by the scene and physics engine and the new properties are passed to the Renderer to update the corresponding visual components accordingly by modifying their shape (modifying the polygon representing a Voronoi cell) or applying the needed transformations to them (moving a robot to a new position).

```
{
    type: 'Goal',
    svgClass: 'robot-goal',
    dataPoints: { sceneProp: 'robots' },
    shape: 'circle',
    staticAttrs: {
      r: { prop: 'radius', modifier: (val) => val / 2 },
      id: { prop: 'id' }
    },
    dynamicAttrs: {
      cx: { prop: 'goal.x' },
      cy: { prop: 'goal.y' }
    },
    styles: {
      fill: { special: 'schemaColor' },
      stroke: 'white'
    },
    drag: {
      prop: 'goal',
      pause: true,
      onStart: { styles: { stroke: 'black' } },
      onEnd: { styles: { stroke: 'lightgray' } }
    }
  }
```

Listing 5.3: Creating Visual Elements with SwarmJS

Code listing 5.3 shows how renderable configuration is used to create visual elements (circles) corresponding to the robots' goals. Robots in the scene are iterated

and a circle is created to highlight the goal of each robot. The corresponding robot for each circle is bound to this circle as its 'data point', and its properties can be dynamically accessed through the 'prop' keyword. The location of each circle (cx and cy) is set according to the position of the corresponding goal, the radius of the goal circle is set to the robot's $radius/2$, a random color from a predefined color schema is assigned for each goal circle, and finally support for interactivity by dragging and dropping goal circles is added. During drag events, the styling of the goal circle is changed and after the event is ended, the scene is updated to notify the robot of its new goal.

The goal circles are then updated at each simulation time step by updating the defined dynamic Attributes, in this case, the position of the circle (cx and cy) according to the corresponding data point's (robot) properties (goal location).

PhysicsEngine

A physics engine is needed to perform the necessary calculations to simulate the movements of each component in the scene, which requires repeated calculations of their positions based on their velocities and accelerations. Forces such as gravity and friction, if implemented, are also taken into considerations. The physics engine is also responsible for collision detection and resolution, by determining when two objects collide and calculating how the objects move after the collision based on their physical properties such as shape, velocity, and mass.

Physics engines can vary greatly based on their complexity, ranging from advanced engines used for critical physics calculations to simple engines that can run in real-

time. A suitable engine should be selected based on the use case; fast engines might not offer the required precision for certain applications, While complex engines can be very precise but the extra calculations can cause the processing time to take longer and prevent their use in real-time applications.

Most of the calculations needed for our simulation can be implemented from scratch using simple Newtonian physics (positions, velocities, and accelerations), but using a simple engine such as Matter.js is preferred to more easily implement some advanced features such as collision detection and resolution.

Matter.js is a 2D rigid body physics engine built with JavaScript. Rigid body physics calculates object interactions and resolves collisions without considering force-based deformities, so physical objects never deform.

We can use Matter.js in one of two ways; The first is using its own Runner module which provides a simulation loop that continuously updates the engine at fixed intervals. The other approach that we use is implementing our own loop and updating the engine by calling its *Engine.update*() method and passing the time passed (delta) since the last update. Code listing 5.4 shows a pseudo code of the main steps involved during each update in our simulation.

```
update() {
  # Update background calculations such as Voronoi diagram
  voronoi = Delaunay.from(scene.robots);

  # Simulate robots' sensor measurements (neighbors and pucks)
  updateRobotsMeasurements();

  # Calculate new robots' properties (goals and velocities)
  scene.robots.forEach((r) => r.timeStep());

  # Calculate new pucks' properties (direction to goal)
  scene.pucks.forEach((p) => p.timeStep());

  # Update benchmarking data
  scene.updateBenchmark();

  # Update physics engine (calculate new positions)
  Engine.update(timeDelta);

  # If rendering is enabled, call Renderer's update()
  # to redraw the UI elements
  if (renderingEnabled) {
    renderer.update(activeElements);
  }
}
```

Listing 5.4: Pseudo Code of Simulation Update Steps

StaticObject

The StaticObject module contains methods to create rigid bodies with common shapes like a circle or a rectangle. A body for this object is created at startup and added to the engine. This type of object is completely static and passive and does not move nor has any dynamic physical properties such as velocity or acceleration, meaning they do not move as a result of any interactions with any other object in the environment.

```
var staticObjectsDefinitions = [
  {
    type: 'circle',
    center: { x: 250, y: 350 },
    radius: 50,
  },
  {
    type: 'rectangle',
    shape: [{ x: 50, y: 50 },
            { x: 50, y: 100 },
            { x: 100, y: 100 },
            { x: 100, y: 50 },
        ],
  },
];
```

Listing 5.5: Static Objects Definition

The static objects are defined using a JSON object, *staticObjectsDefinitions*, which defines the type and shape of each static object and is passed to the *Scene*

module at startup, which creates the static objects according to these definitions. Code listing 5.5 shows an example of this object defining two static objects of type circle and rectangle.

```
var pucksGroups = [
  {
    id: 0,
    count: 25,
    radius: 10,
    goal: { x: 150, y: 250 },
    color: 'red',
  },
  {
    id: 1,
    count: 25,
    radius: 10,
    goal: { x: 650, y: 250 },
    color: 'blue',
  }
];
```

Listing 5.6: Pucks Definition

Puck

The Puck module contains methods to create dynamic circle-shaped rigid bodies called pucks. A body for this object is created at startup and added to the engine. This type of object is passive, meaning it does not move by itself but only as a result

of its interactions with other active elements in the environment like the robots.

Pucks are separated into different groups, with each group having: a unique id, a goal area where the pucks need to be gathered defined by a circle with a center and a radius, the number of pucks in this group, and a color used for rendering the pucks from this group. The pucks are defined using a JSON object, *pucksGroups* and is passed to the *Scene* module at startup, which creates these pucks according to the definitions. Code listing 5.6 shows an example of pucks definitions. Pucks are dependent on the global planning module to calculate their local waypoint, or direction towards their group's goal.

Robot

The robot module contains methods to create dynamic rigid bodies. A body for this object is created at startup and added to the engine. The body of the robot can be a simple circle-shaped body or more complex compound body. This type of object is dynamic, its movement is simulated at each time step and is associated with a Voronoi cell calculated using the D3Delaunay module. It has sensors and actuators simulated at each simulation step, and its goal and velocities are calculated using the goal and velocity controllers. As the main active objects, robots interact with and move other passive objects in the environment such as the pucks.

Robots are defined using configuration objects specifying the number of robots and their radius, $numberOfRobots$ and $radiusOfRobots$. They are passed to the *Scene* module at startup, which creates these robots according to the configuration.

The robot module depends on controllers which implement the low-level motion

planning and collision avoidance algorithm based on its sensor values, which are simulated using available data such as neighboring robots' positions, nearby pucks' positions, and the Voronoi cells calculated using the D3Delaunay module.

Controllers

Controllers are higher order functions that return functions that control different aspects of the robots behaviors, they are called at each timestep when the simulation is updated. There are 4 types of controllers:

Goal Controller

The main controller that implements the application specific algorithm and sets the goal of the robot at each time-step.

This controller is responsible for defining and implementing the high-level algorithm which can change drastically depending on the scenarios being simulated as it relates to the intention of the simulation and objective of the robots within the environment.

For instance, when running a simulation to evaluate a collision avoidance algorithm, similar to the one proposed in chapter 3.2.1; a robot goal set by the goal controller can be simply a static point for each robot that does not change with time, with the intention of evaluating how fast the robots can reach their goals using the collision avoidance algorithm under review. These goals can either be randomly selected or follow a predefined pattern such as lie on a specific geometrical shape like a circle.

Another scenario can involve dynamically calculating a new goal for each robot at

each time step depending on the position of the robot within the environment and the values of the robot sensors at that particular instance. Such as having a robot orbit the environment when no nearby pucks are sensed, while moving towards a puck in case one is detected. Similar goal selection algorithm is discussed in details in section 4.1.

Waypoint Controller

Provides motion planning (collision avoidance and maneuverability) for the robots. While the goal controller is expected to be application-dependent, the waypoint controller can provide more general motion planning algorithms at can be used across different simulation scenarios.

Velocity Controller

Provides the control signals (velocities) that should move the robot towards the waypoint.

Actuator Controller

Optional controller to control the actuators.

Sensors

Objects that define how the robot can sense a specific aspect of the simulation, such as its own position and orientation, the position of neighboring robots, nearby objects, etc. Other sensors can easily be defined by extending the Sensor class. Sensors can

be implemented as either a class or a function but should implement the following interface:

- sample(): calculates the value of the sensor
- read(): returns the latest sampled value of the sensor
- name: used to access (sample and read) the sensor through the sensor manager
- type: determines when the sensor is sampled, possible values: onStart, onUpdate.
- dependencies: optional, a list specifying any other sensors needed for this sensor to work, sensorManager uses these lists to generate a dependency graph and determine the order in which the sensors should be sampled.

All sensors should be added to the 'availableSensorDefitions' list in sensorManager and the name and sensor object should be exposed by default exporting an object with the following properties:

- name: the name of the sensor
- Sensor: the sensor object

Position Generators

Higher order functions that return a function that generates positions in the environment. They are used to generate starting coordinates for the robots, goals, pucks, etc. They are useful for generating and repeating specific starting configurations for these elements.

Performance Trackers

Special objects that describe the simulation performance. Each object provides a function to calculate a performance metric at each simulation update. Each tracker will result in a separate graph in the benchmarking tab. Trackers should also define functions for reducing and aggregating values. Tracker can be used as a reference and extended as it provides most of the needed functionalities.

Benchmark

This module is responsible for comparing different simulation scenarios by running and recording the simulation performance as defined by the provided Trackers.

Benchmarks provide an easy way to run multiple simulations and compare them across multiple runs using specific metrics. Benchmarks are defined using configuration objects that are passed to the simulation on initialization. Below are the main parameters that should be preset in a benchmark configuration object:

- simConfigs: a list of configuration objects that describe different simulation scenarios, each object should include:
  - name: unique name for this scenario, will be used to refer to this scenario in the benchmarking graphs.
  - simConfig: a simulation configuration object that should adhere to the specifications described in section 5.2.3. Not all parameters should be specified here, but rather only the ones that separate this scenario from the main simulation scenario. These differences can be simple such as

changing the defined static objects, number or radius of robots, number of puck groups or the number or radius of the pucks; or they can be more specific such as comparing different controllers or even changing specific parameters for the same controller. Any property in the main simulation configuration can be overridden here.

- timeStep: minimum reported time step in the graphs.
- maxTimeStep: length of each simulation run while benchmarking.
- trackers: list of special objects that describe the performance and provide a function to calculate a performance metric at each simulation update.

D3Delaunay

This is a fast library for computing the Voronoi diagram of a set of two-dimensional points (robot positions). It is based on Delaunator, a fast library for computing the Delaunay triangulation using sweep algorithms. The Voronoi diagram is constructed by connecting the circumcenters of adjacent triangles in the Delaunay triangulation.

This module might not be needed if the robot's motion planning algorithm does not depend on the Voronoi diagram. But since our collision avoidance algorithm depends on the Voronoi cells, this module's *delaunay.voronoi* function is used by the Scene to calculate the Voronoi diagram of the current robot positions at each time step. The corresponding Voronoi cell and buffered Voronoi cell for each robot are then passed to the robot as a sensed value.

GlobalPlanning

Global planning is used to find the direction a puck should go towards in order to move closer to its goal. This is implemented using the distance transform method, which requires the goal and the environment's shape to be known. It produces a discretized grid representation of the environment where the value of each cell is the smallest distance from the cell's position to the goal.

The distance transform can be calculated using many methods, such as the fast marching method that we use on the physical robots, but due to lack of suitable libraries support in JavaScript, we decided to implement the distance transform from scratch in this module.

We start by creating an array as a discretized representation of the environment, we call it the environment map. Each cell in this map is initialized as follows: 0 for cells representing empty areas in the map where no static obstacles exist, and 1 for cells containing static obstacles. Figure 5.4 shows an example of the environment map.

Then we create a new array that will contain the distance of the closest path from each cell in the map to the goal, we call it the distance map. We start processing this array from the cell representing the goal and give it a distance of 0. Then we move on to the goal's immediate neighbors which are assigned a distance of 1, then their neighbors are assigned a distance of 2, ... etc, increasing the distance by 1 at each scan until all cells in the grid are visited. However, any cell representing a static obstacle in the environment map is a assigned a special distance value NaN, denoting that the goal cannot be reached from this position. 8-connectivity is used to traverse the

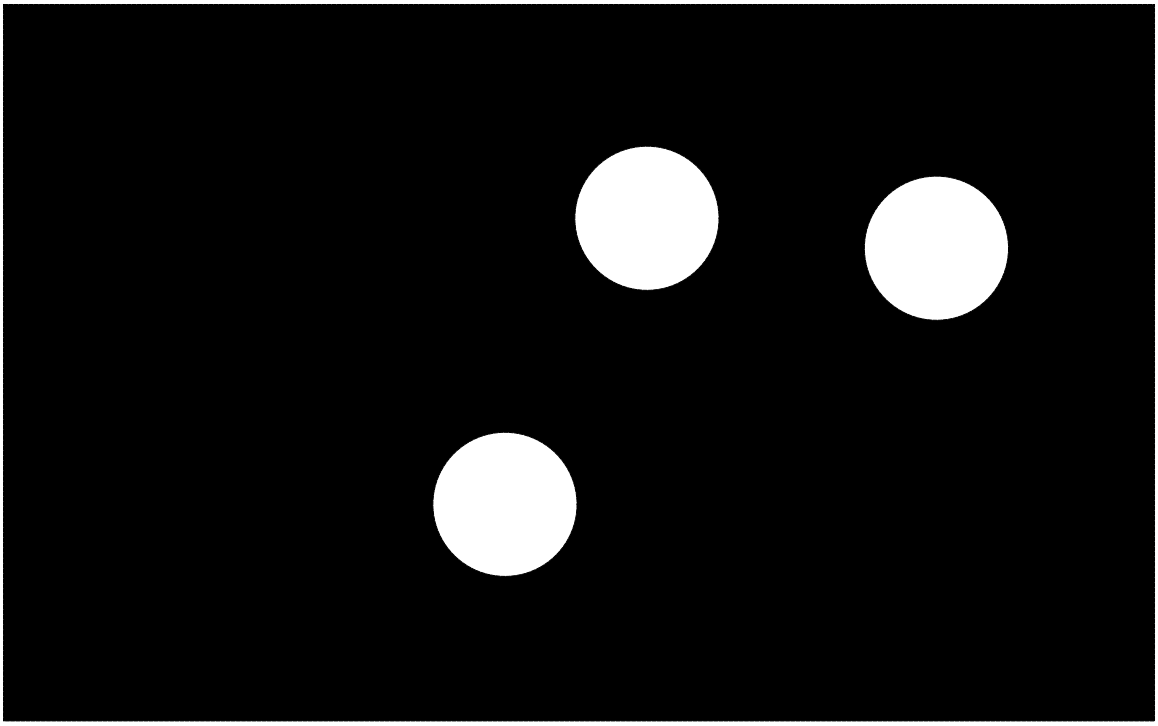


Figure 5.4: Environment Map

map using the neighbors. So each cell, excluding cells on the edges of the map, has 8 neighbors by default which make up the Moore Neighborhood of that cell, consisting of the 4 immediate cells along diagonal offsets, and the cells directly above and below and to the left and right of the cell. Figure 5.5 shows an example of the generated distance map.

The generated distance map is independent of any start point, it expands from the goal similar to a wave, flowing and propagating around obstacles. A path from any point to the goal can be found by walking downhill (following neighbors with minimal distances) towards the goal. However, a full path is not required for each cell, only the direction towards the goal along the shortest path is sufficient.

We generate the final goal map by iterating through all cells in the distance map.

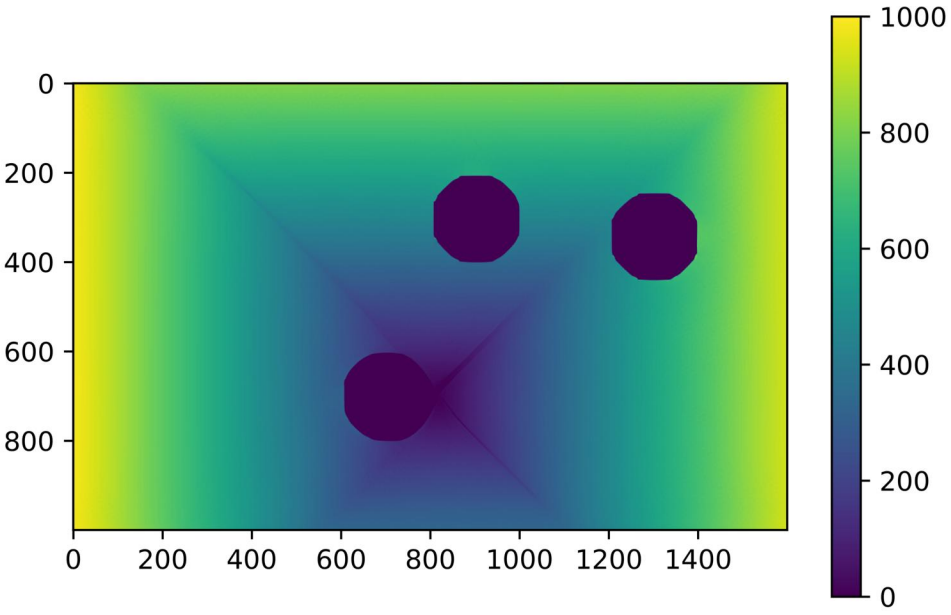


Figure 5.5: Distance Map

If the distance at a cell is NaN, then no path to goal can be found, so the direction to the goal is not defined, so we keep the NaN value for this cell in the goal map as well. Otherwise, we check all the neighboring cells to find the cells closest to the goal those having the minimum distance in the distance map. We then calculate the final direction to the goal as the average of the directions towards each of these cells.

Chapter 6

Experimental Setup

This chapter describes the physical swarm system. It details the hardware design of the robots and the setup of the experimental system including the tracking and sensing simulation system, and the software system running on the robots.

6.1 Experimental Setup

Our experimental platform is a set of Pololu 3pi robots [1] fitted with Raspberry Pi 3 A+ single-board computers [2] to perform the on-board processing. The 3pi robot base is ≈9.5 cm in diameter and provides basic differential-drive mobility.

Each robot is also fitted with 2 acrylic plates, providing mounting points for the Raspberry Pi and two small communication and power interface boards. A unique AprilTag [50] is printed on the top of each robot to identify the robot by the overhead tracking system that provides localization and sensing to the robots.

[1] `https://www.pololu.com/product/975`

[2] `https://www.raspberrypi.org/products/raspberry-pi-3-model-a-plus`

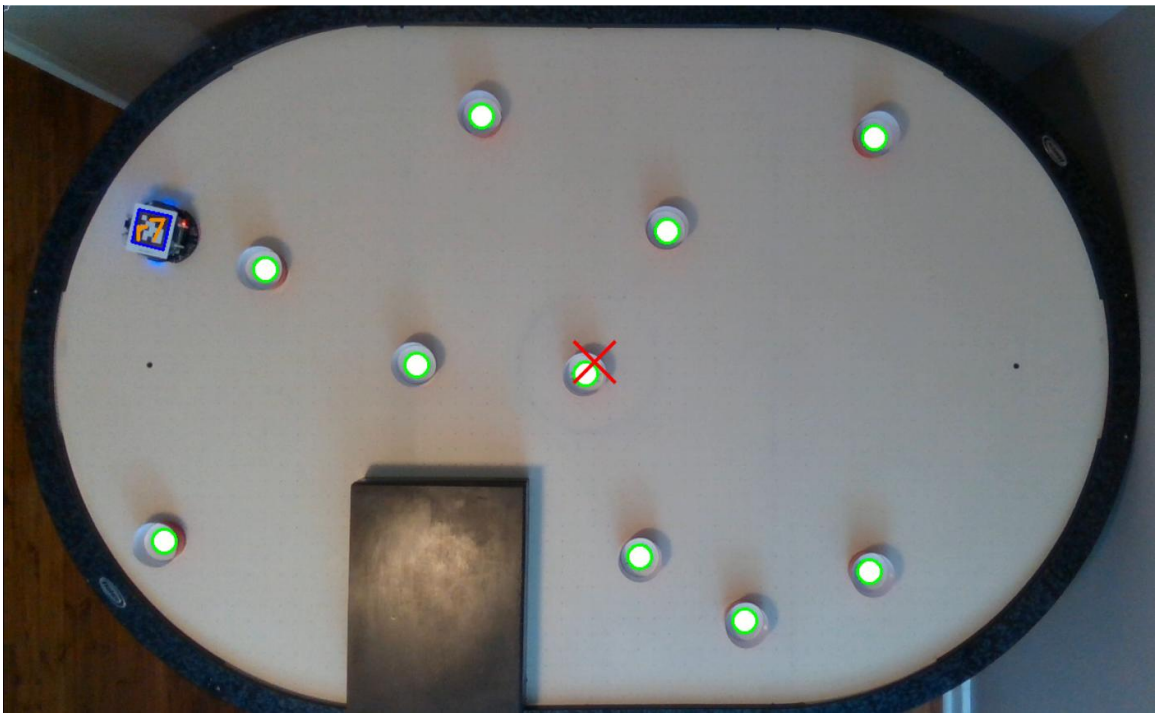


Figure 6.1: Arena

These robots mainly operate on a table with a stadium-shaped boundary. A stadium consists of a rectangular region in the middle with semicircular ends. The overall width and height of the table's surface is 1.82 × 1.21 m.

The "pucks" are cups with a bottom diameter of 6 cm, weighted with a wooden disk on the bottom that also presents a 4 cm green painted dot visible to the overhead camera. The table's boundary has a height of 2 cm and the pucks have a height of 4 cm.

A wooden board was added to the middle of the table to present a more complex boundary for testing. An image of the arena is shown in Figure 6.1, showing the shape of the arena, pucks, and robots.

The robots connect to a central server remotely over WiFi to continuously request

location and sensing data. The server continuously processes the images received from the attached overhead camera, an Intel Real-sense D435. It uses computer vision to detect the positions and orientations of the robots and the positions of the pucks. It uses this data to simulate the values detected by each robot's virtual sensors by passing each robot a message containing its current position and orientation, the positions of nearby robots, and the positions of nearby pucks.

The software used on the central server was adapted from a previously developed system for controlling autonomous surface vehicles [56] with the server component available as open source [3].

Using a centralized computer to simulate local sensing violates a core principle of swarm robotics, local sensing [4]. It is important however to emphasize that only the central server has access to the global knowledge and each robot strictly receives local sensing and makes all the planning and control decisions on board their own local computer. So while the experimental setup falls short of meeting the technical definition of a swarm, it still provides a useful research platform for testing decentralized robotic algorithms.

6.2 Robot Control Software

Each robot has an on-board Raspberry Pi computer that runs a custom controller responsible for autonomously driving the robot. This application was written in Python because it is natively supported by the Raspberry Pi's operating system, in addition to its power, ease of use, and its extensive collection of libraries for many program-

[3] `https://github.com/BOTSlab/kodama_server`

ming tasks. The entire software is available as an open source project on GitHub and can be accessed through the following link (`https://github.com/m-abdulhak/kodama_swarm`).

The controller software consists of multiple modules that handle all the sub-tasks required for operating the robots from low-level tasks such as sending control signals to the wheel motors to high-level ones such as goal selection. Many of the modules were directly ported from the corresponding JavaScript modules in the SwarmJS simulation described in chapter 5, such as the goal selection module, the global planning module, and the local planning module that was implemented as part of the robot module in SwarmJS, so we will not describe them in details again here.

The following is a list of these tasks:

- Sensor update: requesting sensor data from the central server and parsing that data.
- Goal Selection: choosing a goal in the environment depending on the task defined such as the sorting task.
- Local planning: finding a local waypoint that drives it towards its goal using the collision avoidance algorithm defined in section 3.
- Motor control: calculating and sending the appropriate control signals to the wheels' motors in order to reach its local waypoint.

Code listing 6.1 shows a simplified code of the main steps involved during each update in robot controller's loop.

```
def update(self, sensor_data):
    # Get Robot Position and orientation
    x, y = sensor_data.pose.x, sensor_data.pose.y
    theta = sensor_data.pose.yaw

    # Update buffered Voronoi Cell
    bvcCell = bvcNav.update(sensor_data)

    # Get new goal
    self.newGoal = updateGoal(bvcCell, sensor_data, self.env)

    # Get the local waypoint within BVC Cell
    waypoint = bvcNav.setWaypointInCell()

    # Compute relative position of waypoint in polar coordinates
    dx = waypoint["x"] - x
    dy = waypoint["y"] - y
    distanceToGoal = sqrt(dx**2 + dy**2)
    angleToGoal = min_signed_ang_diff(atan2(dy, dx), theta)

    # Get forward and angular speeds (v, w)
    fSpeed, aSpeed = getDiffSpeeds(distanceToGoal, angleToGoal)

    # Convert to right and left wheels speeds and send to robot
    v_left, v_right = getMotorSpeeds(fSpeed, aSpeed)

    # Send speeds to motors
    three_pi.send_speeds(v_left, v_right)
```

Listing 6.1: Simplified Code of Robot Controller Update Steps

6.2.1 Sensor Update

The sensing data calculated by the central server is sent over WiFi and encoded using Google's Protocol Buffer library [57]. It offers language-neutral, platform-neutral, extensible mechanism for serializing structured data. A single definition for the data structure is used to generate source code that can used to easily write and read the structured data to and from a variety of data streams and using a variety of languages. This greatly simplifies transferring the data between the sever software written in C++ and the robot control software written in Python.

```
message SensorData {
  message Position2D {
    int32 x = 1;
    int32 y = 2;
  }
  message Pose2D {
    int32 x = 1;
    int32 y = 2;
    float yaw = 3;
  }
  Pose2D pose = 1;
  repeated Pose2D nearby_robot_poses = 2;
  repeated Position2D nearby_target_positions = 3;
  google.protobuf.Timestamp timestamp = 4;
}
```

Listing 6.2: Sensor Data Message Definition Using Protocol Buffer

Code listing 6.2 shows how the sensor data message is defined using the Protocol Buffer. Each sensor data message contains the following information: the robot's current location and orientation, the position of the neighboring robots, and the positions of neighboring pucks. This data is parsed and stored and made available for access by other modules.

6.2.2 Goal Selection

This implements the high level behavior of the robot. It varies greatly depending on the task assigned to the robot.

When validating the collision avoidance algorithm, the high level goal for each robot was to reach a specific goal point in the environment. In practice, that goal was determined by the location of a specific AprilTag assigned as a goal to each robot. This was implemented by requesting the location of this AprilTag at the first successful connection to the central sensing server and setting that position as the goal of the robot for the remainder of the experiment. Figure 7.6 shows an example experiment where in the initial stat configuration, 9 robots and 9 goal tags can be seen on the field, while in the final configuration only the robots robots can be seen after reaching their tags successfully.

For the sorting task, the goal selection module implements the sorting algorithm involving all the behaviours described in section 4.1. This includes target selection, target conflict avoidance, control strategy and the default behavior of orbiting the environment when no suitable targets can be found.

The environment orbit map and the puck goal maps are generated offline by a

Python script as described in sections 4.1.1 and 4.1.4. The generated maps are saved as raw data using Python's pickle module and loaded on systems startup. Code listing 6.3 shows a script for generating the goal maps for each puck group.

```
for indx, goal in enumerate(goals):
    # Generate environment boundary mask
    X, Y = np.meshgrid(np.linspace(0, envXMax, envXMax+1), \
            np.linspace(0, envYMax, envYMax+1))
    map_goal_mask = -1 * np.ones_like(X)
    map_goal_mask[np.logical_and(X==goal[0], Y==goal[1])] = 1
    # Generate speed mask
    speed = 1 * np.ones_like(X)
    speed[np.logical_or(X>=envXMax, Y>=envYMax)] = 0
    speed[np.logical_or(X==0, Y==0)] = 0
    for x in range(0, envXMax):
        for y in range(0, envYMax):
            curPoint = Point(x,y)
            for o in staticObstaclesPolygons:
                if(pointIsInsidePolygon(curPoint, o)):
                    speed[y][x] = 0
    # Generate Distance Transform
    distTr = skfmm.travel_time(map_goal_mask, speed, 1).filled()
    # Generate Gradients on both axes
    yG, xG = np.gradient(distTr)
    # Calculate goals for each point in the environment
    x_goals, y_goals = -1 * xG + X, -1 * yG + Y
```

Listing 6.3: Simplified Script for Target Goal Maps Generation

```
def setWaypointInCell(self):
    # Set the cell as the current Buffered Voronoi cell
    cell = self.bvc

    # If cell is undefined, set waypoint = goal
    if (cell == None or len(cell)<2):
        log("Error, Cell Not Defined!")
        self.waypoint = self.goal
        return self.waypoint

    # If the goal is within the Buffered Voronoi cell
    # set waypoint = goal
    if (cellContains(cell, self.goal)):
        self.waypoint = self.goal
        return self.waypoint

    # If deadlocked or deadlock is expected
    # or if currently recovering from deadlock
    # set waypoint according to deadlock recovery policies
    if (self.setWaypointByDeadlockRecovery(cell)):
        return self.waypoint

    # Default behavior: set waypoint as the point in cell
    # that is closest to the goal
    self.waypoint = self.findPointInCellClosestToGoal(cell)
    return self.waypoint
```

Listing 6.4: Simplified Local Planning Function

6.2.3 Local Planning

This module implements the collision avoidance algorithm proposed in chapter 3. This includes calculating the Voronoi diagram using the positions of the robot and neighboring robots, calculating the buffered Voronoi cell, modifying the buffered Voronoi cell depending on the locations of nearby static obstacles as described in section 4.2, and finding the best local waypoint that drives the robot towards its goal using the algorithm defined in section 3.2.1. Code listing 6.4 shows a simplified version of the function responsible for finding this local waypoint.

The Voronoi diagram is calculated using the Voronoi module in SciPy [58], which is a free and open-source Python library used for scientific computing.

6.2.4 Motor control

This implements the low-level control strategy for the robot by defining the control signals that should be sent to each of the motors of the two wheels in the differential drive system on the robot. It receives the distance and angle to the local goal, calculated from the local waypoint defined by the Local Planning module, and calculates the corresponding forward and angular speeds required to reach that goal. Code listing 6.6 shows how forward and angular speeds are calculated based on the distance and angle to local goal. These are then translated into left and right motor speeds that are passed through a Serial Gateway to the 3pi robotic base. Code listing **??** shows how forward and angular speeds are translated into left and right motor speeds.

```
def getFowrardAndAngSpeeds(self, distanceToGoal, angleToGoal):
    forwardSpeed, angularSpeed = 0, 0

    # If goal is reached, stop
    if (bvcNav.reached(goal)):
        return forwardSpeed, angularSpeed

    # If angle to goal is small enough,  move in a straight line
    if (abs(angleToGoal) < maxAngleToMoveStraightToGoal):
        forwardSpeed = min(distanceToGoal/10, maxForwardSpeed)
    # Else, turn in place,  No Forward Speed
    else:
        forwardSpeed = 0

    # If robot is not facing goal, turn to goal
    if (abs(angleToGoal) > robotIsFacingGoalMaxAngle):
        angularSpeed = min(angleToGoal / pi, maxAngularSpeed)
    # Else do not turn (move in straight line)
    else:
        angularSpeed = 0

    return forwardSpeed, angularSpeed
```

Listing 6.5: Simplified Function for Calculating Forward and Angular Speeds

```
def getMotorSpeeds(self, forwardSpeed, angularSpeed):
  v_right = forwardSpeed - angularSpeed / 2
  v_left = forwardSpeed + angularSpeed / 2

  if forwardSpeed == 0 and angularSpeed != 0:
    if(v_right > 0 ):
      v_right = max(min(v_right, maxMotSpeed), minMotSpeed)
    else:
      v_right = max(min(v_right, minMotSpeedBck), maxMotSpeedBck)

    if (v_left > 0):
      v_left = max(min(v_left, maxMotSpeed), minMotSpeed)
    else:
      v_left = max(min(v_left, minMotSpeedBck), maxMotSpeedBck)
  else:
    v_right = max(min(v_right,1), -1)
    v_left = max(min(v_left,1), -1)

  return v_left, v_right
```

Listing 6.6: Simplified Function for Calculating Motor Speeds

6.2.5 Summary

In this chapter we described the experimental setup including the hardware and the software of the robots and the setup of the experimental system. In the next chapter we will describe how this setup was used to validate and test the proposed algorithm and show the results of the performed experiments along with the data obtained from simulating the algorithms in SwarmJS.

Chapter 7

Experiments and Results

This chapter describes the the simulation and physical experiments used to validate the proposed collision avoidance and object sorting algorithms and a detailed analysis of the results of the various experiments performed.

7.1 Collision Avoidance Algorithm Experiments

7.1.1 Simulation

In this section we will detail the results of multiple simulation experiments to compare our collision avoidance algorithm -integrating the proposed deadlock avoidance algorithm- and a buffered Voronoi cell based collision avoidance algorithm with right-hand deadlock recovery heuristics.

[1]All simulations are performed in an interactive web-based multi-robot simulation based on the SwarmJS platform described in chapter 5.

[1]`https://github.com/m-abdulhak/Buffered-Voronoi-Cell-Deadlock-Avoidance`

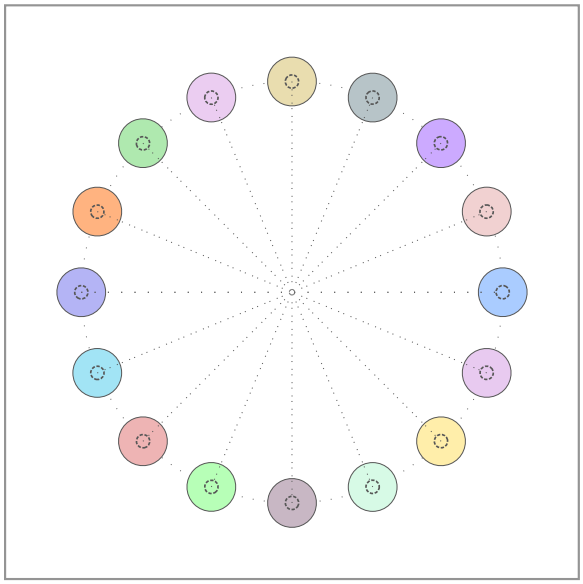
A set of disk-shaped single-integrator robots moving in a planar space are simulated, each robot is assigned a specific goal, and for each simulation, the same collision avoidance algorithm is applied for all robots. Two metrics are tracked and compared: the total distance between the robots and their goals and the minimum distance between the robots at each time step. We compare the performance of 2 algorithms, our collision avoidance algorithm integrating the proposed deadlock avoidance algorithm and a baseline buffered Voronoi cell based collision avoidance algorithm with the right-hand deadlock recovery heuristics. Performance is compared across 4 different cases, all performed in a $800 * 500$ cm environment, with 100 robots and 100 simulations per algorithm. No collisions occurred in any of the simulations.

First, 100 robots with 3 cm radius are placed around a circle, with their respective goals assigned to the point directly across them on the same circle, with a small displacement to break the symmetry.

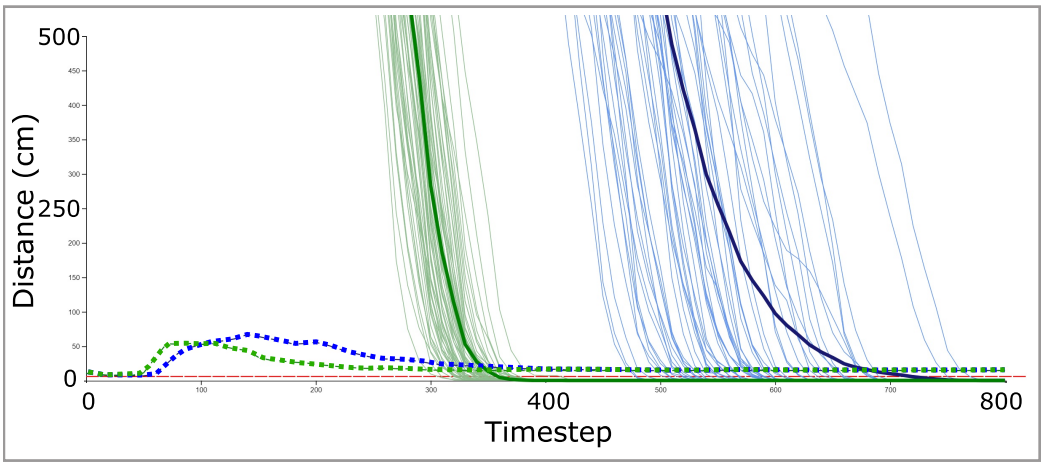
This configuration presents a considerable challenge for collision avoidance algorithms since the default paths of all robots intersect at the same point in the environment, the circle's center. All robots start at the same time and head along their intersecting paths, causing a huge congestion at the center of the circle.

With the baseline algorithm, the robots keep going along their path until they find themselves stuck in a deadlock configuration. While the proposed deadlock prediction algorithm allows the robots to predict the deadlocks caused by the congestion at the center of the circle earlier and proactively find alternative paths to avoid these deadlocks.

Figure 7.2 shows the difference between the states of the robots at multiple times during an experiment for both algorithms. While both algorithms are able to eventu-



(a) Starting configuration for simulation case 1 with 16 robots instead of 100 for clarity



(b) Total distance to goal (solid) and minimum distance to neighbors (dashed) with safety distance in red (dashed). Each simulation is presented as a single line, with the average for each algorithm in bold. Proposed algorithm in green and BVC algorithm with right-hand heuristics in blue.

Figure 7.1: Collision Avoidance Simulation Experiment - Case 1

ally drive the robots to their destination, this look into how the experiment progresses with time clearly shows how the proposed algorithm successfully prevents the congestion and leads the robots into alternative paths around it and into their goals much faster than the baseline algorithm.

This translates into a clear performance improvement as shown in figure 7.1. The figure shows the performance metrics across the run time of the 100 experiments for this case, the total distance to goal is shown for each simulation individually. The baseline algorithm is shown in light blue and our algorithm with the proposed deadlock avoidance algorithm is shown in light green, the means of these simulations are also highlighted in blue and green respectively. The safety distance ($2r_s$) is also shown for each case as a dashed red line, and the minimum distances between robots are shown as two dashed lines in blue and green for the two previous algorithms, respectively.

While both algorithms are eventually able to drive all the robots to their respective goals, the time taken clearly shows the benefits of the early deadlock prediction algorithm, with the proposed algorithm driving the robots to their goals in almost half the time. The figure shows how all robots reach their goals by the 400th time step in all experiments using the proposed algorithm, while needing 800 time steps to reach their goals for most experiments using the baseline algorithm.

For the next experiments, 100 robots (5 cm radius) are placed on a square, and the goals are set as their positions reflected across the y-axis (case 2). Then, the same configuration is used but the goals are reflected across the origin of the environment (case 3) as shown in Figures 7.3 and 7.4. These two cases along with the previous one represent particularly challenging configurations for collision avoidance algorithms

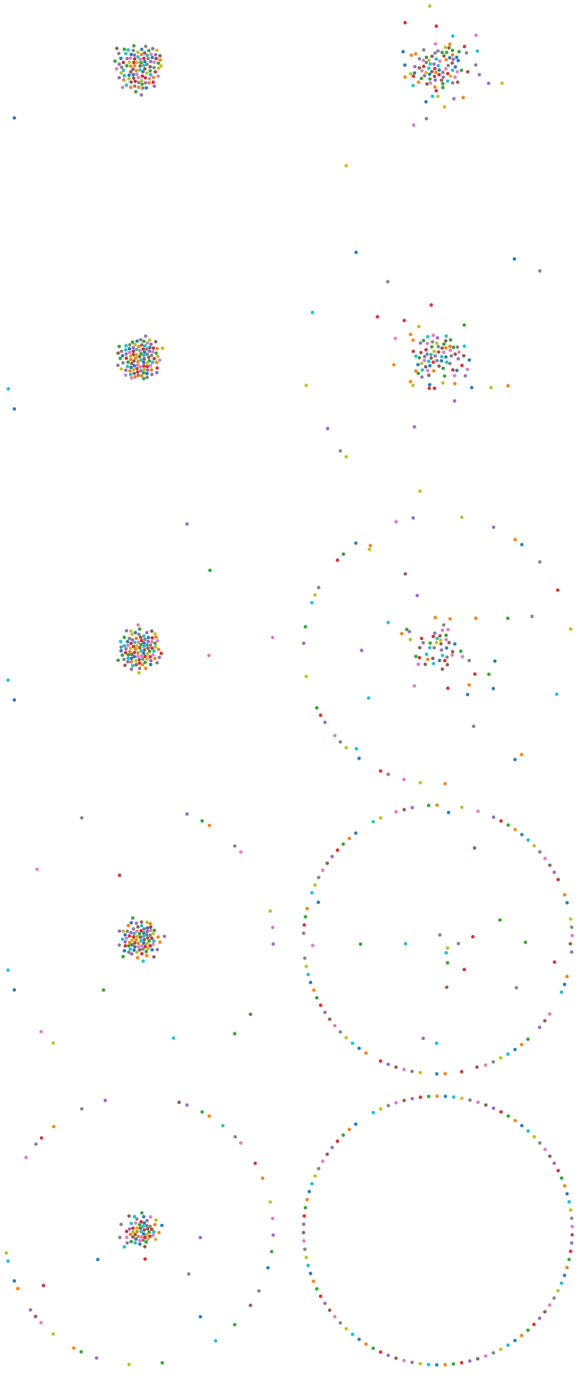
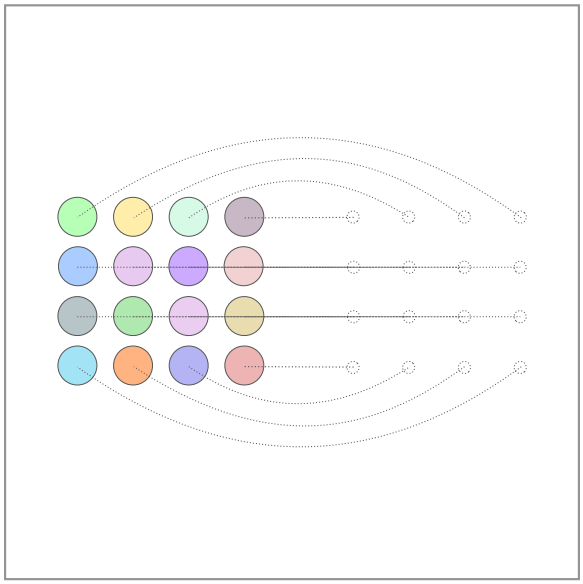


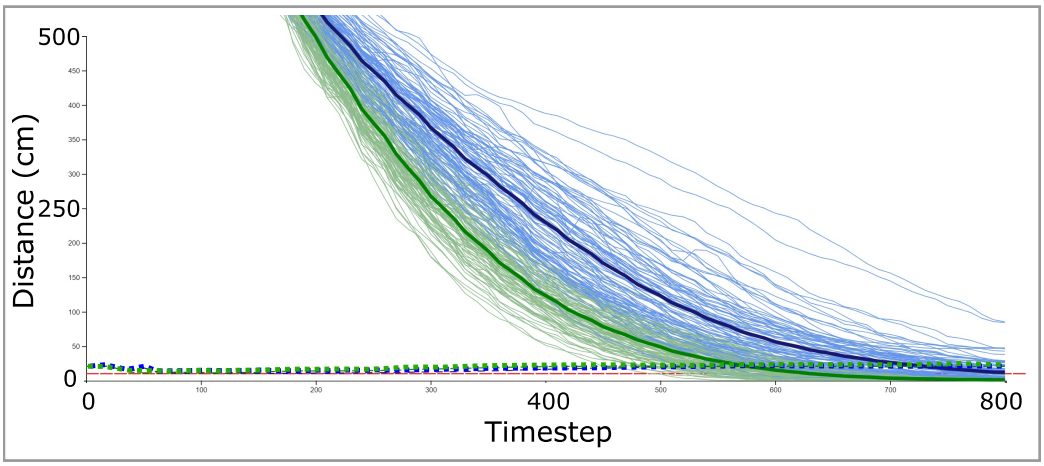
Figure 7.2: Congestion at Multiple Time-steps During a Case 1 Experiment

Left: baseline algorithm, Right: proposed algorithm

Time-steps shown: 100, 200, 300, 400, and 500

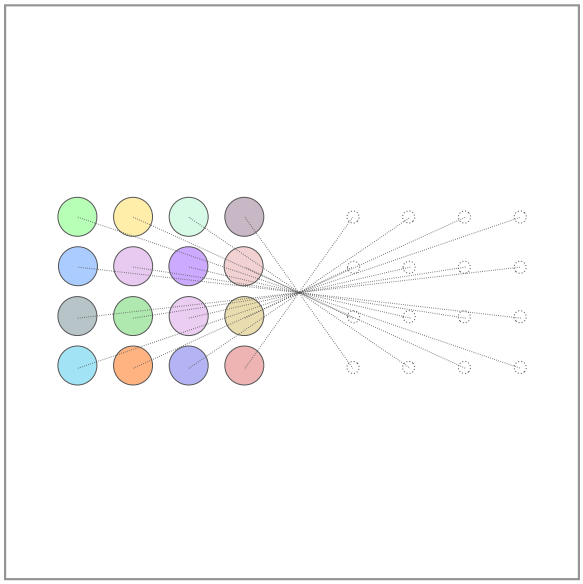


(a) Starting configuration for simulation case 2 with 16 robots instead of 100 for clarity

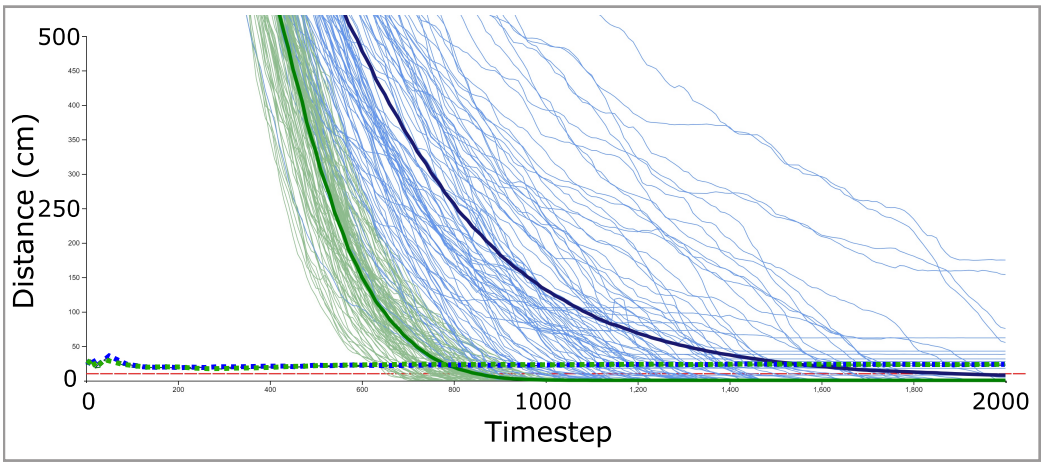


(b) Total distance to goal (solid) and minimum distance to neighbors (dashed) with safety distance in red (dashed). Each simulation is presented as a single line, with the average for each algorithm in bold. Proposed algorithm in green and BVC algorithm with right-hand heuristics in blue.

Figure 7.3: Collision Avoidance Simulation Experiment - Case 2



(a) Starting configuration for simulation case 3 with 16 robots instead of 100 for clarity



(b) Total distance to goal (solid) and minimum distance to neighbors (dashed) with safety distance in red (dashed). Each simulation is presented as a single line, with the average for each algorithm in bold. Proposed algorithm in green and BVC algorithm with right-hand heuristics in blue.

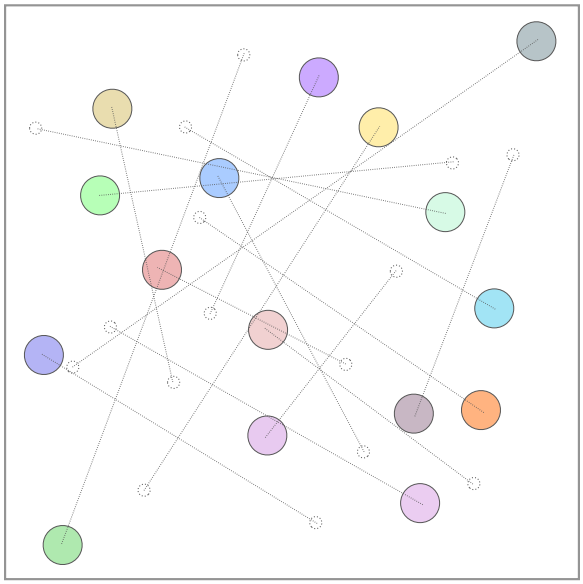
Figure 7.4: Collision Avoidance Simulation Experiment - Case 3

and are considered as performance benchmarks [11]. The results of the experiments for these cases are shown in the figure in similar fashion to the results of the first case shown in figure 7.1.

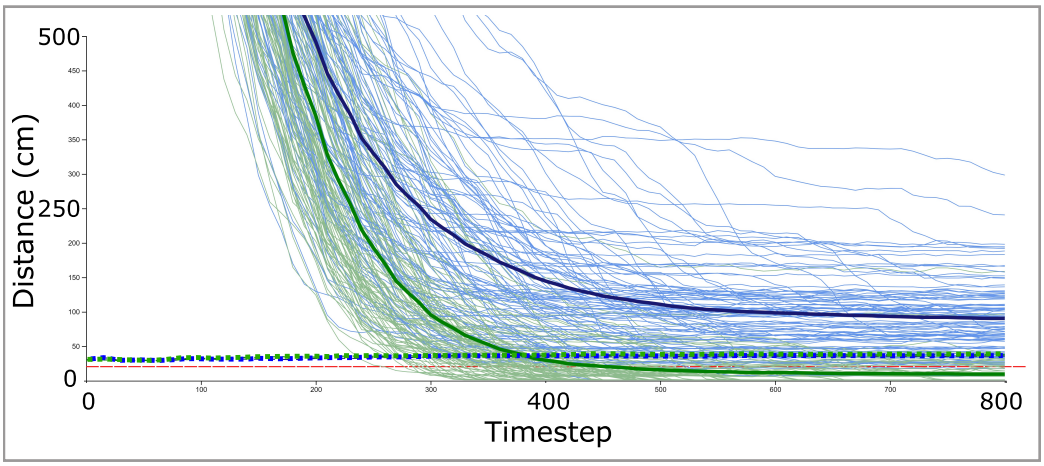
The results show clear performance improvement when using the proposed algorithm. While the proposed algorithm is able to drive the robots to their goals in all the experiments, we can clearly see many experiments where many robots are not able to reach their goals by the end of the recorded timeline for both cases, especially in the more challenging case 3. These cases can be identified by the total distance line (solid blue) not reaching 0 at the final time step in the graph. It is also clear how the proposed algorithm drives the robots to their goals much faster in both cases, and almost twice as fast in case 3.

In the last set of experiments, a set of 100 robots (5 cm radius) are assigned random starting positions in the environment and random goals. The only constraint is that both the starting and final positions need to be in a collision-free configuration. The experiment is repeated 100 times for both algorithms and the results recorded. Figure 7.5 shows the starting configuration and recorded results for these experiments.

This case is particularly important since it demonstrates the advantages the proposed deadlock prediction and recovery stages offer in real scenarios. It clearly shows how the baseline algorithm with the right-hand heuristics fails to drive many robots to their goals due to unsolved deadlocks. This can be identified by the total distance between the robots and their goals (solid blue lines) reaching a minimum of $100cm$ on average across all simulations, and in some simulations as high as $250cm$, meaning that at the end of these experiments many robots were still far from their goals, and the flatter slopes near the end of the experiments, with minimal decreases in distances



(a) Starting configuration for simulation case 4 with 16 robots instead of 100 for clarity



(b) Total distance to goal (solid) and minimum distance to neighbors (dashed) with safety distance in red (dashed). Each simulation is presented as a single line, with the average for each algorithm in bold. Proposed algorithm in green and BVC algorithm with right-hand heuristics in blue.

Figure 7.5: Collision Avoidance Simulation Experiment - Case 4

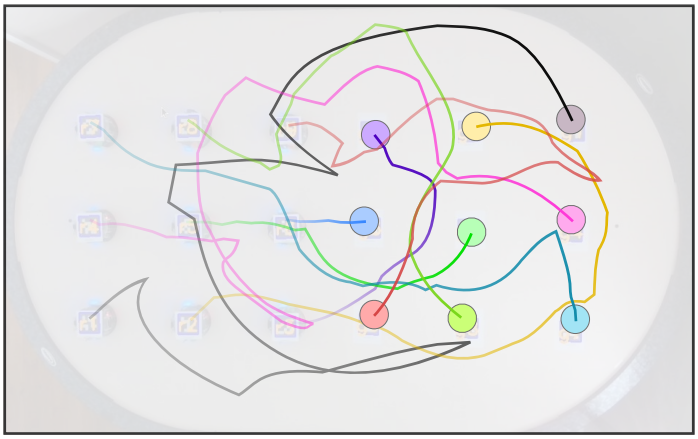
to goals indicate that the robots are not able to proceed any further towards their goals due to being stuck in deadlocks.

In contrast, the proposed algorithm is able to resolve most deadlocks and drive many more robots to their goals, with the total distance between the robots and their goals reaching a minimum of around 10 cm on average across all simulations, almost 10 times better than the baseline algorithm.

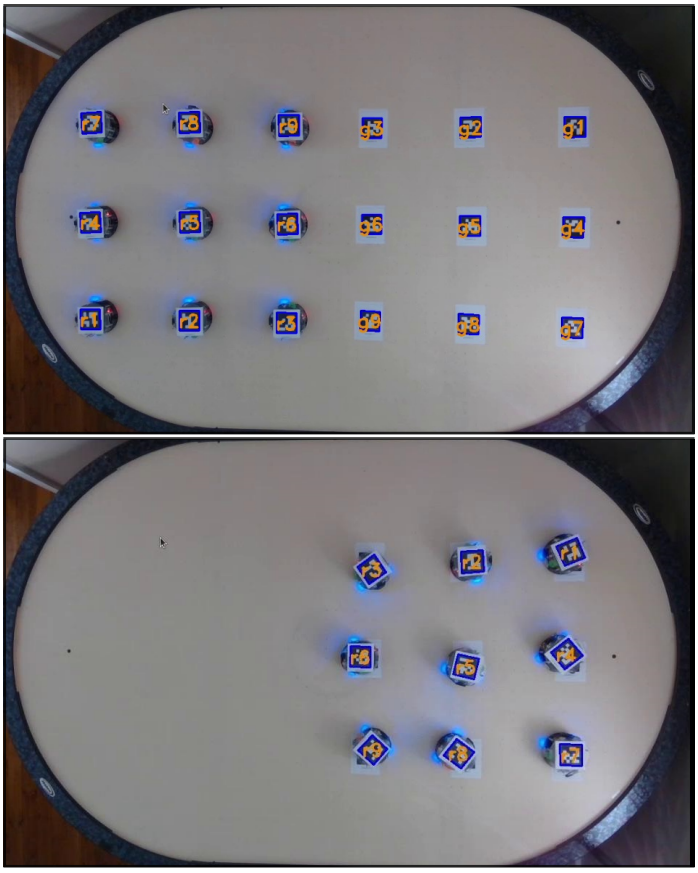
It is also important to note that some of these unresolved deadlocks are due to the random nature of this case where some robots become completely surrounded by static robots (who already reached their goals) and it is impossible to resolve these cases with the current restrictions (distributed algorithm without any communication between agents).

7.1.2 Real-world Validation

Our experimental platform is a set of Pololu 3pi robots fitted with Raspberry Pi 3 A+ single-board computers to perform the on-board processing. Each robot is also fitted with a unique AprilTag [50], and the localization and sensing are performed with an external server fitted with an overhead camera using software previously developed for unmanned surface vehicles [56]. The server continuously detects the robots' positions and simulates their sensor data by restricting the passed positions of neighbors for each robot to only those within their own sensing distance. Each robot is assigned a unique AprilTag as its goal, its position is obtained by the robot once at the start of the experiment and used throughout the experiment. The experiment setup is described in details in chapter 6.



(a)



(b)

Figure 7.6: Collision Avoidance Real-world Validation for Case 3
(a) paths with time encoded by transparency; (b) initial and final configurations.

We performed validation experiments to give confidence to the results obtained in our simulator. We used initial and final configurations arranged in cases 2 and 3 (see figures 7.3 and 7.4), with 9 robots instead of 100. Figure 7.6 shows the paths taken by all robots, as well as the initial and final configurations of robots. These experiments were recorded and are available online through the following link: `https://youtu.be/tvH3xAL9YD0`

The relationship between the robots' initial configuration and the configuration of goals here is as described in case 3. For this case all robots reached their goals without any collisions and the same result was obtained for case 2 (not shown). We repeated both experiments with tighter spacing between the robots and between the goals so that a robot could not pass between two goals. In these experiments all robots reached their goals except the one in the center, since its path was blocked by surrounding robots who had already reached their goals.

We also performed targeted experiments where robots were placed in challenging configurations certain to lead to deadlocks. Figure 7.7 shows the initial configurations and the paths taken by a single robot under the control of each algorithm. It shows how the algorithm with right-hand heuristics failed to resolve the deadlocks and ended up in livelock, while the proposed algorithm successfully drove the robot to its goal every time.

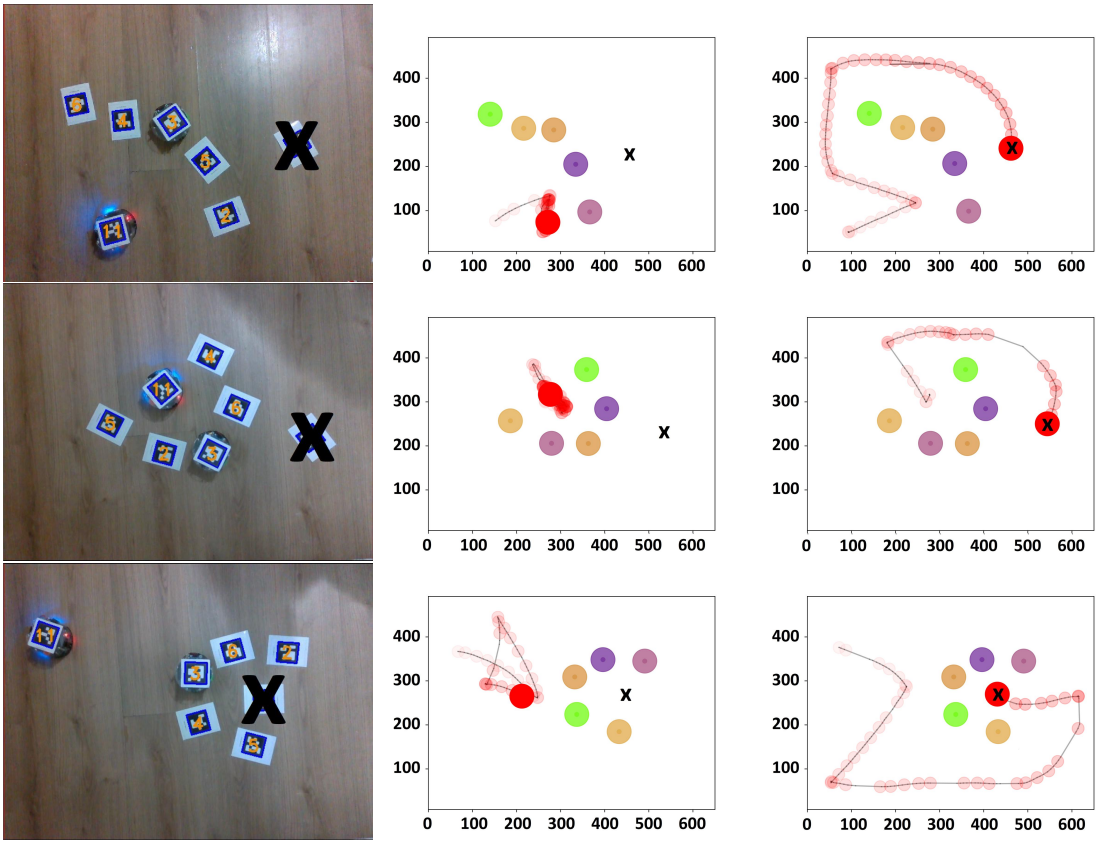


Figure 7.7: Collision Avoidance Real-world validation with Targeted Experiments Left column: initial configuration (robot with tag 11 has X as its goal). Middle column: paths taken with the BVC collision avoidance algorithm with right-hand heuristics. Right column: paths taken with the proposed algorithm.

7.2 Swarm Sorting Algorithm Experiments

7.2.1 Simulation

In this section we will describe the experiments performed in simulation to test and benchmark the proposed swarm sorting algorithm. The simulations are performed using the web-based SwarmJS simulation platform [2] described in chapter 5.

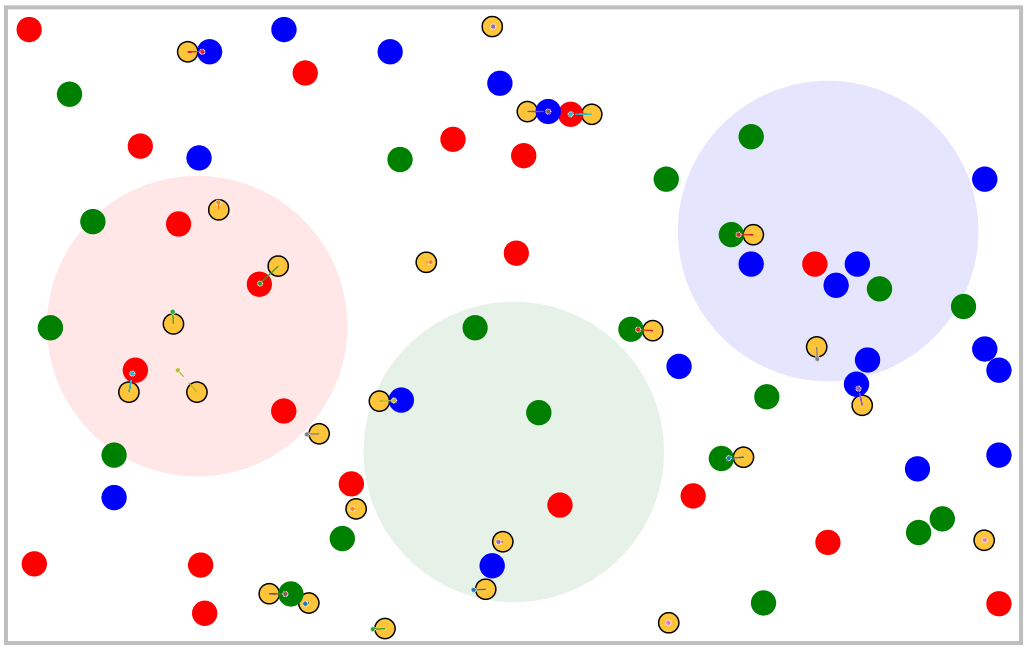
For this experiment setup we simulate a set of disk-shaped single-integrator robots moving in a planar space, with static obstacles in the environment, and multiple groups of targets (pucks); each group has a unique goal area (a circular area of a specific radius around a unique goal position). The pucks are completely passive and only move as a result of their interactions with the robots and the static environment.

The robots are tasked to gather pucks in their respective goal areas using the proposed sorting algorithm. This involves finding pucks and pushing them around obstacles and into their respective goals, and taking care to extract any pucks from another group's goal area and not to disturb pucks who are already in their goal areas.

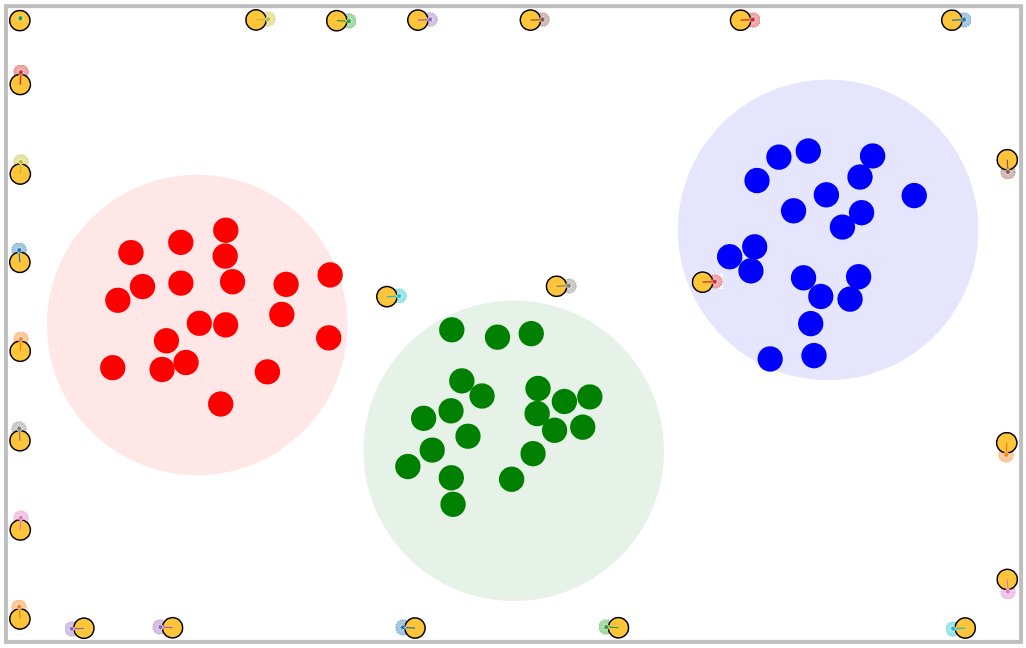
For the first set of experiments, the proposed sorting algorithm is used and two metrics are tracked to evaluate the performance of the algorithm; the total distance between all pucks and their goal areas, and the number of pucks outside of their goal areas at each time step.

Performance is recorded and evaluated for two experiment setups, differing only by the shape of the environment. In the first one, an $800x500$ cm empty environment is used, with a set of 25 robots having a radius of 8 cm, and three groups of pucks,

[2] `https://github.com/m-abdulhak/swarm`

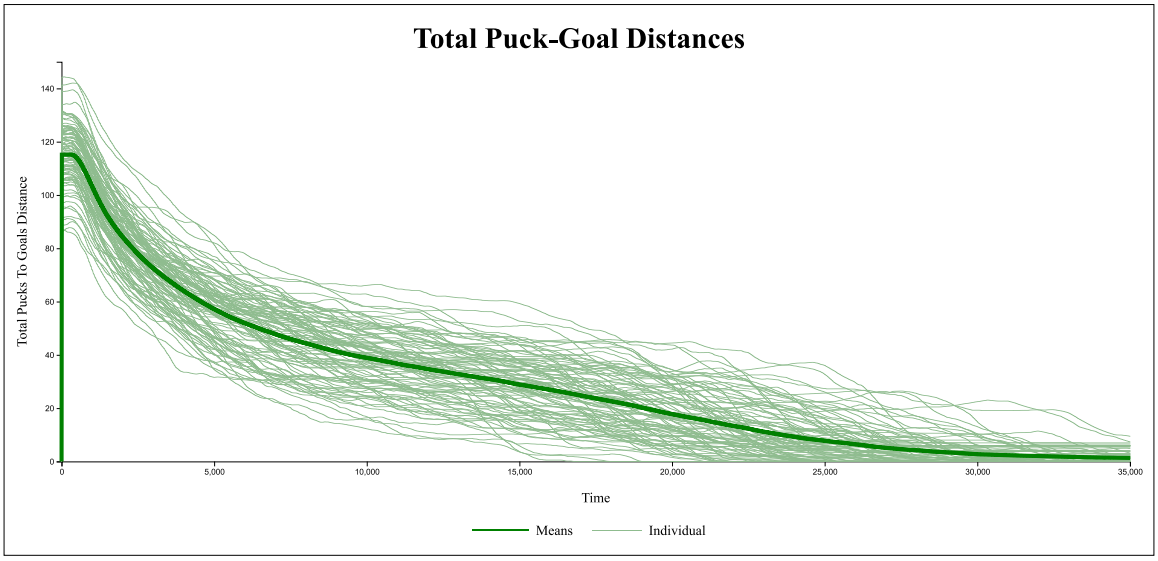


(a) Starting state for a case 1 experiment

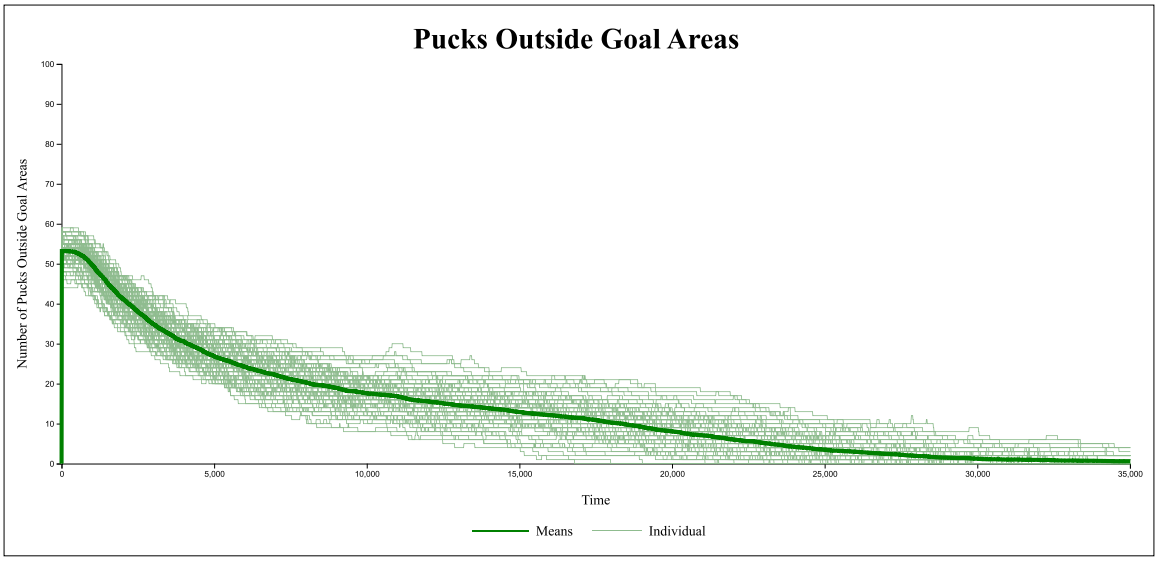


(b) Ending state for a case 1 experiment

Figure 7.8: Swarm Sorting Simulation Experiment Setup - Case 1



(a) Total Puck-Goal Distances for All Case 1 Experiments



(b) Number of Pucks Outside Goal Areas for All Case 1 Experiments

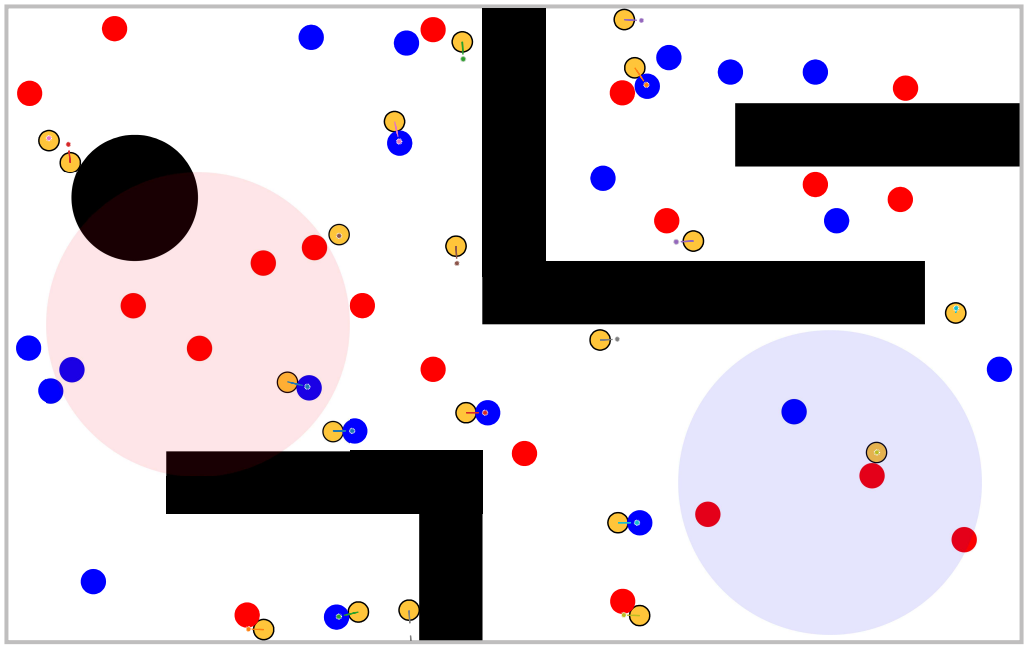
Figure 7.9: Swarm Sorting Simulation Experiment Results - Case 1

each having 20 pucks with a radius of 10 cm and three distinct goal areas as shown in figure 7.8. This figure shows the start and end state of one experiment where all pucks were successfully sorted and placed in their goal areas. The experiment was repeated for 100 times each lasting for $35,000$ time steps.

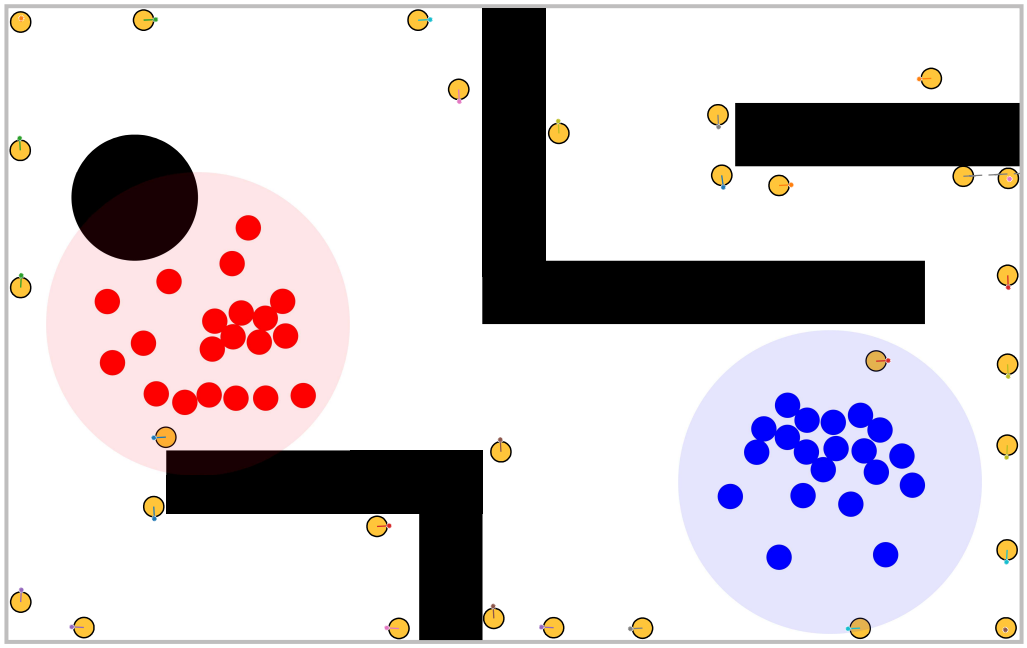
Figure 7.9 shows the results of these experiments. We can see how the total puck to goal distance, and number of pucks outside of their goal areas start high and decrease as the robots push more and more pucks toward their goals, reaching zero by the end of most experiments. In the experiments where the distance does not reach zero, not all pucks reach their targets. This occurs mostly due to pucks getting stuck on the boundary of the environment.

The second experiment setup asses a more realistic use case by adding static obstacles to the environment. Adding multiple static objects produces a more complex map and forces the robots to utilize the global planning and produce more complicated orbits around the environment to be able to cover the inner locations between the static objects. Figure 7.10 shows this experiment setup inside a $800x500$ cm environment, with a set of 25 robots having a radius of 8 cm, and two groups of pucks, each having 20 pucks with a radius of 10 cm and two distinct goal areas. This figure shows the start and end state of one experiment where all pucks where successfully sorted and placed in their goal areas.

This experiment was repeated for 100 times each lasting for $35,000$ time steps and the same previous metrics were recorded. Figure 7.11 shows the results of these experiments where we can see how the algorithm is still able to drive most pucks to their goals in most experiments, with the problem of some stuck pucks still present in some experiments.

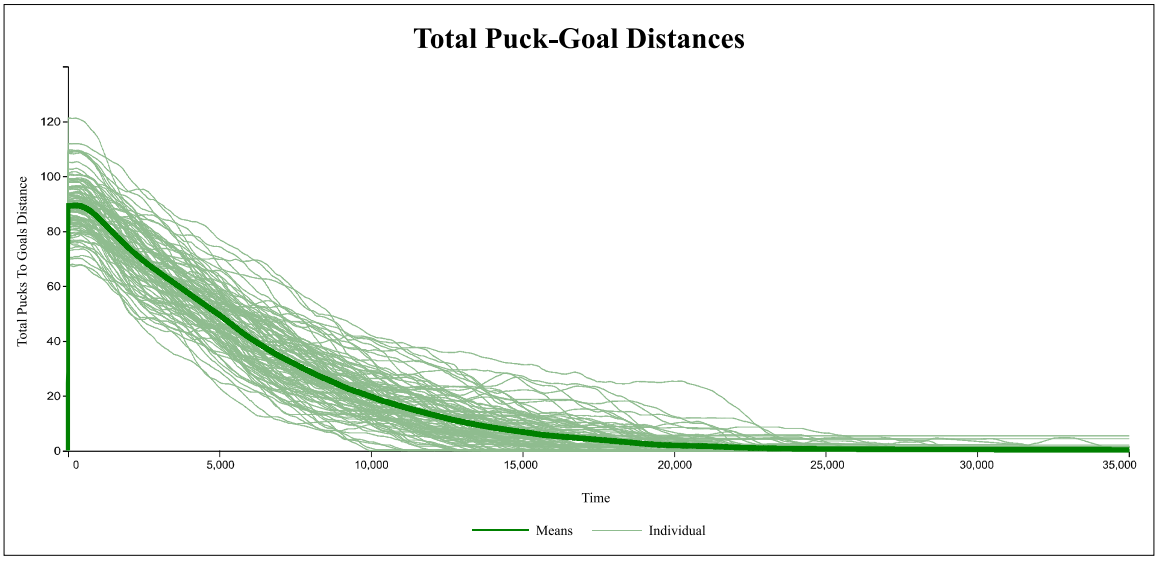


(a) Starting state for a case 2 experiment

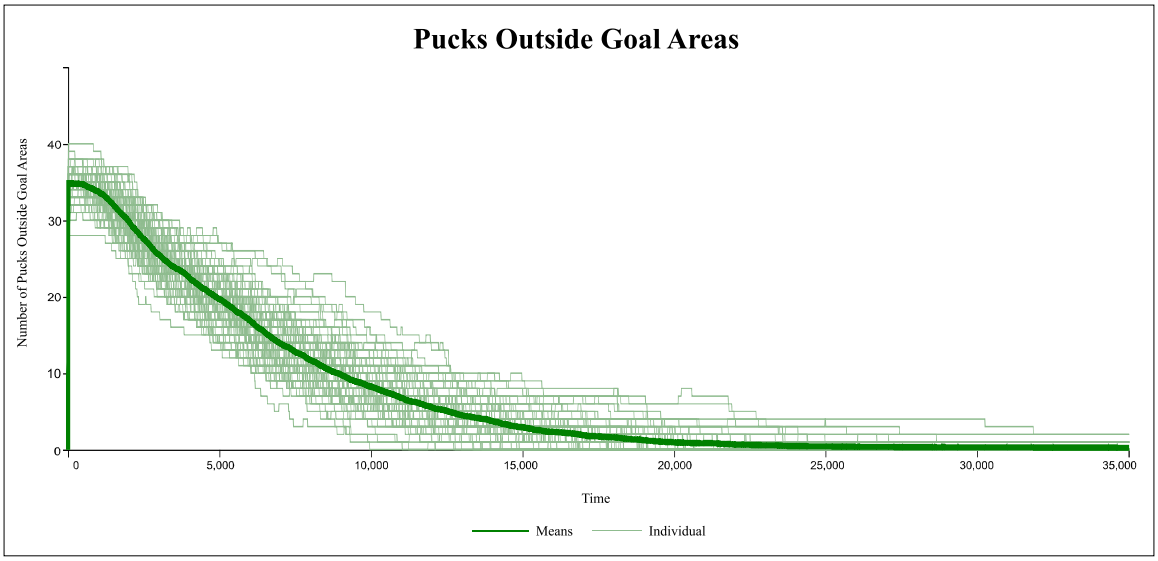


(b) Ending state for a case 2 experiment

Figure 7.10: Swarm Sorting Simulation Experiment Setup - Case 2



(a) Total Puck-Goal Distances for All Case 2 Experiments



(b) Number of Pucks Outside Goal Areas for All Case 2 Experiments

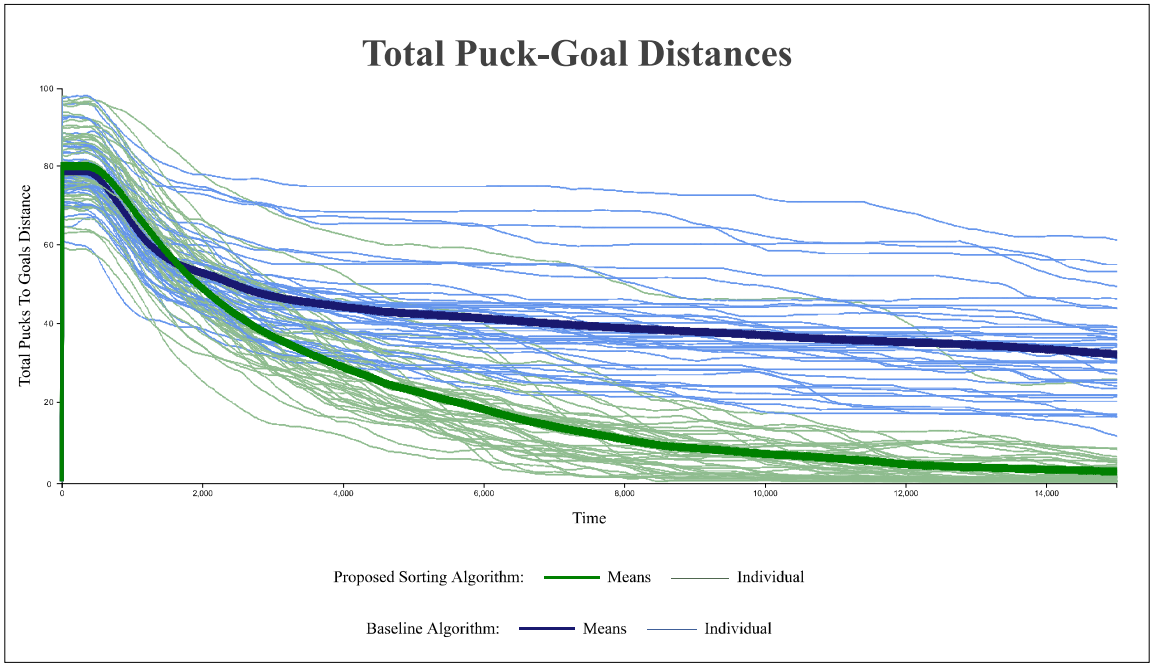
Figure 7.11: Swarm Sorting Simulation Experiment Results - Case 2

In the next set of experiments, the proposed algorithm is compared against a baseline algorithm. For each of these experiments a core feature of the proposed algorithm is disabled and the performance is compared against the full algorithm to measure the impact of this proposed feature and the effectiveness of the proposed approach to solving this sub-problem.

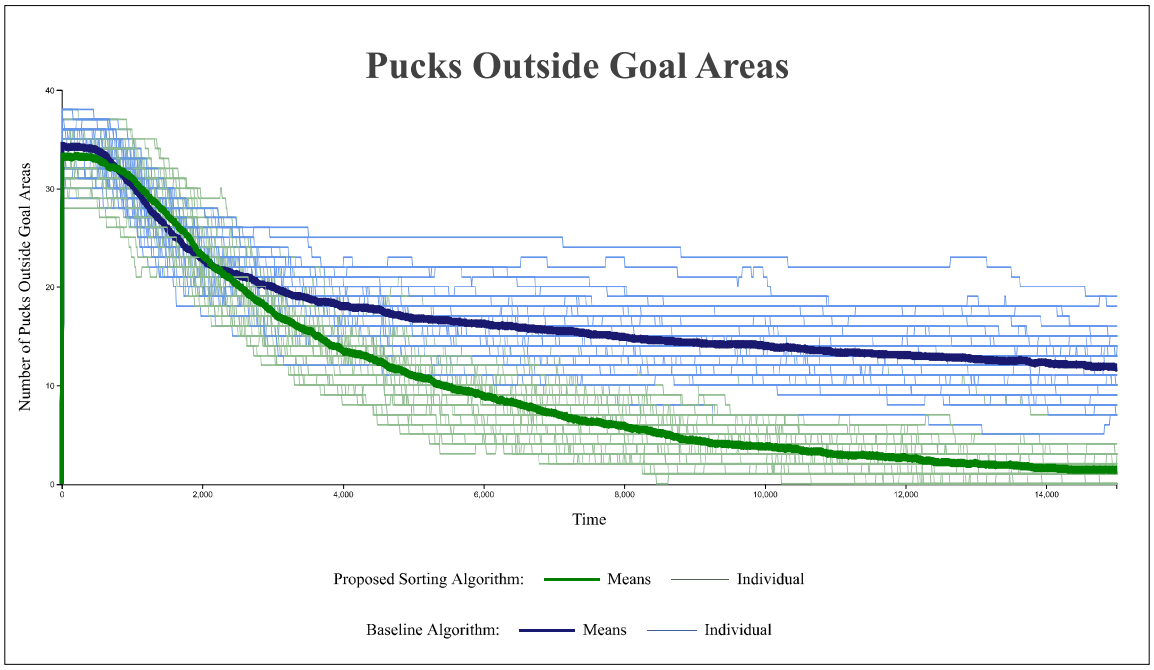
First, we measure the impact of the proposed target conflict avoidance, which works by limiting the possible targets to ones in the current buffered Voronoi cell of the robot. We disable this feature in the baseline algorithm by allowing any robots within the sensing distance to be valid targets for a robot. We use the same experimental setup shown in figure 7.8 but with only two groups of pucks, each having 20 pucks. This particular setup was chosen because including static obstacles in the environment while allowing robots to pursue targets outside of their buffered Voronoi cells leads to very poor performance due to cases where the robots get stuck chasing unreachable targets behind static obstacles (on the other side of a wall). Limiting the allowed targets to the buffered Voronoi cell in The full proposed algorithm prevents these cases because the proposed static obstacle avoidance algorithm automatically limits the buffered Voronoi cell to the visible part with respect to close by static obstacles.

Figure 7.12 shows the results of the 100 performed experiments. Similar to previous benchmarks, the figures show the total puck to goal distances, and the number of pucks outside of their goal areas for each individual experiment as well as the means of all experiments.

The results show that while the proposed algorithm allows the robots to gather most pucks into their goal areas with a mean of only 2 pucks outside of their goal areas



(a) Total Puck-Goal Distances for All Case 3 Experiments



(b) Number of Pucks Outside Goal Areas for All Case 3 Experiments

Figure 7.12: Swarm Sorting Simulation Experiment Results - Case 3

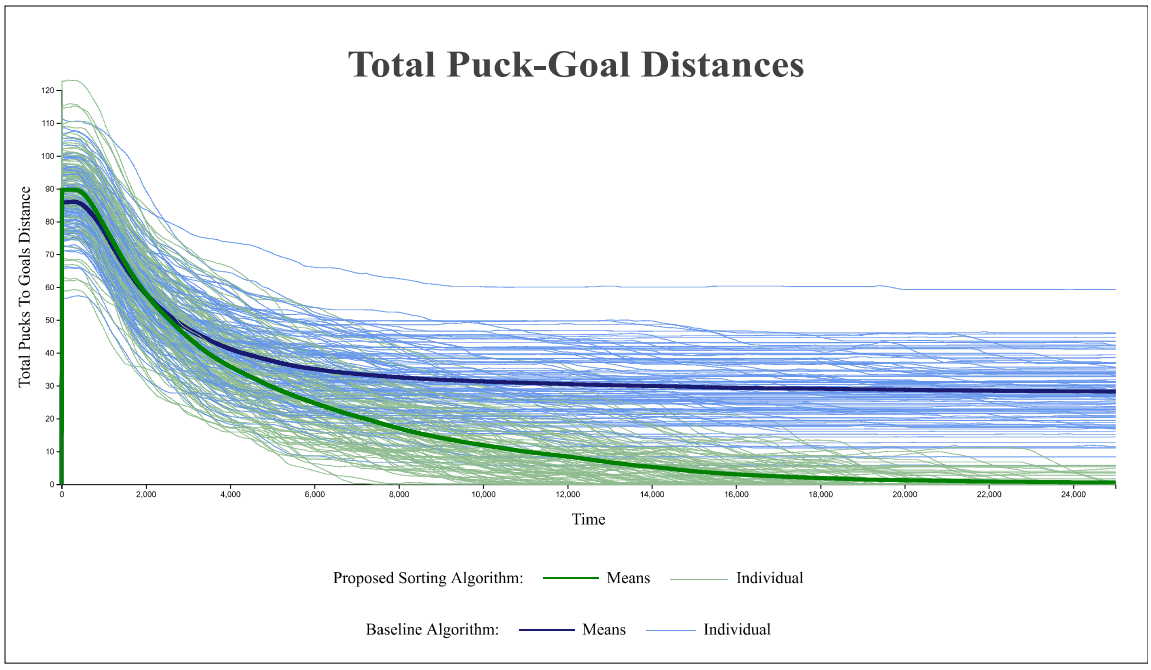
by the end of the experiments; disabling the target conflict avoidance has a profound effect on this success rate, with the mean number or unsorted pucks reaching 15 by the end of the experiments when disabling this feature, with some experiments having as many as half the total number of pucks not sorted by the end of the experiments. The same effect can also be seen in the total distance graph with the mean total distance for these experiments reaching as much as 40 m compared to just 3 m for the full proposed algorithm.

The same approach is used in the next set of experiments, where we assess the impact of the proposed expanding environment orbits. For these experiments we compare the full proposed algorithm against a baseline algorithm where the robots pursue random points in the environment when no suitable targets are found instead of following the proposed expanding orbits.

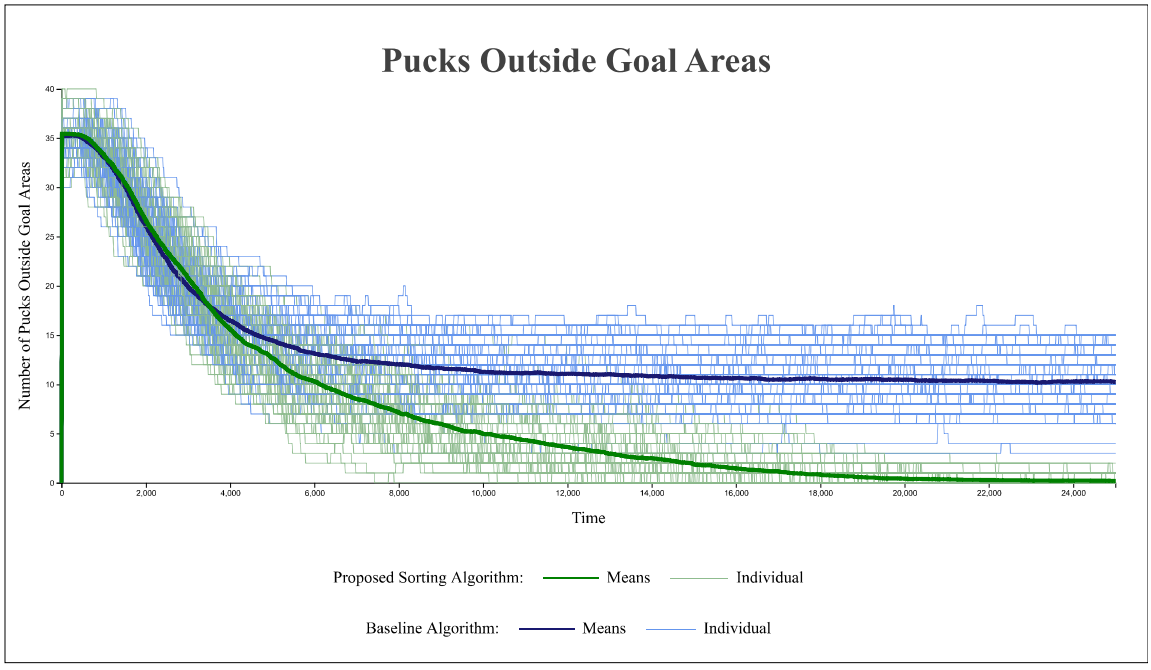
We are interested in studying the effects of this feature in both simple cases with empty environments as well as more complex environments with multiple static obstacles, thus we perform two sets of experiments with the environment setups shown in figures 7.8 and 7.10 but with only two groups of pucks, each having 20 pucks.

The results of the first set of experiments shown in figure 7.14 show the advantage these orbits provide in this complex environment. While the proposed algorithm is able to sort most pucks into their goal areas even with all the static obstacles, the one using random goals is not able to keep up. The mean number of unsorted pucks for the proposed algorithm is almost 0 while the one for the baseline algorithm is much higher at almost 12, and the corresponding mean distances is less than 1 m for the proposed algorithm compared to around 32 m for the baseline algorithm.

The next set of experiments show that the expanding orbits have an impact on

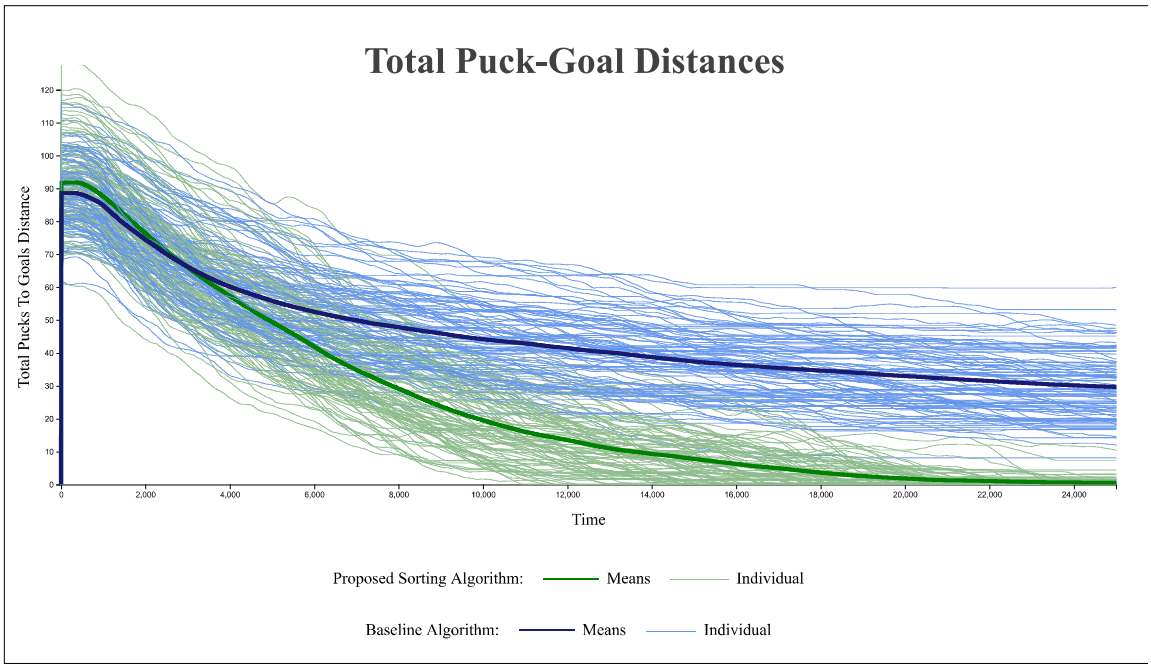


(a) Total Puck-Goal Distances for All Case 4 Experiments

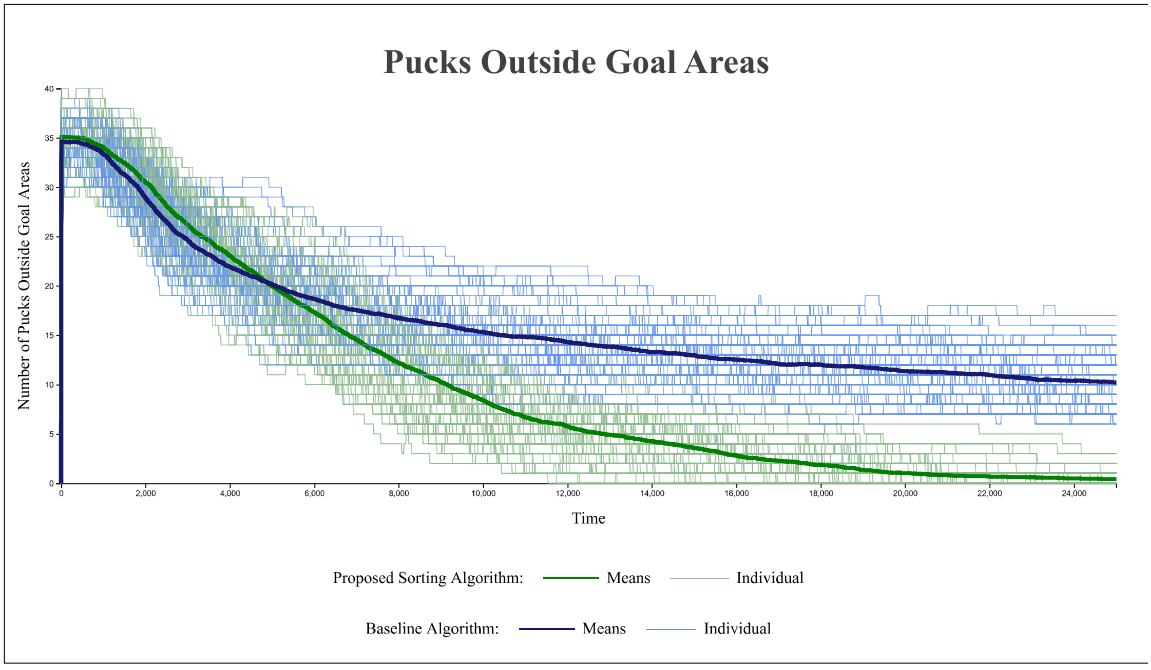


(b) Number of Pucks Outside Goal Areas for All Case 4 Experiments

Figure 7.13: Swarm Sorting Simulation Experiment Results - Case 4



(a) Total Puck-Goal Distances for All Case 5 Experiments



(b) Number of Pucks Outside Goal Areas for All Case 5 Experiments

Figure 7.14: Swarm Sorting Simulation Experiment Results - Case 5

the speed and effectiveness of the sorting process even in simple environments as seen in figure 7.13, with the mean number of unsorted pucks by the end of the experiments decreasing from around 10 when using random goals compared to almost 0 when using the proposed expanding orbits, and the corresponding mean puck to goal distances also decreasing from around 28 m to less than 1 m.

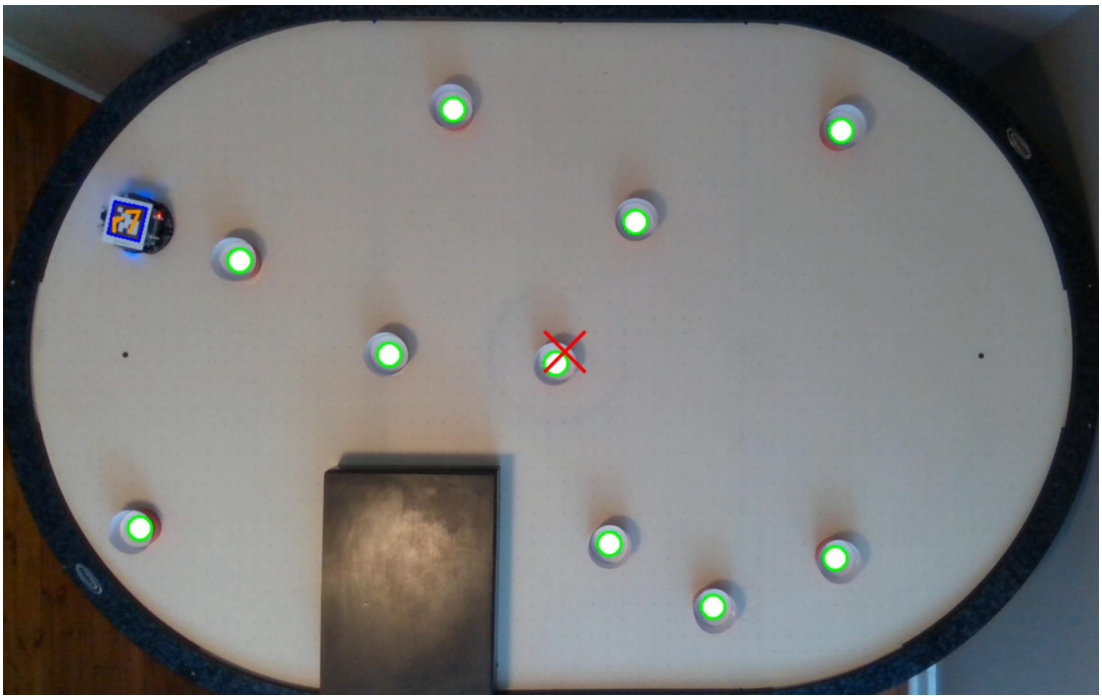
7.2.2 Real-world Validation

A validation experiment using real robots was performed to insure that the results obtained with the simulation are credible and that the proposed algorithm performs well in real-world scenarios.

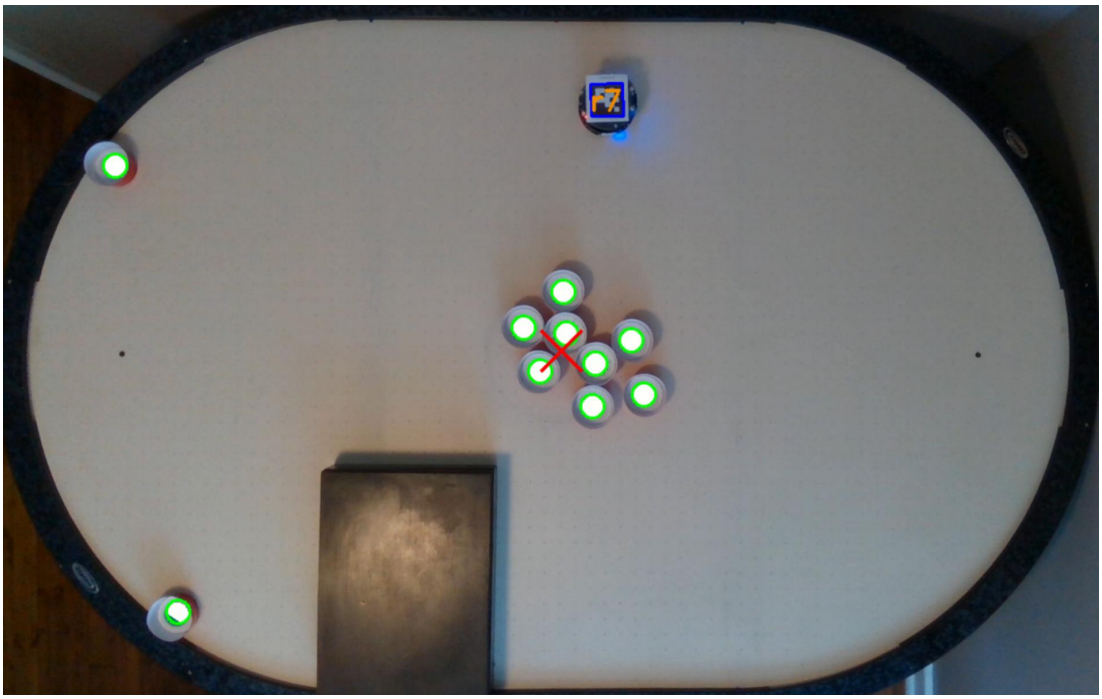
The same experimental platform discussed in chapter 6 is used to perform this experiment. We use the setup shown in figure 7.15 where the proposed algorithm is deployed on a single robot with the task of gathering 10 randomly distributed pucks around the stadium-shaped table into a single goal area at the center of the table.

Figure 7.15 shows the starting and final states of this experiment. The robot was able to gather all pucks into their goal areas except for two pucks that got stuck near the edge of the table and the robot could not retrieve them.

The algorithm performed similar to the expectations set out by the simulation results; the pucks getting stuck near the boundary of the environment was also observed in the simulation when the radius of the pucks was smaller than that of the robots. These results show that the proposed algorithm shows promising results with drawbacks that still need to be addressed.



(a) Starting State for Real World Validation Experiment



(b) Ending State for Real World Validation Experiment

Figure 7.15: Swarm Sorting Validation Experiment Result

Chapter 8

Conclusion

This thesis presented a distributed multi-robot collision avoidance algorithm integrating a novel deadlock avoidance algorithm with a buffered Voronoi cells based approach. Collision avoidance is guaranteed and only the position of the robot and neighboring robots is needed. While our algorithm does not completely eliminate deadlocks, both simulations and experiments showed that it offers drastic improvement in resolving deadlocks compared to the buffered Voronoi cells based approach with right-hand heuristics. We also proposed a new mechanism for static obstacle avoidance based on the same concept of buffered voronoi cells. The algorithm works by finding the closest obstacle to the robot at each time step and modifying the voronoi cell in a way that prevents the robot from colliding with that obstacle.

We also presented a distributed sorting algorithm for gathering pucks into specific goal areas in the environment. It uses the collision avoidance algorithm as the local movement planning algorithm and adds algorithms for target selection, target conflict avoidance, control strategy, and global planning for calculating directions from tar-

gets' current positions towards their goals. The simulations showed that the proposed algorithm works reliably even in complex environments, and the physical experiment validated that the proposed algorithm works similarly on real robots. Both the simulations and real experiments showed that there are still issues that can be improved such as pucks getting stuck near the boundaries of the environment.

8.1 Future Work

While the sorting algorithm was demonstrated on real robots, more experiments can be done with more general use cases, since only a subset of the algorithm's behaviours were assessed, namely a single group of pucks and a single robot were used for the experiment. The currently available experimental setup and timing restrictions prevented us from performing further tests in this area.

In terms of improvements, the following is a list of known issues or feature suggestions that could be the focus of future work in this area:

- Due to the collision avoidance algorithm's lacks of any communication capabilities (by design), some scenarios arise when a robot's path is blocked by one or more static robots, causing the robot never be able to reach its goal. This is seen in both simulations and the real world experiments. Introducing a simple communication mechanism that allows a robot to signal to nearby neighbors that it is stuck and needs them to make way for it to pass could dramatically increase the success rate of avoiding such deadlocks. It would only be used in these situations, while keeping all the processing and decision making fully distributed.

- An always present issue with the swarm sorting experiments and simulations is pucks getting stuck near the edges of the environment with no way for robots to retrieve them due to their circular shape. This was mostly avoided in the simulations by making the pucks larger than the robots allowing them to slide behind them and retrieve them while following their expanding environment orbits. A better approach could be implemented to fix this issue, such as changing the shape of robots by adding a pointed wedge that would allow the robots to retrieve targets that are near the boundaries of the environment, this approach is used by work in the BOTS lab [59]. This would require modifying the collision avoidance algorithm to allow the robots to take the wedge's shape into account when recognizing and avoiding other robots.

- A possible continuation of this work could be investigating the possibility of using the same algorithm to build specific structures from multiple materials by gathering different type of targets into specific areas with different shapes. Only circular areas around a goal position were used in this work, but the underlying distance transform algorithm that is used for find pucks' paths to their goals could be used with goal areas of any shape.

Bibliography

[1] Mohammed Abdullhak and Andrew Vardy. Deadlock prediction and recovery for distributed collision avoidance with buffered voronoi cells. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 429–436. IEEE.

[2] Kazuhiro Ohkura, Toshiyuki Yasuda, Yoshiyuki Matsumura, and Masaki Kadota. Gpu implementation of food-foraging problem for evolutionary swarm robotics systems. In *International Conference on Swarm Intelligence*, pages 238–245. Springer, 2014.

[3] Gerardo Beni and Jing Wang. Swarm intelligence in cellular robotic systems. In *Robots and biological systems: towards a new bionics?*, pages 703–712. Springer, 1993.

[4] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[5] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications. *Frontiers in Robotics and*

AI, 7:36, 2020.

[6] Giandomenico Spezzano. *Swarm Robotics*. MDPI-Multidisciplinary Digital Publishing Institute, 2019.

[7] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.

[8] Heiko Hamann. *Swarm robotics: A formal approach*. Springer, 2018.

[9] Marco Dorigo, Mauro Birattari, and Manuele Brambilla. Swarm robotics. *Scholarpedia*, 9(1):1463, 2014.

[10] Scott Camazine, Jean-Louis Deneubourg, Nigel R Franks, James Sneyd, Eric Bonabeau, and Guy Theraula. *Self-organization in biological systems*. Princeton university press, 2003.

[11] D. Zhou, Z. Wang, S. Bandyopadhyay, and M. Schwager. Fast, on-line collision avoidance for dynamic vehicles using buffered voronoi cells. *IEEE Robotics and Automation Letters*, 2(2):1047–1054, 2017.

[12] L. Aguilar, R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Ten autonomous mobile robots (and even more) in a route network like environment. In *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, volume 2, pages 260–267 vol.2, 1995.

[13] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.

[14] J. van den Berg, Ming Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *2008 IEEE International Conference on Robotics and Automation*, pages 1928–1935, 2008.

[15] J. V. D. Berg, Stephen J. Guy, M. Lin, and D. Manocha. Reciprocal n-body collision avoidance. In *ISRR*, 2009.

[16] M. Jager and B. Nebel. Decentralized collision avoidance, deadlock detection, and deadlock resolution for multiple mobile robots. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*, volume 3, pages 1213–1219 vol.3, 2001.

[17] J. van den Berg, J. Snape, S. J. Guy, and D. Manocha. Reciprocal collision avoidance with acceleration-velocity obstacles. In *2011 IEEE International Conference on Robotics and Automation*, pages 3475–3482, 2011.

[18] J. Snape, J. v. d. Berg, S. J. Guy, and D. Manocha. The hybrid reciprocal velocity obstacle. *IEEE Transactions on Robotics*, 27(4):696–706, 2011.

[19] D. Wilkie, J. van den Berg, and D. Manocha. Generalized velocity obstacles. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5573–5578, 2009.

[20] J. Alonso-Mora, A. Breitenmoser, P. Beardsley, and R. Siegwart. Reciprocal collision avoidance for multiple car-like robots. In *2012 IEEE International Conference on Robotics and Automation*, pages 360–366, 2012.

[21] A. Giese, D. Latypov, and N. M. Amato. Reciprocally-rotating velocity obstacles. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3234–3241, 2014.

[22] A. Best, S. Narang, and D. Manocha. Real-time reciprocal collision avoidance with elliptical agents. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 298–305, 2016.

[23] D. H. Shim, H. J. Kim, and S. Sastry. Decentralized nonlinear model predictive control of multiple flying robots. In *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, volume 4, pages 3621–3626 vol.4, 2003.

[24] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous robot vehicles*, pages 396–404. Springer, 1986.

[25] Jean-Claude Latombe, Anthony Lazanas, and Shashank Shekhar. Robot motion planning with uncertainty in control and sensing. *Artificial Intelligence*, 52(1):1–47, 1991.

[26] Daniel Morgan, Soon-Jo Chung, and Fred Y. Hadaegh. Model predictive control of swarms of spacecraft using sequential convex programming. *Journal of Guidance, Control, and Dynamics*, 37(6):1725–1740, 2014.

[27] Daniel Morgan, Soon-Jo Chung, Lars Blackmore, Behcet Acikmese, David Bayard, and Fred Y Hadaegh. Swarm-keeping strategies for spacecraft under j2 and atmospheric drag perturbations. *Journal of Guidance, Control, and Dynamics*, 35(5):1492–1506, 2012.

[28] H. Zhu and J. Alonso-Mora. Chance-constrained collision avoidance for mavs in dynamic environments. *IEEE Robotics and Automation Letters*, 4(2):776–783, 2019.

[29] Y. F. Chen, M. Liu, M. Everett, and J. P. How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 285–292, 2017.

[30] D. E. Chang, S. C. Shadden, J. E. Marsden, and R. Olfati-Saber. Collision avoidance for multiple agent systems. In *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, volume 1, pages 539–543 Vol.1, 2003.

[31] A. D. Ames, X. Xu, J. W. Grizzle, and P. Tabuada. Control barrier function based quadratic programs for safety critical systems. *IEEE Transactions on Automatic Control*, 62(8):3861–3876, 2017.

[32] M. Wang and M. Schwager. Distributed collision avoidance of multiple robots with probabilistic buffered voronoi cells. In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 169–175, 2019.

[33] H. Zhu and J. Alonso-Mora. B-uavc: Buffered uncertainty-aware voronoi cells for probabilistic multi-robot collision avoidance. In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 162–168, 2019.

[34] A. Pierson, W. Schwarting, S. Karaman, and D. Rus. Weighted buffered voronoi cells for distributed semi-cooperative behavior. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5611–5617, 2020.

[35] Gerardo Beni. *Swarm Intelligence*, pages 1–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019.

[36] Gerardo Beni and Jing Wang. Swarm intelligence in cellular robotic systems. In Paolo Dario, Giulio Sandini, and Patrick Aebischer, editors, *Robots and Biological Systems: Towards a New Bionics?*, pages 703–712, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[37] Alan F. T. Winfield. *Foraging Robots*, pages 1–26. Springer New York, New York, NY, 2009.

[38] Elizabeth A Langridge, Nigel R Franks, and Ana B Sendova-Franks. Improvement in collective performance with experience in ants. *Behavioral Ecology and Sociobiology*, 56(6):523–529, 2004.

[39] Alejandro G Farji-Brener, Luciana Elizalde, Hermógenes Fernández-Marín, and Sabrina Amador-Vargas. Social life and sanitary risks: evolutionary and current ecological conditions determine waste management in leaf-cutting ants. *Proceedings of the Royal Society B: Biological Sciences*, 283(1831):20160625, 2016.

[40] Davide Santoro, Stephen Hartley, and Philip J Lester. Behaviourally specialized foragers are less efficient and live shorter lives than generalists in wasp colonies. *Scientific reports*, 9(1):1–10, 2019.

[41] Anna Dornhaus. Specialization does not predict individual efficiency in an ant. *PLoS biology*, 6(11):e285, 2008.

[42] Jean-Louis Deneubourg, Simon Goss, Nigel Franks, Ana Sendova-Franks, Claire Detrain, and Laeticia Chrétien. The dynamics of collective sorting robot-like ants and ant-like robots. In *From animals to animats: proceedings of the first international conference on simulation of adaptive behavior*, pages 356–365, 1991.

[43] Ralph Beckers, Owen E Holland, and Jean-Louis Deneubourg. Fom local actions to global tasks: Stigmergy and collective robotics. In *Prerational Intelligence: Adaptive Behavior and Intelligent Systems Without Symbols and Logic, Volume 1, Volume 2 Prerational Intelligence: Interdisciplinary Perspectives on the Behavior of Natural and Artificial Systems, Volume 3*, pages 1008–1022. Springer, 2000.

[44] Marinus Maris and René Boeckhorst. Exploiting physical constraints: heap formation through behavioral error in a group of robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS'96*, volume 3, pages 1655–1660. IEEE, 1996.

[45] Chris Melhuish, Owen Holland, and Steve Hoddell. Collective sorting and segregation in robots with minimal sensing. In *Proc. 5th Int. Conf. Simulation of Adaptive Behaviour*, pages 465–470, 1998.

[46] Chris Melhuish, Ana B Sendova-Franks, Sam Scholes, Ian Horsfield, and Fred Welsby. Ant-inspired sorting by robots: the importance of initial clustering. *Journal of the Royal Society Interface*, 3(7):235–242, 2006.

[47] Tao Wang and Hong Zhang. Multi-robot collective sorting with local sensing. In *Ieee intelligent automation conference (IAC)*. Citeseer, 2003.

[48] Andrew Vardy. Accelerated patch sorting by a robotic swarm. In *2012 Ninth Conference on Computer and Robot Vision*, pages 314–321. IEEE, 2012.

[49] Andrew Vardy, Gregory Vorobyev, and Wolfgang Banzhaf. Cache consensus: rapid object sorting by a robotic swarm. *Swarm Intelligence*, 8(1):61–87, 2014.

[50] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407, 2011.

[51] Jörg-Rüdiger Sack and Jorge Urrutia. *Handbook of computational geometry*. Elsevier, 1999.

[52] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley Series in Probability and Statistics. Wiley, 2009.

[53] Andrew Vardy. Orbital construction: Swarms of simple robots building enclosures. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 147–153. IEEE, 2018.

[54] James A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[55] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, et al. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1), 2021.

[56] C. Gregory and A. Vardy. microUSV: A low-cost platform for indoor marine swarm robotics research. *HardwareX*, 2020.

[57] Protocol buffers. `https://developers.google.com/protocol-buffers/`. Accessed: 2020-09-30.

[58] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[59] Andrew Vardy. Robot distancing: Planar construction with lanes. In *International Conference on Swarm Intelligence*, pages 229–242. Springer, 2020.