A Reinforcement Learning Approach to Multi-Robot Planar Construction

Caroline Strickland

Computer Science M.Sc. Thesis
Memorial University of Newfoundland
December, 2019

Abstract

We consider the problem of shape formation in a decentralised swarm of robots trained using a subfield of machine learning called reinforcement learning. Shapes are formed from ambient objects which are pushed into a desired pattern. The shape is specified using a projected scalar field that the robots can locally sample. This scalar field plays a similar role to the pheromone gradients used by social insects such as ants and termites to guide the construction of their sophisticated nests. The overall approach is inspired by the previously developed orbital construction algorithm.

Reinforcement learning allows one or more agents to learn the best action to take in a given situation by interacting with their environment and learning a mapping from states to actions. Such systems are well-suited to robotics, as robots often interact with complex environments through a variety of sensors and actuators. When reinforcement learning is applied to a multi-agent system, it is called 'multi-agent reinforcement learning' (MARL). The main feature that MARL offers is flexibility — a multi-agent decentralised system can have agents added, removed, or reconstructed without need for rewriting the system. This allows for more robust solutions due to its ability to cope with failure.

With the use of simulators paired with MARL, we can effectively learn policies that result in the formation of unique shapes. This is a vast improvement over hand-coded solutions, as it removes dependence on hard-coded actions. Reinforcement learning eliminates the need for writing control algorithms in the first place — which tend to be be extremely task-specific and time-consuming.

Contents

List of Figures

List of Tables

Nomenclature

MAL Multi-Agent Learning. 17

MARL Multi-Agent Reinforcement Learning. 14

MAS Multi-Agent System. 14

ML Machine Learning. 20

OC Orbital Construction. 1

RL Reinforcement Learning. 11

SAS Single-Agent Systems. 18

1 Introduction

Shape formation is a generic term for the problem of determining how a desired geometrical pattern can be both obtained and then maintained by a swarm of robots without any centralized coordination [42]. We are interested in the capacity of swarms of robots to form shapes using objects in their environment. This has direct application to topics such as cleaning (merging all waste objects into a single cluster) and recycling (segregating objects by type and moving them to desired collection points). It is also a useful tool in construction–related tasks such as forming walls and enclosures. In our research, shapes are formed from ambient objects which are pushed into a desired shape specified by a *projected scalar field* that robots can sample locally. This scalar field plays a similar role to the pheromone gradients used by social insects such as ants and termites to guide the construction of their sophisticated nests. We take inspiration from the previously developed Orbital Construction (OC) algorithm [52], its objective, and the environment that it acts within. In our research, we use reinforcement learning to learn a state to action mapping (called a *policy*) that allows agents to construct formations without the need for hand-coding algorithmic solutions, reduces parameter tuning and testing, and increases abstraction.

1.1 Orbital Construction

One mechanism that has been proposed to explain some aspects of social insect construction is the use of a *template*, often consisting of some sensed environmental parameter or pheromone emitted by the insects themselves [50].

A well-known example of such a template is the pheromone emitted by the queen

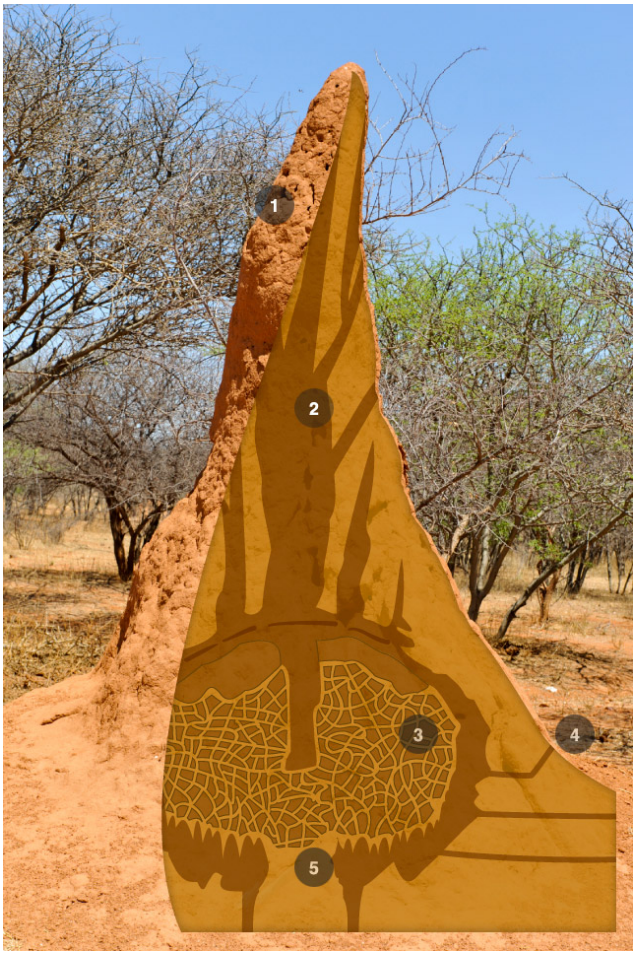


Figure 1: A figure showing the layout of a typical termite mound provided by PBS [34]. Pheremones used by insects to create this structure are used as inspiration in the OC algorithm. The royal chamber can be seen at point 3 of the digram.

in a colony of termites (genus Macrotermes). The concentration of this pheromone is used to guide the construction of a "royal chamber" that encloses the queen, which can be seen in the bottom of Figure 1.

In a recent review of collective robot construction, a number of different organizing principles were identified, including the use of such templates [35]. Experiments have been conducted on physical robots showing the construction of walls where the template is sensed as a light field [47] or through sensing landmarks combined with odometry [45]. An example of this construction is demonstrated in [24]. Ladley and Bullock simulate the construction of a royal chamber using a fixed-rate source of queen 'pheromone', seen in Figure 4. The approach that we take is to assume that a set of robots can sense such a template within an enclosed environment. In this environment, circular agents are free to move around in a rectangular enclosed space which is also occupied by circular puck objects whose position can only be changed by applying a force to them via an agent collision. In addition to the rectangular boundary, it contains a *scalar field* grid projected onto the 'floor' surface, where the grid values range between 0 and 1. This grid is what is used as the template. The scalar field is illustrated in Figure 2 where lightest spaces have a scalar value of 1, while the darkest have a scalar value of 0.

Figure 2: Scalar field template used to create annulus (ring-shaped) shapes in both the OC algorithm and in our RL research. The small squares within the grid compose the environment's projected scalar field, with each square ranging from a value of 0 (black) to 1 (white), which guides shape formation.

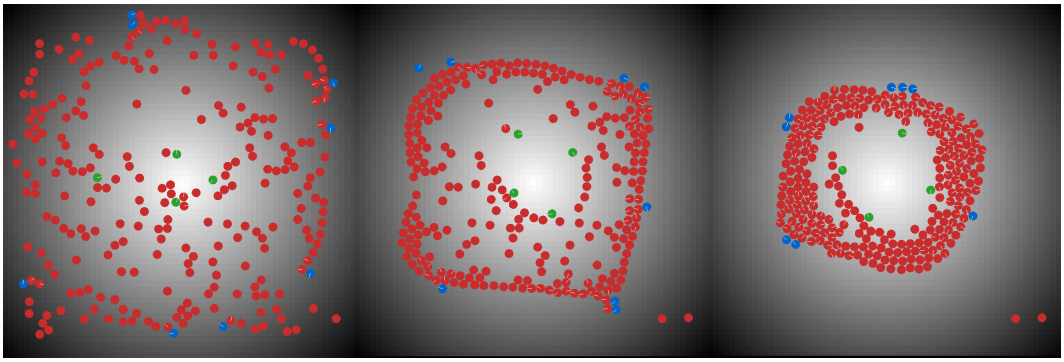


Figure 3: Three states of our environment demonstrating the progression of orbital construction annulus formation using outies (blue circles), innies (green circles) and 250 pucks (red circles). Shown are an initial randomized configuration of pucks (left), an intermediary state during construction (middle), and a successfully constructed shape (right).

In [52], an algorithm is proposed which can form various enclosed shapes based on sensing scalar values from a field that serves as template to specify the shape.

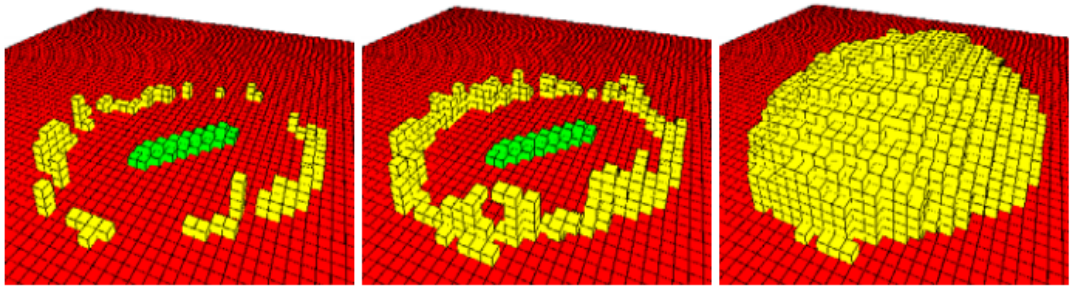


Figure 4: The simulated construction of a termite royal chamber (Ladley et al., 2004. Used with permission).

This algorithm, Orbital Construction (OC), is guided by the scalar field to orbit the growing structure, pushing objects inward or outward to join the structure. The OC algorithm is intentionally minimalistic which allows for ease of implementation and reduces the need for parameter tuning.

The OC algorithm, demonstrated running in Figure 3 is based on the ability of agents to locally sample the scalar field with the use of sensors. A fixed threshold value, τ ($0 \leq \tau \leq 1$) specifies a contour line of the scalar field. If two thresholds are specified then we can define a region which is to be filled in with objects. In the simplest case, the scalar field is defined by a single seed point (a point on the scalar field given by a pseudo-random number generator to ensure randomness) at scalar value 1 with every other point having a value of $1 - d$ where d is the Euclidean distance from the seed point. In this case, specifying two thresholds defines an annulus — a shape that can be seen forming in Figure 3.

There are two types of sensors that each agent is equipped with, each positioned symmetrically about the forward heading of the robot. Within the simulator, these sensors do not have any physics associated with them and do not collide with agents, pucks, or walls in the environment. The first sensors are the 4 circular

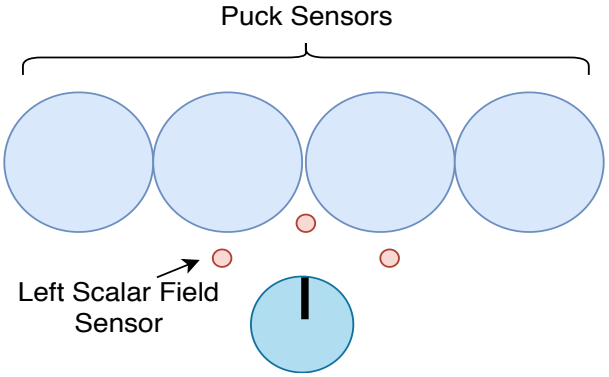


Figure 5: The sensor configuration for each agent. Agent shown on bottom as a teal circle with its current heading (black line). Sensors are rigidly fixed relative to the position and heading of the agent, moving with the agent as it moves.

puck sensors which give a reading of 0 or 1, representing whether a puck in the environment currently intersects the sensor's area. Next, there are 3 *scalar field sensors* which can read the floating point value of the scalar field directly below the center of each sensor.

For the OC algorithm, two different types of agents were defined—*innies* which nudge objects outwards from the seed point, and *outies* which nudge objects inwards. Both innies and outies are equipped with the same sensors in the same configuration as described above. The puck and scalar field sensor layout for each of these agents can be seen in Figure 5. The OC algorithm combines the use of a scalar field for guidance with the minimalistic approach to clustering objects discovered by Gauci et al [12]. OC uses the scalar field to define the direction of these movements so that robots can nudge objects inwards or outwards depending upon what is required. For outies, sensed objects are nudged inwards until the threshold value of the scalar field is reached.

For innies, sensed objects are nudged outwards until the threshold value of the scalar field is reached. To do this, an agent must first figure out its direction orien-

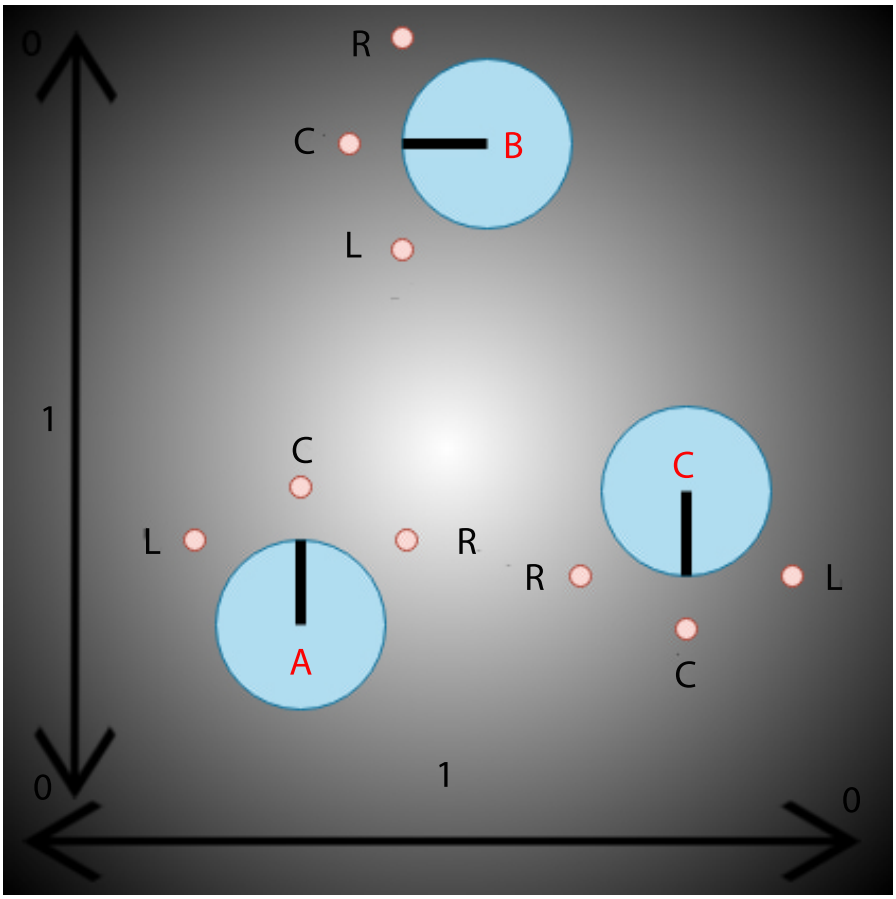


Figure 6: Three example configurations of agents within the environment. The arrows on the bottom and right-hand-side represent the transition of scalar values between 0 and 1. The three teal circles represent three agents, 'A', 'B', and 'C'. The three red circles in front of each represent the left, center, and right scalar field sensors. Agents 'A' and 'C' are positioned to navigate in a clockwise orbit, while agent 'B' is not, and must adjust its alignment to return to a clockwise orbit.

tation, and then make a decision based on whether there are pucks surrounding it. Agents observe the sensor values for the left, center, and right scalar field sensors ($floor_l$, $floor_c$, and $floor_r$ respectively). If $floor_r \geq floor_c$ and $floor_c \geq floor_l$ like agent 'A' or 'C' in Figure 6, then the agent is aligned to orbit the scalar field in a clockwise manner. If the agent is an innie, it checks its right puck sensor ($puck_r$) for any pucks to its right. If it returns 1, it nudges them outwards. If the agent is instead an outie, it checks its left puck sensor($puck_l$) for any pucks on its left. If it returns 1, it nudges them inwards as it continues moving clockwise. If the agent is not positioned clockwise (i.e, it is not true that $floor_r \geq floor_c$ and $floor_c \geq floor_l$) like agent 'B' in Figure 6, it realigns itself to return to a clockwise orbit.

The OC algorithm has a number of attractive properties: (1) it is minimalistic and requires only a coarse sensing of whether objects lie in the left or right field-of-view; (2) the shape constructed can be varied by changing the scalar field; (3) robots tend to circumnavigate clockwise about the growing structure, therefore moving in the same direction and avoiding collisions; and (4) no special grasping capability is required—the robots just bump into the objects to move them (assuming sufficient mass). The control algorithms for orbiting and construction are demonstrated in Algorithms 1 and 2, respectively. These algorithms are carried out by n agents with various sensors shown in Figure 22 and reference the following variables:

ω_{max} : The maximum angular speed permitted.

ϕ: The coefficient for ω_{max}, factoring the angular speed.

τ : The threshold value defining our shape's distance from the center (0-1).

obs_l : The value of the left obstacle sensor (0 or 1).

$floor_x$: Float value of the left(l), right(r), or center(c) scalar value (0-1).

$pucks_x$: The value of the left(l), or right(r) puck sensor (0 or 1).

Algorithm 1: The Orbiting Algorithm. Processes sensory state and returns angular speed, ω

Input : $obs_l, floor_l, floor_c, floor_r$

Output: The angular speed, ω_{max}

```
if obs_l then
    return ω_max
if floor_r ≥ floor_c ∧ floor_c ≥ floor_l then
    if floor_c < τ then
        return φ * ω_max
    else
        −φ * ω_max
else if floor_c ≥ floor_r ∧ floor_c ≥ floor_l then
    return −ω_max
else
    return ω_max
```

Algorithm 2: The orbital construction algorithm. Processes sensory state and returns angular speed, ω

Input : $obs_l, floor_l, floor_c, floor_r, pucks_l, pucks_r$

Output: The angular speed, ω_{max}

```
if obs_l then
    return ω_max
if floor_r ≥ floor_c ∧ floor_c ≥ floor_l then
    if innie ∧ pucks_r then
        return ω_max
    else if outie ∧ pucks_l then
        return −ω_max
    if floor_c < τ then
        return φ * ω_max
    else
        −φ * ω_max
else if floor_c ≥ floor_r ∧ floor_c ≥ floor_l then
    return −ω_max
else
    return ω_max
```

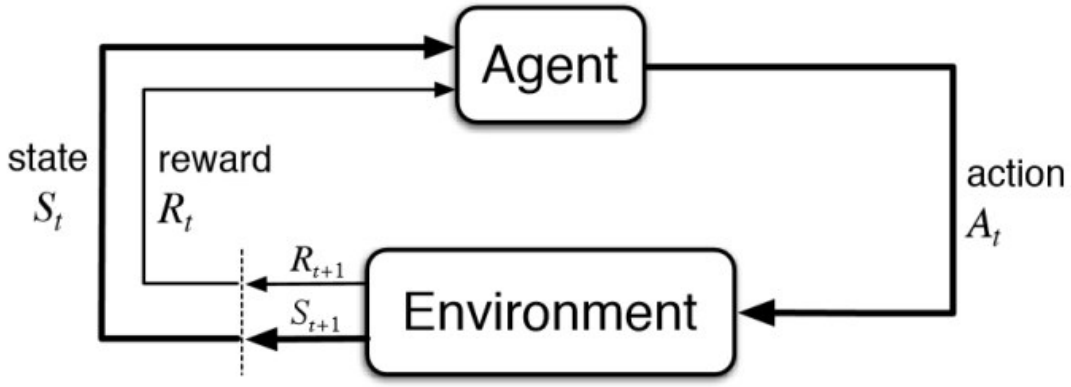


Figure 7: A figure representing the flow of a reinforcement learning model. The agent takes an action, A_t, and the environment returns both a reward, R_{t+1} and the next state, S_{t+1} to the agent based on the action it took. The policy is then updated, the agent moves to state S_{t+1} and the process is repeated [49].

1.2 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning. It is the problem faced by agents that must learn specific behaviours through trial-and-error interaction with the dynamic environment it exists within [19]. RL consists of five primary elements – *environment*, *state*, *reward*, *policy*, and *value*, defined below.

Environment: Physical world that the agent(s) operates in.

State: The agent's belief of a configuration of the environment.

Reward: Feedback from the environment.

Policy: A mapping from the agent's state to actions.

Value: Future reward an agent receives taking an action in a certain state

Before discussing the particular implementation of RL that we decided to use, we must first justify our decision to use RL as opposed to other machine learning techniques. Machine interactions usually involve a number of self-interested

entities. Interactions within this type of dynamic inherently have significant complexity. Lately, there has been interest in using machine learning techniques to deal with this heightened complexity. Using RL to handle this type of problem has attracted interest due to its *generality* and *robustness* [10]. Both the generality and robustness of RL were major factors in our decision to implement it over other machine learning techniques. When we discuss generality, we are referring to an algorithm's ability to be effective across a range of inputs and applications. In RL, the computer is only given a goal that it is expected to achieve. It has no prior knowledge about the environment of the problem at hand, and is therefore very malleable and can fit into the mold of other problems. This is an extremely important advantage of RL for our research, as we often want to adjust factors of the environment to accomplish a unique task. When we discuss robustness, we are referring to the sensitivity of an algorithm to discrepancies between the assumed model and reality. This is especially important in systems containing multiple agents, as a poor model of reality may lead to imperfect decision-making [14].

In robotics, RL is used to enable one or more agents to create an efficient control system which learns from its own experience and behavior. It offers the appealing promise of enabling autonomous robots to learn large repertoires of behavioral skills with minimal human intervention [15]. This is another main reason why we chose RL — we wanted to remove the need for human interaction and intervention as much as possible while still ensuring the system accomplishes a given task.

Within RL, there exist a number of unique algorithms. For example, temporal difference (TD) learning, Q-learning, State-Actions-Reward-State-Action (SARSA), and Deep Q-Network (DQN). Our main requirements were that the algorithm we picked be relatively straightforward to implement, and be powerful enough to

quickly construct an effective policy. Because of these requirements, we decided to implement the foundational algorithm of *Q-learning* [53]. Q-Learning is an off-policy (learns the value of the optimal policy independently of the agent's actions) TD control policy. It does not follow a policy to find the next action, but instead chooses the action in a greedy fashion. If we view Q-learning as updating values within a 2D array (action-space * state-space), we see that having large action and state spaces may be disadvantageous as the size of the array increases. If we were dealing with a large state-space or action-space, we might consider instead using a more complex algorithm like DQN [17], which is more robust. However, because our state and action spaces are kept quite small, Q-learning fits our requirements well.

The value function for Q-Learning is stored as a mapping of state-action pairs $Q(s, a) = v$, the expected future return (sum of rewards) of taking action a at state s. The Q-Learning policy $\pi(s) = a$ maps states to actions, with action a being chosen as the one which maximizes the value $Q(s, a)$. One iteration of Q-Learning happens with each time step t of the environment simulation. For a given state s_t an action is chosen separately for each agent from the current policy and carried out, advancing the current state s_t to s_{t+1}. Then, the reward R_t is calculated, and the Q-values are updated with the following rule:

$$Q(s_t, a_t)' = Q(s_t, a_t) + \alpha(R_t + \gamma max_a Q(s_{t+1}, a_t) - Q(s_t, a_t))$$

Using the new Q-values the policy is then updated, and the process repeats itself. This update is applied once for every agent at every time step using the global reward function R_t. In the above equation, α is the *learning rate* (or *step size*).

The learning rate refers to the extent that newly acquired information overrides previously aquired information. γ is the *discount factor*. The discount factor refers to how much future events lose their value in accordance with how far away in time they are.

1.3 Multi-Agent Systems and Multi-Agent Reinforcement Learning

Multi-Agent System (MAS) are defined as distributed systems of independent actors, called 'agents' that cooperate or compete to achieve a certain objective. These agents could be computer games characters, robots, or even human beings [51]. There is often a lack of knowledge of the existence of other agents, known as 'social awareness', within MAS. Because of this, agents often lack the ability to understand and respond to the needs of other agents. While it may seem beneficial for agents to have maximum social awareness, it is not always necessary to achieve optimal or near-optimal performance [20] as long as agents do not require information from other agents and they have a method to avoid collisions. In the cases where social awareness is not a priority, RL is a fitting technique to use.

When reinforcement learning is applied to an MAS, it is called Multi-Agent Reinforcement Learning (MARL). One of the main features that MARL offers is flexibility — a multi-agent decentralized system can have agents added, removed, or reconstructed without the need for rewriting control algorithms. There are some unique challenges that come with MARL, however. Some of these challenges are inherited from RL itself, such as the curse of dimensionality and the issue of exploration versus exploitation. The curse of dimensionality refers to the phenomena

that occurs when the dimensionality of the data increases. The more dimensions that are added to the data, the more difficult it becomes to find patterns [13]. Exploration versus exploitation refers the tradeoff between systems acquiring new knowledge versus maximizing their reward. Other issues are introduced solely by combining MAS with RL, such as the partial observability problem (the inability to know details on all aspects of the environment), costly communication between agents, and a possible need for coordination and communication between agents.

1.4 Hardware of Individual Agents

Our research was done with specific hardware in mind to implement our findings on. Using RL, we aimed to train a policy in simulation with online learning (learning as the data comes in) and then apply that policy offline (using a static dataset) on the physical robots due to agents not currently having the capacity for on-board learning. To do this, we aimed to create a version of our simulator that uses agent shapes similar to that of the physical robots (i.e, pointed wedge shape at the front). The physical robots consisted of a Zumo32U4 base which contain line sensors consisting of infrared emitter-detector pairs for sensing a black line on a white surface. We replaced these sensors with visible-light photo-transistors to read the pattern projected from below by a 75" LG 4.0 Smart TV. Visual input comes from a Pixy vision sensor which does on-board colour-based segmentation and connected components labeling (i.e, blob detection). There is a skirt formed from layers of foam board laser cut into a circular profile with a pointed wedge shaped at the front. Links to these products are listed below:

- Zumo32U4: https://www.pololu.com/category/170/zumo-32u4-robot
- LG 75" 4K HDR LED webOS 4.0 Smart TV: https://www.lg.com/us/tvs/lg-75UK6190PUB-4k-uhd-tv
- Pixy: https://pixycam.com/pixy-cmucam5/

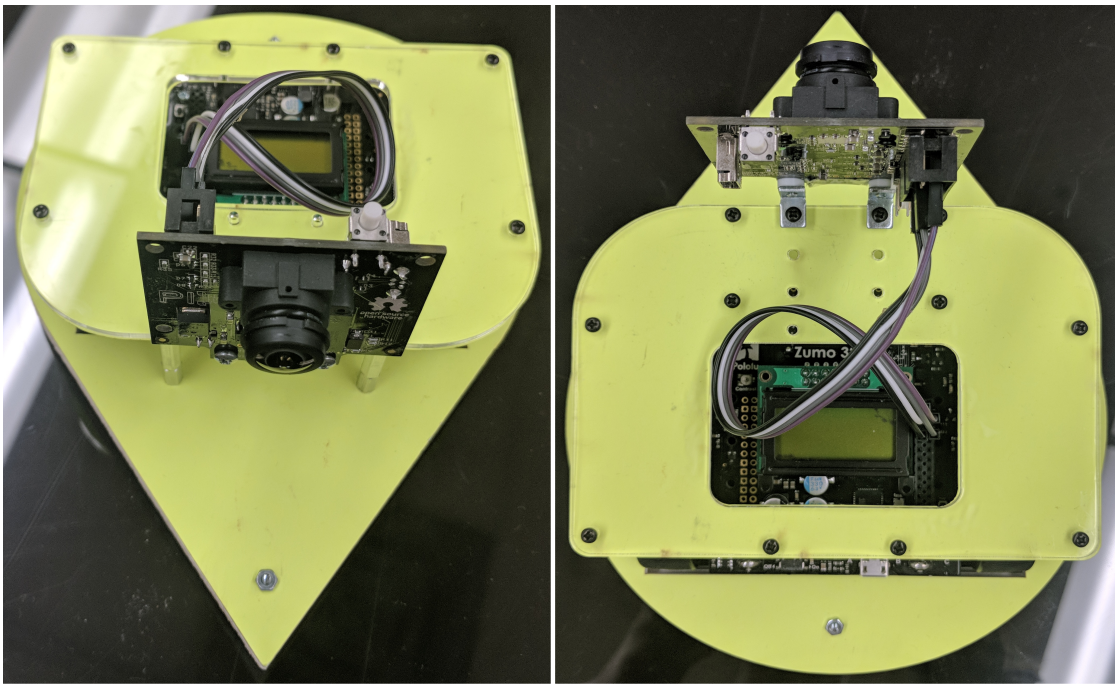


Figure 8: Design of the physical robot intended to be used for multi-agent shape formation. The angular skirt is laser-cut, with a Zumo32U4 placed in the centre and a Pixy placed near the front of the robot for sensing.

2 Literature Review

2.1 Overview of Related Works

With the increasing interest in research and practical applications of MARL and MAS, there has been a lot of focus on providing some solutions to the challenges facing these types of systems [56]. A big part of the success of MAS lies in Multi-Agent Learning (MAL). The last 30 years have seen immense progress in the field of MAL, which we will discuss in this section.

2.1.1 Biologically-Inspired Systems

Nature offers plenty of extremely powerful mechanisms, refined by millions of years of evolution [32]. These mechanisms are well-suited to handle evolving environments. Throughout history, humans have sought to mimic the functionality, thinking process, and operations of these biological systems [2]. In biology, these complex systems can be viewed as a collection of simple entities that work in unison displaying seemingly straightforward behaviors. This type of multi-entity behavior found in many colonies of insects can be defined as "the emergent collective intelligence of groups of simple and single entities" [5]. In this thesis we have focused primarily on the behaviors of ants and termites using pheromones to create mounds and nests, however this type of conduct can also be seen in bird flocking, bacterial growth, and fish shoaling/schooling [30]

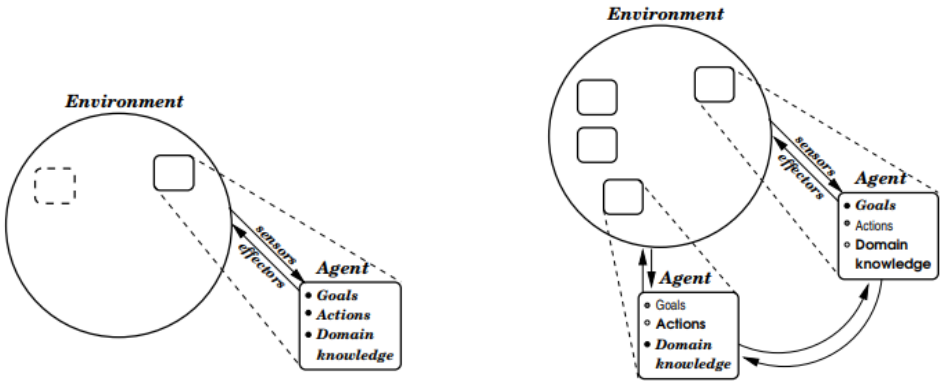
There has long been a potential to use biologically-inspired systems to solve complex problems in the real-world. The MAS paradigm has already inherited biological insights [32]. These insights can be broken down into three distinct types, as described in [3]. The first of these is the distributed nature of MAS.

The systems are based on a number of autonomous agents within the system, and functioning of the system is focused on interaction or collaboration between these agents. The second type is the division of labor. MAS often have distinct types of agents with distinct responsibilities and skills. Typically, an agent does not have to perform all tasks but instead specializes in smaller, distinct tasks. Finally, the MAS paradigm utilizes the emergence from collective straightforward behavior of entities. MAS applications that capture these insights offer a unique way of designing adaptive systems, replacing those with traditional centralized control [32].

2.1.2 Multi-Agent Systems

MAS can often be used to solve tasks that are difficult to accomplish when using a Single-Agent Systems (SAS), especially considering the presence of incomplete information and uncertainties. Agents in SAS are seen as independent entities with their own goals, actions, and knowledge. In SAS, other agents that may be present within the environment are not seen as having their own goals, but rather as part of the environment. Conversely, MAS can have multiple agents modeling each others' goals and actions. The most noticeable difference between SAS and MAS is that the environment's attributes can be determined by multiple other agents in a MAS, therefore making MAS inherently dynamic. This distinction is shown in Figure 9.

The methods used by MAS often allow problems to be solved in a more practical and financially cheaper way. Recently, there have been many practical solutions to real-world problems using MAS. For example- unmanned aerial vehicle (UAV) coordinated flight [16], scheduling systems for swarms of Earth remote sensing



(a) Single-Agent System (b) Multi-Agent System

Figure 9: Single-agent framework versus multi-agent framework. In (a), the agent models intself, the environment, and their interactions. Other agents that may exist are considered to be part of the environment. In (b), agents model each other's goals, actions, and domain knowledge and may also interact directly [48].

satellites [44], and disaster search-and-rescue teams [22].

As discussed in [48], there are a number of specific benefits to solving a task using MAS, for example:

– Parallelism

– Simpler Programming

– Cost Effectiveness

– Geographic Distribution

When creating a MAS, it is difficult to consider all potential situations that an agent may find itself in. Agents within these systems must have a way of learning from and adapting to their environment and robot counterparts. Therefore, control and learning become two important avenues of research in MAS. There are a multitude of techniques that have been proposed for dealing with control

and learning in this type of system, however we will be focusing on the Machine Learning (ML) subset of these solutions, which we will discuss later. Spears [46] raised four questions that need to be addressed when dealing with learning agents:

1. Is learning applied to one or more agents?
2. If multiple agents, are the agents competing or cooperating?
3. What element(s) of the agent(s) are adapted?
4. What algorithm(s) are used to adapt?

The way in which researchers have answered these questions has established the majority of research directions in the domain [21] however classifications of these numerous directions is difficult. Weiss and Dillenbourg [55] proposed a classification scheme for MAL in order to work out key differences in MAL. This schema split MAL into three primary types: 1) multiplied learning, 2) divided learning, and 3) interactive learning [55]. In 'multiplied learning', there exist several agents learning independently of one another. They may interact with each other; however this does not change the way that any agent is learning. Thus, each individual agent is capable of carrying out all activities that, when combined, make up the learning process. Divided learning involves dividing a single-agent learning task amongst multiple agents before the learning process starts. These tasks may be split by different aspects (i.e, location of an agent). Unlike multiplied learning, the agents within the system have a shared overarching learning goal. In divided learning, interaction between agents is required in order to put the individual learning results together. This interaction only involves the input and output of agents'

individual learning processes. Finally, interactive learning involves all agents being engaged in a single learning process that becomes possible through a shared understanding of the learning task and goal. The purpose of interaction between agents is to construct a successful path to learning a solution to a task. Each agent in an interactive learning process works together with equal importance to construct a solution — no agent has any unique role within the system.

Within our MAS problem, a modified version of multiplied learning was decided to be the best fit. Assigning each agent with equal importance allows us to add or remove any agent at will. As well, MARL requires all agents to work together to solve an overall problem without knowing anything about other agents in the system. Rather than splitting each agent into groups or relying on other agents to develop a reliable set of instructions (like in divided and interactive learning), we want each agent to benefit from the discoveries of other agents while also having limited to no information about them [51]. Dissimilar to classic multiplied learning, the agents are not necessarily learning 'independently', but instead contributing to a shared policy (π) using a shared reward signal, ($\mathfrak{R}$) that allows for faster policy convergence.

In a team of agents solving a task with combined forces, it often seems useful to have agents with individually unique sensors and effectors split into different subtasks in order to share the effort to finish the overarching task. This type of method is explored in [33] — however, this is not always the best solution. It is not always obvious what the best distribution method is for types of sensors or effectors between agents. It is also expensive to design a team of distinct agents to interact with the environment. Because of this, we chose to use identical agents within our system.

2.1.3 Machine Learning Techniques for Multi-Agent Systems

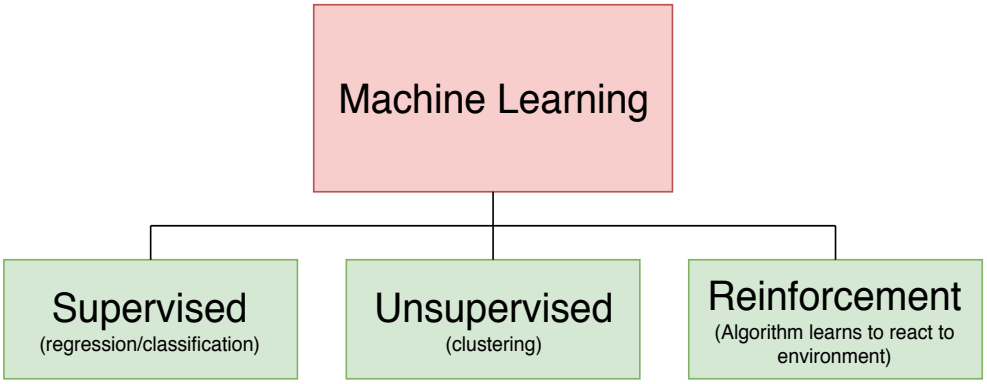


Figure 10: Breakdown of Machine Learning Subcategories.

In MAS, it is important to distinguish between two types of learning: 1) centralized (isolated) learning, and 2) decentralized (interactive) learning. Learning is centralized if the learning process is executed by a single agent and does not require any type of interaction with other agents. Conversely, decentralized learning involves several agents engaged in the same learning process. Thus, compared to centralized learning, decentralized learning requires the presence of several agents capable of carrying out specific activities. Centralized and decentralized learning are best thought of as two distinct paths in an MAS that cover a range of possible forms of learning. There are six relevant aspects for characterizing learning in MAS: 1) The degree of centralization, 2) Interaction-specific aspects, 3) Involvement-specific features, 4) Goal-specific features, 5) The learning method, and 6) The learning feedback [54]. Agent learning partially relies on the underlying algorithm being used. This algorithm is often one of several relevant ML techniques, all of which fall under one of three categories: supervised learning, unsupervised learning, and reinforcement learning. These categories are distinguished by what kind of feedback the critic provides to the learner [21].

2.1.3.1 Supervised Learning Supervised learning in ML is the task of learning a model that maps an input to an output based on example labeled input-output pairs [40]. Algorithms such as classification, regression, and estimation fall under this category. Supervised learning derives a model from training data consisting of a set of training examples [31]. In MAS implementing supervised learning, the accuracy of the system at each iteration is usually decided by an external teacher that evaluates the system output. Furthermore, the inputs are separated into two distinct sets — one for training and one for testing. In general, supervised learning tends to be easier to implement than unsupervised due to the consistancy of the data throughout iterations [37]. Supervised learning is not feasible for our problem because while we have training data for annulus construction through the OC algorithm, we do not have training data for any other potential scalar fields. Without data to learn from, supervised learning is not practical.

2.1.3.2 Unsupervised Learning Unsupervised machine learning algorithms infer patterns from a dataset without reference to known, or labeled, outcomes [4]. Instead of learning from feedback, unsupervised learning identifies commonalities and patterns in the data and proceeds based on similarities or dissimilarities within each new piece of data. Clustering and data-compression algorithms fall under the category of unsupervised learning. Unsupervised learning techniques are used on occasion within MAS. For example, in [38], Pugh uses unsupervised learning in an MAS so agents learn obstacle-avoiding behaviours within a noisy environment. In [8], agents learn a model of their opponent's strategy based on past behaviour, and uses the model to predict its future behavior by using an unsupervised learning algorithm termed '$US - L^*$'.

2.1.3.3 Reinforcement Learning As discussed previously, RL was originally studied from the perspective of simplistic animal behaviors [39]. It has since become a major class of ML methods to solve decision-making problems that contain uncertainties [19]. Although we have described RL in Section 1.2, we will discuss it here in the framework of MAS. In general, RL is the most commonly studied technique for MAL. Some RL algorithms (Q-learning, for example) are guaranteed to converge to the optimal behaviour eventually, assuming that the environment an agent is experiencing is Markovian and the agent is allowed to try out all actions [51]. Unlike the supervised and unsupervised learning ML techniques described above, the learner does not need to be supplied with any data before learning begins during RL. It allows an agent that has no knowledge of a task or environment to learn more efficient behaviours by progressively improving its performance based on being given positive or negative reward signals (determined by the environment) [41]. These reward signals were separated into three distinct types by Balch [1]:

1. **Local performance-based reinforcement:** Each agent receives rewards individually after achieving the task.

2. **Global performance-based reinforcement:** All agents receive a reward when one of the team members achieves the task.

3. **Local shaped reinforcement:** Each agent receives rewards continuously while it gets closer to accomplishing the task.

Balch applied these different reward functions to MARL systems in the context of multi-agent foraging, robotic soccer, and formation movements. He observed that

globally reinforced agents converge towards a heterogenous (agents specializing in different subtasks) team, while local reinforcement leads to homogeneous agents (specializing in the same task). As well, in multi-robot foraging, a locally reinforced team outperforms a globally reinforced team — however, robotic soccer showed the opposite effect [20].

The reward signal approach discussed in this thesis lies between global perfomance-based and local shaped reinforcement. Each agent in our system receives the same reward as all others, but also the rewards are delivered each time step as the system gets closer to accomplishing the task (formation of the prescribed shape).

2.2 Discussion

MAL still has many open research challenges and issues. These challenges include goal setting, scalability, communication bandwidth, dynamic systems, and domain problem decomposition [21]. While there has been a fair amount of research within shape formation and object manipulation in MAS using MAL, few have delved into the distinct comparisons between hand-coded and MARL solutions. In our research, we also discuss the outcomes of creating a state representation using information provided by agent sensors working atop a projected scalar field. Although this scalar field was created initially for a hand-coded solution, we would like to prove the versatility of RL and how it can adapt to new environments (even if those environments were not made with RL in mind).

In this brief review of related history, we have discussed a variety of distinguishing features between different types of MAS. We have also discussed the assortment of algorithms available for systems using ML, and which type is best-

suited for achieving specific goals. From there, we discussed the benefits of using RL, a form of ML, in MAS (resulting in MARL). From choosing a type of reward signal to deciding on using a centralized versus decentralized system, there are a considerable amount of factors to take into consideration when constructing a MARL system, as we have previously discussed. Due to the unique nature of our environment, we decided upon a decentralized system using global reward signals and having each agent be completely independent and unaware of the positions and actions of all other agents. These were all decisions that were made based on the type of environment we were working within, and the overall goal we were looking to achieve. In this thesis, we will discuss the outcomes of the techniques we chose, and whether they proved to be the most effective measures to attain the types of results we were anticipating.

3 Framing the Orbital Construction Problem in a Reinforcement Learning Framework

To eliminate the need for hand-coding algorithms such as the OC algorithm, we have chosen to use RL in an attempt to automatically learn the robot actions necessary to construct desired shapes. In order to apply RL methods to the OC problem, we must frame it in an RL context. An RL problem is defined by several key elements, described in Section 1.2. The solution method to that problem is some algorithm which carries out the process of generalized policy iteration to learn a policy for the agent(s) to follow. In this section, we will describe how we initially frame each of these elements within the context of the OC problem.

3.1 Environment

The environment of an RL problem is the physical or simulated world in which the agent(s) operate. The environment used for our OC experiments is the same as described by Vardy in [52], an example of which can be seen in Figure 11. This environment is identical to that discussed in 1.1; it is an enclosed space with a projected scalar field template on the floor for agents to sense. The simulation of this environment is implemented via the CWaggle[1] open-source software project. Cwaggle supports static and dynamic circle-circle and circle-line collision resolution, with each agent and puck having mass, radius, and velocity, with constant deceleration. Scalar fields, discussed in Section 3.1.1, can be defined in CWaggle manually, or loaded from an image saved in jpg or png format. CWaggle is written in C++, and is based on the existing Waggle Javascript simulator described in

[1] https://github.com/davechurchill/cwaggle

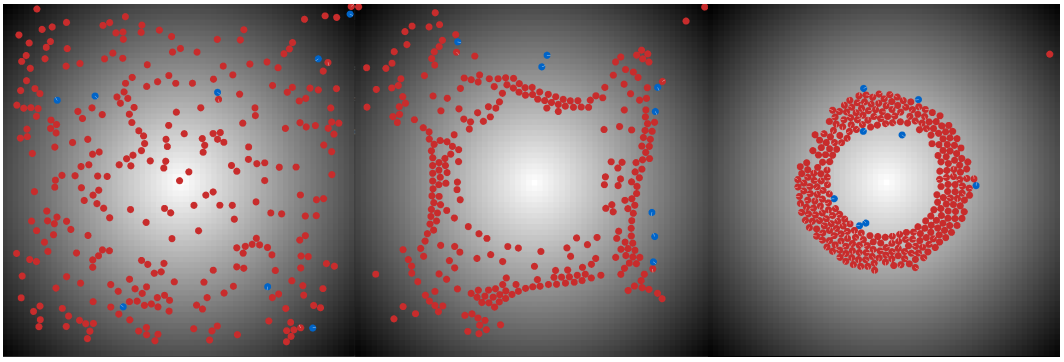


Figure 11: Three states of our environment demonstrating the progression of RL annulus construction using 8 robots (blue circles) and 250 pucks (red circles). Shown are an initial randomized configuration of pucks (left), an intermediary state during learning/construction (middle), and a successfully constructed shape (right). The background color of each state shows the value of the environment's projected scalar field, ranging from a value of 0 (black) to 1 (white), which guides shape formation.

[52]. The Q-Learning algorithm was implemented within the CWaggle framework from scratch, and supports the saving and loading of learned values and policies to disk.

3.1.1 Scalar Fields

The projected scalar fields used for all experiments were composed using MATLAB, and are comprised of attractive potential, repulsive potential, and total potential as demonstrated in Figure 12. Based on these potential field values, we use a distance transform function to calculate the distance between pixels. This results in a potential field similar to that shown in the bottom right of Figure 12. A variation of this procedure that can be used to create non-symmetrical shapes will be described later in this thesis.

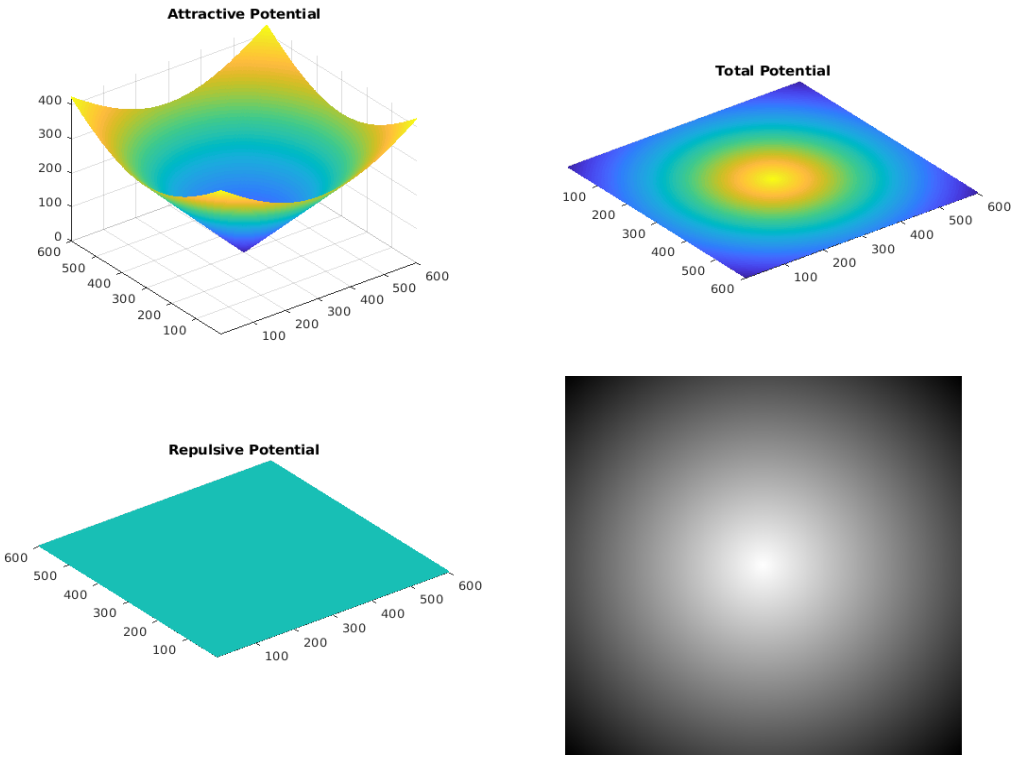


Figure 12: 3-dimensional graphs of the attractive potential (top left), total potential (top right), repulsive potential(bottom left), and the potential field (bottom right) that they create using the distance transform function provided by MATLAB.

3.2 State

A *state* of an environment is the instantaneous perception of the current configuration of the environment from some viewpoint. MARL methods typically use one of two forms of state representation: a *global* state recording the positions of all agents and objects in the environment, or a *local* state storing just the sensory information available to any given agent at some time step. As global state information is rarely available in a real-life robot setting, we chose the latter local representation for our implementation. One key result of this choice is that our RL method will learn a single policy that will be followed by each agent in the environment, a much simpler task than using global states to learn a separate policy for each agent. In Chapter 5 of this thesis, we discuss the results of changing this technique and learning a separate policy for each agent.

In most RL problems, the smaller the state representation, the easier the learning task becomes [49]. Also, the less complex the sensor configuration, the more feasible it becomes to implement as a real-world swarm robot system. For both of these reasons, we wish to implement the smallest possible set of sensors that can facilitate the task. This minimum functionality requires that our agent a) be able to sense pucks in its vicinity, and b) detect its scalar field values. Using the scalar field values, we can implicitly deduce the heading of an agent by comparing scalar values to each other. Having the minimal possible state representation does come at a cost, however. Less sensors means compromising specific state information, so we have to be sure that the information we are not encoding does not have a significant impact on the performance of our system.

The sensory configuration that was used for our experiments was introduced in

Section 1.1. For our experiments, we combine the two left of center puck sensors as the 'left puck sensor' and the two right of center sensors as the 'right puck sensor', which approximates the left and right rectangular puck sensors described in [52] As well, we require each state to have a finite length binary representation, and so we discretize this real number scalar field value *svr* into into an integer *svi*, based on dividing the range of $[0, 1]$ into n equally sized areas using the equation:

$$svi = \lfloor svr * n \rfloor$$

Next, instead of simply encoding all three field sensor values in the state, we employ a hash function to both minimize the state representation and determine the relative agent orientation without having to include all three sensed scalar values into the state representation. We use the middle field sensor to obtain the agent's scalar field position value, and the relative values to the middle sensor of the left and right field sensors to determine the agent's heading within the scalar field. For example, if both the left and right field sensors have a value less than the middle sensor, we know the agent is heading towards a 'lighter' area of the scalar field. Our final state representation is then formed as a binary string of length $4 + log_2(n)$ with the following bits:

PL	PR	FM	FL	FR

- **PL** (puck left): 1 bit representing if either of the 2 left puck sensors is active (1) or inactive (0)
- **PR** (puck right): 1 bit representing if either of the 2 right puck sensors is active (1) or inactive (0)
- **FM** (field mid): an integer of $log_2(n)$ bits representing the svi of the middle field sensor
- **FL** (field left): 1 bit representing if the left field sensor is less than the field mid sensor (1) or not (0)
- **FR** (field right): 1 bit representing if the right field sensor is less than the field mid sensor (1) or not (0)

If we use $n = 16$ field divisions, this would yield $2^{4+log_2(16)} = 2^8 = 256$ possible states for an agent. If we then have an agent with: left puck sensor active (PL=1), right puck sensor inactive (PR=0), field mid sensor value of 0.4 (FM=$\lfloor 0.4 * 16 \rfloor$=6=0110), field left sensor value of 0.35 (FL=1), and field right sensor value of 0.45 (FR=0), this would yield a state presentation of 10011010 (154 of 256 possible). The state observed by an agent at time step t of the environment simulation is denoted as s_t. In Chapter 5 of this thesis, we discuss the results of altering this state representation and why this type of representation is the most efficient.

Our simulations are run under ideal conditions. When applied to real-world robots, these state representations may not be ideal due to noise in the environment or any array of issues that could arise. Thus, while a specific state representation may work best in simulation, it may need adjustments in order to work outside of simulation.

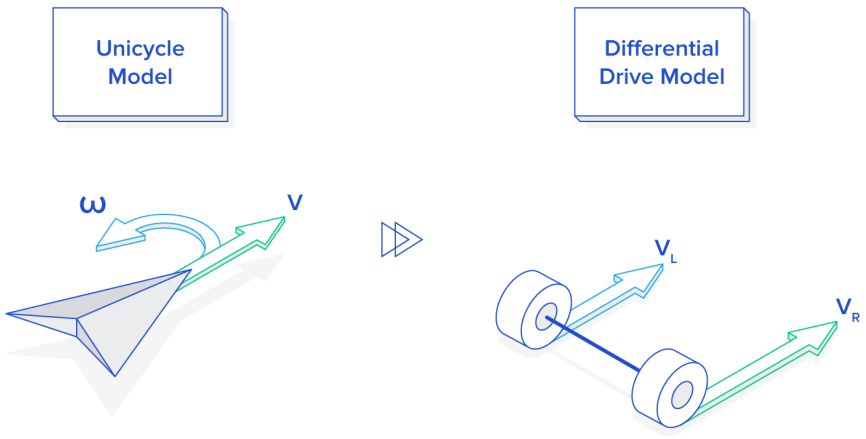


Figure 13: Representation of the unicycle model (left) and the differential drive model (right). Our agents use the unicycle model with a constant forward velocity (v) and a changing w value to avoid having to incorporate two changing velocities V_L and V_R like in the differential drive model, which significantly reduces the complexity of our problem [29].

3.3 Actions

In our OC environment, each agent moves maintains a constant linear speed, but alters its angular speed. The only action that an agent can take is to rotate on every time step (i.e, there is not an option for no movement, which cuts down on our action space, thus reducing the size of the RL problem). The agents are modeled after the 'unicycle model', shown in Figure 13. The unicycle model is one step simpler than using a 'differential drive' which assumes independent wheel control. For our RL framework, we discretized the real-valued space of angular speeds into a finite set of turning angles that can be chosen to guide the agent.

3.4 Reward

A reward function, $R : SxA \rightarrow \Re$ specifies an agent's task [27]. It is arguably the most important element of any RL problem, as it defines the objective measure that is to be optimized by the learning process. As we will talk about later in this thesis, altering the reward function can be beneficial depending on the type of shape the system is trying to form. In this section, we will discuss just our initial implementation. Intuitively, the reward for our RL framework should become more positive as the pucks in the environment get closer to the desired formation. The desired formation for our OC environment is defined by a given scalar field, along with two threshold values: an inner threshold T_i that defines the inner limit of our desired shape, and an outer threshold T_o that defines the outer limit of our desired shape. Due to the nature of the scalar field, we must be sure to choose these thresholds such that $T_i > T_o$. We can therefore construct our reward signal as a function of the distances of the pucks in the environment from the desired

location within the scalar field.

In an environment with P pucks, we define $SV_t(P_i)$ as the scalar field value located at the center of puck i with $1 \leq i \leq P$ at simulation time step t. Next we define a distance function $D_t(P_i)$ which yields 0 if the puck is inside the desired thresholds T_i and T_o at time step t, or the difference from the closest threshold if outside it:

$$D_t(P_i) = \begin{cases} SV_t(P_i) - T_i, & \text{if } SV_t(P_i) > T_i \\ T_o - SV_t(P_i), & \text{if } SV_t(P_i) < T_o \\ 0, & \text{otherwise} \end{cases}$$

We then define a global evaluation function $Eval_t(E)$ on an environment E with which averages $D_t(P_i)$ for all pucks:

$$Eval_t(E) = 1 - \frac{1}{P} \sum_{i=1}^{P} D_t(P_i)$$

which ensures $0 \leq Eval_t(E) \leq 1$. An ideally constructed shape will therefore yield a reward $Eval_t(E) = 1$. Our final RL reward function R_t for an environment at time step t then simply subtracts the current evaluation from the evaluation of the previous time step $t - 1$. If the environment has come closer to the desired shape then the reward will be positive:

$$R_t = Eval_t(E) - Eval_{t-1}(E)$$

4 Shape Formation

To evaluate the overall performance of RL versus the OC algorithm, we must provide and test both algorithms on an array of distinct scalar field backgrounds. These backgrounds should vary in shape and complexity. Two primary experiments were carried out to demonstrate the effectiveness of reinforcement learning for shape formation. These experiments tested the overall performance of reinforcement learned policies versus the original OC algorithm on: (1) an annulus shape, and (2) a collection of letters. We decided on using letters because they present a unique combination of curved segements, straight segements, and angles in each different letter.

4.1 Performance Metric

For each experiment, all agents and pucks were initially set to random positions within the environment, and each agent was also randomly oriented. Agents moved forward at a constant speed each time step, with the decision of angular speed given by different planar formation methods. Each simulation was executed for a given maximum number of time steps, and two metrics were kept to determine the effectiveness of each method during that time period: (1) the number of formations that were successfully formed, and (2) the number of times the method got 'stuck' and could not form a formation for a specific number of time steps. A formation was 'successfully formed' if the environment E reached $Eval(E) > 0.94$, a threshold value determined experimentally by observing repeated shape formations.

We wanted to ensure that the shape the agents were forming was clearly identifiable, and we also wanted to ensure that the threshold was achievable, as there

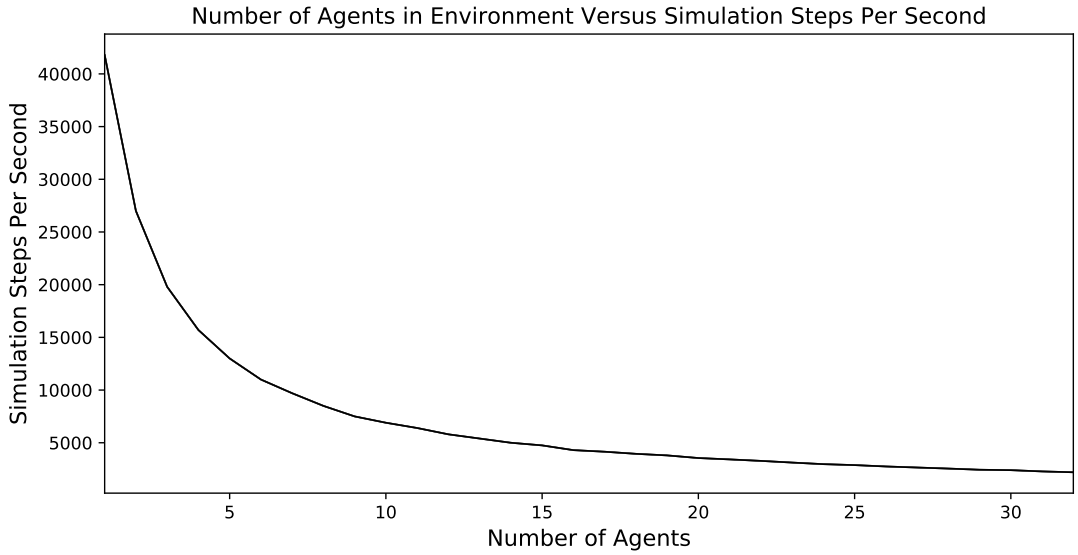


Figure 14: Number of simulation steps achievable per second using 1 to 32 robots and 250 pucks with the GUI turned on for visualization.

may be a small number of pucks that were lost around the periphery of the environment. Upon trial and error, 0.94 seemed a good balance. Once a successful formation was completed, the number of formations was recorded and the environment was 'reset', with each puck and agent placed in a random position within the environment. A simulation was deemed to be 'stuck' if the agents could not create a configuration of pucks that reached the threshold evaluation within a given number of time steps. Once the simulation was declared stuck, the environment was reset again. Within this framework, we consider one method to be more successful than another if it is able to successfully construct more formations than another within a given amount of time steps. All experiments were run on an Intel i7-7700K CPU processor running at 4.20GHz with 32GB of RAM using Ubuntu 18.04.

4.2 Experiment 1: Annulus Formation

Our first experiment was run to test the effectiveness of our RL solution to that of the original OC algorithm on the annulus shape shown in Figure 11 to see which method could successfully construct more formations in a given amount of time steps. In this experiment, the environment contained 8 agents and 250 pucks. We chose these numbers because 1) we wanted to have a significant number of agents to show that agents in RL could work together efficiently while following the same policy without interfering with each other too much, and 2) we wanted the environment to be densely populated with pucks so that agents were hitting the maximum number of possible states. The number of time steps was set to 5,000,000, as the RL policy converges quickly and additional simulation steps were deemed unnecessary in annulus formation. As well, we depending on the goal shape, we keep or remove the innies from the OC simulations. This is because they serve no purpose in shapes that do not have a cavity in the center, as there are no pucks to push out. The 'innie' row in each configuration table will address whether the innies have been removed or not. Additional variables values can be seen in Table 1.

The results of this experiment can be seen in Figure 15, with the x-axis denoting simulation time steps, and the y-axis denoting the number of successfully constructed formations by time step t. Three separate plots are visible, which show the performance of the following solutions over 10 trials: (OC) the original orbital construction algorithm, (RL) the on-line performance of the RL method as it was learning from scratch, and (RL2) the on-line performance of the RL method using a pre-trained policy as a starting point. The pre-trained policy used for RL2

Configuration Value	**Reinforcement Learning**	**Orbital Construction**
Number of Robots	8	8
Number of Pucks	250	250
Forward Speed	2.0	2.0
Angular Speed	0.3	0.3
Outie Threshold	0.6	0.6
Innie Threshold	0.8	0.8
Hash Function	Original	-
Actions	0.3, 0.15, -0.15, -0.3	0.3, 0.15, -0.15, -0.3
Max Simulation Steps	5,000,000	5,000,000
Initial Q	1.0	-
Alpha	0.2	-
Gamma	0.9	-
Epsilon	0.0	-
Reset Percentage (%)	94	94
Q-Learning	On	-
Innies?	-	Yes
Steps Before 'Stuck'	150,000	

Table 1: Configuration values used while testing the performance of both RL and OC on an annulus-shaped scalar field.

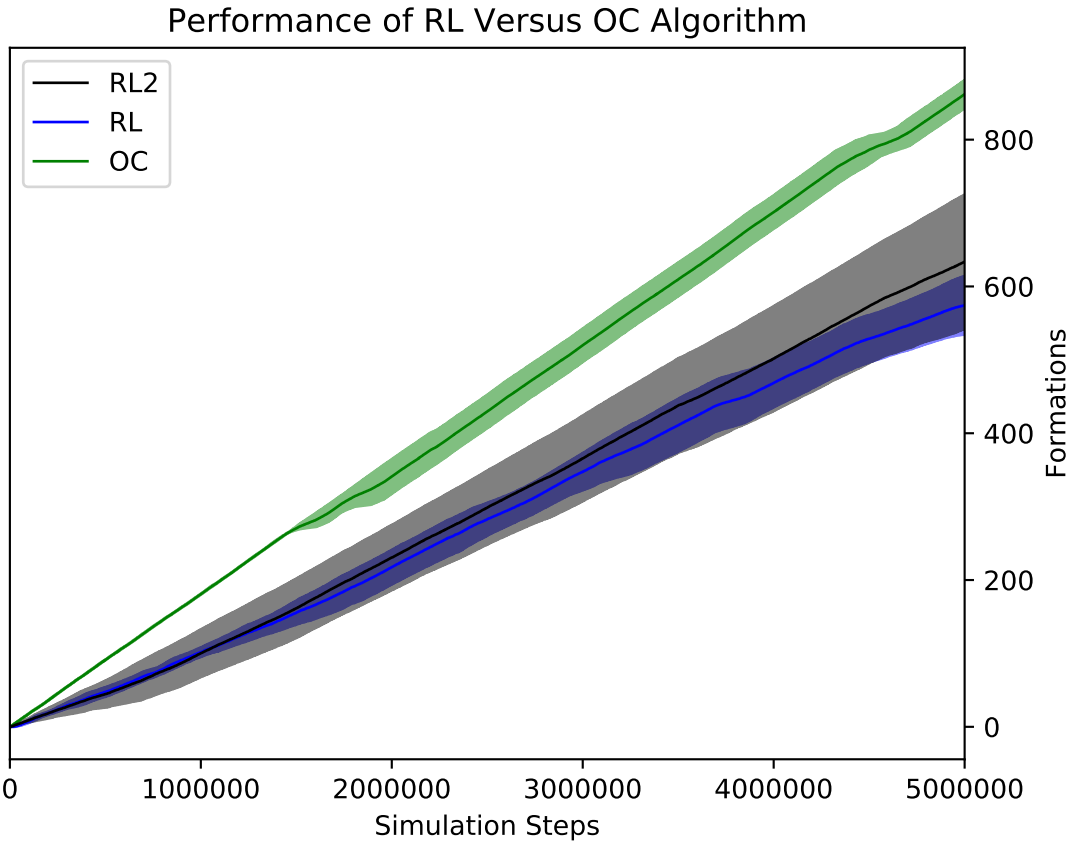


Figure 15: OC vs RL Creating Annulus Shape with 8 Agents. OC = Orbital Construction algorithm, RL = Reinforcement Learning from scratch, RL2 = Reinforcement Learning seeded with previously learned policy. Shaded areas above and below each line represent the maximum and minimum values of each trial of the experiment.

was trained using the same variables shown for RL in Table 1. The line shown is the average performance of each method, with the shaded area enclosing a 95% confidence interval defining a range of values that you can be 95% certain contains the population mean. As expected, the hand-crafted OC algorithm produces formations at a constant rate over time, with its first formations appearing around $t = 5000$. The on-line learning of the RL algorithm requires a longer time period before the first shape is created later at $t = 34000$, while the pre-trained RL2 creates its first formation around $t = 9000$. The RL2 line demonstrates that seeding the RL algorithm with a pre-trained policy yields better results than seeding it from scratch — an intuitive result which demonstrates the increase in performance of RL as more training steps are allowed.

This experiment showed that the RL method is a viable solution for shape construction, yielding 73% as many annulus formations as the OC algorithm, which was specifically designed and tuned by hand to construct only this specific shape. Using a single CPU core the experiment ran at around 8500 simulation time steps per second for a total of 9.8 minutes for the 5,000,000 total time steps. It is important to note that simulations steps per second decreases while the number of agents increases. While learning from scratch, the RL method took less than 10 seconds to successfully construct its first shape.

4.3 Experiment 2: Transfer of Learned Policies

In our second experiment, we wanted to test whether a policy learned on a single agent could be applied to multiple agents to construct formations. Because an increase in agents means an increase in running time, as shown in Figure 14,

Configuration Value	**Reinforcement Learning**
Number of Robots	1 - 32
Number of Pucks	250
Forward Speed	2.0
Angular Speed	0.3
Outie Threshold	0.6
Innie Threshold	0.8
Hash Function	Original
Actions	0.3, 0.15, -0.15, -0.3
Max Simulation Steps	2,000,000
Initial Q	1.0
Alpha	0.2
Gamma	0.9
Epsilon	0.0
Reset Percentage (%)	94
Q-Learning	On
Innies	-
Steps Before 'Stuck'	100,000

Table 2: Configuration values used while testing the performance of 2, 4, 8, 16, and 32 agents loading a policy learned during a single-agent simulation.

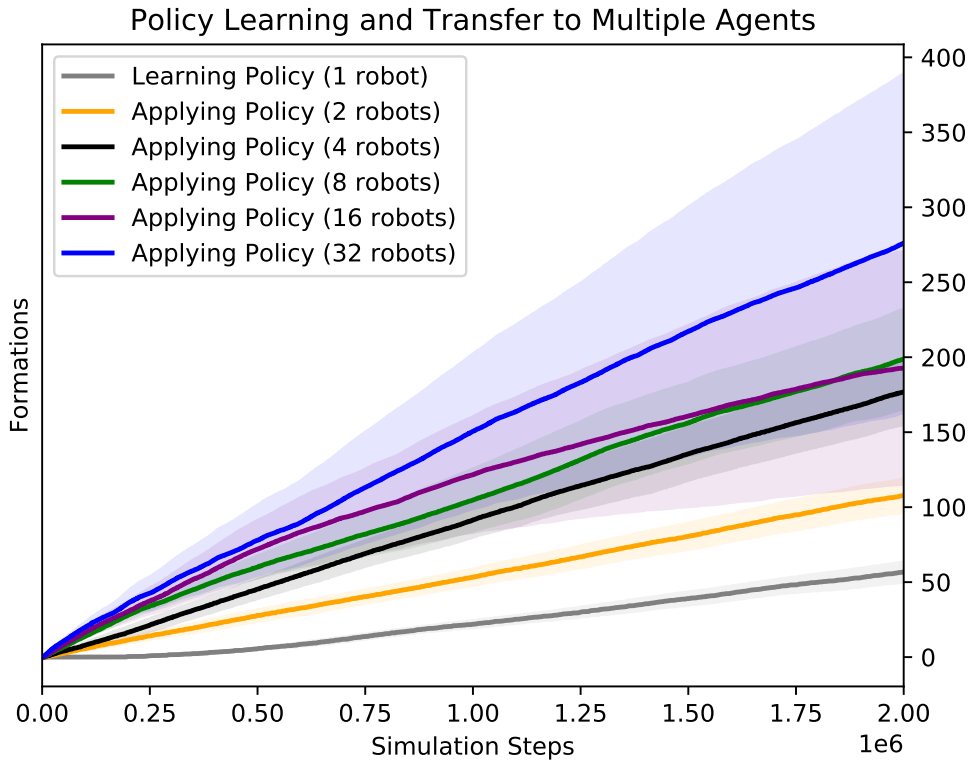


Figure 16: Policy learned on single agent and transferred to 2, 4, 8, 16, and 32 agents with Q-Learning on. Shaded areas above and below each line represent the maximum and minimum values of each trial of the experiment. Exact numbers shown in Table 3.

it would be beneficial if we could train a policy on a smaller number of agents and then apply that policy to a larger number of agents. In this experiment we tested the most extreme case by learning a policy with a single agent and then we applied that same policy to 2, 4, 8, 16, and 32 agents. Using the configuration variables shown in Table 2, the results of this experiment are demonstrated in Figure 16.

We can see that the policy transfer was successful, as multiple agents were able to use the single-agent trained policy and complete more formations in fewer time steps than the single agent. This increase is not linear however. By doubling the number of agents within the simulation, we are not necessarily doubling the productivity of the system as a whole. For example, by increasing the number of agents from 1 to 2, we see an increase of 91.1% (from 56 to 107) in the number of formations. By increasing the number of agents to 32, we see just a 392.9% increase in the number of formations (from 56 to 276). This is likely due to agents getting in the way of each other, colliding, and having to maneuver around each other. This all takes time that could have been spent pushing pucks to the goal. The policy learned initially on a single agent is not accounting for the number of agents within the system.

Despite not having a linear increase, these results are still very promising. Policies learned quickly by fewer agents can be carried out by a larger number of robots with impressive results. This significantly reduces the amount of time needed to learn policies from scratch in systems with large numbers of agents, and also allows us to reuse policies that we have previously learned within a separate system containing a different number of agents. This allows us to save time as we do not need to learn a unique policy for each system containing a different number of agents.

Number of Agents	Total Number of Formations
1	56
2	107
4	176
8	199
16	192
32	276

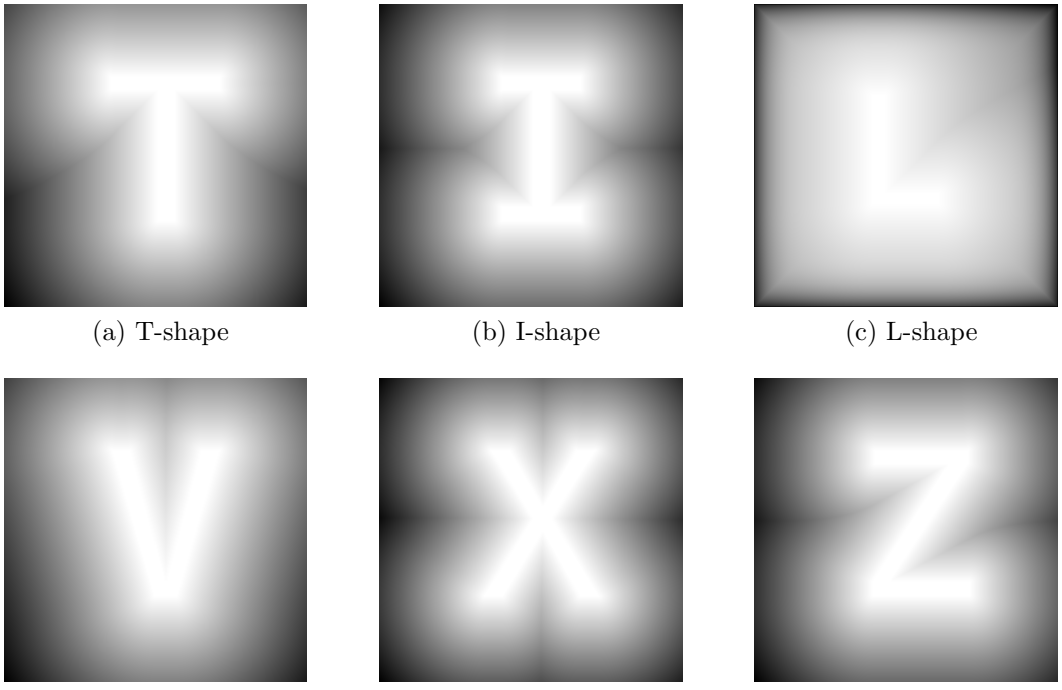
Table 3: Results from experiment 2 showing the averaged results of all 10 trials.

4.4 Experiment 3: Letter Formation

Our final set of experiments were run to test the effectiveness of our reinforcement learning solution relative to that of the original OC algorithm on a multitude of letter-shaped scalar fields shown in Figure 17 to see which method could successfully construct more formations in a given amount of time steps, similar to the previous experiment on the annulus shape. In these experiments, however, the environments contained 200 pucks instead of 250. This is due to the decreased surface area of these shapes compared to the annulus. Often, all 250 pucks would not fit inside of the optimal section of the field. Aside from the alteration in the number of pucks, we still used 8 agents and the maximum number of time steps was set to 5,000,000. In this set of experiments, we changed the 'steps before stuck' variable from 150,000 simulation steps to 50,000 simulation steps. This change was made solely to aid with the results of OC algorithms and ensure simulations created at least a single formation in the allotted number of simulation steps. Often times, OC would get stuck with a formation that did not reach the reset percentage and would then waste 100,000 simulation steps performing non-effective actions. This greatly impacted the final results. The reduction of this value fixed that issue. Additional configuration values can be seen in Table 4.

Configuration Value	**Reinforcement Learning**	**Orbital Construction**
Number of Robots	8	8
Number of Pucks	200	200
Forward Speed	2.0	2.0
Angular Speed	0.3	0.3
Outie Threshold	1.0	1.0
Innie Threshold	1.0	1.0
Hash Function	Original	Original
Actions	0.3, 0.15, -0.15, -0.3	0.3, 0.15, -0.15, -0.3
Max Simulation Steps	5,000,000	5,000,000
Initial Q	1.0	-
Alpha	0.2	-
Gamma	0.9	-
Epsilon	0.0	-
Reset Percentage (%)	94	94
Q-Learning	On	Off
Innies	-	No
Steps Before 'Stuck'	50,000	

Table 4: Configuration values used while testing the performance of both RL and OC on multiple letter-shaped scalar fields.



(a) T-shape (b) I-shape (c) L-shape

(d) V-shape (e) X-shape (f) Z-shape

Figure 17: Collection of letter-shaped scalar fields used as projected backgrounds in experiments testing RL versus OC performance.

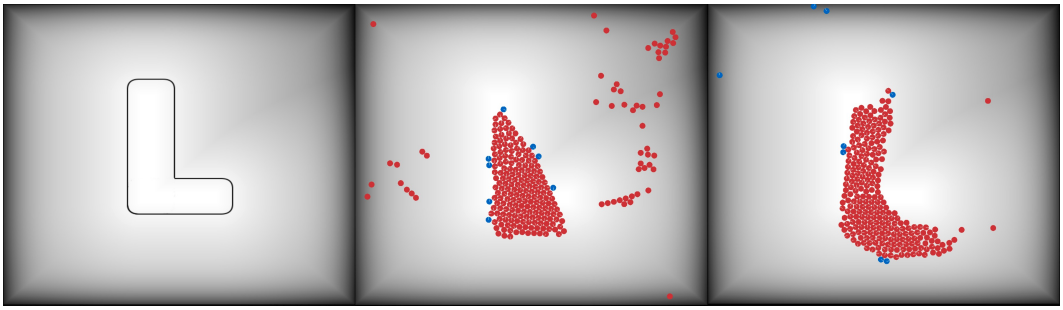
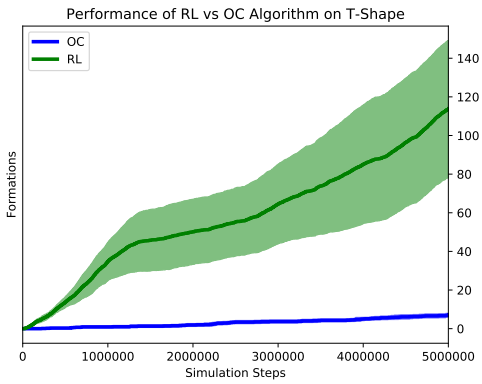


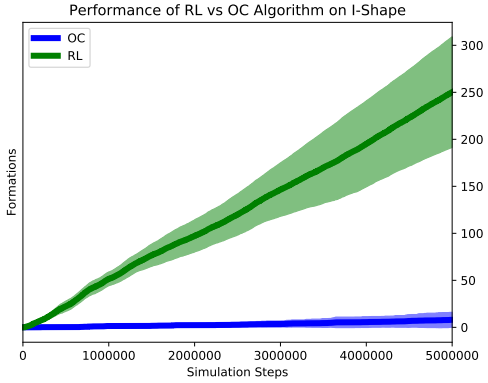
Figure 18: Graph of OC versus RL formations on an L-shaped scalar field (left), and OC algorithm construction example (middle), and an RL construction example (right).

The results of these experiments can be seen in Figure 19, and are detailed in Table 5. In each of the six graphs in Figure 19, two separate plots are visible, showing the performance of the OC and the on-line RL method as it was learning from scratch. For each letter, the hand-crafted OC algorithm produces formations very slowly in comparison with RL, and on-line learning repeatedly takes less time to create its first formation.

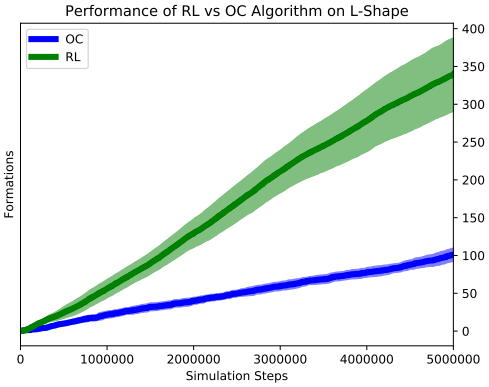
These experiments provide strong evidence to back our claim that the RL approach is more maleable and is often better suited than hard-coded algorithms to form miscellanious shapes within multi-agent systems.



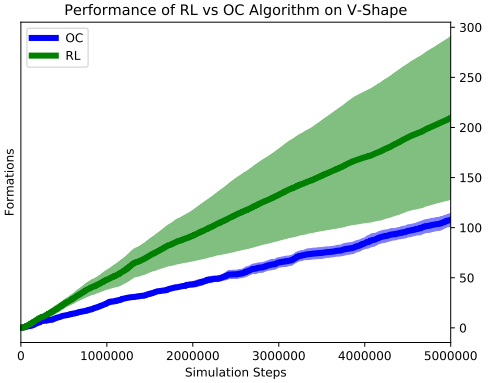
(a) T-shape



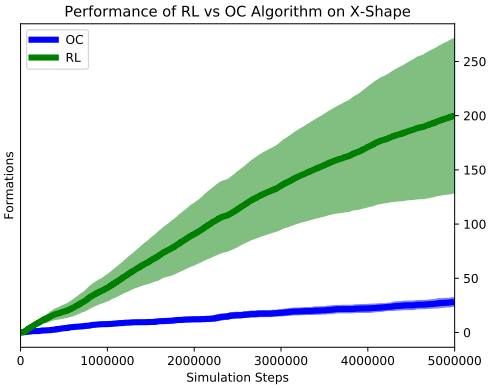
(b) I-shape



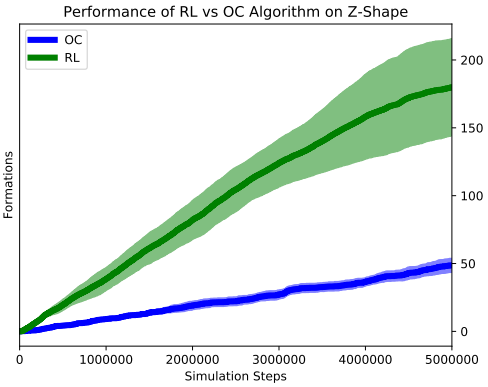
(c) L-shape



(d) V-shape



(e) X-shape



(f) Z-shape

Figure 19: Resulting performance of using RL versus OC algorithms on the letters T, I, L, V, X, and Z.

	RL			OC		
Shape	Formations	Stuck	Steps/Form	Formations	Stuck	Steps/Form
T	113.7	58.1	**43,975**	7	96.7	**714,286**
I	250.2	28.8	**19,984**	8	97.4	**625,000**
L	339.2	15	**14,740**	102.6	80.5	**48,732**
V	209.6	37.8	**23,854**	107.9	77.1	**64,850**
X	200	43.2	**25,000**	28	93.1	**178,571**
Z	180	38.4	**27,777**	48.7	83.4	**102,669**

Each experiment was run 10 times and then averaged to retrieve the final result displayed within the table.

Table 5: Numerical results for RL versus OC letter formation on the letters T, I, L, V, X, and Z.

4.5 Summary of Results

In this chapter, we investigated three distinct themes — RL versus OC in annulus formation, the transfer of a policy learned on n agents to a system using p agents, and RL versus OC in unique shape formation. In all of our experiments, we focused entirely on the number of successful formations that a system could create within a given number of time steps.

In our first set of experiments, we noticed that by starting with an empty policy using RL, OC began creating successful formations about 6.8 times faster than RL. However, on a pre-trained policy (RL2), OC only began creating formations about 1.8 times faster. We learned that pre-training a policy helps the system start creating formations much faster, thus increasing the chances that it will create more formations in the time frame allotted. In terms of overall formations, RL created about 71% as many successful formations as OC in the same amount of time steps, while RL2 created about 76% as many successful formations as OC. Clearly, OC outperforms RL in annulus formation, however it is important to note that by pre-learning a policy, the performance of RL is improved.

In our second set of experiments, we explored the effects of policy transfer between systems containing different numbers of agents. We learned a policy on a single agent and then applied that policy to system containing 2, 4, 8, 16, and 32 agents. We wanted to know both how the number of agents in a system affected performance, and how applying a policy learned on n agents to a system containing p agents (where $p > n$) affected performance. The results of this experiment showed us two things. First, an increase in agents does increase performance. This seems intuitive because with more agents following the same policy, the amount of work done should be increased at a rate equivalent to the number of agents in the system. While an increase in agents does improve performance, it is not linear. The more agents added, the more they tend to get in the way of each other, slowly impairing the productivity. We also learned that the transfer of a policy learned on a single agent to systems containing multiple agents is beneficial. It allows for systems to begin creating successful formations earlier, and removes the need for each different system to learn a brand new policy from scratch.

Our final experiment look at the performance of RL versus OC with forming a variety of letters. We created scalar fields depicting the letters T, I, L, V, X, and Z and RL significantly outperformed OC on each shape. Many of these shapes had distinct angles and curves that OC was not written to handle. This resulted in the agents circling angles instead of following the turns tightly, which forced pucks to build up inside of the angles. In this experiment we see one of the biggest benefits of RL in action — its adaptability. We did not need to write six new hand-coded algorithms for these shapes, but instead RL learned policies on its own to handle the unique turns and angles we included in the scalar fields.

In multi-robot planar construction, the ability of agents to create a variety of

shapes containing varying angles, curves, and concavities is extremely important. The ability to deviate from familiar shapes and create new, unique ones with relative ease is a major benefit of using RL. In this chapter, we determined that OC does outperform RL in the forming of annulus shapes. However, OC was hand-coded to do just that — create a singular shape. When it came to forming different shapes with a myriad of unique angles and curves, RL significantly outperformed OC every time. Using RL, agents began creating successful formations almost immediately while OC struggled with forming a newly proposed shape while also following an algorithm designed for annulus formation. Through these experiments, we noticed that because RL learns a unique policy for each new scalar field it is presented with, it is superior in handling features like straight lines and angles within shapes.

5 Altering the State Representation

In Section 3.2, we introduced the concept of a 'state', and discussed the five main elements within the environment that our primary 'original' hash function (the function responsible for converting factors of an agent's perceived environment into a state) is comprised of. A more specific description of our primary state representation can be shown as:

PL	PR	FL	FR	FM(bit 0)	FM(bit 1)	FM(bit 2)	FM(bit 3)

- **PL** (puck left): 1 bit representing if either of the 2 left puck sensors is active (1) or inactive (0)
- **PR** (puck right): 1 bit representing if either of the 2 right puck sensors is active (1) or inactive (0)
- **FM** (field mid): an integer of 4 bits representing the scalar field value (translated to an integer between 0 and 15) of the middle field sensor
- **FL** (field left): 1 bit representing if the left field sensor is less than the field mid sensor (1) or not (0)
- **FR** (field right): 1 bit representing if the right field sensor is less than the field mid sensor (1) or not (0)

Thus, we can easily deduce the state value of any agent in the system at any given point in time. Picturing an agent on an annulus-shaped grid with multiple pucks detected in each of its puck sensors, an orientation of a 90° angle, and a center scalar field value of 0.71345, we can create an exact state value to reference within our policy. In this case, both PL and PR are 1 due to the puck sensor values, FM is $\lfloor 0.71345 * 16 \rfloor = 11$ which in binary is 1011, and since the agent is at a 90° angle,

everything directly to the left will have a smaller scalar field value than FM and everything directly to the right will have a larger scalar field value than FM. This results in FL having a value of 1 and FR having a value of 0. Combining all of these values together, we get an 8-bit state of *11101110* using our 'original' hash function.

There is still the question of why, and even *if*, this is the best state representation to use. In this chapter, we will discuss alternatives to the 'original' hash function that we have considered, and their effects on the performance of RL.

5.1 Experiment 1: Modifying Bits Representing the Scalar Value

By adding or removing bits to the binary representation of the agent's current greyscale location within our state representation, we can increase or decrease the granularity of where an agent believes it is in the environment. Currently, as mentioned above, we have 4 bits representing the scalar field value of an agent. This allows 2^4 possible greyscale values. Having n bits representing the scalar field value underneath an agent correlates to 2^n distinct contours of the scalar field that an agent could be in. In this experiment, we wanted to explore the best balance of specificity and compactness. We wanted to keep our state space small, while still giving the agents enough information to create an efficient policy.

5.1.1 Decreasing the Number of Bits

First, we removed a single bit from the representation of the scalar field value. Thus, our new state representation consisted of 7 bits and was structured as shown

in the diagram below:

PL	PR	FL	FR	FM(bit 0)	FM(bit 1)	FM(bit 2)

This results in $2^3 = 8$ distinct contours of the scalar field.

As we can see from Figure 20, there is an increase in the average number of formations made when using just 3 bits to represent the greyscale value underneath an agent. The 7-bit state representation creates an average of **520** formations over 10 trial runs, while the original 8-bit state representation only creates an average of **461** formations. Within our simulations, agents cannot possibly visit all states with equal frequency. Thus, convergence of a policy depends on focusing only on the relevant parts of a state and maximizing the amount of information learned from each trial. The smaller the state space, the fewer learning trials are required [28]. Thus, we see that by reducing the state space from $2^8 = 256$ to $2^7 = 128$, (cutting it in half), we can still efficiently represent the relevent parts of the environment and formulate an effective policy. By reducing the number of bits detecting the scalar value, we also begin creating formations faster than our 'Original' state representation. While this reduction in state space proved to be beneficial, it is possible to reduce the granularity to a point where the shapes cannot be formed.

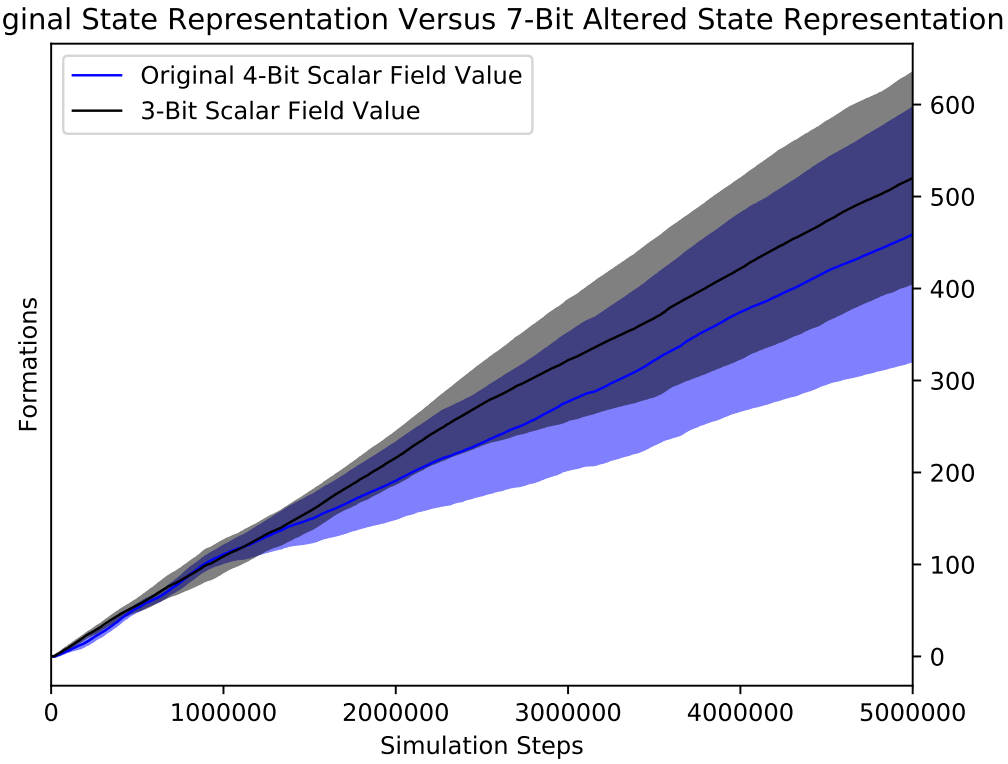


Figure 20: RL using the original 8-bit state representation versus RL using the altered 7-bit state representation. The altered 7-bit state representation uses just 3 bits to represent the scalar field value.

5.1.2 Increasing the Number of Bits

Next, we increased the number of bits used to represent the scalar field value from 4 to 5. Thus:

PL	PR	FL	FR	FM(bit 0)	FM(bit 1)	FM(bit 2)	FM(bit 3)	FM(bit 4)

This results in $2^5 = 32$ variations in the scalar field.

As we can see in Figure 21, there is a decrease in performance when we add an extra bit to represent the scalar field value underneath an agent. The 9-bit state representation creates an average of **356** formations over 10 trial runs, while the original 8-bit state representation creates an average of **461** formations. Our new state representation increases the state space from $2^8 = 256$ to $2^9 = 512$, resulting in more states to explore and the construction of a larger policy. In our case, increasing the accuracy of an agent's scalar value had a negative effect in performance. The policy constructed was less accurate due to the increased number of states to visit. The nature of our environment implies that, given identical values for scalar field and puck sensor readings, the action taken by two agents in two very similar scalar value positions will often be the same. When we increase the accuracy of our scalar value representation, two states that would have been identical in a less-specific state representation become unique, and thus now we have to learn an optimal action for two unique states (which will likely end up being the same). This is one explanation for the decrease in performance in this new state representation.

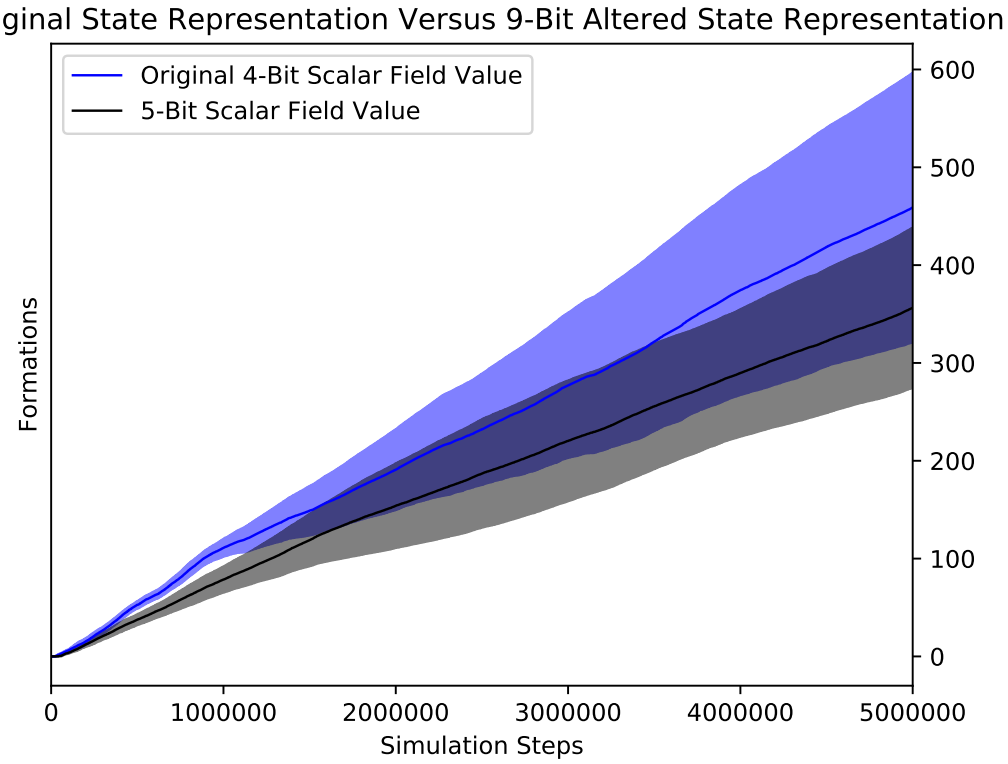


Figure 21: RL using the original 8-bit state representation versus RL using the altered 9-bit state representation. The altered 9-bit state representation uses 5 bits to represent the scalar field value.

5.2 Experiment 2: Incorporating Obstacle Sensors

In this experiment, we added two additional sensors to each agent. We call these new sensors 'obstacle sensors', and they are placed on the left and right of each agent. We experimented with the reach of these sensors, as we required they be broad enough to be effective but also not so large as to detect agents too far away. We began with using obstacle sensors the same size as our floor sensors. This seemed too small, as by the time the agent was detected, they were already incredibly close. Then, we tried using obstacle sensors half the size of the puck sensors. The performance was slightly increased; however we wanted to see how increasing the sensor size even more would affect the system. Ultimately, we decided on using obstacle sensors the same size as the puck sensors. The model we stuck with is shown in Figure 22. Like puck sensors, each obstacle sensor can have a binary reading of 0 or 1. 0 means there is another agent sensed, 1 means there is not. These sensors are used to detect agents to the left and right of a robot, and will be used later in this paper to try and help reduce the number of agent-agent collisions. In this experiment, we were interested in the performance gap resulting from incorporating the obstacle sensor readings into the state representation.

We added the binary values from the obstacle sensors into a new state representation, 'Original_Obstacle'. 'Original_Obstacle' appends to the 'Original' state representation. It turns the 8-bit state representation into a 10-bit state by adding both 'obstacle left'(OL) and 'obstacle right'(OR) bits, thus resulting in a state representation of:

PL	PR	FL	FR	FM(bit 0)	FM(bit 1)	FM(bit 2)	FM(bit 3)	OL	OR

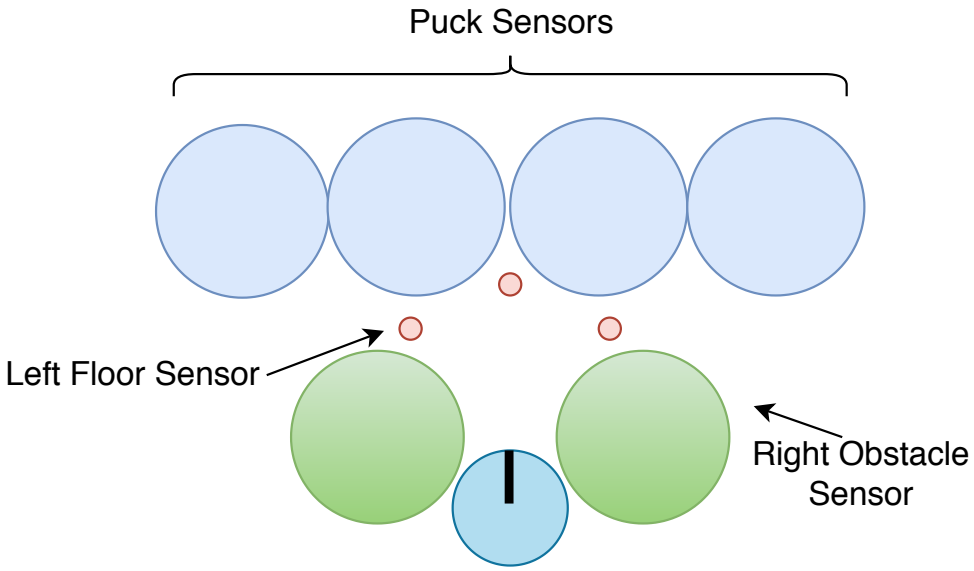


Figure 22: The updated sensor configuration for each agent (obstacle sensors added). Agent shown on bottom as a teal circle with its current heading (black line). Sensors are rigidly fixed relative to the position and heading of the agent, moving with the agent as it moves.

As demonstrated in Figure 23, the 'Original_Obstacle' state representation significantly outperforms the 'Original'. The system using the 'Original_Obstacle' state representation averaged at 682 successful formations, while the 'Original' state representation averaged at 421 successful formations. This translates to a 62% increase in formations when we include the results from obstacle sensors into our state representation. One explanation for this is that agents learn to avoid each other, thus having more time to focus on puck transfer.

By using the 'Original_Obstacle' state representation, a policy is developed that incorporates the consequence of colliding with another agent. For example, if choosing to turn right while $OR = 1$ results in less pucks being moved towards the threshold, our policy eventually learns to avoid turning right in that particular state. This becomes helpful in later chapters when we are trying to reduce damage

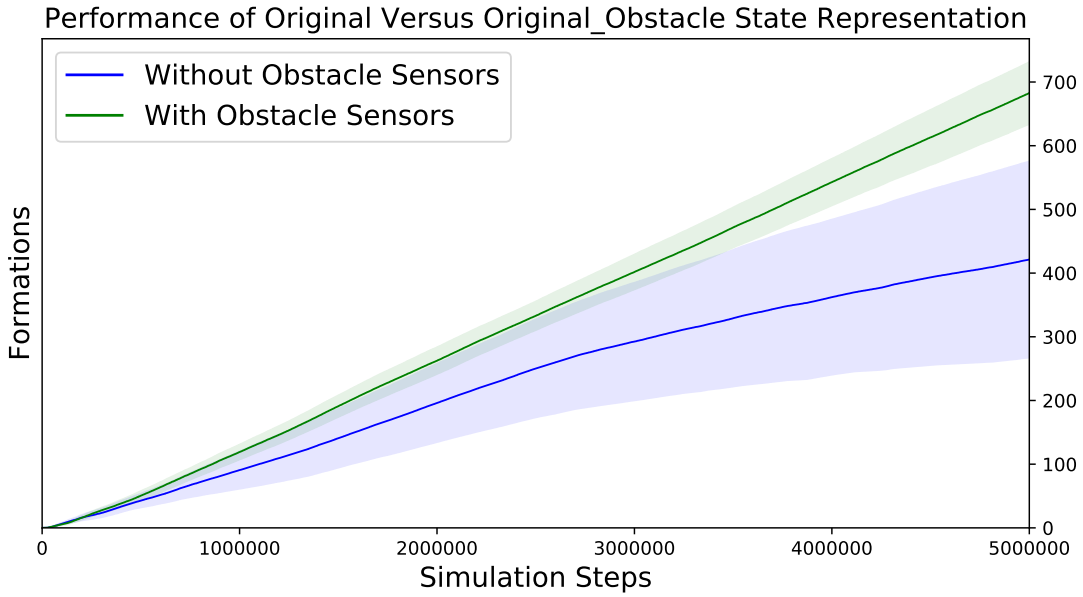


Figure 23: Averaged Results from 10 runs of RL using the original 8-bit state representation versus RL using the altered 10-bit state representation incorporating the left and right obstacle sensors.

done to physical robots through agent-agent collision.

5.3 Summary of Results

The way in which we represent the state of an agent in our system is very important. The state encompasses everything that an agent knows about its environment, and bases its decisions off of that. We wanted to keep our state representation as concise as possible while also maintaining all of the critical information about an agent's surroundings.

In our first experiment, we altered the amount of bits being used to represent the scalar field value within the state representation. First, we decreased the number of bits being used from 4 to 3, resulting in 2^3 (128) possible scalar field values.

Doing this resulted in a 13% increase in the number of successful formations. This demonstrated to us that using 4 bits to represent the scalar field value might be excessive. By using only 3 bits, there are less possible states that an agent can be in, and thus the policy becomes viable at a faster rate. Next, we increased the number of bits used to represent our scalar field value from 4 to 5. This had the opposite effect that decreasing the number of bits did. By increasing the granularity, there was a 29% decrease in the number of formations. It is likely that the agents were not ending up in certain states often enough to build a robust policy equivalent to our 'Original' state representation.

Next, we wanted to explore what kind of impact including binary information from object sensors to the left and right of each agent would have on the performance of our system. We hypothesized that the agents would learn to avoid each other and thus increase the efficiency of the system — and it did just that. Adding information from these object sensors increased the complexity of our state representation but had a positive impact on the number of successful formations completed. We saw a 62% increase in performance when using a state representation using obstacle sensors versus our 'Original' state representation that did not.

In this chapter we larned the importance of carefully selecting the way in which we represent our states. By providing too much or too detailed information, our performance can suffer. However by providing too little, our system may fail to function in the way that we want. We found that when deciding on a state representation, there is a balance between detail and minimalism that should be reached in order to achieve optimal efficiency.

6 Agent-Agent Collision

The performance of large groups of robots is often limited by a commonly shared resource [36]. This effect, termed interference, can have a large impact on the effectiveness of robotic swarms. Interference is one of the key problems in large cooperating groups. The time each robot spends doing behaviours not related to the main task increases whenever the density of individuals increases, effectively reducing the performance of the system as a whole [36]. These behaviours could be things like obstacle avoidance or colliding into each other.

The goal of simulation softwares is to model a real phenomenon with a set of mathematical formulae so that we do not have to use real (often expensive) hardware for testing. We would like to eventually transfer what we have learned to real hardware so that it can have tangible benefits in the real world. In previous experiments, we did not track the number of collisions between objects within the environment; however, exploring methods in which we can decrease agent-agent collisions is vital in preparing for the transfer of our research to real robots. The less agent-agent collisions we have, the less damage our robots incur, and the more efficient our system is due to agents spending less time being stuck.

Three experiments were carried out to both explore the natural incline and decline of agents colliding into each other throughout a simulation, and find ways in which it can be reduced without interfering with the RL algorithm: (1) analysis of the frequency of agent-agent collisions in learned policies with no alterations made, (2) adding obstacle sensors to each agent so it can detect whether there is another agent on the left or right, then adding those values to the state representation, and (3) altering the reward function so that an increase in agent-agent collisions

decreases the returned reward, and a decrease in agent-agent collisions increases the returned reward.

6.1 Performance Metric

Instead of focusing on the exact number of collisions during every simulation, we decided to count the number of agent-agent collisions that occurred every 1000 time steps. Initially, we wanted to record the percentage of collisions that were specifically one agent colliding with another, however this percentage would decrease based on unchanging numbers of agent-agent collisions and an increasing number of any other type of collision often resulting in inaccurate readings. Thus, for the entirety of our simulations, we recorded the number of agent-agent collisions that occurred during the previous 1000 simulation steps.

6.2 Experiment 1: Agent-Agent Collisions in Unaltered Systems

First, we wanted to determine the typical number of agent-agent collisions throughout a simulation without making changes to the system. This data acted as a benchmark to compare with the performance of future alterations made to decrease collisions between agents. During this experiment, we used configuration values identical to those in Table 1. The average collision results after forming an annulus shape are shown by the green line of best fit in Figure 24. From this graph we see that there is a trend towards increased collisions as learning progresses. Further, the average number of collisions per 1000 time steps is high at 1528 collisions. This means it took about 0.65 time steps for a collision to occur.

6.3 Experiment 2: Inclusion of Obstacle Sensors into State Representation

In our second experiment, we wanted to determine the effect that adding in obstacle sensors (and adjusting our state representation accordingly) would have on agent-agent collisions. While keeping all other sensors the same, we added a left and right obstacle sensor to the agent, shown in Figure 22. These obstacle sensors were discussed previously in greater detail in Section 5.2.

Aside from incorporating new obstacle sensors into the state representation, this experiment does not focus on doing anything to explicitly reduce the number of agent-agent collisions. Our goal was to determine if the agents learned to avoid each other naturally. Based on the nature of RL, this should happen if agents learned that avoiding collisions resulted in a higher reward. Since our reward function has not been altered, the reward would only increase if avoiding other agents resulted in more pucks being pushed towards the defined threshold(s). This seems plausible, as avoiding interference leaves extra time to accomplish puck-related tasks. The results of this experiment are demonstrated by the blue line of best fit in Figure 24. The average number of collisions per 1000 time steps during this experiment was 752. This means it took on average 1.33 time steps for an agent-agent collision to occur, which is a clear improvement over not having the obstacle sensors included.

In Figure 24, we can see the difference caused by incorporating obstacle sensors. The agents learn to avoid each other better than when there was no sense of the location of other agents relative to a single agent. As stated previously, this is likely due to agents learning that a decrease in interference correlates to an increase in

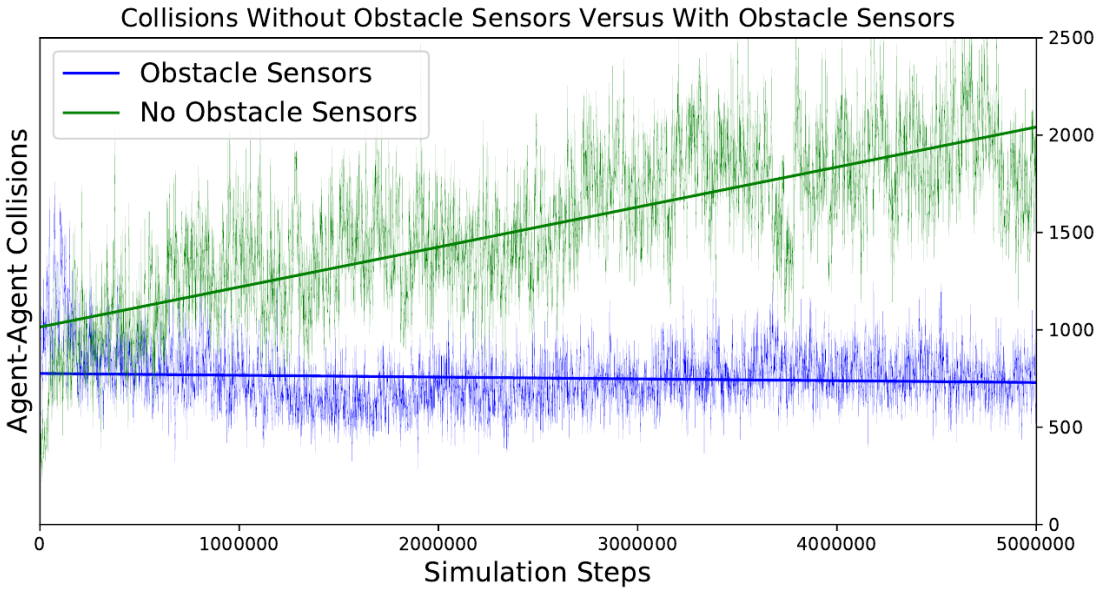


Figure 24: Agent versus agent collisions occurring every 1000 simulation steps. The blue line represents the line of best fit determined (using the polyfit function for numpy) for agent-agent collisions using the obstacle sensors, while the green line represents the line of best fit for agent-agent collisions that are not using obstacle sensors, but instead just the 'Original' state representation discussed in Chapter 5.

reward. This is a result of agents having more time and space to push pucks towards the goal instead of colliding with each other.

6.4 Experiment 3: Adjusting the Global Reward Function

As discussed in Section 3.4, reward functions are one of the most important elements of any RL problem. The reward function describes how the agent(s) should behave and stipulates what we want the agent to accomplish. By changing the reward function, we change how much of a positive or negative reward our agents receive for performing certain actions. In our current reward function, we focus solely on the movement of pucks within the environment — the reward signal is not affected by any type of collision (be it agent-agent collision, agent-puck collision, or puck-puck collision).

In our third experiment, we wanted to write a new reward function so that our environment learned not only to push the pucks into a desired shape, but also to avoid colliding into each other in the process. Recall from Section 3.4 that in an environment with P pucks, we define $SV_t(P_i)$ as the scalar field value located at the center of puck $1 \leq i \leq P$ at simulation time step t. As well, we define a puck distance function $PD_t(P_i)$ which yields 0 if the puck is inside the desired thresholds T_i and T_o at time step t, or the difference from the closest threshold if outside it:

$$PD_t(P_i) = \begin{cases} SV_t(P_i) - T_i, & \text{if } SV_t(P_i) > T_i \\ T_o - SV_t(P_i), & \text{if } SV_t(P_i) < T_o \\ 0, & \text{otherwise} \end{cases}$$

Next, assuming A agents in the environment, we define an agent distance function $AD_t(A_j)$ where $1 \leq j \leq A$ which gets the summed distance between an agent j and all other agents in the environment and divides by A to get the average distance between agent j and other agents in the environment. $SV_t(A_j)$ is defined as the scalar field value located at the center of agent j. The pseudocode describing $AD_t(A_j)$ is shown below in Algorithm 3:

Algorithm 3: The Agent-Distance Algorithm. Finds the distance between an agent and all other agents, returns the average distance

Input : Agent j, total number of agents A
Output: The average distance between agent j and all other agents

```
sum ← 0
for k ← 0 to A by 1 do
    sum += abs(SV_t(A_j) − SV_t(A_k))
return (sum/A)
```

We then define a new global evaluation function, $Eval_t(E)$ on an environment E with which averages $PD_t(P_i)$ for all pucks, and $AD_t(A_j)$ for all agents. This function uses two variables, φ and ν, that when summed add up to one and prescribe weights to both of our goals:

$$Eval'_t(E) = \varphi\left(1 - \frac{1}{P}\sum_{i=1}^{P} PD_t(P_i)\right) + \nu\left(1 - \frac{1}{A}\sum_{j=1}^{A} AD_t(A_j)\right)$$

The higher the value of φ in $Eval'_t(E)$, the more importance (or 'weight') we place on the task of moving pucks towards a defined threshold. Conversely, the higher the value of ν, the more importance we place on having agents avoid each other. In this experiment, we learned a policy from scratch on 2, 4, and 6, and 8 agents for 5,000,000 time steps. Additional variables can be seen in Table 6.

Initially, we noticed that 8 agents using our new reward function $Eval'_t$ per-

Configuration Value	**Reinforcement Learning**
Number of Robots	2
Number of Pucks	200
Forward Speed	2.0
Angular Speed	0.3
Outie Threshold	0.6
Innie Threshold	0.8
Hash Function	Original
Actions	0.3, 0.15, -0.15, -0.3
Max Simulation Steps	5,000,000
Initial Q	1.0
α	0.2
γ	0.9
φ	0.6
ν	0.4
Epsilon	0.0
Reset Percentage (%)	94
Q-Learning	On
Reward Function	$Eval'_t$
Steps Before 'Stuck'	150,000

Table 6: Configuration values used while testing the performance of the new reward function, $Eval'_t$.

Experiment	Successful Formations	Collisions/1000 Steps
1 (Original)	421	1528
2 (Original_Obstacle)	682	752
3 (New Reward Function)	47	420

Table 7: Averaged results on successful formations and average collisions collected from Chapter 6.

formed significantly better than 8 agents using the on-board obstacle sensors. Over time, as the policy converges, the agents learn not to collide into each other. This is demonstrated in Figure 25. The average number of collisions per 1000 time steps is 420. This evens out to about one agent-agent collision every 2.4 time steps, much better than using just obstacle sensors alone.

While the number of collisions is significantly reduced by using the new reward function, we wanted to see how the performance of the system was affected. Since the system is spending time learning how not to collide with other agents, learning a way to push pucks towards the desired threshold will take longer as there are now two primary goals. As seen in Figure 26, using the new reward function drastically affects the speed in which formations are created. Using just the 'Original_Obstacle' hash function, we averaged 682 successful formations. By using $Eval'_t$, we averaged just 47 successful formations. From this, it is clear to see that there huge is a tradeoff between reducing collisions and creating formations. Our policy learns to create successful formations much slower when we are also learning to avoid agent-agent collisions.

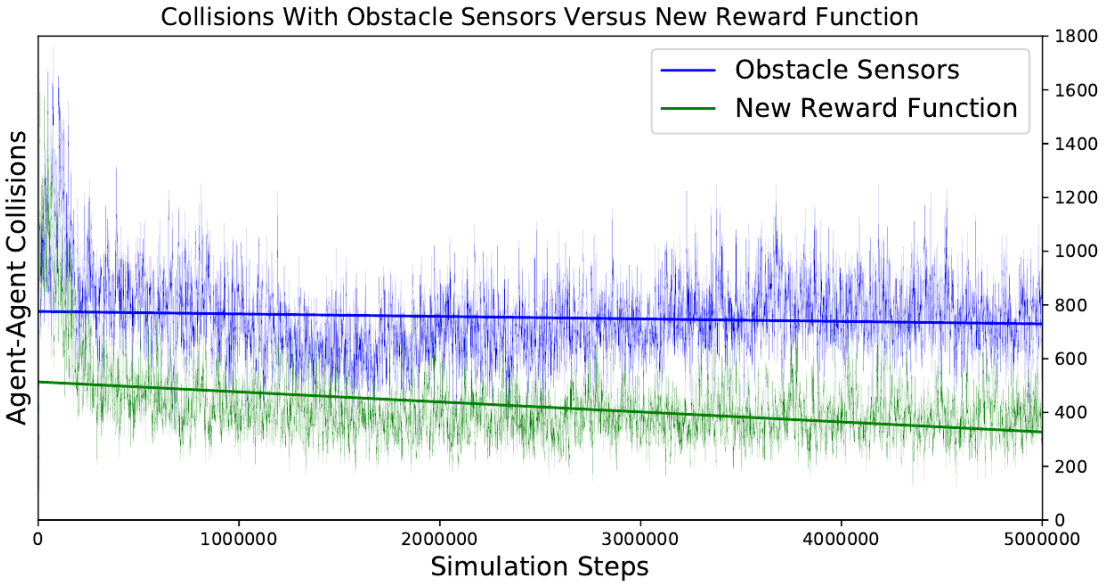


Figure 25: Agent versus agent collisions occurring every 1000 simulation steps. The blue line represents the line of best fit for agent-agent collisions using the obstacle sensors, while the green line represents the line of best fit for agent-agent collisions using the updated reward function, $Eval'_t$.

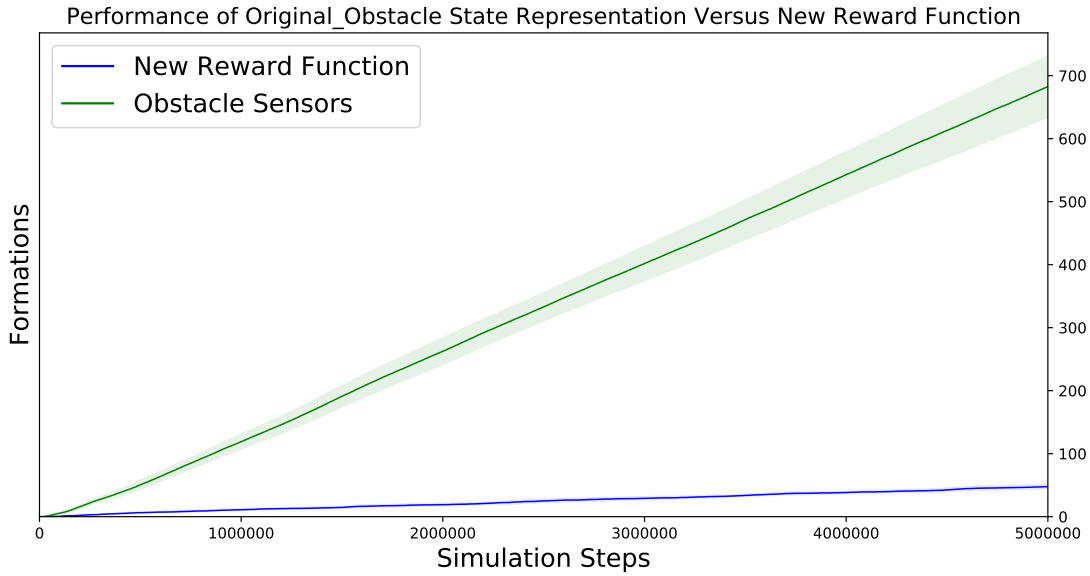


Figure 26: Performance of using just 'Original_Obstacle' state representation versus using the new reward function, $Eval'_t$ using 8 agents for 10 trials.

6.5 Summary of Results

Finding inexpensive ways to reduce the number of agent-agent collisions is vital when working with real-world MAS. Hardware can be extremely expensive. In our case each agent would cost approximately $250.00. Because of this, reducing the amount of contact each agent has with each other helps ensure that hardware on-board the agents won't be damaged by constant force being applied to them. In this chapter, we explored how the system currently fares with agent-agent collisions, as well as ways to reduce these collisions.

In our first experiment, we wanted to create a benchmark for future alterations to the system. We wanted to see how many agent-agent collisions our system experienced on average without any changes made. What we found was that agents were colliding with each other more than once per time step (1528 collisions per 1000 time steps). As well, the slope of the line of best fit for collisions was positive, so the number of collisions were increasing as the policy was progressing. This showed us that changes needed to be made to our system in order to reduce collisions between agents.

In our next experiment, we explored the effect that using the 'Original_Obstacle' hash function and obstacle sensors had on agent-agent collisions. By being able to sense other agents to the left and right, agents collided with each other **51%** less than they did without the obstacle sensors (752 collisions per 1000 time steps). The upwards slope demonstrated by collisions occuring from using the 'Original' state representation is something to be avoided, as the longer the test runs, the more frequently collisions occur. Ideally, we want to be able to train our policy for as long as we wish without worrying about the increase in agent-agent collisions.

In addition, as we learned in Chapter 5, there is an increase in the number of successful formations when we use the information retrieved from obstacle sensors. So as the number of agent-agent collisions decreases, our performance is also positively affected.

Finally, we altered our reward function to increase the returned reward the further agents are from each other, and decrease the returned reward the closer they are to each other. This increase or decrease in reward is combined with the reward given for pushing pucks towards the threshold(s), and both are weighted using φ and ν. This technique resulted in just 420 agent-agent collisions per 1000 time steps. This translates to a **73%** decrease in collisions compared to the original system. As well, the line of best fit for this technique is negative. This means that as the policy progresses, the amount of agent-agent collisions decreases over time. Again, performance in terms of shape formation is negatively affected by altering the reward function. With just 47 successful formations in 5,000,000 time steps, it seems that the more we emphasize avoiding collisions between agents, the worse the performance of the system is.

Through the experiments in this chapter, we learned that reducing the number of agent-agent collisions is a difficult task. There appears to be a trade-off between reducing collisions and forming successful formations. With this being said, finding a balance between this two things seems a necessity if we plan to using physical robots to demonstrate our work. For porting our work to physical robots, we would likely perform on-line learning in simulation to learn a policy using $Eval'_t$ and then apply that policy to physical agents. Because the average number of collisions is reduced over time while learning a policy using $Eval'_t$, the policy that we feed the robots should result in a fairly low number of collisions between agents.

7 Summary and Conclusions

In this thesis we have proposed a system using RL to improve upon previous methods of shape construction within MAS. Our system removes the dependence on hand-coding algorithms and has the benefit of being able to adapt to unique environments and shape formations. Unlike OC, RL does not need distinct types of agents to complete a task. Instead, agents learn to adapt and fulfill multiple roles based on their current state. The RL solution can also form different types of shapes that the OC algorithm struggles with (likely due to right-angles and concave features). We have also demonstrated the promising results for policy transfer, in which policies learned quickly by fewer agents can be carried out by more robots. Furthermore, we have unearthed some plausible solutions to collision reduction by including obstacle sensors and altering the reward function. This translates to a reduction in collisions between agents, and thus a method to reduce financial costs when policies are loaded onto physical agents.

There were also a number of experiments that were executed but not included in this thesis. For example, we examined the results of altering the ϵ value, thus altering the ratio of agents' decision to follow the policy or explore a random action in a given state. We also experimented on a variety of combinations of possible action spaces and examined how more or less possible actions affected the system's ability to function. As well, we examined how RL performed when the scalar field was modeling open versus closed shapes and symmetrical versus asymmetrical shapes. There are an array of questions to delve into and explore within this topic and within our specific environment, however we tried to focus on what we deemed to be the most imperative.

Many ideas relating to planar construction using MARL were discussed throughout the course of this research; however due to time and financial constraints, were not able to be fully explored. The first of these ideas was implementing deep neural network state representations coupled with deep RL to test the limits of what complex shapes can be formed by more modern learning algorithms. We also would have liked to learn a policy on-line through the Cwaggle simulator and then apply that policy to the physical agents in our lab. This would have indicated to us the issues surrounding the transition from simulation to physical hardware. We would have seen how noise in the physical environment affects performance, and if latency in data transfer affects agents' abilities to properly execute the policy.

We believe that these experiments have only scratched the surface of using RL for swarm shape formation, as we have shown results using the most basic form of tabular Q-learning. In the future, we would like to expand on the power of using alternate or multiple reward functions. We would like to look into adjusting weights attached with specific goals of the reward function and seeing how it affects the resulting policy. We would also like to implement deep neural network state representations coupled with deep RL to test the limits of what complex shapes can be formed by more modern learning algorithms. We believe that there are many ways to move forward with this research, and much to be uncovered with more powerful machine learning techniques that will progress the world of multi-robot planar construction.

References

[1] Tucker Balch. Behavioral diversity as multiagent cooperation. In *Sensor Fusion and Decentralized Control in Robotic Systems II*, volume 3839, pages 55–63. International Society for Optics and Photonics, 1999.

[2] Yoseph Bar-Cohen and Cynthia Breazeal. Biologically inspired intelligent robots. In *Smart Structures and Materials 2003: Electroactive Polymer Actuators and Devices (EAPAD)*, volume 5051, pages 14–20. International Society for Optics and Photonics, 2003.

[3] José Barbosa and Paulo Leitão. Modelling and simulating self-organizing agent-based manufacturing systems. In *IECON 2010-36th Annual Conference on IEEE Industrial Electronics Society*, pages 2702–2707. IEEE, 2010.

[4] Horace B Barlow. Unsupervised learning. *Neural Computation*, 1(3):295–311, 1989.

[5] Eric Bonabeau, Marco Dorigo, Directeur de Recherches Du Fnrs Marco, Guy Theraulaz, Guy Théraulaz, et al. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford university press, 1999.

[6] Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 195–210. Morgan Kaufmann Publishers Inc., 1996.

[7] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations in Multi-Agent Systems and Applications*, pages 183–221. Springer, 2010.

[8] David Carmel and Shaul Markovitch. Learning models of intelligent agents. In *AAAI/IAAI, Vol. 1*, pages 62–67, 1996.

[9] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *In AAAI/IAAI*, 1998:746–752, 1998.

[10] Ginni Devi and Harjot Kaur. A novel step towards deep-reinforcement learning in a cooperative multi-agent system. *International Journal of Information Systems & Management Science*, 1(1):87–95, 2018.

[11] Kids Discover. *Insect Nests*, accessed June 24, 2019. URL: `https://online.kidsdiscover.com/unit/insects/topic/insect-nests`.

[12] Melvin Gauci, Jianing Chen, Wei Li, Tony J Dodd, and Roderich Gross. Clustering objects with robots that do not compute. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems*, pages 421–428. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

[13] Camron Godbout. *What Killed the Curse of Dimensionality?*, September 6, 2017 (accessed July 10, 2019). URL: `https://hackernoon.com/what-killed-the-curse-of-dimensionality-8dbfad265bbe`.

[14] Laura Rodríguez Gómez. *Machine Learning Support for Logic Diagnosis*. PhD thesis, Universität Stuttgart, 2017.

[15] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[16] Jian Han, Chang-hong Wang, and Guo-xing Yi. Cooperative control of UAV based on multi-agent system. In *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, pages 96–101. IEEE, 2013.

[17] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[18] Giuseppe Iazeolla, Alessandra Pieroni, Andrea D'Ambrogio, and Daniele Gianni. A distributed approach to the simulation of inherently distributed systems. In *Proceedings of the 2010 Spring Simulation Multiconference*, page 132. Society for Computer Simulation International, 2010.

[19] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[20] Dimitar Kazakov and Daniel Kudenko. Machine learning and inductive logic programming for multi-agent systems. In *ECCAI Advanced Course on Artificial Intelligence*, pages 246–270. Springer, 2001.

[21] Khaled M Khalil, M Abdel-Aziz, Taymour T Nazmy, and Abdel-Badeeh M Salem. Machine learning algorithms for multi-agent systems. In *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, page 59. ACM, 2015.

[22] Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsuhi Shinjou, and Susumu Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics*, volume 6, pages 739–743. IEEE, 1999.

[23] Jelle R Kok, Eter Jan Hoen, Bram Bakker, and Nikos Vlassis. Utile coordination: Learning interdependencies among cooperative agents. In *IEEE Symp. on Computational Intelligence and Games, Colchester, Essex*, pages 29–36, 2005.

[24] Dan Ladley and Seth Bullock. Logistic constraints on 3D termite construction. In *International Workshop on Ant Colony Optimization and Swarm Intelligence*, pages 178–189. Springer, 2004.

[25] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.

[26] Kristina Lerman. A model of adaptation in collaborative multi-agent systems. *Adaptive Behavior*, 12(3-4):187–197, 2004.

[27] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[28] Maja J Matarić. Learning in multi-robot systems. In *International Joint Conference on Artificial Intelligence*, pages 152–163. Springer, 1995.

[29] Nick McCrea. *An Introductory Robot Programming Tutorial*, October 16, 2018 (accessed July 29, 2019). URL: `https://www.toptal.com/robotics/programming-a-robot-an-introductory-tutorial`.

[30] P Miller. The Genius of Swarms. *National Geographic*, 212:126–147, 2007.

[31] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. MIT press, 2018.

[32] TTH Ng and GSB Leng. Engineering applications of artificial intelligence. *Department of Mechanical Engineering, National University of Singapore*, pages 439–445, 2005.

[33] Lynne E Parker. *Heterogeneous multi-robot cooperation*. PhD thesis, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1994.

[34] PBS. *The Incredible Termite Mound*, October 28, 2011 (accessed August 18, 2019). URL: `https://www.pbs.org/wnet/nature/the-animal-house-the-incredible-termite-mound/7222/`.

[35] Kirstin H Petersen, Nils Napp, Robert Stuart-Smith, Daniela Rus, and Mirko Kovac. A review of collective robotic construction. *Science Robotics*, 4(28), 2019.

[36] Giovanni Pini, Arne Brutschy, Mauro Birattari, and Marco Dorigo. Interference reduction through task partitioning in a robotic swarm. In *Sixth International Conference on Informatics in Control, Automation and Robotics–ICINCO*, pages 52–59, 2009.

[37] Jim Pugh and Alcherio Martinoli. Multi-robot learning with particle swarm optimization. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent S=ystems*, pages 441–448. ACM, 2006.

[38] Jim Pugh, Alcherio Martinoli, and Yizhen Zhang. Particle swarm optimization for unsupervised robotic learning. In *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.*, pages 92–99. IEEE, 2005.

[39] George John Romanes. *Animal intelligence*. D. Appleton, 1883.

[40] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[41] Maryam Sadeghlou, Mohammad Reza Akbarzadeh-T, and Mohammad Bagher Naghibi-S. Dynamic agent-based reward shaping for multi-agent systems. In *2014 Iranian Conference on Intelligent Systems (ICIS)*, pages 1–6. IEEE, 2014.

[42] Erol Şahin, Sertan Girgin, Levent Bayindir, and Ali Emre Turgut. Swarm robotics. In *Swarm intelligence*, pages 87–100. Springer, 2008.

[43] Madhhu Sanjeevi. *Different Types of Machine Learning*, 2007 (accessed June 24, 2019). URL: `https://medium.com/deep-math-machine-learning-ai/different-types-of-machine-learning-and-their-types-34760b9128a2`.

[44] PO Skobelev, EV Simonova, AA Zhilyaev, and VS Travin. Application of multi-agent technology in the scheduling system of swarm of earth remote sensing satellites. *Procedia Computer Science*, 103:396–402, 2017.

[45] T Soleymani, V Trianni, M Bonani, F Mondada, M Dorigo, et al. An autonomous construction system with deformable pockets. *IRIDIA Technical Report Series*, 2014.

[46] Diana F Spears. Assuring the behavior of adaptive agents. In *Agent Technology from a Formal Perspective*, pages 227–257. Springer, 2006.

[47] Robert L Stewart and R Andrew Russell. A distributed feedback mechanism to regulate wall construction by a robotic swarm. *Adaptive Behavior*, 14(1):21–51, 2006.

[48] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[49] Richard S Sutton, Andrew G Barto, et al. *Introduction to Reinforcement Learning*. MIT press Cambridge, 1998.

[50] Guy Theraulaz, Jacques Gautrais, Scott Camazine, and Jean-Louis Deneubourg. The formation of spatial patterns in social insects: From simple behaviours to complex structures. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 361(1807):1263–1282, 2003.

[51] Karl Tuyls and Gerhard Weiss. Multiagent learning: Basics, challenges, and prospects. *AI Magazine*, 33(3):41–41, 2012.

[52] 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.

[53] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[54] G Weiss. Multiagent systems a modern approach to distributed modern approach to artificial intelligence, 1999, 1999.

[55] Gerhard Weiß and Pierre Dillenbourg. What is multi in multiagent learning. *Collaborative learning. Cognitive and computational approaches*, pages 64–80, 1999.

[56] Erfu Yang and Dongbing Gu. Multiagent reinforcement learning for multi-robot systems: A survey. Technical report, tech. rep, 2004.

[57] William Yeoh and Makoto Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–53, 2012.