General Coating of Arbitrary Objects Using Robot Swarms

Ahmad Reza Cheraghi
Technology of Social Networks
Heinrich Heine University
Düsseldorf, Germany
ahmad.cheraghi@hhu.de
https://tsn.hhu.de

Gorden Wunderlich
Technology of Social Networks
Heinrich Heine University
Düsseldorf, Germany
gorden.wunderlich@hhu.de
https://tsn.hhu.de

Kalman Graffi
Honda Research Institute Europe GmbH
Offenbach am Main, Germany
kalman.graffi@honda-ri.de
https://www.honda-ri.de

***Abstract*—(Nano) robot swarms promise new applications in the field of health and entertainment. One of the tasks such a swarm could solve is coating, in which the robots in the swarm locate themselves as close to an object of arbitrary shape as possible. In this paper, we present a swarm coating algorithm for simple robots using only local communication and decentralized coordination only. The robots calculate distances to the coating target object for themselves and all adjacent spaces around them, trying then to move to a lower distance than they are currently at. Through simulations, we show that our approach is able to efficiently coat various object forms with a 100% success rate, even objects with cave-like structures are coated correctly.**

***Keywords*—Robot, Swarms, Coating, Caging, Boundary, Coverage, Scanning**

I. Introduction

Swarm robotics is an emerging field of research. The individual entities of a swarm work hand in hand together to fulfill a task that would be impossible to solve by only one of them. In [1], swarm robotics is defined as "using a fully distributed control paradigm and relatively simple robots to achieve coordinated behavior at the group level." It follows a self-organizing approach, in which microscopic local decisions of individual robots, based on locally observed information, emerge to a collective or macroscopic behavior.

To create a swarm algorithm, we purely rely on local coordination between the robots. The robots are equipped with a simple algorithm, which operates only on local information, and creates a complex system only by means of communication. The same paper gives us three important features of a swarm robotic system: robustness, flexibility and scalability. A system can be called **robust** if the failure of individuals does not impede the operation as a whole. **Flexibility** is the capacity to handle a multitude of tasks in many different kinds of environments, with coating being only one task. And **scalability** is defined as the ability to handle many different amounts of robots. With our algorithm we aim to address all three attributes.

Many applications of swarm robotics require the robots to first move into specific positions, often directly around the object. Whether to protect, isolate, scan, shape or transport it or simply to make it easier to spot for larger entities, coating is essential to ensure a smooth procedure. Many applications for this technology can be envisioned, whether as part of a transportation system or as a coating to isolate damaged cells or viruses in medical patients. Even as a standalone technology the applications are numerous. Depending on the size of robots deployed, it could be used to reinforce walls or form a thin protection layer around objects to protect them from chemical or biological influence. Using a virtual environment and virtual objects as coating targets for the robots, even completely new objects could be constructed. In this paper we present a novel coating algorithm for swarm robots that can coat any arbitrary shaped object. Examples for valid coatings are presented in Figure 1 depicting robots/particles and objects/tiles.

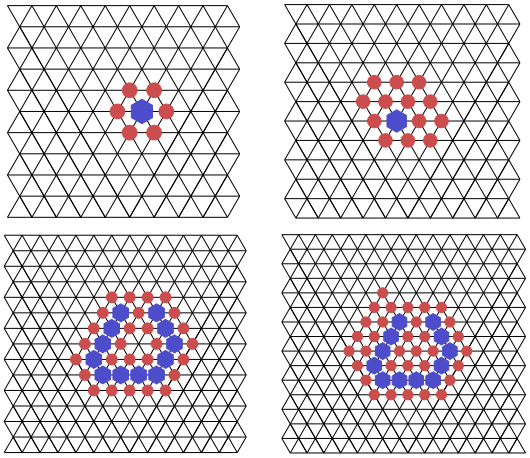


Fig. 1: Valid coating examples. Red: *Particles*, blue: *Tiles*

At first, in the Sections II and III, the coating problem and the needed definitions are proposed. This allows to discuss related work in Section IV. Section V presents the methodology of our proposed coating algorithm and how the algorithm is working. Last in Section VII, an thorough evaluation with different kinds of shaped objects is presented. Finally, the paper ends with a conclusion and a future work.

II. Problem Statement

Our goal is to create an algorithm to solve the universal coating problem for hexagonal 2D structures, which is similar to the anonymous multi agent path finding problem, but the

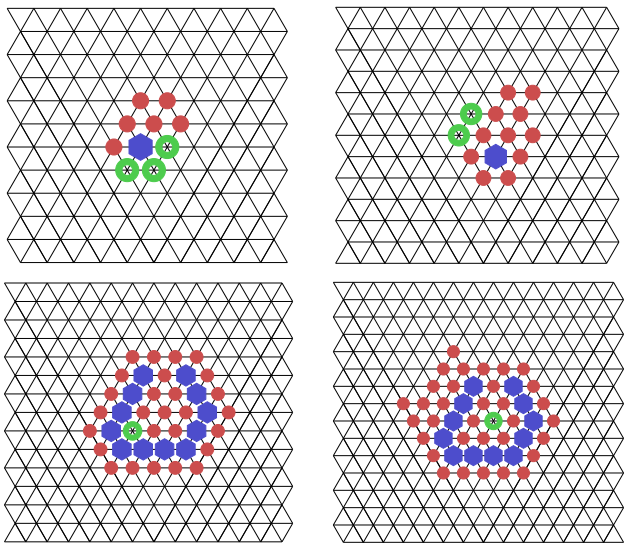


Fig. 2: Invalid coating examples. Red: *Particle* blue: *Tiles* green: free *Location* that should be occupied

targets and the environment are unknown to the agents and the number of targets and agents is variable even during a single simulation.
So, given a set of randomly distributed robots or *Particles* the goal is to distribute these *Particles* as closely and evenly around an object shape or *Tile Island* as possible as shown in the examples in Figure 1. Figure 2 on the other hand, shows some examples of unfinished states during the coating algorithm, where we can see some *Particles* in higher *Layers* while there are still free *Locations* in lower *Layers*.
In general, a coating algorithm is valid if the condition

$$max(Own_Distance(p)_{\forall\ p \in P}) \leq min(Own_Distance(fl)_{\forall\ fl \in FL}) \quad (1)$$

is satisfied. In other words, the maximum distance for all *Particles* to the nearest *Tile* must alway be smaller or eqaul to the minimum distance of all *Locations* to the neares *Tile*.

The coating problem can be split up into different smaller problems, that each *Particle* has to solve:

- Finding the *Tile Island*.
- Approximating its *Distance* to the next *Tile* and likewise calculating the *Distances* for all neighboring *Locations*.
- Efficient propagation of collected information.
- Finding free *Locations* with low *Distances* to reduce the *Particles* own *Distance* or to allow other *Particles* to reduce their *Distance*.

After describing the problem in this Section, we introduce a terminology that is used throughout this paper.

III. Definitions

In this section, the necessary definitions for the coating algorithm are described that are used throughout this paper.

Matters, i.e. the elements in the world, are divided into two types, the first being passive *Matters* called **Tiles**. *Tiles* in the Swarm-Sim simulator model objects that cannot act themselves, like obstacles or other inanimate objects. The set of all *Tiles* is called T. Like all *Matters* a *Tile* t has a state st_t containing only the *Location* $\lambda(t)$ it occupies, therefor $st_t = \lambda(t)$. A *Tile* has no further memory MM_t or capabilities.

The second kind of *Matter* is the active *Matter* called **Particle**. In contrast to *Tiles* a *Particle* can act and move on its own, such as a robot. The state st_p of a *Particle* p contains its memory and the *Location* it occupies:

$$st_p = (MM_p, \lambda(p))$$

Particles have a predefined set of possible actions. They can move, write into the memory of other *Particles*, check if their *Neighborhood* contains any *Matters* and, if so, what kind of *Matter* they are. *Particles* can also access their own memory.
Moving: A *Particle* p can move from its current *Location* $l = \lambda(p)$ to any *Location* in its *Neighborhood* $l_t \in N_l$, given that the *Location* is not already occupied, so $st_{l_t} = (N_{l_t}, 0)$.
Writing: As only *Particles* have a memory, *Particles* can write into each others memory. This is used to implement communication between *Particles* enabling coordinated actions. Each *Particle* can write into the memory of itself and of any *Particle* which occupies a *Location* in its *Neighborhood*.
Scanning: A *Particle* can scan its direct *Neighborhood* to identify the presence and type of *Matters* in its *Neighborhood*. This is important as a *Particle* has to know that there are other *Particles* in range to send information to and has to check whether the target *Location* of a move operation is already occupied or not. *Matters* can store and to some extent also exchange information. For this they use their memory MM. Each memory has an infinite number of slots $MS = \mathbb{N}_0$ and each slot can hold a set of information $I = \{i \mid i \in \Sigma^*\}$ for any alphabet Σ. So the memory of a *Matter* m is defined as:

$$MM_m = \{(ms, i) \mid ms \in MS \wedge i \in I\}$$

The *Locations* and *Matters* are combined in the **World**. Thus, a *World* W is the tuple of a set of *Matters* M and a set of *Locations* L

$$W = (M, L)$$

so that any matter has location, $m \in M \Rightarrow \lambda(m) \in L$. As *Matters* have to occupy a *Location*, the *Locations* dictate the movement and action range of these *Matters*. A *Location* can hold only one *Matter* at maximum, so the we define the state st_l of a *Location* l as its *Neighborhood* N_l and a value $oc_l \in \{0, 1\}$ indicating if the *Location* is occupied or not

$$st_l = (N_l, oc_l)$$

Any *Locations* is connected with other *Locations* through *Links*. In our work, we use an infinite regular hexagonal grid for the simulator, which means that each *Location* is connected to exactly six others *Locations* through *Links*, so that each of those *Locations* is in turn connected to exactly two of the other five. So for all *Locations* $l \in L$ the set of all *Links* LI contains the *Links* $LI_l = \{\{l, l_i\} \mid l_i \neq l \wedge i \in \{1..6\}\}$ related to the specific location l, so that for each pair of sets LI_{l_a}, LI_{l_b}: $\{l_a, l_b\} \in LI_{l_a} \cap LI_{l_b} \Leftrightarrow$
$\exists\ l_i, l_j \in L : \{l_a, l_i\}, \{l_a, l_j\} \in LI_{l_a} \wedge \{l_b, l_i\}, \{l_b, l_j\} \in LI_{l_b}$.

The set LI is then be defined as

$$LI = \bigcup_{l \in L} LI_l$$

Each *Location* also has coordinates, which are mainly used for rendering and limiting the simulation area, in the form of euclidean $(x, y) \in \mathbb{R}^2$ pairs. So, the grid graph G, which is the visible simulation is defined as a tuple of *Locations* and *Links* between those *Locations*:

$$G = (L, LI)$$

So, for each *Location* l the set LI_l is called its **Neighborhood** N_l and all $l_b \in N_l$ are called *Neighbors* of l. All *Links* N_l of a *Location* l are also ordered in relation to l, as each *Location* l has a set of **Directions** DIR_l, with

$$DIR_l = \{dir \mid 1 \leq dir \leq 6\}$$

and a function assigning

$$\wp_l : DIR_l \to LI$$
$$\{\wp(dir) \mid dir \in DIR_l\} = N_l$$

The *Neighborhood* N_m of a *Matter* m is then the *Neighborhood* of the *Location* it occupies, so $N_m = N_l$, with $l = \lambda(m)$.

A **Tile Island** TI is a set of *Tiles* so that the set of locations of the *Tiles*, $LTI = \{\lambda(t) \mid t \in TI\}$, is connected in the *Location* graph, i.e., there exists a path from each $l_1 \in LTI$ to each other $l_2 \in LTI \setminus l_1$ using only *Links* $li = (l_a, l_b)$ between *Locations* $l_a, l_b \in LTI$. Aside from a single *Tile Island* there will be no *Tiles* in any of our simulations.

The function **Own_Distance** calculates the *Distance* of a *Location* $l \in L$ to the closest *Tile* of the *Tile Island*.

$$Own_Distance : L \to \mathbb{N}_0$$
$$Own_Distance(1) = \min_{t \in TI} D(l, \lambda(t))$$

Own_Distance is also defined for *Particles* and *Tiles*. For *Particles* $p \in P$, we have $Own_Distance(p) = Own_Distance(\lambda(p))$ and for *Tiles* $t \in T$ $Own_Distance(t) = 0$. When we talk about the *Distance* of a *Particle* p or *Location* l without specifying a reference point, we mean Own_Distance(p) or Own_Distance(l) , respectively.

We can now define a **Layer** $L_i \subseteq P$, so that $p_1, p_2 \in L_i \Rightarrow Own_Distance(p_1) = Own_Distance(p_2) = i$, i.e. a set of particles of the same distance to the *Tile Island*.

The aim of the coating is to fill all layers around the *Tile Island*, starting with the layers of smallest distance and to not leave any position in a layer unoccupied when starting the next further distant layer. In the following, we discuss related work and present then in Section V our approach.

IV. Related Work

In the area of swarm robotics the problem of object coating has been touched several times, but it is rarely studied as an isolated problem, as it is often a part of a bigger problem. Thus, a detailed analysis of this case is desired. Coating is addressed partially in collective transport research, such as [2], where a group of robots that are each too small to carry a large object have to cooperate and organize themselves around an object shape to maneuver it in different directions. A full and perfect coating, as envisioned here is not needed. Target surrounding and boundary coverage are similar problems as object coating, but unlike object coating, target surrounding, as presented in [3], deals with fewer larger robots, the swarm effect is limited. The problem of moving targets while boundary coverage, as in [4] and [5], is about coating the inner boundary of an enclosed space. It does not consider arbitrary object shapes, as we do in this paper. The article "Universal coating for programmable matter" [6] is dealing with a similar problem, but states different assumptions and goals. In contrast to their work, we use a less specific robot model, as our robots to not need to be able to extend and retract their bodies. We assume limited, close range communication capabilities, implemented through reading each others memory. Also, while their algorithm ignores small cracks in the objects surface, our solution coats the whole object surface as evenly as possible independent of the surface layout. Our solution is more robust to robot failure as our robots do not need to be connected and work completely independent without the need of a leader robot. Outside the area of swarm robotics, a similar problem named anonymous multi-agent path-finding or sometimes goal-invariant multi-agent path-finding is studied, which can be solved in polynomial time [7]. There are multiple articles generalizing this problem [8] or mapping it to real world problems [9] and some that propose different algorithms for solving the problem ([10], [11], [12], [13]). Though the basic idea of the problem is the same, multi-agent path-finding requires the target locations and all possible paths to be known and, as it is not a swarm algorithm, is solved by a central computation unit, which sends the solution to all agents. Thus, the approach is highly centralized, which limits its application.

V. General Coating Algorithm

After defining the problem, we now present our fully decentralized and robust swarm algorithm for coating objects. The general idea of the algorithm is to find the *Particle* p_max with the highest *Own_Distance*, as well as a free *Location* with a lower Own_Distance , so that we can then move a *Particle* to that *Location*. This is repeated until the *Particle* p_max has a lower *Own_Distance* than any free *Location*, so that the total distance of the *Swarm* to the *Tile Island* can be minimized.
An overview of the **G**eneral **C**oating Algorithm (GCA) is shown in Figure 3. First, each individual particle has to move (**Movement**) until it reaches a neighboring *Particle* or *Tile*. It is best, if it reaches a Tile, then it has a **Distance** of 1 to it. However, often it reaches a neighboring *Particle*, then it calculates its own distance based on the neighbors distance (**Distance Calculation**). Information calculated or collected from its neighborhood is propagated to its other neighbors, if it is relevant to that neighbor. (**Information Propagation**). Over time, each *Particle* learns its distance to the *Tile Island*and whether closer free positions exist in the graph. Thus, they learn whether movement is needed and in which direction the particle has to go. On its turn, each particle follows the steps depicted in Figure 3.

A. Movement

To select the next *Direction* to move to, which is done right before messages are sent, the *Particle* collects all free *Locations* from its *Neighborhood*that have a lower Own_Distance

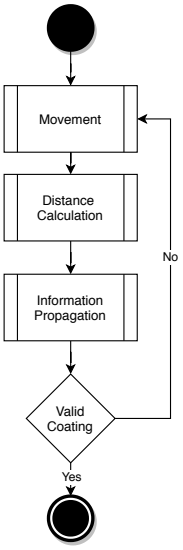


Fig. 3: An overview of the General Coating Algorithm

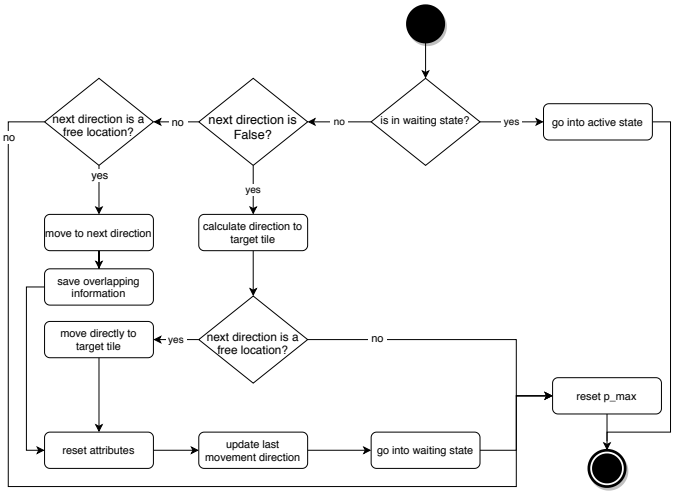


Fig. 4: Movement Flow Diagram

value than its current *p_max*and selects the one with the smallest Own_Distance estimation, excluding the one it came from during its last move action. In the movement phase, if the *Particle* successfully calculated a next *Direction*, it will attempt to move there. If the next *Direction* calculation did not yield a result, the *Particle* will try to move along the shortest path to its assigned *Tile*. In both cases the *Particle* will reset all of its memory. In the first case however it will save the distance information of its current, target and both shared *Locations*. If the *Particle* did not move at all, only its *p_max* will be reset. Additionally, each *Particle* that moved saves the *Direction* it came from and will go into a waiting state which will prevent it from moving during the next movement phase. An overview of this step is presented in Figure 4.

B. Distance Calculation

Each *Particle* that runs into a *Tile* or another *Particle* will start approximating its own and its *Neighbors Distance*. So each *Particle* $p \in P$ calculates the function

$$D(p) = \begin{cases} \min(D_{N_p}) + 1, \ \min(D_{N_p}) \neq \infty \\ \\ \infty, \ \min(D_{N_p}) = \infty \end{cases}$$

for itself, where N_p denotes the *Particles Neighbors* and $D_{N_p} = \{D(n) \mid n \in N_p\}$. For its *Neighbors* $n \in N_p$ the function $D(n)$ is calculated in a different way, because the *Particle* might have a message msg_n regarding that *Neighbors Distance* in its memory MM_p.

$$D(n) = \begin{cases} msg_n, \ msg_n \in MM_p \wedge D(p) \neq \infty \\ \min\limits_{x \in \{n_a, n_b, p\}} (D(x)) + 1, msg_n \notin MM_p \wedge D(p) \neq \infty \\ \infty, D(p) = \infty \end{cases}$$

where n_a and n_b are the *Neighbors* adjacent to n.
There is also the special case that the *Neighbor* n is a *Tile* t. In that case $D(t) = 0$. From here on when talking about the Own_Distance or *Distance* of a *Particle* or one of its *Neighbors*, we refer to this estimation, since the *Particles* have no information regarding their true Own_Distance.
If a *Distance* is calculated, for the *Particle* itself or any of its *Neighbors* that is higher than the *Distance* currently in that *Particles* memory, it is discarded.
If, however, the *Particle* is missing a message from any of its *Neighbors* it will go into the waiting state, because it can not be sure about the distances it has yet calculated.

In the same round each *Particle* p calculates the highest Own_Distance value of all *Particles* they know of, termed **p_ max**. Each *p_max* has a lifetime and the combination of a *p_max* and its lifetime will be termed *p_max* package. At the beginning the *p_max* value of each *Particle* is initialized as $-\infty$ and its lifetime will be zero. In addition each *Particle* saves all *Directions* from which it receives a *p_max* with the highest value.
If the *Particles* Own_Distance is not ∞ it compares its current *p_max* to the Own_Distance value of itself and all its *Neighbors* $n \in N_p$ and if the *Particle* finds a greater Own_Distance value than its *p_max* it updates its *p_max* with the new value and.
Then the *Particle* checks its memory MM_p for messages containing a *p_max* value from another *Particle*, compares these values to its own *p_max* and copies the whole *p_max* package if necessary and resets its own initial lifetime counter. The *p_max* value of a message will be referenced as $p_max(msg_n)$. This will be $-\infty$ if there is no such message or it doesn't contain this information.
So the formula for calculating the *p_max* is as follows:

$$p_max(p) = \begin{cases} \max\limits_{x \in \{p\} \cup \ N_p} \max(D(x), \ p_max(msg_x)), \ D(p) \neq \infty \\ \max\limits_{x \in N_p} p_max(msg_x), \ D(p) = \infty \end{cases}$$

If the *Particle* receives a *p_max* package with the same value as the package it holds, it will save the higher of both lifetimes in its package.
If the *Particle* has more than two *Neighbors* with a higher *Distance* than itself it will only use *p_max* packages from messages. If on the other hand the *p_max* calculated based on its *Neighborhood* is higher than any *p_max* from a message, the *Particle* will increase its own initial *p_max* counter by one and will copy it into the *p_max* lifetime, if it was lower.
If at the end of this step the *Particle* still has an Own_Distance

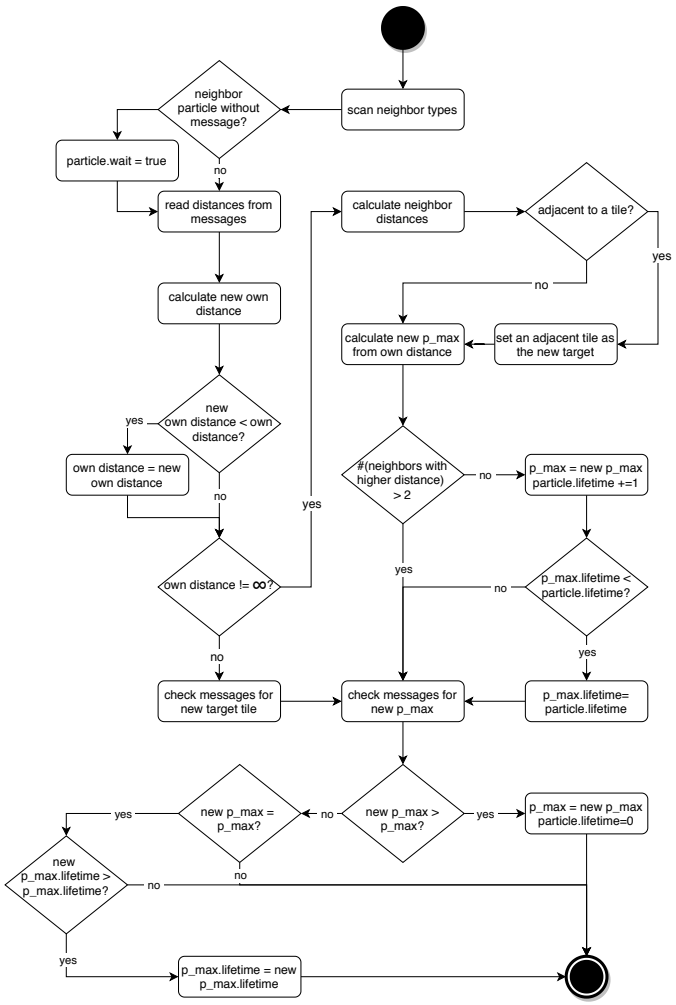


Fig. 5: Flowchart for the distance calculation step

of ∞ it will check if it received the coordinates of a tile to set as its own target tile. This is necessary to resolve the rare scenario of two *Particles* trying to move through each other. The concrete order of these steps is shown in the flowchart Figure 5.

C. Information propagation

In this step each *Particle* first checks if their Own_Distance is estimated to be ∞. In case so, it calculates its next movement *Direction* based on the coordinates of its target *Tile*. Else, it calculates its next movement *Direction* based on its neighborhood (how this is done is explained in the Movement Section).
If, in the latter case, no *Direction* could be calculated, the *Particle* will send data to its *Neighbors*.
If the *Particles* $p_max > 0$, it will send its Own_Distance estimation and its *p_max*, otherwise only the *Distance* will be send. if the former was the case and the *Particles* path is blocked by another *Particle* it will send the coordinates of its target *Tile* to the other *Particle*.
When sending its *p_max* to *Neighbors*, it will instead send its Own_Distance if the *Direction* is in the set of *Directions* where its current *p_max* came from.

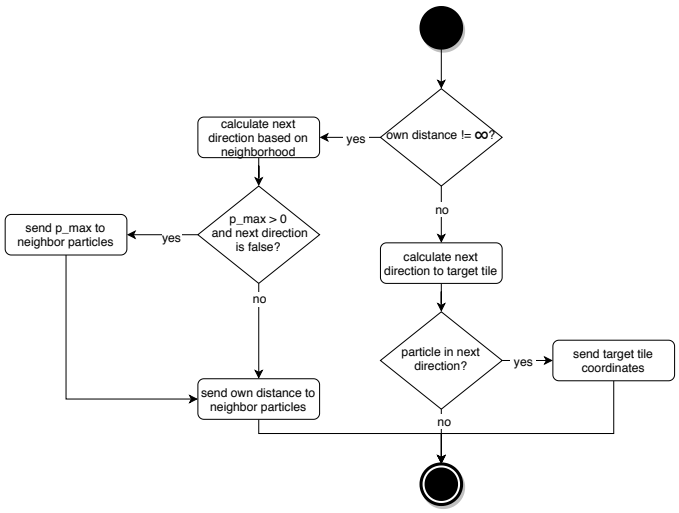


Fig. 6: Flowchart for the information propagation step

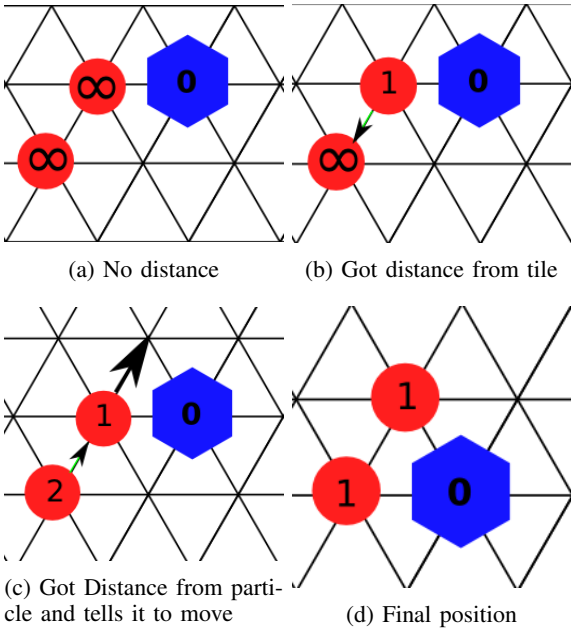
When sending only its Own_Distance , the *Particle* will check how many rounds it has been sending without moving or otherwise resetting its memory. It will always send its Own_Distance to all *Neighbors* with a higher estimated Own_Distance and if it has been sending for more than 15 rounds, it will send a dummy messages containing and Own_Distance of ∞ to all other *Neighbors*. This step is shown in Figure 6.

VI. Example

A simple example of the GCA is shown in fig 7. At the beginning (fig. 7a) both red particles have an undefined distance (i.e. the ∞ sign) and the tile has a distance of zero as described earlier. Hence one of the subjects neighbors is a tile, it will get the distance of one as shown in fig. 7b. After calculating its own distance it will share its information with its neighbor particle as indicated by the green lined arrow to the particle marked with infinity. The other particle gets this information and calculates its own distance, in this case two, as shown in fig. 7c. This particle is then telling the other particle that he is the one with the highest distance p_max. This causes the the particle with distance one to make space for the other one and it will move to the next location with a distance that is either equal or less than p_max. This causes both subjects to move closer to the tile and have the final position shown in fig. 7d.

VII. Evaluation

After having defined the algorithm, we will now begin its evaluation. For the evaluation of the GCA, the simulator **Swarm Sim** has been used. This simulator is created at the "Technology of Social Networks" lab of the Heinrich Heine University in Düsseldorf in Germany. It is a 2D and 3D simulation core that provides interfaces for creating individual scenarios and solutions with the main focus on simulating swarms. This simulator is open source and can be downloaded in [14] and the GCA algorithm in [15]. This algorithm will be tested with the following test settings:



(a) No distance (b) Got distance from tile

(c) Got Distance from particle and tells it to move (d) Final position

Fig. 7: Example of simple GCA

- Nine Scenarios shown in fig. 8
- Each scenario will be tested with 20 to 100 *Particles* in steps of 20 placed randomly anywhere between $x = \pm 600$ and $y = \pm 800$ in euclidean coordinates
- 50 same tests by using random seeds
- A round limit of 10000

A. Test scenarios

To test our algorithm, we prepared 9 tests *Scenarios*. during a test run all *Scenarios* will be tested with the same number of randomly distributed *Particles*. The *Tile* layout of each *Scenario* can be seen in Figure 8.

Scenario	***Particles* needed to coat up to *Layer***				
	1	2	3	4	5
Single Tile	6	18	36	60	90
Multiple Tiles	27	60	99	144	195
Small Holes	41	76	116	162	214
Thin Tunnel	47	88	134	185	242
Small Cave	27	53	84	121	164
Medium Cave	57	110	158	210	268
Large Cave	40	86	134	181	233
Bottle	58	113	167	221	281
Small Bottle	43	83	124	170	222

TABLE I: Coating *Layer* sizes: *Particles* required to reach a coating of thickness 1 to 5 in each *Scenario*

To better compare the *Scenarios* with each other and to investigate the behavior of the algorithm Table I shows the number of *Particles* required to coat the *Tile Island* up to the first five full *Layers*.

B. Metrics

The objective of this paper is to create a swarm algorithm as described in the introduction, the main targets for optimization will be, in order, flexibility, i. e. the ability to handle all described *Scenarios*, scalability, i. e. the ability to handle many different *Particle* counts. The robustness,i.e. the ability to handle failure of individual *Particles*, will only be tested in the last iteration of the algorithm.
To evaluate our algorithm in respect to these attributes we will first need to define measurable metrics that reflect these attributes.
Thus our first metric will be the success rate, because the main target is flexibility and, since the only criteria to label a simulations as unsuccessful is it reaching the round limit, the runtime in rounds will also be an important metric, as a very high runtime is nearly equivalent to failure. So even though we will optimize our algorithm in respect to success rate, the main metric we will evaluate is the runtime, since it is directly tied to the success rate.
The next Section will cover the tests of the GCA Algorithm and what these tests tell us about the algorithm.

C. Runtime

In this Section we will examine the runtime of the algorithm . Table II and fig. 9 shows the runtime using the same *Scenarios* introduced in Section VII-A. As we can see the GCA Algorithm did not fail a single test. As a result, the GCA Algorithm is working. After evaluating the GCA Algorithm in the same *Scenarios* we will now conduct some further tests on scalability, robustness and flexibility.

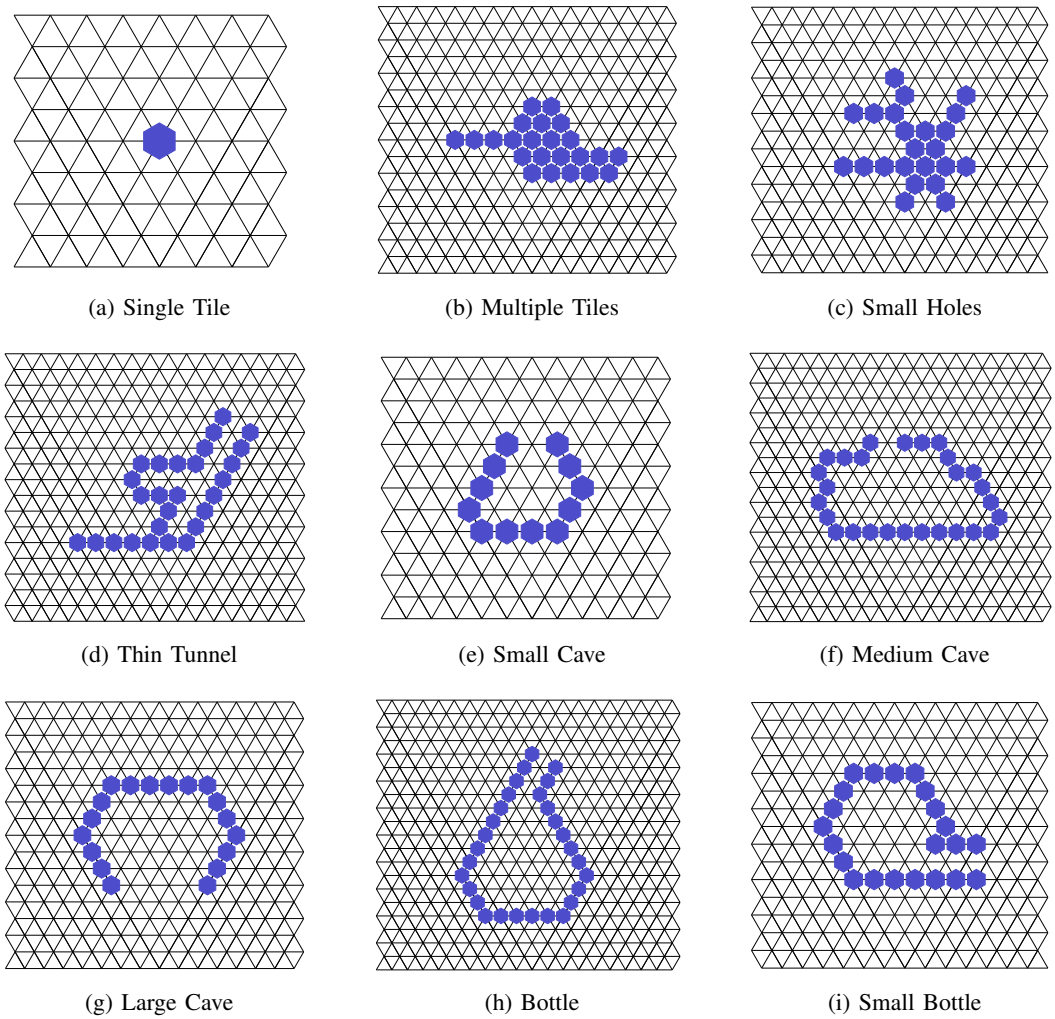
D. Scalability

Now to test the scalability of the GCA Algorithm, we ran it on our test *Scenarios* with *Particle* counts from 120 to 500 in steps of 20 and the runtime of these test is visualized in the plot in Figure 10. Again, all simulations were successful, and the runtime increased linearly with the *Particle* count, aside from the fluctuations caused by the fill level of the outer *Layer*, while the standard deviation is either nearly constant or also increases linearly depending on the *Scenario*. Both of these predictions are not very precise as our test were not made at specific fill levels of the outermost *Layer*, so most of the tests are not comparable enough to predict the trend. The next Section will deal with the robustness of the algorithm.

E. Robustness

To test the robustness of the GCA Algorithm with respect to failing *Particles*, we run the test *Scenarios* again but delete some of the *Particles* a few rounds before the solution state is reached.
So, in round one, five percent of the *Particles* are chosen at random and in round 300, which is shortly after most of the *Particles* have reached the *Tile Island*, the chosen *Particles* will be deleted. As we can see in Table III, this had little effect on the runtime. The small changes come from either the reduced amount of *Particles* or the changes in the random generator from randomly selecting the *Particles*. But since, again, no simulations failed we can say that our algorithm is robust against deleting *Particles*.

But as our *Particles* have no mechanism to move around obstacles other than *Tiles*, *Particles* that do not disappear block the way for other *Particles*. Even worse is that these dead



(a) Single Tile (b) Multiple Tiles (c) Small Holes

(d) Thin Tunnel (e) Small Cave (f) Medium Cave

(g) Large Cave (h) Bottle (i) Small Bottle

Fig. 8: *Scenarios*: Figure showing the *Tile* layouts for the different *Scenarios*

Particles catch all other *Particles* traveling past them, since *Particles* wait for messages from their *Neighbors* before they move but solving this is not part of this paper and is left for future work.

F. Flexibility

The flexibility we want to achieve is the ability to solve all possible *Tile* formations. We have shown some *Tile Islands* with features that proved to be difficult in early testings, but in this Section we test our algorithm in an unexpected *Scenario* of two separates *Tile Islands* as shown in Figure 11a.

The success rates of the tests are shown in the bar plot Figure IV. Here we can see that the algorithm fails at low *Particle* counts but has no difficulties at higher one. This is the case, because the *Particles* coating each *Tile Island* cannot touch each other if the amount of *Particles* is to low, as shown in Figure 11b, and thus cannot share their information, which causes them to coat each *Tile Island* individually, but as the condition for a successful simulation does not differentiate between the two *Tile Islands*, the simulation does not finish if the *Tile Islands* are coated unevenly. If the amount of *Particles* is sufficient however, the two splits *Swarms* start to touch each other and exchange information, which leads to a successful coating as depicted in Figure 11c. But further testing and adjustment of the goal condition is left for future work. After assessing the functionality of our algorithm, we talk next about possible extensions and improvements in the next Section.

VIII. Conclusion & Future Work

The coating algorithm presented in this paper is flexible, i. e. it can solve the coating problem for all possible Tile formations and it is scalable. Further, the Particles needed for this algorithm are completely independent and identical which makes the Swarm more robust, as there is no single point of failure and increasing the Swarm size is also easier, even while other robots are already working on the problem. Using this algorithm is also efficient for smaller problems, because there is no minimum of Particles required, so the Swarm size is fully up to the requirements of the specific application. Since all Particles have identical and only minimal functionality, their production is cheap as well and no central control unit is necessary, which further reduces the cost of using this algorithm. As already discussed in the last Section, the existence of dead *Particles* or other obstacles that are not to be coated is a problem of the GCA Algorithm. This is a real

Particle count	*Scenario*											
	Single Tile				**Multiple Tiles**				**Small Holes**			
	Min	Avg	Max	SD	Min	Avg	Max	SD	Min	Avg	Max	SD
20	335	388	503	30	316	393	461	32	308	382	425	29
40	371	424	518	38	347	404	533	35	395	545	830	83
60	446	597	832	89	401	639	923	104	379	456	602	54
80	479	634	875	88	425	546	755	81	434	618	1058	126
100	521	676	926	85	533	795	1157	143	461	614	833	86
	Thin Tunnel				**Small Cave**				**Medium Cave**			
	Min	Avg	Max	SD	Min	Avg	Max	SD	Min	Avg	Max	SD
20	302	388	485	31	332	418	542	34	296	377	455	33
40	403	527	740	64	377	446	524	38	466	600	782	80
60	560	681	836	61	425	512	647	56	1079	1260	1559	97
80	545	680	1007	79	461	611	902	85	914	1031	1211	73
100	557	705	983	84	467	616	875	94	1118	1415	1802	139
	Large Cave				**Bottle**				**Small Bottle**			
	Min	Avg	Max	SD	Min	Avg	Max	SD	Min	Avg	Max	SD
20	311	388	461	29	323	384	455	30	308	383	455	31
40	508	720	941	81	457	594	884	77	692	856	1061	88
60	464	545	671	46	1370	1592	1850	106	686	782	944	55
80	626	884	1267	126	1136	1271	1523	71	896	1200	1619	150
100	674	906	1205	122	1231	1422	1715	117	1046	1276	1649	135

TABLE II: Runtime test overview of the GCA Algorithm: Table showing the minimum, average, maximum and standard deviation of the number of rounds taken for each combination of *Scenario* and *Particle* count.

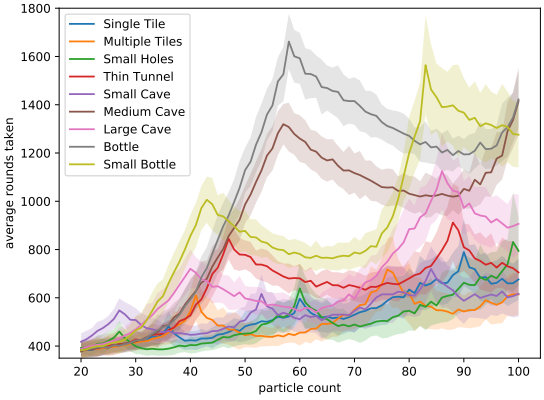


Fig. 9: Plot showing the average runtime and its standard deviation of the GCA Algorithm for each *Scenario*

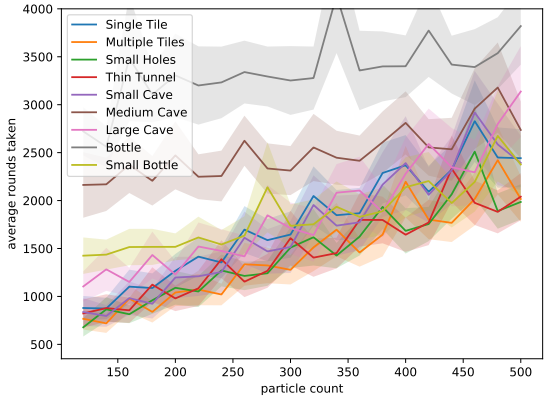
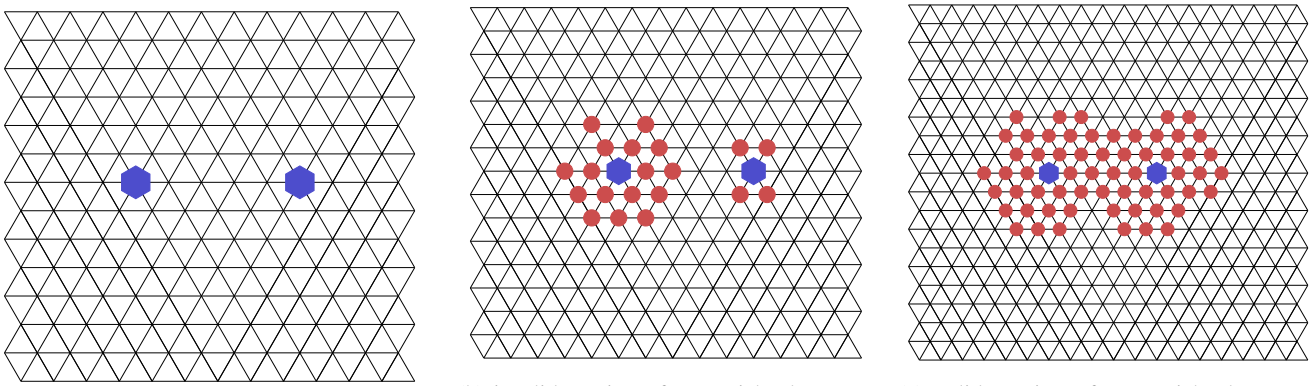


Fig. 10: Average runtime plot of the final algorithm for *Particle* counts 120 to 500



(a) *Scenario* with two *Tile Islands*

(b) invalid coating of a two island *Scenario* with few *Particles*

(c) valid coating of a two island *Scenario* with many *Particles*

Fig. 11: Comparing how different *Swarm* sizes coat a two *Tile Island Scenario*

problem when using the algorithm in practice, since *Particles* that malfunction normally don't disappear. This can possibly be solved by extending the message system of the *Particles* to send extra messages that show the *Particle* is still working

Particle	*Scenario*											
count	**Single Tile**				**Multiple Tiles**				**Small Holes**			
	Min	Avg	Max	SD	Min	Avg	Max	SD	Min	Avg	Max	SD
20	317	395	493	36	300	387	449	38	300	380	437	34
40	365	435	545	41	359	395	461	24	407	494	701	56
60	410	522	746	68	415	530	656	60	386	439	511	32
80	449	587	833	81	410	513	683	60	437	635	905	95
100	503	680	890	88	530	681	941	93	448	581	866	85
	Thin Tunnel				**Small Cave**				**Medium Cave**			
	Min	Avg	Max	SD	Min	Avg	Max	SD	Min	Avg	Max	SD
20	287	384	461	34	322	414	499	36	295	374	446	35
40	398	499	703	63	377	453	560	43	428	551	748	76
60	578	696	836	60	413	539	695	65	1081	1295	1570	119
80	509	660	857	68	440	621	932	112	929	1066	1229	75
100	599	745	959	89	410	607	827	88	860	1144	1490	131
	Large Cave				**Bottle**				**Small Bottle**			
	Min	Avg	Max	SD	Min	Avg	Max	SD	Min	Avg	Max	SD
20	310	385	454	28	320	376	446	30	300	381	467	36
40	508	669	884	87	446	555	745	75	523	761	959	110
60	443	555	698	57	1022	1555	1880	157	707	806	1073	68
80	526	772	1010	118	1109	1312	1499	89	725	905	1240	124
100	728	949	1400	133	1010	1242	1478	107	1145	1351	1736	132

TABLE III: Runtime test overview of the robustness test: Table showing the minimum, average, maximum and standard deviation of the number of rounds taken for each combination of *Scenario* and *Particle* count.

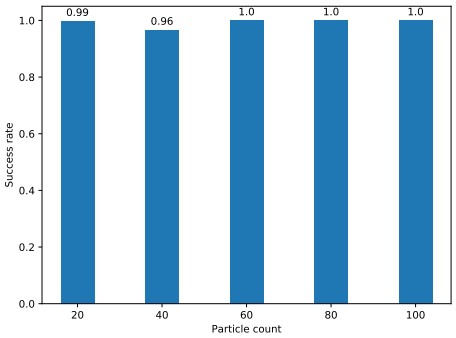


TABLE IV: Plot showing the success rate in the *Scenario* with two *Tile Islands*

and having the other *Particles* react accordingly if the *Particle* is dead.

Another extension mentioned in the last Section is the ability to coat multiple different *Tile Islands*. This is supported by the GCA Algorithm to some extend but needs further testing and some modification might also be necessary.

There is also the problem of simulator limitations, as the simulator is designed to have all *Particles* carry out the steps of the algorithm in order. Even though the order can be randomized each round, it is still unclear how well the *Particles* work together if they run the algorithm without any notion of rounds or even time.

Other possible enhancements could be a three-dimensional version or an algorithm that works on an unlimited number, or at least a higher number, of *Directions*. Application specific extensions are also numerous, like giving *Particles* the functionality to measure or manipulate certain properties of the object, which will lead to a multitude of more complex applications than simple coating.

ACKNOWLEDGMENTS

We would like to thank the Arab-German Young Academy of Sciences and Humanities for funding this research as well as our collaboration partner Dr. Ahmed Khalil for the valuable discussions on future nanorobot technologies.

REFERENCES

[1] K. Lerman, A. Martinoli, and A. Galstyan, "A review of probabilistic macroscopic models for swarm robotic systems," in *International workshop on swarm robotics*. Springer, 2004, pp. 143–152.

[2] S. Wilson, T. P. Pavlic, G. P. Kumar, A. Buffin, S. C. Pratt, and S. Berman, "Design of ant-inspired stochastic control policies for collective transport by robotic swarms," *Swarm Intelligence*, vol. 8, no. 4, pp. 303–327, 2014.

[3] L. Blazovics, K. Csorba, B. Forstner, and C. Hassan, "Target tracking and surrounding with swarm robots," in *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE, 2012, pp. 135–141.

[4] G. P. Kumar and S. Berman, "Statistical analysis of stochastic multi-robot boundary coverage," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 74–81.

[5] T. P. Pavlic, S. Wilson, G. P. Kumar, and S. Berman, "An enzyme-inspired approach to stochastic allocation of robotic swarms around boundaries," in *Robotics Research*. Springer, 2016, pp. 631–647.

[6] Z. Derakhshandeh, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann, "Universal coating for programmable matter," *Theor. Comput. Sci.*, vol. 671, pp. 56–68, 2017.

[7] J. Yu and S. M. LaValle, "Multi-agent path planning and network flow," in *Algorithmic foundations of robotics X*. Springer, 2013, pp. 157–173.

[8] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," *arXiv preprint arXiv:1612.05693*, 2016.

[9] H. Ma, S. Koenig, N. Ayanian, L. Cohen, W. Hönig, T. Kumar, T. Uras, H. Xu, C. Tovey, and G. Sharon, "Overview: Generalizations of multi-agent path finding to real-world scenarios," *arXiv preprint arXiv:1702.05515*, 2017.

[10] K.-H. C. Wang and A. Botea, "Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees," *Journal of Artificial Intelligence Research*, vol. 42, pp. 55–90, 2011.

[11] P. Obermeier, "Scalable Robotic Intra-Logistics with Answer Set Programming," in *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 64, Dagstuhl, Germany, 2018, pp. 24:1–24:5. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9890

[12] D. Lekic and A. Hampus, "Extending multi-agent pathfindingtowards item transportation, with comparisons between heuristics indifferent settings," 2018.

[13] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '18. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2018, p. 757–765.

[14] A. R. Cheraghi and U. Duesseldorf. (2020) Swarm simulator. [Online]. Available: https://gitlab.cs.uni-duesseldorf.de/cheraghi/swarm-sim

[15] A. R. Cheraghi and G. Wunderlich. (2020) General coating algorithm source code. [Online]. Available: https://gitlab.cs.uni-duesseldorf.de/cheraghi/swarm-sim/-/tree/coating