Real-time Learning and Planning in Environments with Swarms: A Hierarchical and a Parameter-based Simulation Approach*

Lukasz Pelcner[1], Shaling Li[2], Matheus Aparecido do Carmo Alves[3], Leandro Soriano Marcolino[1], Alex Collins[1]
[1] School of Computing and Communications, Lancaster University
[2] Portsmouth Business School, University of Portsmouth
[3] Institute of Mathematics and Computer Science (ICMC), University of São Paulo (USP)
l.pelcner@lancaster.ac.uk, shaling.li@port.ac.uk, matheus.aparecido.alves@usp.br, l.marcolino@lancaster.ac.uk, a.collins5@lancaster.ac.uk

ABSTRACT

Swarms can be applied in many relevant domains, such as patrolling or rescue. They usually follow simple local rules, leading to complex emergent behavior. Given their wide applicability, an agent may need to take decisions in an environment containing a swarm that is not under its control, and that may even be an antagonist. Predicting the behavior of each swarm member is a great challenge, and must be done under real time constraints, since they usually move constantly following quick reactive algorithms. We propose the first two solutions for this novel problem, showing integrated on-line learning and planning for decision-making with unknown swarms: (i) we learn an ellipse abstraction of the swarm based on statistical models, and predict its future parameters using time-series; (ii) we learn algorithm parameters followed by each swarm member, in order to directly simulate them. We find in our experiments that we are significantly faster to reach an objective than local repulsive forces, at the cost of success rate in some situations. Additionally, we show that this is a challenging problem for reinforcement learning.

ACM Reference Format:
Lukasz Pelcner, Shaling Li, Matheus Aparecido do Carmo Alves, Leandro Soriano Marcolino, Alex Collins. 2020. Real-time Learning and Planning in Environments with Swarms: A Hierarchical and a Parameter-based Simulation Approach. In *Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), Auckland, New Zealand, May 9–13, 2020,* IFAAMAS, 9 pages.

1 INTRODUCTION

Swarms have a great range of applications, such as patrolling [6], mapping [13], foraging [7], rescue [18], etc. Despite following simple local rules, the system may exhibit complex behavior, and its decentralized nature makes them robust and fault tolerant. Hence, we expect a wide presence of swarms across many domains.

However, a swarm may not always be under an agent's control, and may sometimes even be an agent's antagonist. In those scenarios, they usually follow unknown algorithms, making it hard to predict their behavior. Therefore, it is a great challenge for an agent to take decisions towards reaching a certain objective, in an environment containing an unknown swarm. In particular, an agent must estimate swarm members' behavior while planning its actions under strong real time constraints, since a swarm usually moves quickly in an environment following fast reactive algorithms, and leading to a fast-paced changing world state.

Previous work has studied how to add agents to influence a swarm [4, 5], or how to add agents to a swarm in order to disrupt coverage tasks [14]. However, these would still require full knowledge of the underlying algorithm used by the swarm system. Recently, inverse reinforcement learning has been proposed to create a model of an unknown swarm [20], but without considering using such models for decision-making, and real time constraints for learning and planning.

Therefore, in this work we introduce the *novel problem* of taking decisions in an environment containing an unknown swarm *under real-time constraints* and *without any pre-training*. That is, *all learning happens on-line, in a single execution, and no knowledge is used across multiple runs*. Hence, the agent can always handle *new unknown swarms*.

We present two *novel algorithms for this problem*, which integrate on-line learning and on-line planning. The first one proposes an ellipse abstraction for the whole swarm, based on statistical models, and uses time-series to predict how they will move in the future. Our second algorithm uses simulations to estimate an error function, in order to continuously estimate the parameters of the swarm potential algorithm by gradient descent. These models are then used to perform Monte Carlo simulations for real-time decision-making. Our approaches could be used in cooperative or competitive settings, but we focus on the latter in our experiments.

We evaluate our technique in an "infiltration game", where an agent must reach a target without being captured by an unknown patrolling swarm, without any pre-training. We are able to reach the target significantly faster than applying reactive local decision-making rules, at the cost of some success rate loss in some cases. On the other hand, in very hard situations, we actually obtain the same success rate than the reactive algorithm, with a significantly better time. Additionally, we show that reinforcement learning approaches have great difficulties in solving this problem.

*Equal contribution by the first two authors.

Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), B. An, N. Yorke-Smith, A. El Fallah Seghrouchni, G. Sukthankar (eds.), May 9–13, 2020, Auckland, New Zealand. © 2020 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

2 RELATED WORK

Planning in scenarios with a large number of agents is very common in Real Time Strategy Games. However, usually a centralized controller decides the actions of all units (agents), and scripts are employed to reduce the action space and guide the search [10, 11]. Reinforcement learning can also be applied, given a set of scripts to suggest actions to all agents [19]. In our case, however, we focus on single-agent decision-making, given an unknown swarm in the scenario. We also do not assume a set of scripts to suggest actions.

On the other hand, agents that model other agents for decision making is a very active research area, as shown in a recent survey [2]. In particular, agents model others in ad-hoc teamwork, where teams must coordinate without pre-programmed rules. Many works in that area assume a known algorithm template, and the main focus is on learning parameters [1]. We take inspiration on this idea, but focus on learning parameters of a complex swarm system.

Many works in multi-agent systems consider how to place and move agents in order to influence the behavior of a swarm [4, 5]. In those works a perfect model of the swarm is given to the influencing agents, while we study how to achieve a certain objective given an environment with an unknown swarm.

Inverse reinforcement learning for modelling swarm systems has been explored [20], where an unknown swarm is assumed to follow a Markov Decision Process (MDP) policy, which is learned from several observations. In our work we also learn a swarm model, but our methods allow us to quickly learn in real-time in large domains, without any pre-training.

Hierarchical abstractions have been applied for the *control* of robotic swarms [15, 16]. In such works the robots are directly controlled to move within a specific formation (e.g., an ellipse), and controlling that formation allows easy control of the swarm. In our work, however, we are *learning* a shape given a swarm that is not under our control, and using that for predicting their behavior.

Finally, to the best of our knowledge, Sanghvi et al. [14] is the first work to consider agents that must accomplish a certain objective in a scenario given an antagonist swarm. They study how to insert "fake" members that will disrupt a swarm performing a coverage task, *given a perfect model of the swarm and its parameters*. On the other hand, our approach is not focused in disrupting coverage, and we learn swarm models on-line.

3 METHODOLOGY

We assume a swarm, composed of a set of agents Ω, where each $\omega \in \Omega$ decides its actions (e.g., acceleration vector) based on a local decision given the subset of neighbor agents $\mathbf{N} \subset \Omega$ that is within a certain (unknown) distance r from ω. This local decision is performed by an unknown algorithm.

Additionally, our approaches can be applied when agents in Ω perform waypoint navigation, according to an (unknown) set of waypoints $\mathbf{W}$. That is, the local algorithm may push the agents towards the next waypoint $\mathbf{w}_i$, and once any agent reaches $\mathbf{w}_i$, it would push all agents towards $\mathbf{w}_{i+1}$. This situation happens, for example, when the swarm is patrolling a certain area (as how we will see later in our "infiltration game"). Note, however, that our approaches *support* waypoint navigation, but that is not required.

We also consider an agent ϕ, where $\phi \notin \Omega$, but ϕ acts in the same environment as the agents in Ω, and must accomplish a certain objective. That is, ϕ does not follow a reactive algorithm like the swarm, ϕ is able to observe the full state of the environment s, and the corresponding output of a reward function $R : \mathbf{S} \rightarrow \mathbb{R}$, where $\mathbf{S}$ is the set of all possible states. Hence, ϕ must find actions that maximizes the expected sum of discounted rewards $E[\sum_{j=0}^{\infty} \gamma^j \tau_{t+j}]$, where t is the current time and τ_{t+j} is the reward received j steps in the future ($\gamma \in [0, 1]$ is a discount factor).

For all agents in Ω and ϕ, we consider a continuous action space (e.g., acceleration vectors). Additionally, ϕ does not control the *time* of the state transitions, since it has no control of Ω. That is, while ϕ is processing (learning or computing next action), the current state s changes, since agents in Ω are moving. Like-wise, ϕ may have a current speed and acceleration, and thus ϕ's own movement changes the state s while it is learning/computing its next action. Therefore, we consider strong *real-time constraints*.

We also consider that *the swarm is unknown at every execution*. Hence, there is no knowledge being accumulated across multiple runs (like in reinforcement learning approaches), and *all learning happens within a single run*. Additionally, we consider that ϕ does not communicate with agents in Ω, and its objective may be either cooperative or competitive against the swarm system (defined by the reward function).

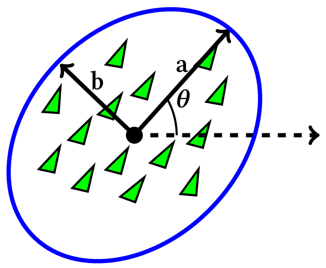
Our approaches have an *on-line learning* and an *on-line planning* component, since models to predict Ω's behavior must be learned for ϕ's decision-making, and they are both executed at every iteration in real-time. In the next sections we show novel algorithms to predict the swarm behavior: one that performs hierarchical abstraction simulations (HAS), based on statistical models; and another that directly simulates the swarm (DSS) for parameter learning. We will then explain how these models are used for decision-making.

3.1 Hierarchical Abstraction (HAS)

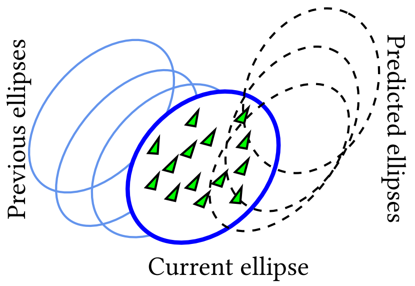
Instead of modelling each swarm member individually, we can model the whole swarm as a group. The main idea is to define one ellipse at each state covering (most of) the swarm members, and use those to predict their future positions. Our main insight comes from seeing the swarm as a sample of a 2-D Gaussian distribution, allowing us to calculate an ellipse from the covariance matrix, which covers the members at any arbitrary confidence level.

That is, for each state we obtain the location of all swarm members in a matrix $\mathbf{L}$, which we will use to calculate the ellipse parameters: center location (x_c, y_c), width (a), height (b) and angle (θ). θ is the anticlockwise angle between the ellipse's largest axis and the positive x-axis. An affine transformation of the coordinates using θ and (x_c, y_c) is performed, leading to the ellipse equation: $\frac{((x-x_c)\times\cos(\theta)-(y-y_c)\times\sin(\theta))^2}{a^2} + \frac{((x-x_c)\times\sin(\theta)+(y-y_c)\times\cos(\theta))^2}{b^2} = 1$.

Given $\mathbf{L}$ at one state, we use the covariance matrix Σ to define an ellipse covering the members at an arbitrary confidence level (Figure 1 (a)). That is, assuming that $\mathbf{L}$ is a sample from a 2-D Gaussian distribution, the eigenvectors of Σ $(\vec{e}_0, \vec{e}_1)$ point to the direction of largest spread of the data in $\mathbf{L}$, and the eigenvalues represent the variance along the eigenvectors' direction. Hence, the eigenvectors $\vec{e}_i$ define the two fundamental axes of the ellipse. By using the standard deviation ($\sqrt{||\vec{e}_i||}$), we can multiply the normalized axes by a certain constant $c\sqrt{||\vec{e}_i||}$ in order to get a desired confidence level (i.e., the probability of a point from the estimated distribution to lay within the ellipse). c can be directly obtained from the Cumulative



(a) Ellipse's parameters are calculated from covariance: a and b are its axes, θ is its angle with the x-axis (dashed line).



(b) History of ellipses is used to predict future ellipses. Previous locations are in blue, and future swarm locations are predicted in dashed black.

Figure 1: Hierarchical abstraction of a swarm.

Distribution Function (*CDF*) of the χ^2 distribution with 2 degrees of freedom [17], leading to $a := \frac{\sqrt{c'||\vec{e}_0||}}{2}$, and $b := \frac{\sqrt{c'||\vec{e}_1||}}{2}$, where c' is such that χ^2 $CDF(c')$ equals our desired confidence level.

The center of the ellipse (x_c, y_c) can be directly calculated as the mean x and y across $\mathbf{L}$. Finally, θ is the angle between the largest eigenvector ($\vec{e}_l$) and the x-axis: $\theta = \arctan \frac{\vec{e}_l^{\,y}}{\vec{e}_l^{\,x}}$, where the superscript indicates vector components.

Hence, each state s_t leads to 5 ellipse parameters: $(x_c, y_c, a, b, \theta)^t$. Since there is a sequence of states up to the current one (t), we model a time-series for each parameter: e.g., $[x_c^0, x_c^1, \ldots, x_c^t]$ for parameter x_c. We use the time-series up to t in order to forecast future ellipse parameters over time, allowing us to predict the location and shape of the swarm in the future (Figure 1 (b)). Note that this approach allows one to make predictions *without any knowledge of the swarm algorithm*, and *does not require any estimation of waypoints*.

For each parameter the sequence of values up to t is used as input to a time series prediction model. We used non-seasonal ARIMA (Auto-Regressive Integrated Moving Average) for time-series prediction. Since 1970s, ARIMA has been widely used as a general class for time series forecasting [3]. It has a family of models, and each hyper-parametrisation (*order*) (p, d, q) describes a particular one: e.g., $(x, 0, 0)$ is AR(x) model (autoregressive model); $(0, x, 0)$ is I(x) model (differencing model); and $(0, 0, x)$ is MA(x) model (moving average 'smoothing' model). This is useful when the structure of the time series is unknown, as in our case (given the complexity of swarm movement). We noticed that a particular order could eventually lead to converging predictions, which are not suitable for later usage in decision-making.

Therefore, we run the predictions using a certain ARIMA order, and switch the order with a differencing function to fit stationary or non-stationary time series data. For example, when $d = 0$ in ARIMA order (p, d, q), it would fit stationary data, and when $d \neq 0$, it would fit non-stationary data. Hence, we define two orders $o := (p, d, q)$, $o' := (p', d', q')$, and a "switch" condition. The list of orders to be tried starts as $l := (o, o')$, and sp is a function that swaps a list ($sp((o, o')) := (o', o), sp((o', o)) = (o, o')$). Let l_i be the order i in the list. We first run l_1. If the "switch" condition is satisfied, we run l_2, and define $l := sp(l)$ for the next iterations. Whenever the "switch" condition is again satisfied, we change l to $sp(l)$. This algorithm could be generalized to trying multiple different orders, but we use two due to the real time constraints.

Our "switch" conditions is: we calculate the difference of the mean and the standard deviation between the predicted time series and the observed time series (Δ_μ, Δ_σ). If the mean or the standard deviation of the predicted series is too different to the observed series ($\Delta_\mu > t_\mu$ or $\Delta_\sigma > t_\sigma$), signaling a poor fit of the current ARIMA order, we switch the order, where t_μ and t_σ are pre-defined percentage thresholds. Note that in iterations where we switch, we have run both l_1 and l_2 for the respective parameter. Hence, we can choose the best time series for our prediction. If l_1 fails because $\Delta_\sigma > t_\sigma$ we still use l_1's prediction in case l_2 has even higher Δ_σ or Δ_μ. Otherwise, use use l_2's prediction. We use this approach because we noticed that it leads to more stable predictions.

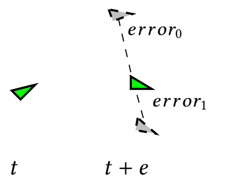
HAS does not require any previous knowledge of the algorithm being used by the swarm. However, it does assume that the swarm moves in a coherent, ellipse-like fashion. The algorithm can still be applied when this assumption does not hold, but the performance would eventually degrade. In cases where this assumption is not expected to hold well, one can apply the algorithm that we present next, *DSS*, which does not depend on this assumption.

3.2 Direct Swarm Simulations (DSS)

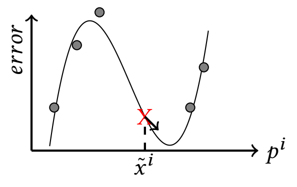
3.2.1 On-Line Parameter Learning. *DSS* needs more assumptions than *HAS*. We consider that the swarm (Ω) algorithm is parametrised by a list of parameters $\mathbf{p}$. We consider that ϕ knows the algorithm used by Ω, *but does not know the list of parameters*. Therefore, ϕ must learn an estimation $\tilde{\mathbf{p}}$ of the list of parameters, and plan its actions in real-time.

Our parameter estimation is based on Monte Carlo simulations, followed by polynomial regression and gradient descent to minimize error. The main idea is to run multiple simulations assuming different parameters, fit a polynomial to the errors, and then update the estimated parameters following the gradient descent. We write as p^i the parameter in position i of $\mathbf{p}$, as x^i a certain value for parameter p^i, and as $\tilde{\mathbf{p}}$ the list of current estimated parameters.

Let $\mathbf{q}_\omega^t$ be the position of a swarm member ω at time t, and $t + e$ be the time step where we run one parameter estimation iteration. For each parameter p^i, we will fit one polynomial. Let $\tilde{x}^i$ be the current estimated value for p^i in $\tilde{\mathbf{p}}$. We create a parameter list $\mathbf{p}_r$ to be simulated where all parameters p^j are fixed to their current estimated value $\tilde{x}^j$, except for the parameter p^i that we are currently



(a) Parameter simulations. Swarm member is in green, and dashed grey are simulated ones to learn a parameter.



(b) Gradient descent updates parameters after n simulations. Gray balls show error of randomly chosen x_r values.

Figure 2: On-line parameter learning. Each simulation has an error in predicting time $t+e$ from time t ($error_i$).

learning. That is, $\mathbf{p}_r = < \tilde{x}^0, \dots, \tilde{x}^{i-1}, x_r, \tilde{x}^{i+1}, \dots, \tilde{x}^m >$, where m is the size of $\mathbf{p}$, and x_r is a random number within p^i's range.

We then run a simulation from time step t to time step $t+e$. That will lead to estimated positions $\tilde{\mathbf{q}}_\omega^{t+e}$ for all $\omega \in \Omega$. For a certain agent ω, we calculate the *error* of $\mathbf{p}_r$ as $error_r = dist(\mathbf{q}_\omega^{t+e}, \tilde{\mathbf{q}}_\omega^{t+e})$, where $dist$ is some distance metric. In this paper we use the *Euclidean distance*. We show an example in Figure 2 (a), where we display the error of two simulations.

This process repeats n times. Therefore, we have a set $\mathbf{R}$ of random parameters $\mathbf{p}_r \in \mathbf{R}$, and their corresponding errors $error_r$ for each ω. Considering only a fixed position p^i (the one that we tried different x_r values), we fit a polynomial f using the pairs $(x_r, error_r)$ for each ω. That allows us to update $\tilde{x}^i$ by gradient descent as: $\tilde{x}^{i\prime} := \tilde{x}^i - \alpha \times f'(\tilde{x}^i)$, where $f'(\tilde{x}^i)$ is the derivative of f calculated at point $\tilde{x}^i$ and α is the learning rate (Figure 2 (b)).

This whole process repeats for each parameter p^i, leading to a new list of estimated parameters $\tilde{\mathbf{p}}'_\omega$. That is, we fit m polynomials using n points for each, where m is the size of $\mathbf{p}$, for each $\omega \in \Omega$. Note that for each ω we may find a different estimated list $\tilde{\mathbf{p}}'_\omega$. Therefore, the final estimated list $\tilde{\mathbf{p}}'$ will be the average estimated values across all ω. The whole process repeats every iteration. Therefore, for the next iteration $\tilde{\mathbf{p}}'$ will be used as the next iteration's current estimated value $\tilde{\mathbf{p}}$.

3.2.2 Waypoint Learning. If Ω's local algorithm performs waypoint navigation, then in order to run the simulations in *DSS*, it is necessary to estimate the set of waypoints $\mathbf{W}$. That is because if the local algorithm has an attraction force towards a waypoint $\mathbf{w}_i$, then the set needs to be estimated to simulate the swarm for learning and decision-making, and we do not assume prior knowledge of such set for greater applicability.

Hence, ϕ observes the movement of all agents Ω for l time steps. Every j time steps, the current center of mass $\mathbf{c}$ across all agents $\omega \in \Omega$ is added to an ordered set $\mathbf{M}$, as a potential waypoint. That is, $\mathbf{c}$ is the average position across Ω at the current time step. Afterwards, $\mathbf{M}$'s size is reduced to produce the final set of estimated waypoints $\tilde{\mathbf{W}}$, since frequently storing $\mathbf{c}$ tends to over-estimate the waypoints. We go across every triple of points $(\mathbf{w}_i, \mathbf{w}_{i+1}, \mathbf{w}_{i+2})$ in $\mathbf{M}$, in order. We consider the vectors $\vec{v}_0$ defined by connecting $\mathbf{w}_{i+1}$ to $\mathbf{w}_i$, and $\vec{v}_1$ by connecting $\mathbf{w}_{i+1}$ to $\mathbf{w}_{i+2}$. If the angle between $\vec{v}_0$ and $\vec{v}_1$ is higher than a certain constant β, we add only $\mathbf{w}_i$ and $\mathbf{w}_{i+2}$ to $\tilde{\mathbf{W}}$, since they are close to a straight line, and hence $\mathbf{w}_{i+1}$ is not providing much extra information. Otherwise, we add all 3 points to $\tilde{\mathbf{W}}$. In the next iteration we perform the same test with $(\mathbf{w}_{i+1}, \mathbf{w}_{i+2}, \mathbf{w}_{i+3})$, and so on, until covering all points in $\mathbf{M}$.

Additionally, it is necessary to estimate what is the *current* waypoint in $\tilde{\mathbf{W}}$, in order to simulate Ω. Hence, we calculate the closest waypoint $\mathbf{w}_c$ to one of the swarm members ω, the angle β_1 between ω's heading and the vector connecting ω to $\mathbf{w}_c$, and the angle β_2 between ω's heading and the vector connecting ω to $\mathbf{w}_{c+1}$. If $\beta_1 < \beta_2$, we estimate that the swarm is going towards $\mathbf{w}_c$, otherwise we estimate $\mathbf{w}_{c+1}$.

3.3 On-Line Planning

In addition to learning a model of the swarm Ω, ϕ takes actions to accomplish a certain objective (i.e., maximize its value function) in the environment with the unknown swarm. Given the real time constraints, ϕ must quickly decide its actions, since $\omega \in \Omega$ follows simple reactive neighborhood-based algorithms, and are usually constantly moving. We propose that ϕ uses Monte Carlo simulations at every iteration for decision-making, for both *HAS* and *DSS*.

That is, ϕ samples a certain action a from the set of possible actions $\mathbf{A}$ using a *sampling strategy*, and simulates the effect of taking that action a for h simulated time steps. ϕ must simulate state transitions at a quicker speed than the real state transitions, since $|\mathbf{A}'| \times h$ transitions must be simulated, where $\mathbf{A}' \subset \mathbf{A}$ is the set of sampled actions. Not necessarily all these simulations will terminate at a single real iteration, but their accuracy will decrease the longer it takes, since ϕ and agents in Ω keep moving, changing the current state s.

For each simulated iteration, ϕ samples an action a until completing h simulated time steps, using a *default policy* for simulations. In the end of the process, ϕ takes the action with the highest estimated value. In Section 4.1 we explain the *sampling strategy* and the *default policy* that we used in an "infiltration" game, which can also be seen as an example to make the process more clear. Note that we are not applying a *Monte Carlo Tree Search*, but just *Monte Carlo* simulations. Each action $a \in \mathbf{A}'$ is tried a single time in simulation. This allows ϕ to quickly take a decision concerning its next action. During the simulations, in *HAS* ϕ uses the predicted sequence of ellipse parameters: $[(x_c^1, y_c^1, a^1, b^1, \theta^1), \dots, (x_c^h, y_c^h, a^h, b^h, \theta^h)]$. Therefore, the swarm is replaced by a single ellipse that moves at each simulated iteration. In *DSS* ϕ uses the current estimated parameters $\tilde{\mathbf{p}}$ and estimated waypoints $\tilde{\mathbf{W}}$ to directly simulate the behavior of each member of the swarm Ω.

The whole process described in Section 3.1 (for *HAS*), 3.2.1 (for *DSS*) and 3.3 repeats at every iteration, once ϕ is again able to run computations. That is, ϕ is constantly updating its model of the swarm, and performing simulations to decide its next action. If the swarm Ω is doing waypoint navigation, in *DSS* ϕ first estimates the set of waypoints $\tilde{\mathbf{W}}$ for l iterations, before it starts moving. In *HAS* it also waits for l iterations, in order to get initial observations before trying to forecast using the time-series. Afterwards, ϕ moves by continuously applying on-line learning and planning, as described.

4 EXPERIMENTS

In order to evaluate our approach, we ran experiments in an "infiltration game". We place a certain target in the environment, and a

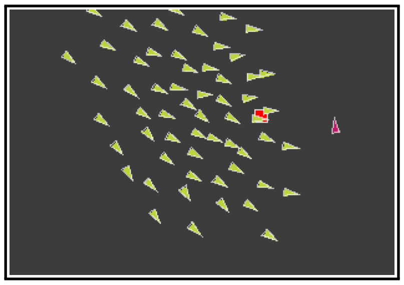


Figure 3: Swarm infiltration game: an attacker (red agent) must reach a target (red square) without being "caught", while a swarm flocks around the area (green agents).

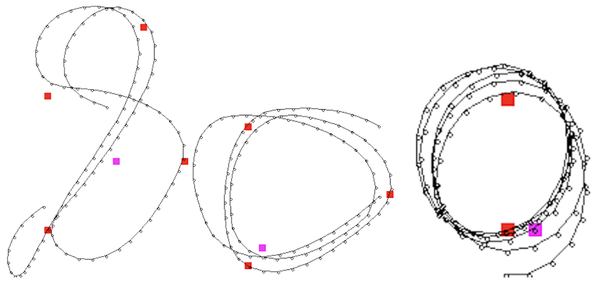
swarm of agents ("defenders") navigate in the target area following a set of waypoints. The swarm moves following the classical boids algorithm [12], with an extra attractive force towards waypoints. That is, we first consider three local rules, as in the classical algorithm: *cohesion*, *alignment*, and *separation*. *Cohesion* calculates an acceleration vector towards the average position of all agents within a radius r_c; *alignment* calculates an acceleration vector towards aligning the heading of the agent with the average heading of all agents within a radius r_a. Finally, *separation* calculates the aggregated acceleration vector towards the opposite direction of all agents within a radius r_s (that is, local repulsive forces), avoiding collisions. These repulsive forces are inversely proportional with the distance (i.e., the closer one member is to another, the higher the repulsive force). Each force vector is multiplied by a weight: λ_c, λ_a, λ_s, and the final acceleration is given by the weighted average of all these forces. Therefore, the classical algorithm has 6 different parameters. In addition, we add an acceleration force towards the next waypoint, of fixed magnitude m_w.

Another agent ("attacker") must reach the target without being touched by the defenders. The attacker must learn a model of the swarm of defenders, and use this model in order to plan its movement towards the target. In Figure 3 we show a screenshot of the infiltration game. Note that defenders may move over the target, and do not simply navigate around it, making the problem significantly hard. For the interested reader, we show videos of our proposed algorithms, and the algorithm we compare against (described later in Section 4.2.2), at https://youtu.be/xLAcUCOuPuQ.

4.1 Sampling Strategy, Default Policy, and Reward Function

As the reader may recall from Section 3.3, our on-line planning methodology needs a *sampling strategy* for deciding actions to be simulated, and a *default policy*, for simulating actions into the future. We also need to define an appropriate reward function for ϕ.

In our "infiltration" game, ϕ considers a set $\vec{\mathbf{V}}$ of acceleration vectors of fixed magnitude. Each $\vec{v} \in \vec{\mathbf{V}}$ is generated by randomly sampling x and y positions from the uniform distribution with range $[-1, 1]$, and then normalizing the resulting vector to the fixed magnitude. For each $\vec{v} \in \vec{\mathbf{V}}$, ϕ simulates a constant $\vec{v}$ acceleration (assuming velocities are saturated at a certain value) for all the h simulated time steps. I.e., our *default policy* repeatedly applies the same action. We define our reward function as: 1 if ϕ reaches the target in the current simulation (terminating the current simulation); the negative of the distance between ϕ and the target in the final simulated state; and $-\infty$ if a swarm member collides with ϕ (also terminating the current simulation). In *HAS* it is not possible to check if ϕ collides with a swarm member in a simulation. Therefore, we consider the $-\infty$ reward if ϕ collides with the predicted ellipse, also terminating the current simulation. We use $\gamma = 1$.



(a) "Easy" (b) "Medium" (c) "Hard"

Figure 4: Three different levels of difficulty, and the corresponding waypoints that we learn in *DSS*.

4.2 Results

4.2.1 Waypoint Learning. We first present our results in waypoint learning in *DSS*, since they clarify the three different difficulty settings that we consider ("easy", "medium", and "hard"). Each difficulty setting is defined by a different set of waypoints. Figure 4 shows the position of the target, the waypoints, and the estimated waypoint sets after cleaning ($\tilde{\mathbf{W}}$) for the "easy", "medium" and "hard" setting. Red squares represent the true waypoints followed by the defenders, while the pink square represents the target position. The circles represent the estimated waypoints, after cleaning the set. As we can see, "easy" mode leads to an "8"-shaped movement centered on the target, while "medium" and "hard" leads to a circular motion around the target. These are waypoints for the whole group, not for the individual agents. That is, depending on the number of agents, there will be agents over the target when following these waypoints (as we can see in Figure 3), making the attacker problem significantly hard.

Note that the estimated waypoints are not close to the true waypoints, but they still represent well the actual movement that is performed by the defenders. That is, the real motion of the defenders follows the 8-shaped pattern in "easy", and the circular patterns in "medium" and "hard" (instead of being, e.g., a triangle in "medium" and a straight line in "hard"), because of the motion dynamics. Therefore, the estimated waypoints still allow us to simulate their behavior. In order to evaluate that quantitatively, we run a scenario with 41 defenders, and we allow them to execute 10 laps following their true waypoints. We then count how many times they go over each estimated waypoints, and we find close to 98% in "easy", 89% in "medium", and 47% in "hard".

4.2.2 On-line learning and planning. We first compare our approaches against a *naive* algorithm, which only applies attraction and repulsion forces. That is, in *naive* the attacker has an attractive force that accelerates it towards the target, with no radius limitations and m_w magnitude. Additionally, the attacker suffers repulsive forces in the opposite direction of any defender within a radius of r_s, and with λ_s weight (inversely proportional to the distance, as before). The summation of all these forces pushes the attacker towards the target, while (almost) ensuring that there will be no collisions against any defenders. This approach is based on the classical potential field algorithm in mobile robotics [8]. Later we also comment how reinforcement learning (DQN [9], Duelling DQN [21]) performs.

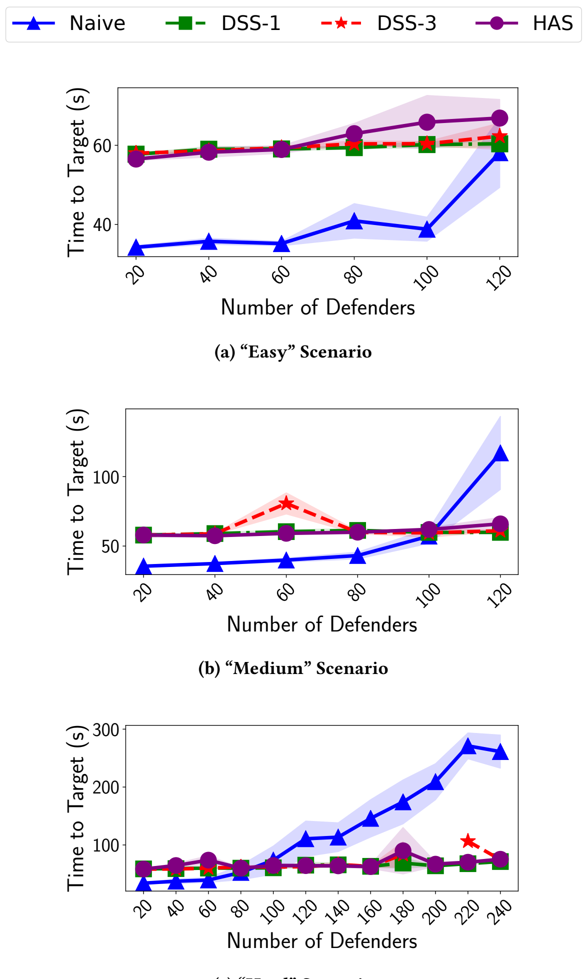
We ran experiments for a varying number of defenders and difficulty level. The initial position of the attacker is randomly chosen on the right hand side of the target, at a fixed initial distance of 3500 pixels. We repeat all experiments 30 times, and evaluate the average results. No knowledge is retained between executions (i.e., there is no pre-training, every execution starts from scratch). In all plots, the error bars show the confidence interval ($\rho = 0.01$), and when we say that a result is significantly better than another we mean with statistical significance considering $\rho \leq 0.01$.

We use the following parameters for parameter estimation: simulation horizon $e = 20$, number of simulations $n = 20$. For online planning: simulation horizon $h = 800$, number of simulations $|\vec{\mathbf{V}}| = 300$. Waypoint learning happens for $l = 1000$ iterations (which are considered when we evaluate our results). For *HAS* we use the χ^2 CDF to 99.9955% confidence level (very close 100%), leading to $c = \sqrt{20}$, ARIMA orders $o = (2, 1, 3)$ and $o' = (2, 0, 3)$, $t_\mu = 15\%$ and $t_\sigma = 40\%$. In the initial trials, we experimentally observed that these two ARIMA orders lead to good predictions.

We evaluate *DSS* when learning the radius of *cohesion*, *alignment* and *separation* (*DSS-3*), and when learning the radius of only *separation* (*DSS-1*). Hence, we consider the case of learning 3 parameters, or a single parameter. We use learning rate $\alpha = 1$ when learning the *separation* radius, and $\alpha = 15$ when learning the radius for *cohesion* and *alignment*. These learning rates are higher than usual because we noticed very small derivative values, and the number of learning iterations is highly limited by the real time constraints. For *DSS-3* we decay the learning rates at the rate 0.9, since we noticed that it leads to more stable executions when considering multiple parameters. The parameters that are not being learned are fixed to the correct values. We consider the following possible ranges for the parameters: *cohesion* and *alignment*: $[50, 90]$; *separation*: $[20, 60]$. Values are measured in *pixels*. Note that *HAS* does not have previous knowledge of any of the swarm algorithm parameters.

For the defenders algorithm, we fix the following parameters: r_c and $r_a = 70$; $r_s = 30$; $\lambda_c = 0.9$; $\lambda_a = 1.2$; $\lambda_s = 2.0$; $m_w = 0.04$. When running *naive*, the attacker considers $r_s = 30$; $\lambda_s = 2.0$; $m_w = 0.5$[1]. When running *DSS* and *HAS*, the simulated actions $\vec{v}$ have magnitude 0.1. For all agents and in all algorithms, velocities saturate at 1, and all distance metrics are in pixels. Additionally, attractive forces are applied after the saturation for all algorithms (leading to a final saturation of $1 + m_w$). Note that we put ourselves in a slight disadvantage, since *naive*'s $m_w > 0.1$.

We evaluate our results in terms of *time to reach the target*, and *success rate*, which is the proportion of times that the attacker



(a) "Easy" Scenario

(b) "Medium" Scenario

(c) "Hard" Scenario

Figure 5: Time to reach target in three different scenarios.

reaches the target without being caught by the defenders, within a time limit of 300s. Additionally, for *DSS-1* and *DSS-3*, we evaluate the normalised squared error of the parameters estimations, and in *HAS* the coverage level of the ellipses. The *time to reach the target* is only counted in successful executions (i.e., not being captured by the defenders). In order to better explore a large number of swarm members, we ran up to 240 agents in the "hard" scenario.

In Figure 5 we show the *time to reach the target* for all scenarios, and a varying number of defenders. When a result is not shown, it means that there was no successful execution for that particular algorithm in that particular scenario. As we can see, we are able to reach the target significantly faster than *naive* in the "medium" and "hard" scenarios (for number of defenders ≥ 120 in both cases). We also notice that *DSS-1*, *DSS-3*, and *HAS* have a very consistent *time to reach the target* around 60s across all scenarios and number of defenders (growing very slowly with the number of defenders), while *naive* quickly grows as the number of defenders and/or scenario difficulty level increases.

[1]We tried different parametrisations but did not see any significant differences.

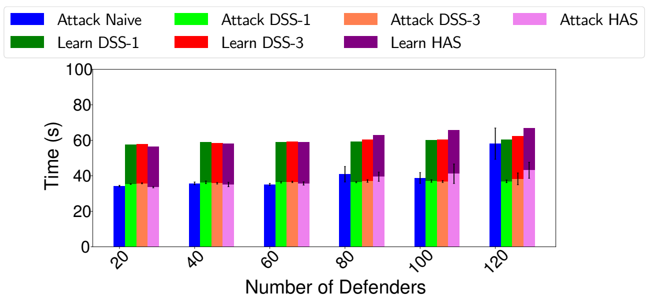


Figure 6: Proportion of time used in actual attack in the "easy" scenario, in comparison with *naive*.

In "easy", we are not faster than *naive* because of the time taken to estimate the waypoints $\tilde{\mathbf{W}}$. If we count only the time after waypoint learning (Figure 6), when the attacker is actually moving, we find that all algorithms are significantly better than *naive* with 120 defenders. Hence, our algorithms scale much better when the swarm size gets higher, and the impact grows with the difficulty level.

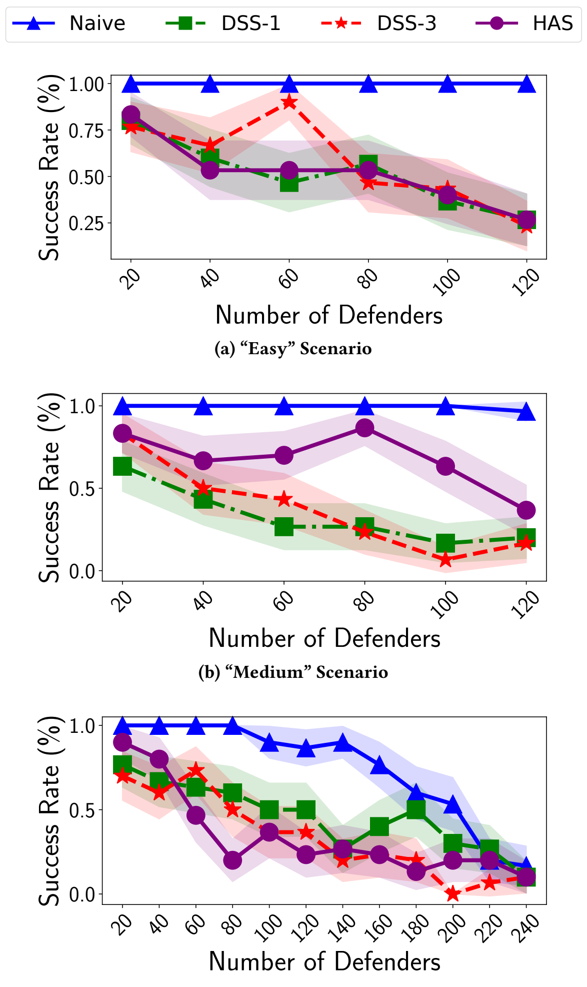
Regarding *success rate* (Figure 7), we find that *naive* is significantly better most of the time, but consistently drops in the "hard" scenario, and is not significantly different than *DSS-1* for $|\Omega| \geq 180$, and for all other algorithms for 220 defenders. Meanwhile, we are able to obtain a significantly better *time to target* in these scenarios.

It is interesting to note that *DSS* obtains higher *success rates* in "hard" than in "medium". This result seems to indicate that different scenarios may require different parametrisations of the algorithm. In the "medium" scenario, *HAS* obtains significantly better *success rates* than *DSS* for $|\Omega| \geq 60$ ($\rho \leq 0.09$ in the 60 case). On the other hand, *DSS-3* is significantly better than *HAS* with 60 agents in the "easy" scenario, and with 60 and 80 in "hard"; and *DSS-1* in "hard" with 120 and 180 agents. In other cases, however, the algorithms are not significantly different.

Furthermore, in Figure 8 (a) we show examples of the learning curves for *DSS-1* and *DSS-3*. We can see that *DSS-3* has much lower learning iterations, due to the real time constraints, and it converges quicker, since we apply a decreasing learning rate in that case. For *DSS-1*, we see that it constantly decreases. In Figure 8 (b) we show an example of the "coverage" results of *HAS*, in the same scenario. That is, we represent the percentage of agents that are within the observed and the predicted ellipses across time. As we can see, we are able to cover most of the swarm even when doing predictions, always covering more than 90% of the members in the observed ellipses, and at least 60% in the predicted ellipses.

Finally, we analyze the time taken for learning and planning at each iteration (Figure 9). *DSS* runs faster than *HAS* for a smaller number of defenders, and thus is able to update the acceleration vector more frequently, leading to a higher number of updating iterations. Even so, *HAS* is able to obtain similar results to *DSS*, and even outperforms it in some cases. For a larger number of defenders (160), however, *HAS* actually starts to run *faster* than *DSS*.

4.2.3 Comparison with Reinforcement Learning. We tested the performance of DQN [9] and Duelling DQN [21] in our setting, algorithms that were able to achieve super-human performance in Atari games. We use the original DQN architecture, hence the



(a) "Easy" Scenario

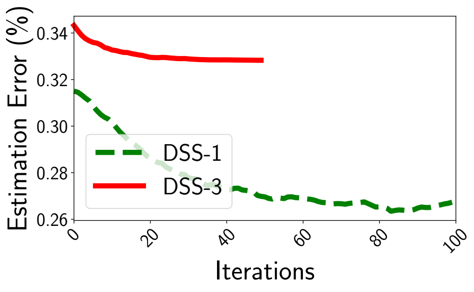
(b) "Medium" Scenario

(c) "Hard" Scenario

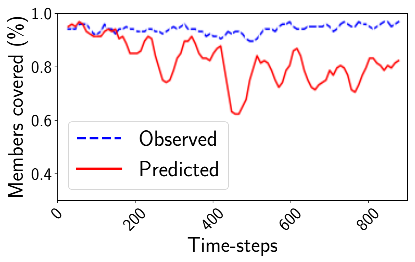
Figure 7: Success rates across three different scenarios.

4 most recent frames defines the current state. The agent uses ϵ-greedy approach, with ϵ decreasing from 1.0 to 0.1 over the course of 5000 steps; and $\gamma = 0.99$. Additionally, we simplified the problem for the DQN agents: (i) We defined a reduced action space with 10 discrete actions, each defining an acceleration vector of the same magnitude as used in our algorithms (0.1). These are uniformly distributed across the whole 2π angular range. (ii) We removed the real time constraints, and "paused" the swarm while the attacker is computing its next action. Therefore, DQN is handling an *easier* problem than the results shown in the previous section. Additionally, in all executions we kept the scenario fixed, and the attacker always started in the same position.

We first analyzed the *hard* scenario with 60 swarm members. We analyzed three variations of DQN: (i) DQN-1: We give a positive reward (200) when the attacker is able to reach the target, and a negative reward when there is a collision with a swarm member (−200). After 10 minutes the attacker receives a *timeout* reward (−200) and the game re-starts. (ii) DQN-2: We use the same rewards as before, but for each non-final state we give a reward $1/d$, where

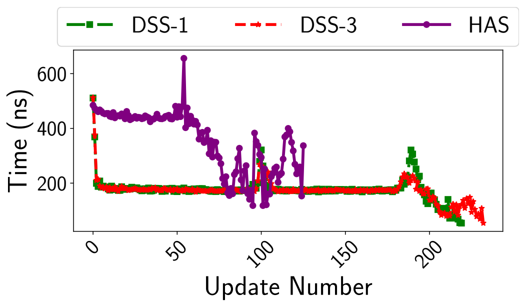


(a) Learning curves for *DSS-1* and *DSS-3*

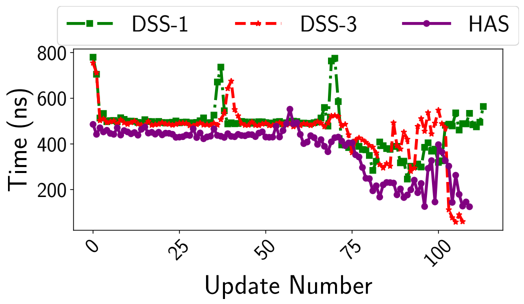


(b) Observed and predicted ellipses coverage

Figure 8: Example of learning curves and ellipses coverage, in "medium" scenario with 100 defenders.



(a) 60 members



(b) 160 members

Figure 9: Processing time for *DSS* and *HAS* algorithms, in the "hard" scenario.

d is the current distance between the attacker and the target, in order to help guide the agent towards the target. (iii) Duelling DQN: The same as (ii), but using the Duelling DQN algorithm instead.

After many days of training, however, the DQN agents were still unable to reach the target. DQN-1 ran up to 235 games, DQN-2 ran 848 games, and Duelling DQN ran 549 games. In DQN-1, 72% were timeouts, and the remaining 28% were collisions with defenders. In DQN-2, 77% and 23% were timeouts and collisions, respectively. Finally, in Duelling DQN, 73% were timeouts, and 27% were collisions. Hence, the agent was not even able to experience a successful game. In the same scenario, however, our algorithms obtain up to 75% success rate, and reach the target in around 55s, learning in a single game (i.e., no training, there is no knowledge carried across runs).

In order to investigate further, we also analyzed DQN-2 in the *easy* scenario, with 20 swarm members. Similarly, however, after 512 games we saw 77% timeouts, 23% collisions, and no victories. We can see, therefore, that this problem is quite challenging for reinforcement learning.

We observed in the resulting DQN-2 policies that the agent learned to get closer to the target, but to avoid the area where the swarm navigates. Therefore, it would not be able to reach the target. This shows a fundamental issue with reinforcement learning approaches: there is a very high probability of receiving the most negative reward when the agent is approaching its goal. Therefore, the agent could end up learning to avoid its goal completely.

5 CONCLUSION

We introduce the novel problem of learning a model of an unknown swarm in real-time, within a single execution, and using that model for decision-making. We present two novel algorithms: one that creates a hierarchical abstraction of the swarm, and another that simulates the individual swarm members in order to learn the swarm algorithm parameters. The first one does not require any knowledge of the swarm algorithm, but it assumes that they would move in a coherent, ellipse-like fashion. The second is free of this assumption, but needs knowledge of their algorithm, as it focuses on learning its parameters. We compare against local repulsive forces across three different scenarios, and we find that we are able to reach our goal significantly faster, at the cost of success rate in some cases. Additionally, we find that the hierarchical approach has significantly better success rate than directly simulating the swarm members in one of the scenarios, while in the others their performance is not significantly different. We also analyzed several variations of DQN in a simplified version of our problem, showing that it imposes great challenges for reinforcement learning.

For the interested reader, our source code is available at https://github.com/lsmcolab/dss-has.

Acknowledgements: This research was partially supported by EPSRC EP/R01860X/1, and AUSPIN. We would also like to thank the School of Computing and Communications, the Portsmouth Business School, Jó Ueyama, and Andrew Wood for their support. Additionally, we thank Caio Ferreira Bernardo for useful comments.

REFERENCES

[1] S. Albrecht and P. Stone. 2017. Reasoning about Hypothetical Agent Behaviours and their Parameters. In *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'17)*.

[2] S. V. Albrecht and P. Stone. 2018. Autonomous Agents Modelling Other Agents: A Comprehensive Survey and Open Problems. *Artificial Intelligence (AIJ)* 258 (2018), 66–95.

[3] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. 2015. *Time series analysis: forecasting and control*. John Wiley & Sons.

[4] K. Genter and P Stone. 2016. Ad Hoc Teamwork Behaviors for Influencing a Flock. *Acta Polytechnica* (2016).

[5] K. Genter, S. Zhang, and P Stone. 2015. Determining Placements of Influencing Agents in a Flock. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)*.

[6] Arnaud Glad, Olivier Simonin, Olivier Buffet, and François Charpillet. 2010. Influence of different execution models on patrolling ant behaviors: from agents to robots. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

[7] Nicholas Hoff, Robert Wood, and Radhika Nagpal. 2013. Distributed Colony-Level Algorithm Switching for Robot Swarm Foraging. In *Distributed Autonomous Robotic Systems: The 10th International Symposium*, Alcherio Martinoli, Francesco Mondada, Nikolaus Correll, Grégory Mermoud, Magnus Egerstedt, M. Ani Hsieh, Lynne E. Parker, and Kasper Støy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 417–430.

[8] O. Khatib. 1986. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research* 5, 1 (1986), 90–98.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level Control through Deep Reinforcement Learning. *Nature* 518 (2015).

[10] Rubens O. Moraes and Levi Lelis. 2018. Asymmetric Action Abstractions for Multi-Unit Control in Adversarial Real-Time Games. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

[11] Rubens O. Moraes, Julian Marino, Levi Lelis, and Mario Nascimento. 2018. Action Abstractions for Combinatorial Multi-Armed Bandit Tree Search. In *Proceedings of the AAAI Conference on AI and Interactive Digital Entertainment (AIIDE)*.

[12] Craig W. Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics (SIGGRAPH 87)*. ACM Press, 25–34.

[13] Joseph A. Rothermich, M. İhsan Ecemiş, and Paolo Gaudiano. 2005. Distributed Localization and Mapping with a Robotic Swarm. In *Swarm Robotics*, Erol Şahin and William M. Spears (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–69.

[14] Navyata Sanghvi, Sasanka Nagavalli, and Katia Sycara. 2017. Exploiting Robotic Swarm Characteristics for Adversarial Subversion in Coverage Tasks. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

[15] V. G. Santos and L. Chaimowicz. 2011. Hierarchical congestion control for robotic swarms. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[16] Vinicius Graciano Santos and Luiz Chaimowicz. 2014. Cohesion and segregation in swarm navigation. *Robotica* 32, 2 (2014), 209–223.

[17] Vincent Spruyt. 2014. How to draw a covariance error ellipse? *Computer vision for dummies* (2014). https://www.visiondummy.com/2014/04/draw-error-ellipse-representing-covariance-matrix/

[18] D. P. Stormont. 2005. Autonomous rescue robot swarms for first responders. In *Proceedings of the 2005 IEEE International Conference on Computational Intelligence for Homeland Security and Personal Safety (CIHSPS)*. 151–157.

[19] A. R. Tavares, S. Anbalagan, L. S. Marcolino, and L. Chaimowicz. 2018. Algorithms or Actions? A Study in Large-Scale Reinforcement Learning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*.

[20] A. Šošić, W. R. KhudaBukhsh, A. M. Zoubir, and H. Koeppl. 2017. Inverse Reinforcement Learning in Swarm Systems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

[21] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML'16)*. 1995–2003.