Decentralized Swarm Robotics: Discrete vs Continuous Shape Formation of Swarm Robotics

Varun Jhunjhunwalla, Shreyas Agarwal, Jonathan Portillo

***Abstract*— The problem of shape formation in decentralized swarms has been tackled in many ways. However, most methods must make a choice between providing well-planned paths or low runtime results. In this paper, we evaluate an algorithm by Wang and Rubenstein that provides low runtime results by sampling from discrete space, and develop a continuous version of this algorithm. We analyze theoretical safety guarantees of this algorithm and compare its performance to the discrete version using metrics such as runtime, total distance traveled by the swarm, and efficiency. We also implement an asynchronous decentralized swarm simulator to run experiments and verify results.**

I. INTRODUCTION

Swarm robotics is a field of robotics in which multiple robots work together to complete a task. Research in this field has several applications ranging from nanotechnology to defense. There are two main approaches in swarm robotics. The first is a centralized approach where all robots have access to global information and a centralized controller drives them to a goal state. The other type is decentralized control where each robot can only communicate with its neighbors but the swarm must still be able to coordinate itself. While all robots have global information and it is easier to control the swarm into particular patterns in the centralized approach, it is also computationally heavier than the decentralized one. In the decentralized approach, robots work with reduced access to information allowing for faster computations and faster maneuvers though it is harder to coordinate such an unorganized swarm.

One of the fundamental problems in swarm systems is shape formation. Wang and Rubenstein [1] propose an algorithm for the decentralized version of this problem, where robots can provably converge to a desired shape. This is appealing for both the speed of convergence and the fact that they have proven the algorithm to converge successfully, an assurance that is difficult to provide in the field of decentralized control. We further examine their claims and look for ways to extend their work for better efficiency.

We have two primary goals in this project. The first is to implement a simulator for asynchronous decentralized swarms which simulates communication as well as robot motion. The second is to extend Wang and Rubenstein's algorithm [1] to the continuous domain. We detail our work in this paper and also on our website[1].

A. Motivation

A large part of the motivation for this project came from sharing the same skepticism as Professor Sastry when we first read Wang and Rubinstein's paper [1]. It seemed too good to be true and we were curious to see if we could replicate their results. Many aspects of the paper seemed like they could be further explored, such the use of a discrete space and non-complex metrics. Furthermore, work by other groups as discussed in Section II largely shaped the direction of our project.

As explained in the introduction, we are interested in improving the decentralized control of swarms. We are specifically interested in exploring [1] and it serves as our primary motivation. We are also inspired by content covered in EECS 106B projects such as path planning with object avoidance, trajectory planning and intelligent goal picking. We made use of our insights from exploring these topics in lab throughout this project. We also made use of our previous work with Rapidly-Exploring Random Trees (RRTs) and goal selecting when working on the goal manager for our algorithm. We also used our understanding of optimization problems from class to evaluate the performance of the algorithm and used our understanding of controllers to build the asynchronous simulator.

B. Problem statement

In this paper, we implement an algorithm where a decentralized swarm of n identical robots, represented by set A, is controlled from their initial positions to an arbitrary target formation represented by Q, the set of goal coordinates, using collision-free and deadlock-free paths. For this paper we assume $|A| = |Q|$.

Each agent $a_i \in A$ is modeled as a circle with radius r. There are three main methods by which the agents operate: motion planner, new goal selector, and broadcaster.

The motion planner continuously checks for messages from the agent's neighbors at intervals of Δt. It then checks for a valid space to move to and moves the agent. We assume that each agent has the same clock frequency f_{clock} and velocity v. The new goal selector continuously sends and receives requests to swap goals with neighbors which helps prevent deadlock and ensures fast convergence. It uses a 2-way swap in which an agent acts as a client and another acts as a server. The client sends a swap request to the server and the server can either accept or deny the request.

The third method, broadcaster, enables the agents to communicate with each other. Each agent communicates its state with negligible latency and communication error to all robots within a communication range $R > 2\sqrt{2}r$ at a fixed rate f_{comm}. To ensure collision-free paths, we enforce

[1]Link to our Website

$\Delta t = 2/f_{comm}$. We define $p_{a_i}(t)$ as the position of a_i at time t. Similarly, $T_{a_i}(t)$ is the position assigned to a_i at time t.

Deadlock-free paths are defined as those where no two robots are assigned the same target position once all robots are guaranteed to be at their assigned position. This is formally defined below by Equation 1: $\exists t_{max} > 0$ s.t. $\forall t > t_{max},\ \forall a_i \in A, p_{a_i}(t) = T_{a_i}(t)$and $\forall a_i \neq a_j, T_{a_i}(t) \neq T_{a_j}(t)$

Collision free paths, as the name suggests, are those in which the robots never collide. This is formally defined below by Equation 2: $\forall t \geq 0$, for any two agents, $a_i \neq a_j, \|p_{a_i}(t) - p_{a_j}(t)\| \geq 2r$

II. Related Work

Reading about progress in the field of decentralized swarm robotics made us realize that there is potential to use what we learn in class about controllers and planning to build on recent swarm research. One inspiring way in which this is done is by [2], who takes advantage of the physical properties of water in their algorithm. They group robots together, treating them like water droplets coming together due to adhesion. This is a natural approach for swarm robotics as the agents on have access to information from the robots in their close proximity. This allows for agents to pass through narrow and complicated pathways and their movement is controlled by hydrodynamic models. This paper also gave us a better idea of how to decentralize our system without restricting our robots to specific environment. The downside of this paper was that it required heavy computation to create harmonic functions that drive the robot towards the goal and away from obstacles. This required solving complex Laplace equations to be able to converge.

We noticed a similar approach in a project called Swarmbots [10] which used aggregation to achieve shape formation. They first collect all robots in a circle, and then use a known trajectory to achieve the desired shape - similar to RRT using primitives. This algorithm presented a lot of problems when we actually ran it and the aggregation step would lead to collisions in most runs. Modifying thresholds only led to convergence for simple shapes like circles and the overall performance was unstable. The runtime was also high as the aggregation step took twice as long as the actual shape formation. While the implementation of [10] did help us understand how to create an asynchronous simulator, the methods used to form shapes showed weak performance that made us decide not to use aggregation. Other works [9][6] have also made use of similar RRT based algorithms but they also use complex heuristics such as shape functions and potentials.

Another project [3] similar to ours worked on a consensus algorithm for decentralized swarms where agents are connected in a network in which each robot is considered to be a probabilistic finite state machine (PFSM). We gained insight from their grouping technique which guarantees convergence for any network size and topology. We also understood how they use continuous probability mass functions to make decisions which gave us insight for the continuous version of our algorithm. The problem with this approach, however, is similar to the problem with Swarmbots [10]. Both require a time consuming aggregation and role assignment step before shape formation begins. The paper also assigns roles to each agent by assigning them a PMF for goal states after aggregation. We also realized that aggregation was not truly decentralized as clusters of connected robots were behaving like centralized swarms where computation was split between agents.

III. Methods

1) Implementation of the asynchronous decentralized swarm simulator: In this portion of this study, we attempted to recreate and simulate Wang and Rubenstein's work [1]. They ran their algorithm on a hundred robots called Coachbot V2.0 which communicated through an FTP server and TCP/IP connection to a base station. We did not have these resources so we had to use a simulator. We tried different simulators but most were for centralized control or they did not allow for simultaneous planning between robots. Thus, we instead developed our own simulator.

In our implementation, each robot is instantiated as an object of a Robot class which has instance methods, one for each of the three algorithms (new goal selector, motion planner, and broadcaster) as described by Wang and Rubenstein [1]. Each robot is run on a separate thread so that all agents can simultaneously run their algorithms and cooperate in actions such as goal swapping and new goal selecting in real time. Additionally, each of the three algorithms on every robot is run on three different threads. Thus, there are three threads per robot running at any given time.

There were additional assumptions made in Wang and Rubenstein's algorithm that each agent has the same clock frequency, each agent is able to constantly transmit messages to its neighbors in communication range at a fixed rate, each agent has the same movement speed v, and latency and communication error is negligible. To incorporate these in simulation, we ensured that every broadcast would be followed by a wait time of $1/f_{comm}$ by the broadcasting method, each robot had a standard speed, and latency and delays in communication were small compared to execution time. The execution time of the methods were in the range of 0.01 - 0.04 seconds indicating that latency was indeed negligible compared to execution rate.

2) Algorithm: The algorithm was largely based off of Wang and Rubenstein's algorithm with a few modifications to work in the context of multi-threading. The three methods (motion planner, new goal selector, and broadcaster) were modified as follows:

a) Motion Planner: The first part that needed to be changed was the receiving of messages by the motion planner. Since we did not have the hardware necessary for receiving broadcasts from neighboring robots, we developed an artificial broadcasting mechanism in which each agent a_i had its own instance variable msg_i containing the message that it would "broadcast" at the current time. Every agent would check its neighbors by iterating through the entire

set of robots and choosing the ones who were within a range R (the communication range) and store its message variable. Also, the algorithm required all messages from the past Δt seconds to be taken. Since, Δt was very small in comparison to agent message updates, we kept sampling the most recent message from all neighboring agents for Δt seconds. We kept the final set of messages and continued the algorithm. Additionally, $2/f_{comm}$ was a very small time interval compared to the latency in the context switching required by multi-threading. Thus, we empirically chose Δt to be $180/f_{comm}$ where $f_{comm} = 200$.

b) New Goal Selector: The new goal selector was altered so that goal swapping could occur safely without race conditions. This was particularly important in the 2-way swaps. Swap requests for each agent were kept in a list of requests stored as a variable for each agent. Since a server could have multiple requests, there needed to be a secure way for clients to add their request to the server's list without causing a race condition with another client trying to swap with the same server. Thus, two kinds of locks were implemented, a request lock (REQ) and an acknowledgement lock (ACK).

Each request list had an REQ and every agent had an ACK. Thus, if a client wanted to send a request to a server, it would need to obtain the corresponding REQ, add its request to the list, and release the REQ. The server would obtain its REQ and look through the list of requests for a client who it wants to swap goals with. If such a client was there, then the server would obtain the client's ACK, send the client its goal, and release the ACK. Otherwise, it would just ignore the requests (the request would be removed after a timeout interval by the client itself). Then the server would release the REQ after looking through its requests. This proved to be an easy and effective way of simulating 2-way requests between multiple clients and a server.

c) Broadcaster: To simulate broadcasting signals from a physical system at a fixed rate of f_{comm}, we set each agent's variable msg_i to the current state of the agent and then had the method sleep for $1/f_{comm}$ seconds.

d) Extension into Continuous Domain: With the basic algorithm established by Wang and Rubenstein simulated, we moved on to simulating the algorithm in the continuous domain. This required some more modification to allow the same convergence properties in the continuous domain.

The first modification was to choose an appropriate distance metric. While we used Manhattan distance in the discrete grid setup, we switched to the L2-norm distance metric in the continuous setup. This allowed for a better approximation of the distance between points.

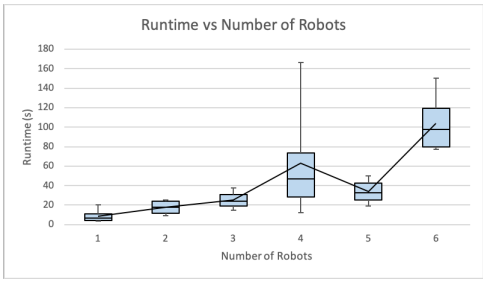
The second modification was to choose a different way of traversing through the space. We couldn't just have up, down, left, and right as candidate spots for the next step. Thus, we used a polar coordinate design for stepping through the space. Each robot would be able to move the same distance per step (same radial distance) but would have the choice of the angle it could move at. We found that the more choices of angles the robot had, the slower the algorithm would be. Thus, we empirically found a resolution of 16 candidate angles to be feasible in the runtime of the algorithm.

IV. Results

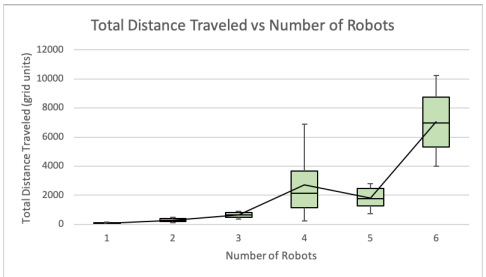
A. Experimental Findings

1) Number of agents: The first experiment we ran was to understand how the size of the swarm affects the performance of the algorithm. We used two metrics to judge the performance: total distance traveled by the robots TDT and time taken to converge. We were confident that time taken and TDT would increase as the number of agents increased since more agents would lead to more path-planning, goal assignment and overall computation. However, we noticed an interesting result in [1] where the algorithm's performance suddenly improves (i.e. the total time and TDT decreases) as the number of agents is increased. Specifically, the performance of the algorithm for 196 agents is far better than the performance for 25 to 169 and 225 to 529 agents.

We performed the same experiment with our implementation, averaging results over 6 trials per swarm size. The results are shown in Figure 1. As expected, we see an overall increase in the runtime and distance traveled as the number of robots increases. We also see the previously mentioned improvement when using five agents. On visualizing these trials on our simulator we noticed some behaviors that might be causing this anomaly. When the number of agents is too high, the number of agents sharing the same goal is larger, causing a large number of 'wait-flags', leading to increased runtime. There is also excess travel when the robots try to swap goals and form clusters of cyclic 'traffic graphs' which are further discussed in Section IV.C. When the number of agents is too small, the performance is good but two or three robots can hardly be called a swarm. The strange behavior of four agents showing a worse performance than five can be explained through certain behaviors identified in the simulations as well. The four agent case specifically demonstrates a strange behavior where two or three agents form a cluster in adjacent squares and alternate between swapping goals and moving back and forth. This behavior accounts for the worse performance, but can actually be controlled by decreasing the probability of swapping goals in the goal manager. Further experimentation is needed to



(a) Convergence time



(b) Total distance traveled

Fig. 1: Results averaged over 36 trials of the discrete version of the algorithm run on our simulator

understand the extent to which this parameter can affect performance.

2) Continuous vs Discrete Experimentation: We compared our discrete space implementation to our continuous space modification. We ran the discrete version for runtimes of 10, 20, and 30 seconds each and recorded TDT, initial distance (IDT), distance to convergence (DTC), percent of initial distance completed (PDC), and efficiency (EFF). We conducted this experiment 20 times and present here the average of the 20 trials. For the continuous case we tested runtimes of 50, 100, and 150 seconds since the continuous version took a longer time to run. We recorded the same data and conducted the experiment 10 times with the average being presented here. IDT is the sum of the distances between each robot and the nearest available goal state. DTC is the same as initial distance except after the time allotted for the simulation has elapsed. PDC completed is $1 - DTC/IDT$ expressed as a percentage. This gives a measure for how close the system is to convergence compared to the starting configuration. This gives a good metric as we are able to compare states that start close to convergence as well as those that start far from convergence on equal footing. EFF is $IDT/(TDT+DTC)$. This allows us to compare how the estimated total distance traveled ($TDT + DTC$) compares to the initial distance of the starting configuration.

As can be seen in Table 1, the TDT of the discrete setting is higher than TDT of the continuous setting (up to 5 times more). We can also see the differences in PDC and EFF between the two settings. PDC for discrete is a lot higher than PDC for continuous. This corresponds to the fact that more computation must be done in the continuous setting causing the runtime to be a lot larger. However, we can see that as runtime increases, EFF of the discrete setting drops below that of the continuous setting. With larger runtime, the discrete setting requires agents to explore the grid more and take exaggerated paths around each other. However, in the continuous case, the EFF remains approximately the same and is higher than the discrete setting. However, this is comparing discrete and continuous settings in different cases where the continuous setting is given a lot more time to converge. Thus, we conducted another experiment to test discrete and continuous settings at the same level.

Discrete Setting					
Time Duration (s)	TDT	IDT	DTC	PDC	EFF
10	305	580.45	121	78.70	1.44
20	842	549.55	61	88.96	0.63
30	1303	591.65	11	98.19	0.52
Continuous Setting					
Time Duration (s)	TDT	IDT	DTC	PDC	EFF
50	167.38	187.38	118.19	37.15	0.67
100	195.43	180.36	86.61	50.11	0.66
150	228.09	183.63	78.18	56.53	0.65

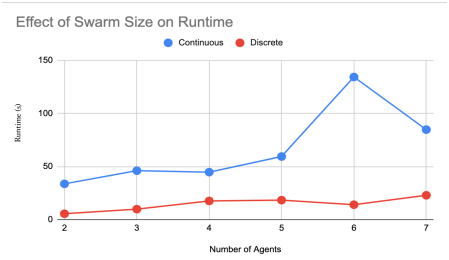
TABLE I: Results for Discrete and Continuous Settings. Average values over 20 and 10 runs are shown.

3) Continuous vs Discrete Analysis: We ran experiments to compare the performance of our continuous space planner with that of the discrete planner where both settings are allowed a maximum runtime of 150 seconds. Figure 2 compares these algorithms with data averaged over 35 trials. Figure 2(a) shows a familiar results, with the red line following a similar trend as the previously discussed Figure 1(a). The blue shows the biggest con of using the continuous version, high running time. As discussed before, this is expected since the continuous version is sampling an exponentially larger number of goals and potential paths than the discrete version. It also faces more behavioral issues such a deadlocks due to an incomplete characterization of the L2-norm distance metric.

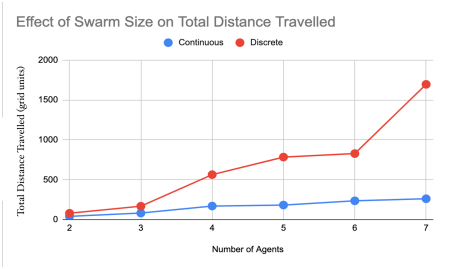
Figure 2(b), on the other hand, shows the benefits of using the continuous algorithm. The total distance traveled by the swarm before convergence is much lower in the continuous case as the swarm isn't restricted to Manhattan paths. While this does increase the runtime, it finds a more optimal path to the goals than [1]. The rate of increase in distance traveled with the swarm size is also worth noting. The discrete swarms face more complexity as the size increases since the number of possible collision paths is comparable to the

total number of paths. This problem is not prominent in the continuous case as there are infinite possible paths.

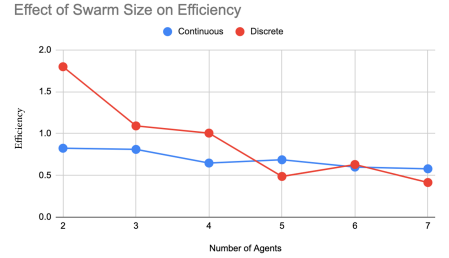
Figure 3(c) shows the effect of swarm size on efficiency. As swarm size increases, the efficiency drops for both continuous and discrete cases drops. This is because with more robots, the agents swap goals more often and tend to explore more before convergence. Thus, $TDT + DTC$ tends to increase with larger swarm size. However, around a swarm size of 5 to 7 robots, the continuous case has an efficiency equaling or exceeding that of the discrete case. The time saved in moving directly to the goal becomes more useful with a larger swarm size.



(a) Convergence time



(b) Total distance traveled



(c) Efficiency

Fig. 2: Results averaged over 35 trials of the discrete and continuous versions of the algorithm run on our simulator

B. Safety Guarantees

In this section we will show that the collision-free and deadlock-free guarantees hold in the continuous using similar arguments as [1]. Let $W_{a_i}(t)$ be the waypoint that's currently assigned to a_i. Also define $\Delta t = \frac{2}{f_{comm}}$. Robots are assigned waypoints which are represented by $W_{a_i}(t)$.

For this proof, let's begin by assuming that all agents start at unique positions with unique initial waypoints. We will prove that when some a_i is assigned a new waypoint at time t^*, it will be assigned a waypoint that no other agent is currently occupying or moving towards by contradiction. From the pseudocode we can show that the following conditions must be met in order to update the waypoint of an agent a_i.

- Condition 1: a_i does not receive a wait-flag message between $t^* - \Delta t$ and t^*.
- Condition 2: For all $t \in (\Delta t - t^*, t^*]$,$W_{a_i}(t) = W^0$. This is because the agent must wait for at least Δt time before the waypoint is updated.

1) Deadlock-free Path Guarantee: First we will prove that the paths are deadlock-free as stated in Equation 1. Assume some robot a_j occupies waypoint W^1 at t^*, when a_i is assigned a new $W_{a_i}(t^*) = W^1$ from original waypoint W^0. WLOG, we assume a_j has higher priority than a_i. We know that a_j was assigned W^1 at some time t^{1j}. We can analyze this situation using two cases.

- Case 1: $t^{1j} < t^* - \frac{\Delta t}{2}$ This situation implies that a_i and a_j share the same waypoint for $t \in [t^* - \frac{\Delta t}{2}, t^*]$. However, by Condition 1, this is not possible.
- Case 2: $t^{1j} \in [t^* - \frac{\Delta t}{2}, t^*]$ This means that a_j was assigned some other waypoint before t by Condition 2 but both a_i and a_j decided to choose the same next waypoint in the interval $[t^* - \Delta t, t^* - \frac{\Delta t}{2}]$. If they are both assigned the same waypoint after this time, they have up to $\frac{\Delta t}{2}$ time to let each other know. Condition 1 will therefore prevent this update from happening since $\frac{\Delta t}{2} = \frac{1}{f_{comm}}$.

Therefore, we have shown by induction that Equation 1 will remain satisfied if initial waypoint assignments are unique.

2) Collision-free Path Guarantee: We can prove the collision-free paths guarantee in a similar way. Let's assume a_i and a_j are moving towards each other in opposite directions at time t^*. At the some $t < t*$, assume $W_{a_i}(t) = W^0$ and $W_{a_j}(t) = W^1$. Since the agents move towards each other at time t^*, we know that $W_{a_i}(t^*) = W^1$ and $W_{a_j}(t^*) = W^0$. This situation can also be analyzed using two cases.

- Case 1: The time $t^u_{a_i} < t$ at which the agents decide to update their waypoints is different. This case is the same as the previously proven deadlock-free guarantee and can never exist.
- Case 2: The time at which the new waypoints are determined is the same. Therefore, the robots share a future waypoint for up to $\frac{\Delta t}{2}$ time, which is enough time for Condition 1 to prevent or delay a waypoint update.

Therefore, there cannot be any collisions on these paths.

C. Convergence

Our implementation of the algorithm in the discrete case guarantee almost sure convergence. The proof from [1] is quite involved and exhaustively analyzes multiple cases to prove the claims being made. Here is a summary.

First we define two loss functions $J_1(t)$ and $J_2(t)$ that reach 0 when the agents achieve the goal state as follows:

- $J_1(t) = \sum_{i=1}^{|A|} d_i(t) =$ the total distance of each agent to its assigned goal position at time t. Here $d_i(t)$ is the Manhattan distance from the agents position to the goal.
- $J_2(t) = |Q| - \sum_{i=1}^{|Q|} 1_i(t) =$ number of goals that no agent is moving to t. Here, $1_i(t)$ is an indicator variable for the event that some agent's target position at time t is goal position i.

It follows from the definition of these loss functions that the algorithm converges if both of these loss functions converge. To show almost sure convergence, we must prove that the loss functions are: (1) Bounded and non-negative, (2) Non-increasing and (3) Strictly decreasing with non-zero probability for all values except $J = 0$.

1) Bounded and non-negative: The loss functions are non-negative by definition. They can also be bounded by considering the fact that agents move greedily at each step. This implies that $J_2 \leq |A|-1$ since at least 1 goal is assigned to the swarm. The total distance from goals also decreases due to this greedy approach, implying that $J_1 \leq k|A|$ where k is the maximum distance between any goal state and any initial position on an agent.

2) Non-increasing: As explained above, J_2 is always decreasing as at least one agent moves towards its target. Once J_2 is 0, each agent is assigned a unique goal state so J_1 can only decrease as the agents move towards it.

3) Strictly decreasing in probability: This part of the proof exhaustively looks at cases to prove the claim. Please refer to Lemma IV-B.4 from [1] for a more detailed explanation. We can form a 'traffic graph' at some t with agents represented by vertices and edges between nodes i and j if a_i wants to move to a point occupied by a_j. Under this formulation, all nodes with zero out-degree are called "head nodes" as their path is not blocked. Consider these cases:

- Case 1: If the traffic graph is acylic and there exists a head agent a_i that hasn't reached it's goal. By definition, this agent has a free path and can reduce J_1 by 1.
- Case 2: If the graph is cyclic. It can be shown by contradiction, that agents will always be able to swap goals in the opposite direction of the cycle without the need to move with non-zero probability.
- Case 3: The graph is acyclic and all head agents have reached a goal. This case can be reduced to one of the above cases by swapping goals with a head agent.

This proves almost sure convergence for our algorithm in the discrete case. Note that parts of this proof, such as the definition of J_1, rely on the assumption that we're only moving along the edges of our grid-world. Therefore, we cannot make this guarantee for the continuous case. In the Section IV.B, we will discuss possible ways in which we can modify the algorithm to achieve convergence in the continuous case.

V. CONCLUSIONS AND FUTURE WORKS

A. Conclusions

This paper aims to explore and extend recent decentralized swarm research. We specifically focus on an asynchronous decentralized shape formation algorithm by Wang and Rubinstein [1]. We implement an asynchronous decentralized swarm simulator and verify the correctness of the algorithm in [1]. We also extend this algorithm to continuous space and compare the two algorithms. The discrete version of the algorithm is able to guarantee convergence with a relatively low runtime. The continuous version cannot guarantee convergence and runs rather slowly but is able to plan efficient paths. These are important trade-offs to consider when choosing between the two versions of the algorithm. The discrete version would work best for tasks where the agents have lower computational power or a small runtime matters more than efficiency. The continuous case, on the other hand, optimizes for agent movement and can be useful if we have battery or fuel constraints.

B. Future Work

The next step for this project would involve integrating more sophisticated ideas from the works described in Section 2. The continuous algorithm can be improved if given a better distance metric for goal swapping. This is because the guarantees provided by the euclidean metric don't exist in the continuous case since the triangle inequality doesn't hold. Ways to do this include sampling paths with noise or creating jerks in the environment if there has been no motion for some number frames. Both of these methods can help reorient our system to prevent deadlock and significantly reduce runtime. We also hope to extend the theoretical results to the continuous case and determine criteria for convergence. Lastly, we're interested in making use of shape functions and their first order information to more intelligently plan paths. However, we'd like to do this without complete aggregation and build on the results of [5] and [6].

REFERENCES

[1] H. Wang and M. Rubenstein, "Shape formation in homogeneous swarms using local task swapping," IEEE Transactions on Robotics, pp. 1-16, 2020

[2] C. A Pimenta L A, S. Pereira G, Kumar V, Chaimowicz L M, Bosque M C, Mesquita R. and Michael N. Swarm Coordination Based On Smoothed Particle Hydrodynamics Technique. [Accessed 10 April 2013].

[3] Liu, Y., Lee, K. Probabilistic consensus decision making algorithm for artificial swarm of primitive robots. SN Appl. Sci. 2, 95 (2020). https://doi.org/10.1007/s42452-019-1845-x

[4] V. J. Lumelsky, Continous Robot Motion Planning In Unknown Environment, pp. 339-358. Boston, MA:Springer US, 1986

[5] G. Li, D. St-Onge, C. Pinciroli, A. Gasparri, E. Garone, and G. Beltrame, "Decentralized progressive shape formation with robot swarms," Auton. Robots, vol. 43, no. 6,pp. 1505-1521, 2019.

[6] Hsieh, Mong-Ying A., Vijay Kumar and Luiz Chaimowicz. (2008). Decentralized controllers for shape generation with robotic swarms. Robotica. Vol 26(5). p.691-701.

[7] P. Ghassemi and S. Chowdhury, "Decentralized informative path planning with exploration-exploitation balance for swarm robotic search," ArXiv, vol

[8] I. Navarro and F. Matia, "An introduction to swarm robotics," ISRN Robotics, vol. 2013, 09 2012.

[9] Joshi, P., Leclerc, J., Bao, D. and T. Becker, A., 2020. Motion-Planning Using Rrts For A Swarm Of Robots Controlled By Global Inputs. [ebook] Available at: ¡https://swarmcontrol.ece.uh.edu/wp-content/papercite-data/pdf/8842916.pdf¿ [Accessed 15 May 2020].

[10] R. Vanarse, "rmvanarse swarmbots," GitHub,09-May-2020.[Online].Available: https://github.com/rmvanarse/swarmbots?fbclid=IwAR0xSmb9N2jsxITejFyZieEP7QGcYKA-M2LHv-8DBksL5vBEMEHRr9OK3ag. [Accessed: 16-May-2020].