Institute of Operating Systems
and Computer Networks

Bachelor's Thesis

Distributed, scalable algorithmic methods for swarms with multiple leader robots

Maximilian Ernestus
Matrikelnummer: 4214647

Dominik Krupke
Matrikelnummer: 4143705

September 16, 2014

Prüfer: Prof. Dr. Sándor Fekete und Prof. Dr. Rüdiger Kapitza
Betreuer: Dr. Michael Hemmer

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund - Abteilung Algorithmik

Statement of Originality

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, September 16, 2014

Abstract

In this bachelor thesis, we examine a scalable and distributed method to control and maintain large swarms of robots. In an attempt to find a balance between local and global swarm manipulation strategies, independent, externally controlled leader robots are used to influence the shape and movement of a swarm. Particular focus is put on the distributed management of swarm connectivity in the face of multiple simultaneous robot failures. Steiner-tree approximations are formed between the leader robots to pursue the antagonistic goals of control and connectivity. A simulator is developed to allow for simulated experiments to verify the feasibility of the proposed methods and an intuitive understanding of the swarm algorithms by its extensive visual analysis capabilities.

Attribution of Work

All algorithmic methods and other ideas, which are not adapted from other cited sources, are a result of discussions and brainstorming sessions between the two authors. Therefore, a concrete attribution of ownership of the different parts of the work is nearly impossible. Nonetheless, there were responsibilities for different areas of the work based on the authors personal interests or based on who initiated the final iteration of an idea. Based on those responsibilities, the major writing work on the corresponding chapters and source code was distributed.

Section 1 (Introduction): This section has been portioned and most of the papers have been read by both authors.

Section 2 (Preliminaries): This section has been done by Krupke, but Ernestus kept him multiple times from getting lost in details.

Section 3 (Flocking): The flocking algorithm has been adapted and explained by Krupke. The plots were made by Ernestus.

Section 4 (Boundary): This section has been performed by Krupke, but the problems were often discussed.

Section 5 (Pheromones): This section has been done by Ernestus, except for the Volatile Pheromone and some graph theory. A lot of work in pheromones is not part of this thesis as they changed from a complex biological inspired system to a simple dynamic broadcasting forest.

Section 6 (Leader): This method has been developed and described by Krupke and tested by Ernestus.

Section 7 (Connectivity): After the ideas of the authors for a continuous connectivity detection failed, Ernestus developed this discrete method. The metric was created by Krupke.

Section 8 (Density): This section has been developed by Krupke.

Section 9 (Steiner Tree): This section has been developed by Krupke.

Section 10 (Integration): The integration was made by Ernestus.

Section 11 (Simulator): The simulator has been partly rewritten multiple times and thus it is difficult to say, who wrote more. The overlay system has been inited by Ernestus and was later elaborated by both authors.

Section 12 (Experiments): The rough plan has been made in discussion. The details, implementation, and interpretation has been done by Ernestus.

Section 13 (Conclusion): The points of the conclusion were made in discussion and written by Ernestus.

Further details can be provided on demand, due to Git logs and source annotations.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Swarm Robotics

Swarms in nature, like ant colonies, fish schools, and bird flocks, are known to be robust, scalable, and flexible. Within some boundaries they can operate on arbitrary sizes and stand the loss of individuals. Also, they can quickly react to changes in their environment. Often swarms are only based on primitive individuals with limited capabilities and local knowledge, but the swarm itself can still have complex behavior. These are properties which are interesting for multi robot systems and led to swarm robotics. Swarm robotics tries to control artificial swarms consisting of numerous simple robots.

The size of a swarm in nature may vary from only a few dozen to multiple thousands or even more. At the moment artificial swarms are either very small or not capable for application outside the laboratory. Not only the price but also the management of large robot swarms is still a challenge because many robots need to be charged, programmed, and maintained. Thus, many researches in swarm algorithms only focus on simulations as we are doing, too.

Artificial swarms are good for searching, discovering, and monitoring large areas. Their strength lies in parallelism in space over many weaker homogeneous individuals, which can be superior to only one strong individual. Also, they are more robust and easier to repair than single but larger robots. If a robot in a swarm fails, it can be simply replaced by adding a new robot from outside or letting another robot move up.

Swarm robots are an active topic of research not only in algorithmic but also in hardware and are getting cheaper and more advanced. An example is the 14 $ Kilobot with a size of only around 3 cm and using vibration for motion [54]. The studies are not only focused on land terrain but also on water or air like the RoboBee project at the Harvard University ([3]), a pico air vehicle.

1.2 Robot Model

Our robot model is inspired by the R-Ones of McLurkin (Fig. 1), but extended by the ability to measure the distance to a neighbored robot. Therefore, our robots have the following abilities:

- Drive forwards and backwards with limited velocity and acceleration
- Rotate left and right with limited velocity and acceleration
- Sense the orientation and distance to a neighbored robot within a limited range
- Sense the motion of a neighbored robot within a limited range
- Communicate with robots within a limited range and identify neighbors by an ID
- Sense collisions and their location with obstacles and other robots

In Fig. 2 the knowledge of a robot is visualized. Knowledge in this context refers to the local position, rotation, and speed of other robots. Please notice that the robot is not able to communicate with every robot in its range, since the communication needs visual contact. The knowledge of a robot can be extended by using multiple hop communication, but this approach scales exponential in the communication complexity over the amount of hops.

Figure 1: The R-One Swarm Robot developed at the Rice University. It has a diameter of circa 10 cm. Photo taken from [2].

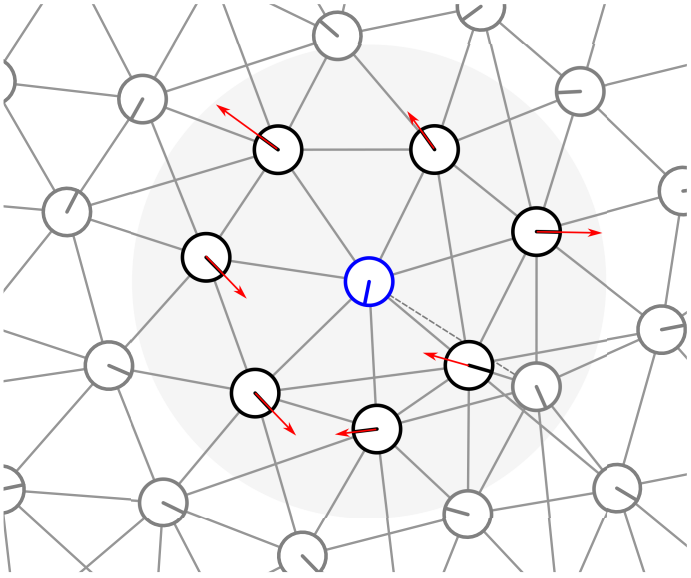


Figure 2: The knowledge of a robot (blue robot) is limited to a specific range (gray circle). In this it knows the local coordinates of its neighbors (black robots), their rotation, and their velocity (red arrows). Neighborhood is visualized by a gray line between the robots. The grey robot within the grey circle is no neighbor due to the line of sight constraint (dotted line).

1.3 Controlling Swarms

There are generally two different approaches to control a multi robot system. The first approach is a global – possibly external – control, which collects all knowledge and decides how each robot has to act. The second approach is the local, where each robot decides by itself how it should react using only local knowledge.

To find the best strategy to perform a task, all information should be taken in consideration. Therefore, a central leader is elected, which can be a random robot, a special robot, or even a server connected with the swarm. All robots send their knowledge to this central leader, which can find an optimal strategy for the swarm and tell it to the robots. Since the robots only have local communication, the communication with the central leader is slow and not scalable. Also, the centralized controlled swarm is vulnerable to failures of the connection to the central leader or failures of the central leader itself. Thus, global control is contradictory to the concept of a swarm and can only be used for small multi robot systems. On the other hand, centralized control allows global strategies and simple external control.

Nature has shown that often there is no need of a leader and local control is sufficient. Each robot can decide on its on perception and its direct neighbors, which strategy it should follow. There is no arrangement of which strategy to follow and thus there can be contradictory behavior of robots. The complexity of computation and communication, however, becomes negligible making the swarm super scalable. Since every robot is equal and there are no delays caused by coordination, the swarm becomes very robust and flexible. Hence, the distributed local control and self-organization is the origin of the strength of a swarm. Unfortunately, it also makes it very complicate to control a swarm, since there is no explicit control in the swarm.

As the goal of swarm robotics is to gain control over the swarm while conserving the properties of it, a hybrid scheme has to be found as exposed by [50] and [7]. In this thesis, we are providing such a schema where some selected leader robots are influenced by a global control while all other robots are self organized.

1.4 Programming Swarms

Brambilla et al. [9] divided the swarm design methods into the two categories *behavior based design methods* and *automatic design methods*. In *behavior based design methods* the behavior of each robot is implemented explicitly and adjusted until the desired swarm behavior is reached. In *automatic design methods* the behavior of the swarm is specified and the implementations for the robots is developed by a computer.

The most common behavior based design methods are *probabilistic finite state machines* and *virtual physics*. In *probabilistic finite state machine design*, the robots act after their actual state (e.g. [61], [47]). The state may changes for example due to sensor input on a probabilistic base. In *virtual physics design*, each robot acts as particle and the environment creates forces onto it like repulsion or attraction from a neighbored robots (e.g. [63], [48]). The robot will move in the direction of the sum of the forces without any state. The resulting programs are accordingly equations.

In *automatic design methods* the desired swarm behavior is specified to the computer, which tries to create a matching behavior for the single robots using evolutionary algorithms or similar methods. A broad review of this topic has been made by Panait and Luke [49].

Of course there are a lot of other or combined methods to engineer a swarm. Brambilla et al. [9] also discuss, which method is preferred for which application.

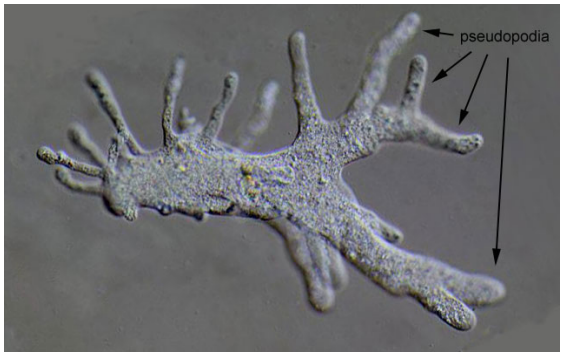


Figure 3: Pseudopodia in cells.

1.5 Flocking

Flocking is the swarm behavior of moving in a group. A first computer simulation of this behavior has been done by Reynolds [53] in 1987 for aesthetic reasons. To do so, he introduced the following three heuristic rules, which lead to flocking:

1. Flock Centering: Attraction to neighbors
2. Collision Avoidance: Repulsion from too close neighbors
3. Velocity Matching: Adaptation of the velocity of neighbors

These rules are used in a lot of flocking algorithms like [48] or [28] and are independent of the dimension. They proved to be sufficient and can be abided by a robot with only knowledge of local environment and close neighbors.

1.6 Pseudopodia

Pseudopodia are protrusions of the cell membrane of varying thickness, that are used for guidance towards chemoattractants, neural growth, embryonic development, and cell migration [41] (see Fig. 3). They are formed by the cell skeleton first pushing dents into the cell membrane and then growing into the dents to further extend them.

We borrow this concept from nature and apply it to swarm robotics by viewing the robot swarm as a cell. Pseudopodia are then parts of the swarm, that explore the environment while staying connected to the swarm. This method is used to intuitively control the rough shape of the robot swarm without the need for completely centralized global control methods. On top of this method of swarm formation, more complex swarm behaviors can be implemented, like for example the bud growth method to explore unknown environments proposed in [51].

1.7 Pheromones

In nature pheromones are used by many swarm animals like ants, bees or termites as a means for global, spatial organization. This scalable and robust organization method inspires a broadcasting system similar to the virtual pheromones introduced by Payton et al.[51].

These virtual pheromones differ from pheromones in nature as they are not bound to the environment but rather use the robots mesh network as a medium to diffuse (see Figure 4). This is due to ease of implementation and because it allows the pheromones to represent parts of the swarm topology.

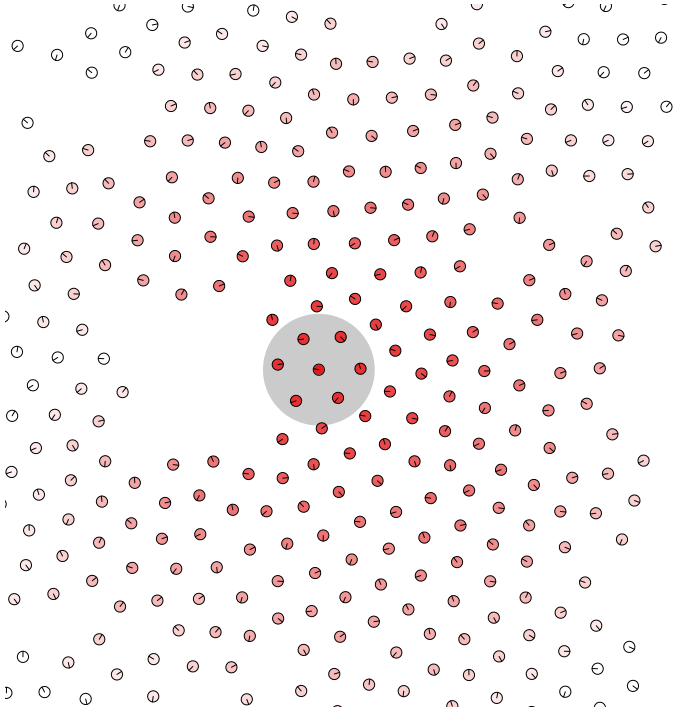


Figure 4: The highlighted robot emits a virtual pheromone into the swarm. The pheromone's intensity decreases with further distance to the emitter.

1.8 Goals and Approach

As a result of our bachelor thesis we envision a robot swarm with the following properties:

- Its robots tend to distribute in an evenly spaced equilateral triangle grid to maximize the covered area while staying in contact. At the same time the overall shape is smoothed out like a water droplet to avoid fragmentation.
- The swarm is controllable by multiple leader robots, that always group their peers around them and drag them along. These leader robots are used to steer and shape the swarm.
- The connectivity of the swarm is always ensured for user defined rates of robot failure and despite of contradicting control inputs by leader robots.
- Except for the leader robots, which might be controlled by a human supervisor, all control strategies are distributed and rely solely on local information.

To reach this goal we use the following techniques:

To distribute the robots in an equilateral triangle grid we use a flocking algorithm based on [48]. For smoothing the overall shape of the swarm, boundary robots apply a boundary pressure directed to the center of the swarm if there are protrusions or outwards if there are concave boundary regions as described in [37].

Leader robots are modeled like normal robots, that can be remote controlled by one or multiple operators of the swarm. They emit pheromones to lure other robots in their direction.

A connectivity metric, that guarantees the connectivity among leader robots despite the failure of an arbitrary but fixed number of robots, is used. This metric is constantly computed in a distributed and scalable way with the help of artificial pheromones. If it falls below a certain threshold, the affected regions of the swarm are identified and enforced with robots, to keep the connectivity above a minimal level.

1.9 Previous Work

For flocking we build upon the first algorithm presented in [48]. It is based on virtual physics and can be filed as a social potential field method. We choose it because it implements Reynolds flocking rules and is proven to stabilize.

The pheromone model is based on [51] and [43]. They both describe message broadcasting systems with characteristics similar to pheromones found in nature with the limitation that pheromones are not bound to the environment but to the members of the swarm.

For boundary detection, the algorithm proposed in [44] by McLurkin and Demaine is extended. To detect the dynamically changing boundaries of a robot swarm, they describe a distributed boundary detection algorithm based only on local properties of a swarm.

Based on the detected boundary, a boundary force algorithm proposed in [37] is extended. This boundary force algorithm applies an effect like the tension of a water droplet on the robot swarm's boundary and is based only on local information about the swarm's boundary.

1.10 Related Work

1.10.1 Control

Most of the studies in swarm robotics relate to fully autonomous swarms with no external control or the goal is given in advance. As there is no centralized control in the swarm and controlling all robots is not feasible, only a small amount of robots can be steered directly and thus it is important to know how to influence a swarm. The following two works are considering the interaction between humans and swarms.

Çelikkanat and Şahin [12] consider how to steer a swarm into a direction by informing some robots. The swarm itself stays totally self organized and is only influenced by these informed robots. They tested it in simulation and real hardware.

Kolling et al. [32] proposes two different approaches of how to control a swarm by a human. The first approach is based on selecting a subset of robots, for example by an area, to give them instructions. The second uses a beacon which can be set on a location and influences the robots in its area. Both methods require the knowledge of the global position of the robots.

1.10.2 Connectivity

Most works only consider connectivity implicitly or in experiments but not explicitly in the algorithms themselves. The following works however do it explicitly.

Esposito [19] considers the navigation of a swarm with the constraint of perseverance of specified links. Links between robots are not only specified by range but also by line of sight due to obstacles. They are using social potential fields based on local information. They made simulations and experiments with a small set of robots.

Bai et al. [5] consider coverage and connectivity for the deployment of wireless sensor nodes with radius limited ranges. The deployment is considered as movement and is therefore related to swarm robots. They proposed a deployment pattern for 2-connectivity and full coverage and

proved its optimality. The coverage is based on the sensing range and the connectivity on the communication range, which are allowed to differ.

1.10.3 Steiner Tree

For our goal to maximize the connectivity between the leader robots, the $\mathcal{NP}$-hard Euclidean Steiner tree problem plays a role.

Hamann and Wörn [27] proposed a robust heuristic with swarm robots based on state machines. The approach is based on building a network to discover paths between the points with a lot of robots. Afterwards the best paths are chosen and optimized. The other paths are removed.

There has been a lot of work in distributed delay constrained Steiner tree in graphs heuristics in (sensor) networks, for the usage as multimedia broadcast backbone. Many of those algorithms are based on the work of Kompella et al. [33], which is based on the distributed minimum spanning tree (MST) algorithm of Gallager et al. [23] as thea MST is a proven Steiner tree heuristic itself. Kun et al. [36] presented an algorithm which is based on simulated annealing and supports dynamic reorganization. A survey of such algorithms can be found in [8].

1.10.4 Multiple Leader Robots

In [38] leader robots are used to control a robot swarm. With just a single leader robot there can be multiple configurations in which the swarm can group around and follow the leader. These different configurations are determined by the multiple local minima of the potential field, that is used to control the swarm. To restrict the number of local minima, multiple leader robots are used that all guide the swarm in the same direction. Unfortunately, there are no experiments to support the results and no sensor based simulations. The leader robots are therefore only used to control the movement in a single direction of the swarm and not the overall shape.

Payton et al. [51] propose special robots acting as *growth buds* that are used as leader robots to control the shape of the swarm similar to plant growth. *Growth buds* explore the environment and thereby pull out robots from the swarm. They inhibit the creation of new growth buds via pheromones as long as they can move freely. This way the swarm can explore complex environments while staying connected.

Jiang-Ping and Hai-Wen [30] use multiple immobile leaders to define a polygon. Within this polygon the remaining swarm robots distribute regularly. This approach allows for accurate control of the swarm shape, but since the leader robots are immobile it is impossible to dynamically adapt to changing environments. Furthermore, the shapes are restricted to convex polygons.

1.10.5 Potential Fields

The work of Khatib [31] is one of the first appearances of virtual potential fields in robotics. Precomputed potential fields around obstacles are used for online robot motion planning. It is shown how a robot arm with multiple joints can reach within a complex enclosure without touching it, but an adaption to mobile robots is obvious. Since the methods are designed for a single robot within a static environment, the potential fields are mostly computed offline and the algorithm is not distributed.

Spears et al. [62] use potential fields to form grids in hexagonal and rectangular patterns. It is practically oriented with simulation, experiments on real robots and some critical code snippets. All computations are based on local information and the algorithms are fully distributed.

In [6] potential fields are used to create formations in a robot swarm. This is archived by virtual *attachment points* at specified angles around the robots. Different arrangements of these *attachment points* result in different global formations similar to tile composition models. The

proposed algorithms are distributed and only local information is used. Furthermore, a smooth flow around obstacles is possible.

1.10.6 Pheromones

Turing introduces in [65] the first mathematical model of so called *morphongenes* that are similar to the pheromone models used in swarm robotics. He shows how patterns can arise from homogeneous systems by disturbing their equilibrium by environmental noise.

Payton et al. [51] propose the first virtual pheromone system applied to swarm robotics. Pheromones have discrete strength and different colors (keys); every robot computes its pheromone based on the strongest pheromone signal in its neighborhood. Pheromones act as attracting or repulsing forces on the robots. A simple version of social potential fields is used to ensure connectivity and coverage within the swarm (simple discrete *distance rings* decide whether robots are attracted, repulsed or neutral towards each other). A distributed method is proposed to find the shortest paths between two points within the swarm and to place all robots on this path. Simulations (which are not documented) are supposed to show that the proposed ideas work.

Shen et al. [58] introduce a continuous digital hormone model (they call it hormones and not pheromones) within a probabilistic cellular automaton. The hormones are bound to the cellular automaton's grid and not to the robots as in [51]. Robots move probabilistically within the CA-grid; the probabilities are dependent on the local hormone distribution around the robots. They examine if this system can form patterns and how the patterns react to parameter changes within the system and to different initial robot configurations. There are experiments within the probabilistic cellular atomaton and the results are shown and discussed.

Later Shen et al. used their digital hormone model to solve tasks like aggregation around goals, covering unknown environments, self repair, and moving around concave obstacles [59].

The use of truly stigmergic pheromones has been studied byMayet et al. [42], who did experiments with phosphorescent glowing paint.

1.11 Overview

In Sec. 2 we give some notations and explain the robot model as used in this thesis.

Our method is based on multiple algorithms and heuristics: In Sec. 3 we explain the flocking algorithm of [48], which is the base algorithm and provides us a regular fragmentation and as well as an equilateral triangle grid. In Sec. 4 we consider the boundary, which includes a detection heuristic based on [44] and a dynamic boundary force based on [37], which provides a size and type sensitive boundary tension. In Sec. 5 we propose a pheromone system, which provides a dynamic broadcast forest. In Sec. 6 we propose a simple method for steering with multiple leader. In Sec. 7 a metric for inter-leader-connectivity is given and a heuristic to maintain a specific value. In Sec. 8 we consider the density for keeping a specific density in the swarm, which is especially useful in combination with boundary tension. In Sec. 9 we tell something about Steiner tree optimization for the swarm shape.

All these parts are integrated in Sec. 10 and tested with the simulator introduced in Sec. 11. The experiments are presented in Sec. 12 and the conclusion is given in Sec. 13.

2 Preliminaries

In this section we explain the notations and give some definitions. The notations of the graph theory used in this thesis are given in Sec. 2.1. In Sec. 2.2 we consider how to interact with the robots. For this the physical model of the robot is described in Sec. 2.2.1. Communication

and graph representation is introduced in Sec. 2.2.2. The physical model for the two wheel drive itself is only used in the simulation; for the algorithms we use the simpler theoretical model based upon vectors, which is mapped onto the physical model in Sec. 2.2.3. A simpler way of inter robot communication based on exposed variables is given in Sec. 2.2.4. Finally, we say some words about angles in Sec. 2.3.

2.1 Graph Theory

Graph Theory plays a fundamental rule in algorithmic methods for swarms. In this subsection we give a short overview over its notions, which is in most parts similar to the book of Diestel [15].

A graph is an abstract concept of objects, called vertices or nodes, and their relations. In this thesis the vertices are mostly robots and the edges represent their ability to communicate.

Definition 1. *An undirected graph $G = (V, E)$ consists out of a vertex set V and an edge set $E \subseteq [V]^2$. A directed graph $D = (V, A)$ consists out of a vertex set V and an edge (also called arc) set $A \subseteq \{(v, w) \in V^2 | v \neq w\}$.*

We denote a vertex $u \in V$ to be adjacent to another vertex $v \in V$ with $u \sim v$ if $\{u, v\} \in E$ for an undirected graph $G = (V, E)$ or $(u, v) \in A \vee (v, u) \in A$ for a directed graph $D = (V, A)$. An undirected graph can also be treated as a symmetric ($(u, v) \in A \Rightarrow (v, u) \in A$) directed graph.

A graph can be visualized by representing vertices as points in $\mathbb{R}^2$ and the edges as lines for undirected graphs or arrows for directed graphs between those points. Such a visualization of a graph is called an embedding. A graph with a definite embedding is called an embedded graph. Most of the graphs in this thesis are embedded graphs, even if we do not denote this explicitly every time.

A path $P_n = (v_0, v_1, v_2, v_3, v_4, v_5, \ldots, v_{n-1})$ of length $n - 1$ (edges) is a sequence of distinct vertices with $\bigwedge_{i=0,\ldots,n-1} v_i \sim v_{i+1}$. An u-v-path is a path with $v_0 = u$ and $v_n = v$. The shortest u-v-path is an u-v-path with minimal amount of vertices. The shortest paths in a graph can be found with Breadth-First-Search. A cycle C_n of size $n > 2$ is a path of size n with $v_0 \sim v_n$.

The graph is called connected if there exists an u-v-Path for all $u, v \in V$. Connectivity can be checked with Breadth-First-Search or Depth-First-Search in linear time. A common metric of connectivity is discussed and a new one introduced later. A vertex, that disconnects the graph on removal, is called an articulation.

A circle-free graph $T = (V, E)$ is called a forest. If it is additionally connected, it is called a tree. A spanning tree of a graph $G = (V, E)$ is a tree $T = (V, E' \subseteq E)$. Spanning trees can be found in linear time using Breath-First-Search.

A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is a graph with $V' \subseteq V$ and $E' \subseteq E$. If the vertex set of the graph G' is abstract, like for the C_3, we call it a subgraph of G if we can find a matching subset of V for V'.

For a graph $G = (V, E)$ with a weighting function $c : E \to \mathbb{R}^+$, the minimum spanning tree (MST) is a tree $T = (V, E' \subseteq E)$ with $\sum_{e \in E'} c(e)$ is minimal. The MST is not unique and if all weights are the same, each spanning tree of G is a MST. A MST of an arbitrary graph can be found in polynomial time for example with Kruskals algorithm [35].

An important graph, where the edges are implied by the positions of the vertices, is the *unit disc graph* ([13]).

Definition 2. *A unit disc graph for a constant $d \in \mathbb{R}^+$ is an embedded weighted graph $G = (V \subset \mathbb{R} \times \mathbb{R}, E)$, with $E = \{\{u, w\} | u, w \in V \wedge |u - v| \leq d\}$. $|x|$ is the euclidean length of x.*

2.2 The Robot Model

Our robot model is inspired by the R-One (Fig. 1) of the Rice University[2]. Similar to the R-One, our robot model can move forwards and backwards as well as rotate, sense the position and orientation of robots within a specific range, communicate with these neighbors, and sense collisions. We did no experiments with real R-Ones but only used simulations with perfect sensing and communication. While the R-Ones have only very low resolutions, our simulator provides 32bit floating point values. Also, we did not limit the communication bandwidth. However, our algorithms are designed (but not proven) to be robust against inaccurate sensor data and fault prone communication, as well as to be of low communication and computation complexity.

Definition 3 (Robot)**.** *A robot r has a constant unique ID denoted by* $\mathrm{id}(r) \in \mathbb{N}$ *and a local time given by* $\mathrm{localtime}(r)$*. All robots have a circle shape with the radius* $\tau \in \mathbb{R}^+$ *and a communication/sensing range of* Γ*. The set of all robots in a swarm is denoted by* $\mathcal{R}$*.*

In this subsection we describe the properties and abilities of the robot model. First we take a look on the physical properties as movement and sensors in Sec. 2.2.1. Then we consider the communication abilities and map the swam into a graph in Sec. 2.2.2. As we use vectors for movement within this thesis, we show how to map vectors to the physical movement in Sec. 2.2.3. Afterward we give an exposed variable model as a replacement for message communication in Sec. 2.2.4.

2.2.1 Physical Model

The physical model does only play a little role in this thesis and is thus only roughly considered. We believe that most of this thesis can also be modified for other models and even extended for the three dimensional space. However, the following model has been used for the simulations:

For simplicity we do not work with a continuous time but with equal time steps. We assume that these time steps are chosen small enough to be accurate. In each time step the robots have a movement state which represents position and movement. The algorithms in this thesis are assumed to be execute within one time step.

Definition 4 (Movement State)**.** *A robot* $r \in \mathcal{R}$ *has a global position* $\mathrm{pos}(r) \in \mathbb{R}^2$*, an orientation* $\alpha(r) \in [-\pi, \pi]$*, a velocity* $\dot{\beta}(r) \in \mathbb{R}$ *and a rotation velocity* $\dot{\alpha}(r) \in \mathbb{R}$*. The set of the movement states of all robot at a specific time is called a configuration.*

To ensure a realistic movement physics, we set up the following constraints to disallow shape intersections and to limit velocity and acceleration.

Definition 5 (Configuration Constraints)**.** *For each robot* $r \in \mathcal{R}$ *the following constraints are set:*

- *The absolute velocity* $|\dot{\beta}(r)|$ *is limited by the maximal velocity* $\dot{\beta}_{\max}$
- *The absolute angular velocity* $|\dot{\alpha}(r)|$ *is limited by the maximal angular velocity* $\dot{\alpha}_{\max}$
- *The absolute acceleration* $|\ddot{\beta}(r)|$ *is limited by the maximal acceleration* $\ddot{\beta}_{\max}$
- *The absolute angular acceleration* $|\ddot{\alpha}(r)|$ *is limited by the maximal angular acceleration* $\ddot{\alpha}_{\max}$
- *The induced velocity and acceleration are always directed into the orientation of the robot.*
- *The shape of r does not intersect with anything.*

Definition 6 (Collision)**.** *Two robots* $r, r' \in \mathcal{R}$ *are called collided if* $|\mathrm{pos}(r) - \mathrm{pos}(r')| = 2 * \tau$*.*

A collision influences the movement by an abrupt stop or pushing away. The associated robots can sense the point of this collision and react to it. However, the algorithms do prevent collisions by repulsion from close robots, so they are not important for us. As collisions are quite complicated, we do not give a mathematical definition but they are handled by the physics engine of the simulator.

2.2.2 Communication and Graph Representation

A robot can communicate with other robots within a fixed range $\Gamma \in \mathbb{R}$ if they have visual contact. An outgoing message is received in the next time step. To check for visual contact a line can be drawn between the position of both robots. If this line does not intersect anything except for the both robots, they are in line of sight. The line of sight can be inhibited by other robots or obstacles, but we do not consider obstacles in this thesis.

Definition 7 (Line of Sight)**.**

$$\mathrm{LoS} : \mathcal{R} \times \mathcal{R} \to \mathbb{B}, (r, r') \mapsto \begin{cases} \textsf{True} & \textit{No shape of a robot in } \mathcal{R} \setminus \{r, r'\} \textit{ intersects the} \\ & \textit{line } (\mathrm{pos}(r), \mathrm{pos}(r')) \\ \textsf{False} & \textit{else} \end{cases}$$

The set of all robots that a robot can communicate with is called its neighborhood. It consists of all robots, which fulfill the line of sight and range constraint.

Definition 8 (Neighborhood)**.** *The neighborhood of a robot is defined as* $N : \mathcal{R} \mapsto 2^{\mathcal{R}}, r \mapsto \{n \in \mathcal{R} | \, \mathrm{LoS}(r, n) \wedge |\,\mathrm{pos}(n) - \mathrm{pos}(r)| \leq \Gamma\}$.

The actual configuration of a swarm can be considered as an undirected graph, where each edge is a communication link. The robot positions are kept in its embedding. Except for the line of sight constraint it is equal to the unit disc graph.

Definition 9 (Communication Graph)**.** *The undirected graph* $S = (\mathcal{R}, \{\{r, n\} \in [\mathcal{R}]^2 | n \in N(r)\})$ *is called the communication graph. The communication graph is embedded in the euclidean plane with* $\mathrm{pos}(r)$ *is the position for each* $r \in \mathcal{R}$.

The graph representation allows a simpler analysis of the swarm as well as the usage of notations and results of the graph theory. For example the distance between two robots in the communication graph correlates to the message/knowledge propagation delay between them.

Definition 10 (Hop Distance)**.** *The hop distance* $d(r, r')$ *between two robots* r *and* r' *denotes the amount of edges in a shortest path between them in the communication graph.*

2.2.3 Vector Representation

For simplicity all our algorithms consider the robots as particles in a two dimensional room, and the movements are implied by force vectors. However, our physical model of the robots does not match to a particle as we have an orientation and a rotation movement. Thus, we transform those force vectors, which do not obey the configuration constraint, to the closest force vectors which do. We consider the global coordinate system and the local coordinate system. In the global coordination system all vectors of all robots are within the same coordinate system, but as the robots work locally this is not the case. Hence, it is only used for a simpler definition and global analysis of the algorithms. The local matches the individual local view of a robot, but needs a lot of transformations for robot interactions.

Definition 11. *The position, velocity and acceleration in the vector space are defined as*

Position $p : \mathcal{R} \to \mathbb{R}^2, r \mapsto \text{pos}(r)$

Velocity $\dot{p} : \mathcal{R} \to \mathbb{R}^2, r \mapsto (\cos\alpha(r) * \dot{\beta}(r), \sin\alpha(r) * \dot{\beta}(r))$

Acceleration $\ddot{p} : \mathcal{R} \to \mathbb{R}^2, r \mapsto (\cos\alpha(r) * \ddot{\beta}(r), \sin\alpha(r) * \ddot{\beta}(r))$

This definition ignores the rotation velocity and acceleration, but as long as it is determined new in every time step and the time steps are small enough, this should be sufficient. However, we do not preclude that there could be better definitions.

Definition 12 (Relative Position)**.** *The relative position of a robot to a robot $r \in \mathcal{R}$ is defined as $p_r : \mathcal{R} \to \mathbb{R}^2, n \mapsto p(n) - p(r)$. The relative velocity of a robot to a robot $r \in \mathcal{R}$ is defined as $\dot{p}_r : \mathcal{R} \to \mathbb{R}^2, n \mapsto \dot{p}(n) - \dot{p}(r)$.*

To translate the velocity vector to the physical model we first check if it is more efficient to move forwards or backwards. As absolute velocity we use the allowed velocity which is closest to the velocity vector length. For the rotation velocity we use the highest allowed rotation velocity which can still brake at the desired orientation. Again, we do not preclude that there could be a better solution, but in simulation it worked well.

Actually, there is no global coordinate system but every robot has its own local coordinate system. This local coordinate system has the associated robot as it origin and the x-axis pointing to the front. Of course it could also be shifted or rotated differently, but its local definition has to be uniform for all robots. As it is much easier to work on a single global coordination system, we show that a robot can transform the vectors of the local coordinate systems of neighbored robots into its own. First we define the local coordinate system for an arbitrary but fixed global coordinate system by giving transformation functions.

Definition 13 (Local Transformation)**.** *A global position can be transformed into a local position of a robot $r \in \mathcal{R}$ by*

$$\text{transform} : \mathbb{R}^2 \to \mathbb{R}^2, x \mapsto \begin{pmatrix} \cos\alpha(r) & -\sin\alpha(r) \\ \sin\alpha(r) & \cos\alpha(r) \end{pmatrix} * (x - p(r))$$

A global direction can be transformed into a local direction of a robot $r \in \mathcal{R}$ by

$$\text{transformDirection} : \mathbb{R}^2 \to \mathbb{R}^2, x \mapsto \begin{pmatrix} \cos\alpha(r) & -\sin\alpha(r) \\ \sin\alpha(r) & \cos\alpha(r) \end{pmatrix} * x$$

As a robot knows the relative position and orientation of its neighbors, it can transform vectors it receives from it to its own local coordinate system. Since the neighbors of the robot can also derive a valid transformation, it can simply broadcast vectors in its own local coordinate system to the neighbors. Thus, algorithms on the global coordinate system can be implemented locally by transforming vectors on every communication hop. However, the accuracy of the transformations heavily depends on the accuracy of the position and orientation determination. As we assume the sensors to be inaccurate, also the vectors will be inaccurate after a few hops, which has to be regarded for robust algorithms.

2.2.4 Exposed Variables

The ability of sending messages is a mighty but also difficult capability. As we can guarantee neither the successful transmissions nor the receipt of all information, the programming of algorithms can become very complicated. Each robot has to send its information's multiple times to lower the probability that some robot might have not retrieved it or a robot coming in its

range will not know it. For receiving, each robot has to save the information in the messages and automatically remove the outdated information. The individual management of messages in algorithms can thus be very elaborating. To lower the complexity of this task we allow robots to have exposed variables, which can be read by their neighbors. This can be implemented using messages. However, we do not guarantee that all exposed variables are up to date or even available. Exposed variables of neighbors, which have not been synchronized for some time, are set to not available.

We do not use point to point messages, as this is a fault-prone concept in swarms. Messages cannot be used like in computer networks. Due to the large amount of hops, the quickly changing topology and the scalability, the classical network protocols like TCP are not feasible. Also, a swarm robot should only act locally and therefore the usage of such a network protocol would be a violation of the swarm principles. By allowing each robot to have exposed variables, we create a local network layer, which is easy to use and analyze.

The usage of these exposed variables, allows a more natural way of programming, similar to object orientated programming. In pseudo code we access these exposed variables using the common '.' notation. Of course, the usage of such variables is common and thus this section shall only proof that the usage of them is actually feasible in practice for swarm robots. To do so, we propose an implementation for fault-prone communication, which has been tested in simulation.

Each robot can access the exposed variables of its neighbors and modify its own. A robot broadcasts its exposed variables as often as it can. If a robot retrieves such messages, it updates the exposed variable of the corresponding local neighbor view. If an exposed variable of a neighbor has not been updated for a specific time, it is set to undefined. We assume that either all or none exposed variable of a neighbor are updated. As we cannot guarantee the currency of the variables, we can at least assume their coherence.

The exposed variable system can be implemented using unreliable asynchronous messages. Each robot r is regularly sending messages $\{r.v|v \text{ is an exposed variable}\}$ with all their exposed variables as broadcast to its neighbors. We call these messages *synchronization packages*. If a robot r receives such *synchronization package* from robot n, it updates its neighbor view of n and also saves the local time. Before a robot uses an exposed variable of a neighbor, it checks if the time since the last synchronization is within a limit. This limit can be set individually for each exposed variable, depending on how up to date the variable has to be for the algorithm and how often the synchronization is expected to fail.

2.3 Angles

Some algorithms of this thesis need angle measurements between 2D-vectors. Angles are measured clockwise from the first to the second vector. The unit is radian for most of the time but we also use degrees. All passed angles are normalized to $[0, 2\pi]$ for radian or $[0°, 360°]$ for degrees. While using arithmetic operations, however, there is no normalization (e.g. $350° + 100° = 450°$ and is not normalized to $90°$).

The angle between two directions $v, v' \in \mathbb{R}^2$ is defined as

$$\measuredangle(v, v') = \text{atan2}(v * v', \det(v, v'))$$

For a robot $r \in \mathcal{R}$ the angle between two neighbors $n, n' \in N(r)$ is denoted by

$$\measuredangle_r(n, n') = \text{atan2}(p_r(n) * p_r(n'), \det(p_r(n), p_r(n'))$$

If the result shall be in radian or degree is context sensitive and apparent, as we never mix these units.

3 Flocking and Grid

Flocking is the behavior of individuals in a swarm to move in a group. This is known from biology by fish schools, bird flocks or different insect swarms. These swarms can also have special formations, like the V-formation of some bird flocks. Most *swarms* in biology do not have leaders, the movement direction is determined by consensus. Within the swarm, the individuals can move freely but should avoid collisions and stay close enough to their neighbors. If a lot of individuals move southwards in the swarm because they sensed a predator in the north, the chances are high that this local movement will propagate to the swarm movement and make the whole swarm move southwards.

The first to implement artificial flocking behavior was Reynolds [53] in 1987. He proposed the following three heuristic rules for flocking, which are used for the most flocking algorithms.

1. Flock Centering: Attraction to neighbors
2. Collision Avoidance: Repulsion from too close neighbors
3. Velocity Matching: Adaptation of the velocity of neighbors

These rules do only require local information and are therefore independent of the swarm size.

In this section we consider the used flocking algorithm which leads to an equilateral triangle grid (of configurable length) and movement consensus. Section 3.1 explains the *algorithm 1* of Olfati-Saber ([48]), which we use in a slightly modified version given in Sec. 3.1.1.

3.1 Algorithm

The *algorithm 1* of Olfati-Saber [48] is a stateless equation based on potential fields. It is the first of three proposed algorithms and considers the regular fragmentation and the consensus. The regular fragmentation forms a pattern of equilateral triangles, which we simply call *grid*. The desired length γ of the edges in the grid can be specified individually for each swarm, but it has to be within the robots range.

Definition 14 (Desired Neighbor Distance)**.** *The arbitrary but fixed desired neighbor distance is denoted by* $\gamma \in (0, \Gamma]$ *and matches the desired grid size.*

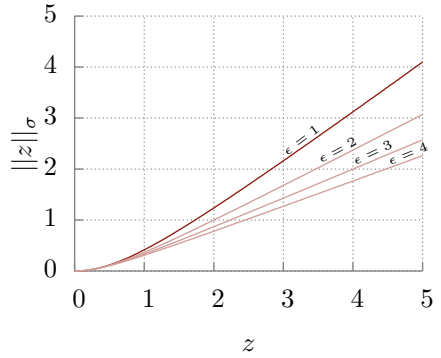
It is advisable to set the length γ long enough, so that each robot in grid with edge length γ is only neighbored to exact six robots. Otherwise, a clean grid, where all edges are part of the grid, is not possible. With consensus is meant that a robot tries to move in the same direction as its neighbors. The further two proposed algorithms of Olfati-Saber extend the *algorithm 1* by obstacle avoidance and a global distributed leader, but do not match our application.

A short version of the *algorithm 1*, which calculates the acceleration for the robot, is given as follows:

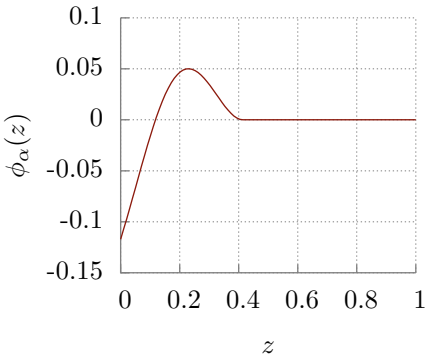
Algorithm 1: Algorithm 1 of Olfati-Saber [48]

$$\mathrm{OS} : \mathcal{R} \to \mathbb{R}^2, r \mapsto \underbrace{\sum_{n \in N(r)} \phi_\alpha(||p_r(n)||_\sigma) * \mu(p_r(n))}_{\text{Grid}} + \underbrace{\sum_{n \in N(r)} \nu(p_r(n)) * \dot{p}_r(n)}_{\text{Consensus}}$$

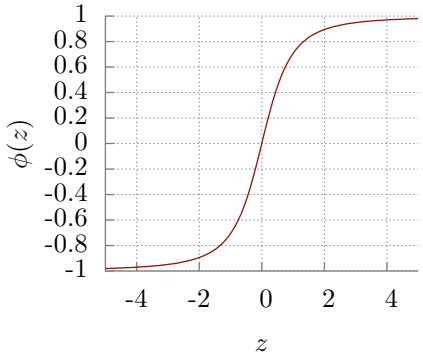
As in the previous section, $p(r)$ denotes the position, $\dot{p}(r)$ the velocity, $N(r)$ are the neighbors of robot r, and the range of a robot is denoted by Γ. All further notations are only for this section and possibly used with different semantic in the later sections.



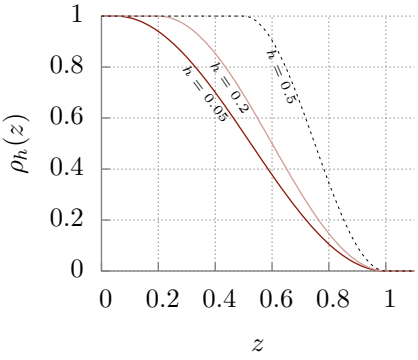
(a) $||z||_\sigma$ for $z \in \mathbb{R}$ and $\epsilon \in \{1, 2, 3, 4\}$.



(b) ϕ_α with $\Gamma = 1, \gamma = 0.5, \epsilon = 1$. $||\gamma||_\sigma = 0.118$ and $||\Gamma||_\sigma = 0.414$



(c) ϕ for $a = 1, b = 1, c = 0$



(d) ρ_h for $h \in \{0.05, 0.2, 0.5\}$

Figure 5: Plots for *algorithm 1*

The grid is formed by repulsion of too close neighbors and attraction of too distant neighbors.

$$||z||_\sigma = \frac{1}{\epsilon}(\sqrt{1 + \epsilon||z||^2} - 1)$$

is a special norm, which is differential at $z = 0$ and smooths the regular norm. $\epsilon > 0$ is a tunable constant, which we have set to $\epsilon = 1$. A plot for $z \in \mathbb{R}$ is given in Fig. 5a. The function

$$\phi_\alpha : \mathbb{R} \to \mathbb{R}, z \mapsto \rho_h(z/||\Gamma||_\sigma)\phi(z - ||\gamma||_\sigma)$$

determines if there is a repulsion (negative) or attraction (positive) between two robots and how strong it is to reach the desired overall grid length. This is based on the relative distance in terms of the range between two robots by ρ_h and the difference to the desired grid length by ϕ.

$$\phi : \mathbb{R} \to \mathbb{R}, z \mapsto \frac{1}{2}[(a + b)\frac{z + c}{\sqrt{1 + (z + c)^2}} + (a - b)]$$

is a sigmoidal function and used to smooth the attraction/repulsion force with the tunable constants ($\in \mathbb{R}$) $a > 0$ and $b > a$, while $c = |a - b|/\sqrt{4ab}$. For this thesis we used $a = 1$ and $b = 1$, and thus $c = 0$. As input it becomes the difference to the desired grid length. First it

raises fast to quickly optimize the small difference and slows down later, so a greater difference is not able to overwrite all other forces. The attraction strength converges to a and the repulsion force to b. $\rho_h(z)$ is a function that goes smoothly between $\rho_h(0) = 1$ and $\rho_h(1) = 0$ using the cosine.

$$\rho_h : \mathbb{R} \to \mathbb{R}, z \mapsto \begin{cases} 1, & z \in [0, h) \\ \frac{1}{2} * [1 + \cos(\pi \frac{z-h}{1-h})], & z \in [h, 1) \\ 0, & \text{otherwise} \end{cases}$$

The tunable constant $h \in (0, 1)$ determines the point from which the function begins to fall. A plot of ϕ_α is given in Fig. 5b with ϕ in Fig. 5c and ρ_h in Fig. 5d.

$$\mu : \mathbb{R}^2 \to \mathbb{R}^2, q \mapsto \frac{q}{\sqrt{1 + \epsilon ||q||^2}}$$

gives the neighbor direction and thus the direction of the force to improve the distance to the neighbor. The strength of this force is defined by ϕ_α. ϵ is identical to the ϵ in $||z||_\sigma$.

The consensus is reached by adapting the movement of the neighbors, where the influence is determined by the distance.

$$\nu : \mathbb{R}^2 \to \mathbb{R}, q \mapsto \rho_h(||q||_\sigma / ||\Gamma||_\sigma) \in [0, 1], i \neq j$$

is a weighted adjacency function used for the influence and is based on the distance. The function ρ_h has been defined above.

3.1.1 Implementation

In experiments we noticed that the force of the consensus part can grow due to slow synchronizations. If a motionless robot notices n neighbors moving into the same direction, it accelerates with n times their speed difference into this direction to catch up. However, if the robots doesn't slow down fast enough, the other robots become faster due to the consensus force. Therefore, we use the average consensus and not its sum, so that a following robot is never moving faster than the robots it follows.

Algorithm 2: Flocking Algorithm

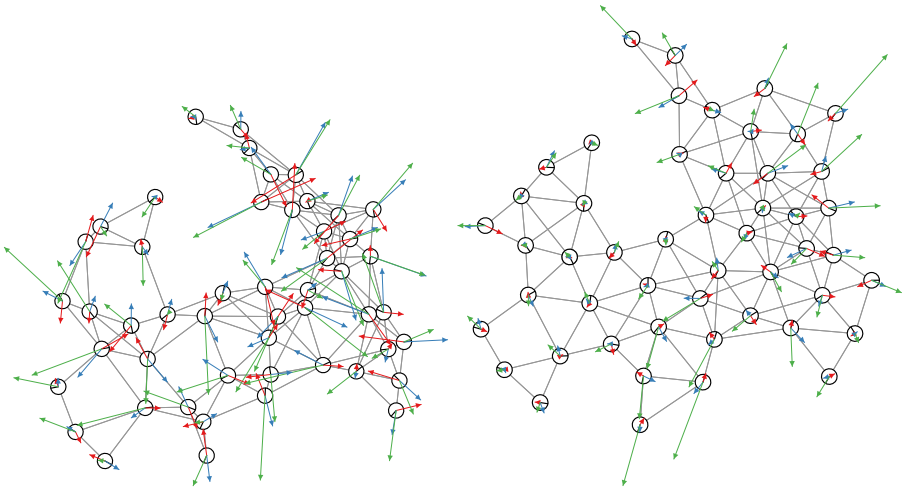
$$\text{flockAlg} : \mathcal{R} \to \mathbb{R}^2, r \mapsto \underbrace{\sum_{n \in N(r)} \phi_\alpha(||p_r(n)||_\sigma) * \mu(p_r(n))}_{\text{Grid}} + \underbrace{\frac{1}{|N(r)|} \sum_{n \in N(r)} \nu(p_r(n)) * \dot{p_r}(n)}_{\text{Consensus}}$$

Please remember that the output vector of Alg. 2 is an acceleration, in contrast to the most other algorithms in this thesis. To gain a velocity the acceleration times the length of a time step has to be added to the robot's velocity.

Fig. 6 shows the algorithm at different times in a simulation with the local algorithm. It slows down the closer it comes to an optimal grid.

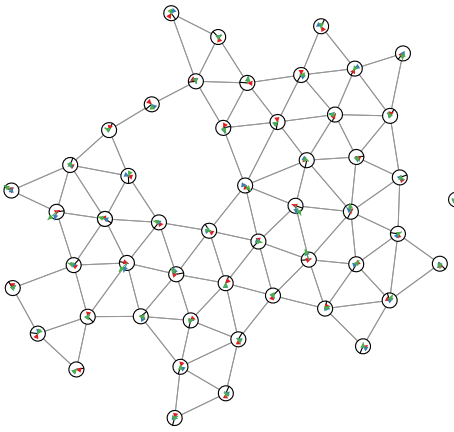
4 Boundaries

The boundaries are of fundamental importance for the geometric and topological self-awareness of the swarm. They define the shape of the swarm and determine if and where there are holes in the swarm configuration, which have to be filled. Additionally, they allow calculating the perimeter

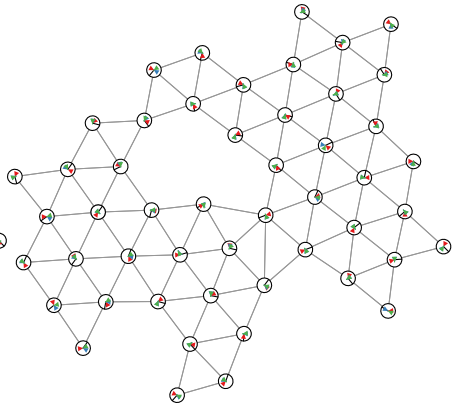


(a) 0s

(b) 2s



(c) 7s



(d) 15s

Figure 6: Blue shows the grid acceleration, red the consensus acceleration and green the combined velocity. The times are not representative as they strongly depend on the simulation.

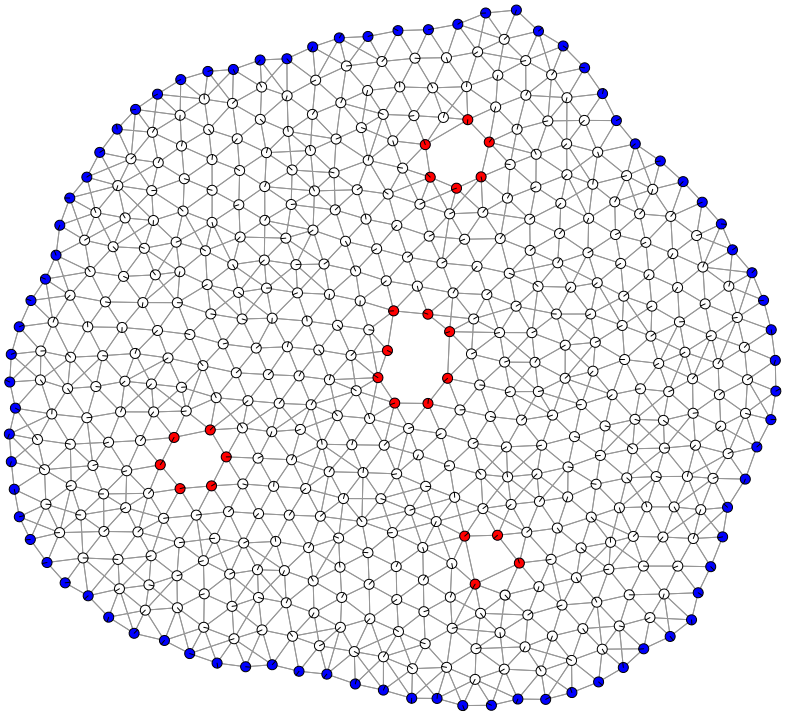


Figure 7: This swarm configuration has some holes margined by interior boundaries visualized in red. The exterior boundary is visualized in blue.

as well as detect objects entering the swarm and local articulation points[1]. A boundary denotes a set of robots lying to a cohesive robot free area but is not well defined. If this robot free area is finite, we call the boundary interior otherwise we call it exterior boundary. An interior boundary is also called hole. Fig. 7 shows a swarm with some holes, where a possible selection of robots for the boundaries is marked.

We try to define the boundary as circle of robots and give local properties for robots within this boundary in Sec. 4.1. A local algorithm for boundary detection is given in Sec. 4.2. Based upon the detected boundary, we give a method to determine the type and size of the boundary is Sec. 4.3. This method uses the average angle of the polygon induced by the boundary. An algorithm to determine the average angle of the boundary is given in Sec. 4.4. Finally, we propose a force that acts upon the boundary similar to the boundary tension of water drops in Sec. 4.5.

4.1 Boundary Definition

A common definition of the boundary is the α-shape of Edelsbrunner et al. [18], which also provides a centralized offline algorithm. In a nutshell it defines each vertex as lying on a boundary, that touches a robot free circle of radius $1/\alpha$. For a unit disc graph with range r, we receive a mostly matching boundary for $1/\alpha$ is minimal greater than $r/2$. As our swarm is only a subgraph of the according unit disc graph, this definition may miss some boundary nodes or even complete interior boundaries. Even for unit disc graphs no fully satisfying α can be found, as discussed

[1]Local articulation points are a superset of the global articulation points, which disconnect the swarm on failure. If a local articulation is not a global articulation point it connects two different boundaries on failure.

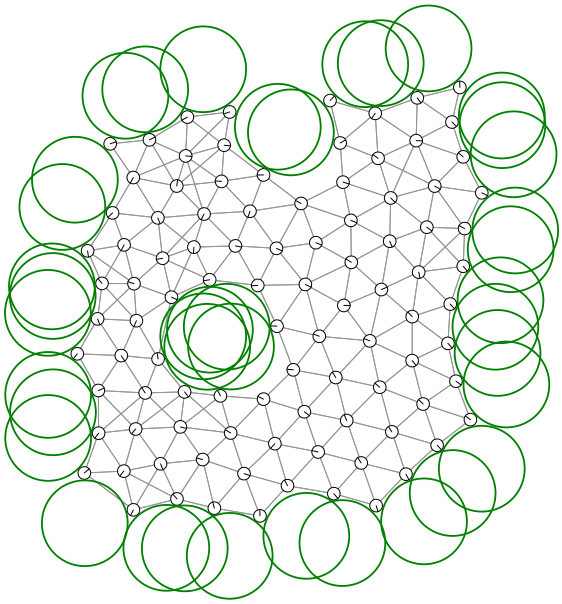


Figure 8: The α-shape defines a boundary based on robot free circles with the diameter little above the robot range.

by McLurkin and Demaine [44]. An example of the alpha shape can be seen in Fig. 8 and an example where no matching α for the alpha shape can be found in Fig. 9.

We use the shape definition of [44] to differentiate between covered and free area. Due to simplicity we assume that every robot is part of at least one C_3 subgraph in the communication graph. Robots which do not abide this are treated separately by adding infinite narrow areas on the edges.

Definition 15 (Covered/Free Area)**.** *The covered area is the union of the finite areas of all C_3 subgraphs and infinite narrow areas of the edges within the communication graph. Other cohesive areas are called free areas. The infinite free area is called open area and the other ones are called holes.*

In Fig. 10 the C_3 areas are filled in blue to visualize the shape. Even so we have a definite

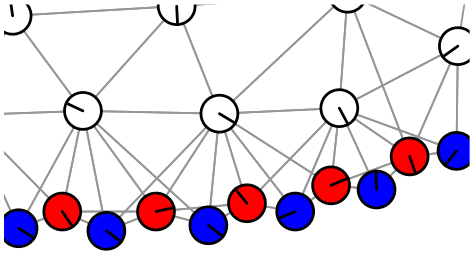


Figure 9: The α-shape would detect the blue robots as boundary and the red robots as interior, but the red robots would also be needed for a connected boundary. If we lower the radius for also detecting the red robots, larger triangles would be detected as boundary.

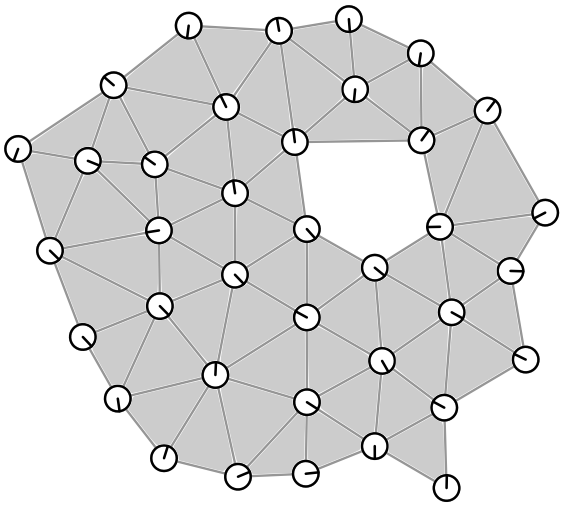


Figure 10: The C_3 areas are filled with grey to visualize the shape.

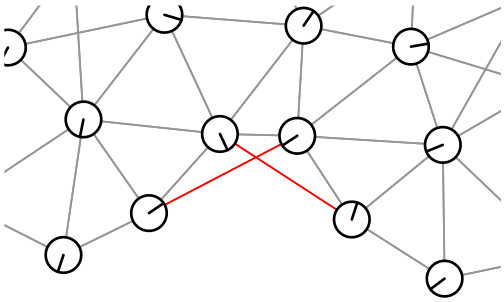


Figure 11: The crossing edges (red) make the definition of the boundary difficult.

covered area, we can have multiple possibilities to chose robots for the boundaries. Fig. 11 shows a case where for the robots with the crossing edges we have three possibilities: Use both, use only the left, or use only the right one. There are also more complicated cases which are less intuitive. However, we can at least determine potential candidates for the boundary. For this we need the term of an open sector.

Definition 16 (Sector)**.** *A sector $(n, n')_r$ consists of a base robot r, a neighbor $n \in N(r)$ and the clockwise next neighbor $n' = \text{NextClockwiseNbr}_r(n)$. The area of a sector $(n, n')_r$ is a circle fragment of the range circle of r delimited counterclockwise by n and clockwise by n'. The angle of a sector $(n, n')_r$ is defined as*

$$\measuredangle((n, n')_r) = \begin{cases} \measuredangle(p_r(n), p_r(n')) & n \neq n' \\ 360° & \textit{else} \end{cases}$$

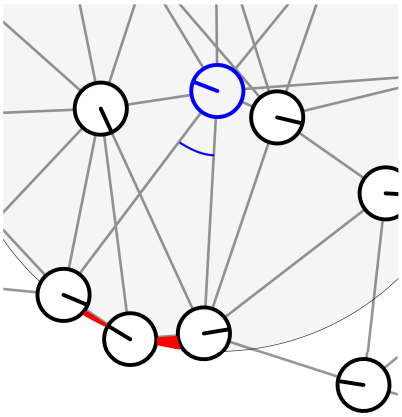


Figure 12: The robot marked in blue is a bad candidate for the boundary but for too small α it has an open sector. The free area intersection is highlighted in red.

Definition 17 (Open Sector)**.** *For an arbitrary but fixed $\alpha \in (0, 1]$, we call a sector $(n, n')_r$ open if*

1. *it is intersecting a free area with at least α of its area. (Intersection Condition)*
2. *$n \not\sim n'$ or $\measuredangle((n, n')_r) \geq 180°$. (Neighborhood Condition)*

The set of all open sectors of a robot r is denoted by OpenSectors(r)

The α is needed for an edge case evoked by the line of sight constraint and thus depends on robot radius and range. Fig. 12 shows such a case, where a robot only slightly cuts the free area. The value of α has to be chosen as small as possible, as for cases like in Fig. 11 robots needed for a connected boundary have open sectors, that only partly intersect free area. If the neighborhood condition is not caused by inaccurate communication or sensing, it is sufficient for most cases. There are cases where the neighborhood condition is kept but the intersection condition is not, like the one shown in Fig. 13, but due to the flocking algorithm they should rarely arise and quickly disappear. We call sectors, which keep only the neighborhood condition, as *false* open sectors. Since the intersection condition is quite difficult to check locally in contrast to the neighborhood condition, we are only using the neighborhood condition with some additional heuristics.

Definition 18 (Boundary Candidate)**.** *A robot is a boundary candidate if it has an open sector. If a robot is no boundary candidate, it is called an interior robot.*

We believe that for a well chosen α a reasonable boundary connecting all the associated boundary candidates can be found. However, it is not always clear how to connect them to form a circle, as a boundary candidate can have more than two neighbors which are boundary candidates too.

4.2 Boundary Detection

There are already a lot of algorithms for local and distributed detection of boundaries in sensor networks. Wang et al. [67] classified those algorithms into geometric, statistical and topological methods. Geometric methods use the most knowledge, as they know about the relative position

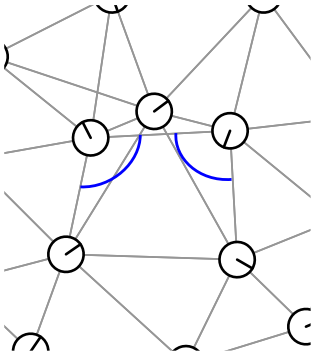


Figure 13: The second condition does not always imply boundary. Two false open sectors marked in blue.

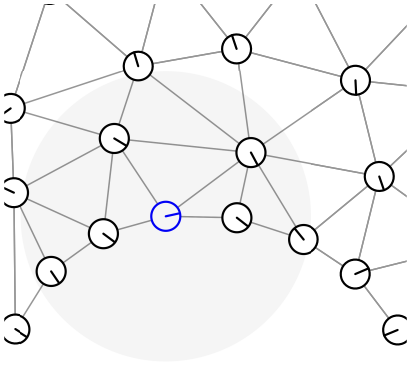


Figure 14: The assumption of a unit disc graph would lead to not detecting the boundary. The two boundary neighbors of each robot are close enough and have an angle below 180°.

of their neighbors (e.g. [44], [20]). This method can work without communication. The statistical algorithms (e.g. [21]) use statistical properties of boundary vertices for randomly deployed sensor nodes. As swarm robots are not randomly deployed, this method does not match our use case. The topological algorithms work only with connectivity information and local communication (e.g. [34]). The needed communication makes it scale bad over big and quickly changing robot swarms. Since we know the relative positions of neighbors, we use a geometric method. More concrete we extend the algorithm of McLurkin and Demaine [44], which is the only work we know considering boundary detection for swarm robots. The original algorithm is designed for unit disc graphs and can infer edges between neighbors only based on their position and without communication. With the additional line of sight constraint, this does not work for us as Fig. 14 shows. Not all robots standing in the line of sight of two others can be perceived by the considered robot, thus the edges between neighbored robot need to be estimated by communication.

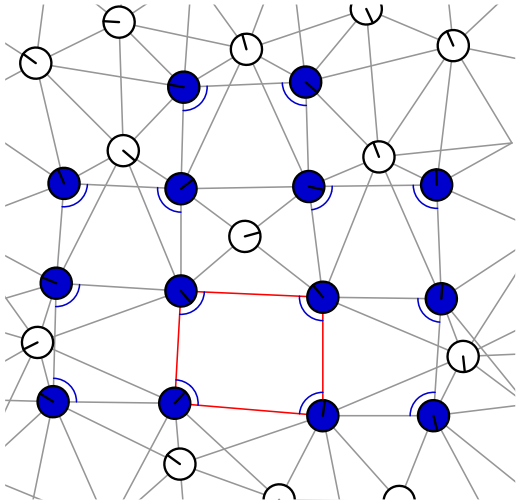


Figure 15: All blue robots are detected as boundary with Obs. 1 but only the red highlighted circle is a true boundary.

4.2.1 Algorithm

It is difficult to check the intersection condition (Def. 17) of an open sector. This is why we use a heuristic algorithm, which can quickly detect open sectors and therefore boundary candidates (but can have false positives). As the neighborhood condition can be checked easily, the main problem is to distinguish false open sectors from real open sectors. The algorithm is based on the following observation, which emerges from the circular nature of boundaries.

Observation 1. *If $(n, n')_r$ is an open sector of robot r, then also the robots n and n' do have open sectors.*

This observation is not sufficient as we can find configurations where this is also true for false open sectors, but for our use case the accuracy in simulations showed to be good enough. An example where Obs. 1 fails is given in Fig. 15.

The routine to refresh the open sectors and the boundary candidate state of a robot r is given in Alg. 3. The routine is assumed to be atomar such that there are no accesses to the public variables of r during its execution. First it removes the old open sectors and resets the states. Then it iterates over all sectors and checks if the second condition for open sectors is fulfilled. If it has such a sector, the robot is denoted as open sector candidate, which means that it has at least a false open sector. Next it is checked if the neighbors of this possibly open sector do also have possibly open sectors in reference to Obs. 1. If this is true, it denotes itself as having matching neighbors. If this is also true for the neighbors of the sector, the sector is assumed to be an open sector and added to the set. After refreshing the open sector set, the boundary candidate state is set. We do not only check if the robot has an open sector but also if it has an open sector for some time, as some fluctuations may be common with fault-prone communication. If the robot has an open sector for longer than a specified time, the boundary candidate state is set to true.

Algorithm 3: Boundary Detection

```
Input: Robot r
OpenSectors(r) ← ∅
r.isBoundaryCandidate ← False
r.isOpenSectorCandidate ← False
r.hasMatchingNbrs ← False
for n ∈ N(r) do
    n' ← NextClockwiseNbr_r(n)
    if n ⊁ n' or ∡((n, n')_r) ≥ 180° or n = n' then
        r.isOpenSectorCandidate ← True
        if n.isOpenSectorCandidate and n'.isOpenSectorCandidate then
            r.hasMatchingNbrs ← True
            if n.hasMatchingNbrs and n'.hasMatchingNbrs then
                Add(OpenSectors(r), (n, n')_r)
if OpenSectors(r) ≠ ∅ then
    if r.isCandidateSince = ⊥ then
        r.isCandidateSince ← localtime(r)
    if localtime(r) − r.isCandidateSince ≥ DELAY then
        r.isBoundaryCandidate ← True
else
    r.isCandidateSince ← ⊥
return
```

4.3 Size and Type

The boundary should be treated based on its size and type. For example large holes have to be closed, while small holes often close themselves on their own. McLurkin and Demaine [44] proposed a method to determine the type of the boundary and there are efficient methods for counting the number of unique values (IDs) like [22], that could be used to estimate the size of a boundary. However, as the boundary can change quickly, the existing distributed counting methods fail because they do not allow the removal of already counted items. We propose a method to estimate the size and type of the boundary, that is based on the average angle of the open sectors.

Due its definition based on triangles, a boundary can be seen as a polygon with the boundary candidate robots as vertices. If we take a look at circles with different amounts of robots, we notice, that the inner angle of the open sectors grows with the number of robots. Fig. 16 shows a circle of six robots and one of ten. The inner angle of the left circle is 120° while for the right it is 144°. Even so the circle is the clearest example, for any polygon the average angle is increasing with the amount of vertices.

The sum of the interior angles of an arbitrary n-polygon is given by $(n-2)*180°$ for $n > 2$ and is thus only dependent on the amount of points in it. Further, the average interior angle can be determined by $\frac{n-2}{n}*180°$. As this formula is monotone rising for $n > 2$ and $\lim_{n\to\infty} \frac{n-2}{n} = 1$, the average interior angle is always below 180°. The sum of the outer angles can be given analogously by $n * 360° - (n-2) * 180°$ and the average exterior angle by $360° - \frac{n-2}{n} * 180°$. Thus, the average exterior angle is always above 180°, which gives us the possibility to determine if the average angle is for interior or exterior. This is important as the boundary angles of the exterior boundary are exterior angles and those of the interior boundary are interior angles. Based upon this we can determine if a boundary is exterior or interior only upon the average boundary angle.

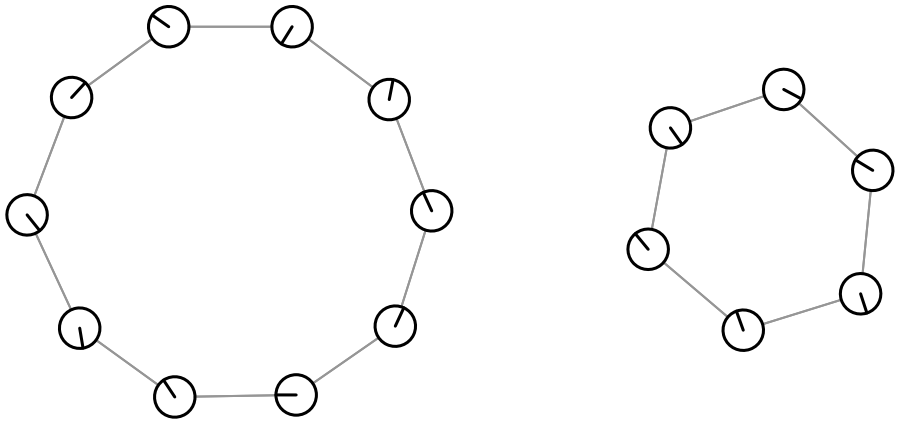


Figure 16: The (interior and exterior) angles of the big circle are closer to 180° than those of the small circle.

Theorem 1. *Let θ be the average boundary angle, then the boundary is interior if and only if $\theta < 180°$. If the boundary is not interior, it is exterior.*

Proof. Follows from the previous paragraph and that the boundary has to be either interior or exterior. □

As the average angle of a polygon only depends upon the vertices within it, we can further determine the amount of robots based solely on the average boundary angle.

Theorem 2. *Let θ be the average boundary angle in degrees, then we can determine the amount of robots in the boundary by*

$$\text{bsize}(\theta) = \begin{cases} \frac{2}{1-\theta/180°} & \theta < 180° \\ \text{bsize}(360° - \theta) & \textit{else} \end{cases}$$

Proof. $\theta = \frac{n-2}{n} * 180° \Rightarrow n = (n-2) * \frac{180°}{\theta} \Rightarrow 2 * \frac{180°}{\theta} = \frac{180°}{\theta} n - n \Rightarrow 2 * \frac{180°}{\theta} = (\frac{180°}{\theta} - 1) * n \Rightarrow n = \frac{2}{1-\theta/180°}$. The average interior angle can be determined from the average exterior boundary by subtracting it from 360° as the sum of interior and exterior angle is 360°. □

4.4 Average Angle

The distributed calculation of the average is an intensively researched topic as it is needed for consensus problems or load balancing. There are a lot of efficient protocols like [45] or [70]. The protocol of [64] does even support topology changes. The main problem, why we cannot use them, is that they do not support the change of values or removal of individuals. We would need to recalculate the average frequently but it takes time to iterate over all individuals and we did not find a way to do multiple calculations at the same time. Therefore, we are using a local heuristic, which is fast but possibly not always accurate.

A naive approach is to simply get the average of the open sector neighbors and merge them with the own value. However, this leads to feedback as the these neighbors are doing the same. Therefore, we are directing the influence such that each robot does not influence itself more than it is influenced by the other neighbors. The base idea is to determine two reverse directed circles upon each boundary. For each circle we do a separate average calculation such that each robot

is only influenced by the incoming neighbor. The two averages are averaged again to obtain the boundary average. The two circles are not only needed to speed up the calculation but also to integrate all robots, as our method does not always generate real circles. Therefore, we do not call them circles but boundary graphs.

As a robot can be part of multiple boundaries and thus have multiple open sectors, we do not build the boundary graphs upon the robots but upon the open sectors. The virtual open sector nodes are managed by their owner robots. Each robot publishes its open sectors as exposed variables and chooses the predecessor open sectors for each of its own open sectors from its neighborhood. It does neither determine nor know the successors since they are not necessary. Beside the open sectors each robot also has to publish the matching values for the calculation. These are the actual average value and the hop count. The hop count is incremented with each hop if it is below an upper bound. It is used to determine the influence of the actual iteration. The upper bound limits the accuracy for large boundaries, but quickens the adaption to changes in long lasting boundaries.

Definition 19 (Boundary Graph)**.** *The counterclockwise boundary graph is a directed graph* $D = (V, A)$ *with the vertex set*

$$V = \{(n, n')_r \in \text{OpenSectors}(r) | r \in \mathcal{R}\}$$

and the arc set

$$A = \{((m, m')_n, (n, n')_r) | r \in \mathcal{R} \wedge (n, n')_r \in \text{OpenSectors}(r) \wedge (m, m')_n \in \text{OpenSectors}(n) \wedge \\ \forall (w, w')_n \in \text{OpenSectors}(n) : \measuredangle(p_n(m), p_n(r)) \leq \measuredangle(p_n(w), p_n(r))\}$$

The clockwise boundary graph is a directed graph $D' = (V, A')$ *with the vertex set*

$$V = \{(n, n')_r \in \text{OpenSectors}(r) | r \in \mathcal{R}\}$$

and the arc set

$$A' = \{((m, m')_{n'}, (n, n')_r) | r \in \mathcal{R} \wedge (n, n')_r \in \text{OpenSectors}(r) \wedge (m, m')_{n'} \in \text{OpenSectors}(n') \wedge \\ \forall (w, w')_{n'} \in \text{OpenSectors}(n') : \measuredangle(p_{n'}(r), p_{n'}(m')) \leq \measuredangle(p_{n'}(r), p_{n'}(w'))\}$$

Figure 17 shows the clockwise boundary graph of a swarm configuration with a leaf and two not definite boundary robots.

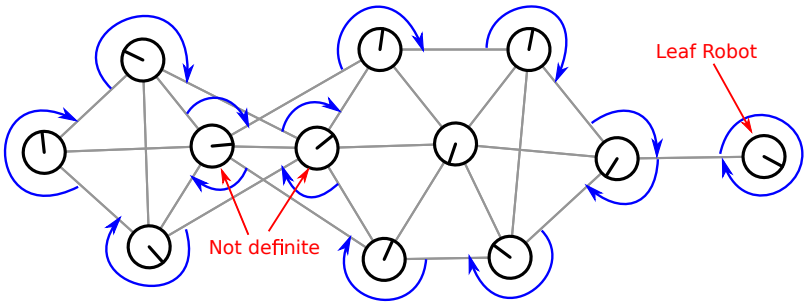
We can assume that each angle is unique amongst the open sectors at one robot, otherwise they are additionally ordered by their neighbor IDs. Thus, each open sector has exactly one incoming arc, as the definition of the boundary graphs allows at most one and the definition of an open sector requires at least one.

Definition 20 (Boundary Predecessor)**.**

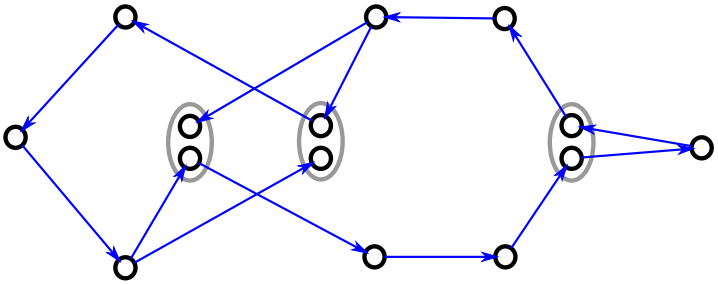
$$\text{bpred}_D((n, n')_r) = (m, m')_s \in N_D((n, n')_r)$$
$$\text{bpred}_{D'}((n, n')_r) = (m, m')_s \in N_{D'}((n, n')_r)$$

The Alg. 4 maintains tuples (a, i) of an angle a and a hop count i for each open sector at the robot r. There is one tuple for each open sector and each direction (cw as well as ccw). The angle a stands for the average angle of the boundary over the $i - 1$ predecessors of the open sector and the open sector itself.

In the beginning any previously computed values for the average boundary angle are cleared. Then for each open sector of the robot r, first the tuple for the counterclockwise average is



(a) The configuration of the swarm with open sectors visualized with blue arrows.



(b) The clockwise boundary graph of the configuration. If a robot has multiple open sectors, they are clustered in grey. Due to the horizontal symmetric property of the configuration, the counterclockwise boundary graph equals the flipped horizontal clockwise.

Figure 17: A swarm configuration and the resulting clockwise boundary graph.

calculated and afterwards for the clockwise. For calculating such a tuple, first the predecessor tuple from the matching boundary graph is token. If it is defined, we increment i by one and merge the average of the tuple with the open sector angle weighted by the hop count i. This tuple is our new tuple for this open sector and the actual boundary graph (cw or ccw). If the predecessor tuple is not defined, we simply use the open sector angle as average value for the new tuple with a hop count of one. This routine is seen as atomar, such that the exposed variables are only synchronized after it has finished.

The average of an open sector is calculated upon the clockwise and counterclockwise average weighted by their hop counts. Let a be the clockwise average and i its hop count, analogously a' and i' for the counterclockwise direction.

$$\text{osavr}((n,n')_r) = \frac{i}{i+i'} * a + \frac{i'}{i+i'} * a'$$

The average angle of a robot is the mean of all averages of its open sectors

$$\text{bavr}(r) = \sum_{(n,n')_r \in \text{OpenSectors}(r)} \frac{1}{|\,\text{OpenSectors}(r)|} * \text{osavr}((n,n')_r)$$

Obviously there are many possibilities for the algorithm to fail if the communication and sensing is inaccurate. Some open sectors may be undetected such that the matching of the open sectors fails, which leads to a local resetting of the average or a wrong value. Smoothing over the previous values can help to reduce this issue but no guarantee can be given. However, this is still the fastest method with a mostly accurate average known to us.

Algorithm 4: Circle Average

$r.\text{CounterClockwiseAvr} \leftarrow \emptyset$
$r.\text{ClockwiseAvr} \leftarrow \emptyset$
for $(n,n')_r \in \text{OpenSectors}(r)$ **do**
  $(m,m')_n \leftarrow \text{bpred}_D((n,n')_r)$
  $(a,i) \leftarrow n.\text{CounterClockwiseAvr}[(m,m')]$
  if $(a,i) \neq \perp$ **then**
    $i \leftarrow \min\{i+1, \text{maxhops}\}$
    $r.\text{CounterClockwiseAvr}[(n,n')] \leftarrow (\frac{1}{i}\measuredangle(p_r(n), p_r(n')) + \frac{i-1}{i}a, i)$
  else
    $r.\text{CounterClockwiseAvr}[(n,n')] \leftarrow (\measuredangle(p_r(n), p_r(n')), 1)$
  $(m,m')_{n'} \leftarrow \text{bpred}_{D'}((n,n')_r)$
  $(a,i) \leftarrow n'.\text{ClockwiseAvr}[(m,m')]$
  if $(a,i) \neq \perp$ **then**
    $i \leftarrow \min\{i+1, \text{maxhops}\}$
    $r.\text{ClockwiseAvr}[(n,n')] \leftarrow (\frac{1}{i}\measuredangle(p_r(n), p_r(n')) + \frac{i-1}{i}a, i)$
  else
    $r.\text{ClockwiseAvr}[(n,n')] \leftarrow (\measuredangle(p_r(n), p_r(n')), 1)$

4.5 Boundary Force

The boundary force is a simple force proposed by Lee [37], which mimics the boundary tension of water drops. It minimizes the boundary by pulling convex boundary robots in and pushing

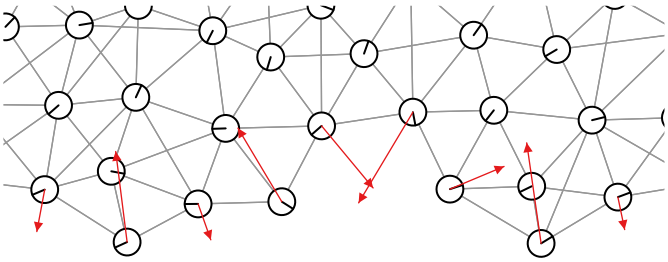


Figure 18: The boundary force

concave boundary robots out, as can be seen in Fig. 18. This straightens the exterior boundary by moving the angle of a boundary robot towards 180°. A useful property of the boundary force is that it closes holes. Together with the flocking algorithm Alg. 2 it produces circle-shaped swarms. However, it has to be weighted carefully so that the boundary force is not compressing the interior too much, but still has enough power for sculpture the swarm.

Let $(n, n')_r$ be an open sector of robot r. We can derive the boundary force by adding up the local positions to $p_r(n) + p_r(n')$. This vector points to the position between the two neighbors and can be used as a steering vector. As the force strength depends on how distanced it is to the optimal position, it automatically slows down as it comes closer. However, as the boundary tension tries to reach an average degree of 180°, which can not be fulfilled, the boundary force emphasis depends strongly on the size of the swarm. Even if the swarm reached an optimal cycle, the boundary force is pressing inside the stronger the smaller the swarm is. For a fixed emphasis of the boundary force, we can not vary the swarm size very much without getting a too weak or too strong boundary force.

As we have a heuristic for the size of the boundary, we can weight the boundary force depending on the boundary size. Also, we can give different weights for interior or exterior boundaries, as the boundary force for interior boundaries can not compress robots but only not gain enough reinforcements (robots moving up). If we want to generate circle shaped swarm, we suggest to weight the exterior boundary force depending on the difference to the average angle. This stops the boundary force in already perfect circles from compressing the robots. However, we want to create more generic shapes, where a difference from the average angle might be desirable, as for example for the shape in Fig. 19. Yet, we see that for a smooth boundary, the angle is always similar to that of the neighbored boundary robots. Therefore, we use the local average instead of the global average for weighting the boundary force. A local average can be obtained with Alg.4 by choosing a small upper bound, which matches the amount of considered robots for each direction. The amount of neighbored robots, considered for this weighting is an important parameter. If it is to big, it corresponds to the global average with the mentioned problems. If it is to small, it removes bigger buckles only slowly. As we introduce additional forces later, which do also remove buckles, a smaller amount should be taken.

Let θ be the local average angle, $\phi : \mathbb{R} \to \mathbb{R}$ which wights the force of the interior boundary based upon the deviation from θ, and $\phi' : \mathbb{R} \to \mathbb{R}$ which does the same for the exterior boundary.

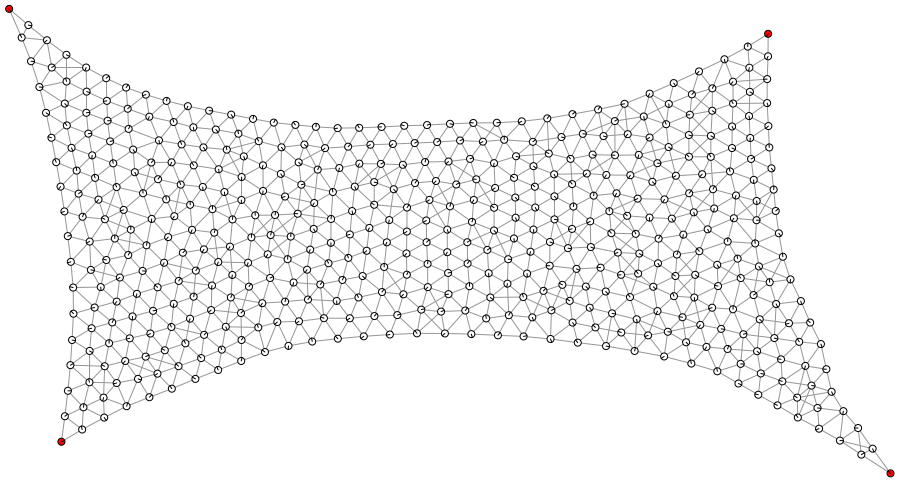


Figure 19: A circle shape is not possible for this configuration if the leader robots (red) are not moved inwards. Bad weighted boundary forces would create multiple holes and make the swarm unstable.

The boundary force is the sum of all open sector forces.

$$\text{dynamicOpenSectorForce} : \mathcal{O} \to \mathbb{R}^2,$$
$$(n, n')_r \mapsto \begin{cases} (p_r(n) + p_r(n')) * \phi(|\theta - \measuredangle((n, n')_r)) & \text{part of interior boundary} \\ (p_r(n) + p_r(n')) * \phi'(|\theta - \measuredangle((n, n')_r)) & \text{part of exterior boundary} \end{cases}$$

Algorithm 5: Dynamic Boundary Force

$$\text{dynamicBoundaryForce} : \mathcal{R} \to \mathbb{R}^2, r \mapsto \sum_{\substack{(n,n')_r \\ \in \text{OpenSectors}(r)}} \text{dynamicOpenSectorForce}((n, n')_r)$$

We chose a constant weighting for ϕ, as interior boundaries do not compress the swarm, and a quadratic[2] one for ϕ'.

5 Pheromones

In nature, pheromones are chemical substances used for intraspecific stigmergic communication. They are secreted into the environment to be perceived by conspecifics where they trigger behavioral or developmental changes. For example the bark beetle uses aggregation pheromone to concentrate on a single tree [66] and ants use trail pheromones to mark paths to food sources in their environment (see Figure 20)[26].

In the context of swarm robotics we consider virtual pheromones as a means to determine the distance and direction towards certain regions of interest in the swarm. For technical reasons the stigmergic communication scheme is omitted in favor of broadcast messages within the swarm and pheromones which are bound to the robots instead of the environment. If a robot considers

[2]In the meantime we observed that other weightings may perform better, but our experiments use the quadratic.

Figure 20: Ants following a pheromone trail.

himself as part of a region of interest – like the boundary of the swarm or leader robots – it can emit a pheromone to inform his swarm mates.

In Section 5.1 we give a formal definition of the pheromones whose local computation is discussed in Section 5.2. Section 5.3 proposes a method to determine the direction of a pheromone based on its gradient. In Section 5.4 a volatile pheromone, that disappears faster than the normal pheromone is introduced. Finally, in Section 5.5 the properties of a pheromone induced broadcast network are analyzed.

5.1 Pheromone definition

The pheromone can be seen as a function over an embedded communication graph which consists of the robots and their communication links. It is chosen in such a way that the function value always represents the hop distance towards the next emitting robot. It therefore has local minima at the emitting robots and its gradient points towards the next emitting robot.

Definition 21 (Emitter set)**.** *$\mathcal{E} \subset \mathcal{R}$ is the set of robots emitting a pheromone.*

Definition 22 (Pheromone Function)**.** *Given a set of emitting robots $\mathcal{E}$ the pheromone function is defined as*

$$l : \mathcal{R} \mapsto \mathbb{N}_0$$
$$l(r) = \min_{e \in \mathcal{E}} d(e, r)$$

This pheromone then represents the hop distance towards the next emitting robot.

Using this pheromone function, we can derive the pheromone direction towards the next emitting robot.

Definition 23 (Pheromone Direction)**.** *The pheromone direction is the direction towards the next emitting robot based on the gradient of the pheromone function:*

$$\dot{l} : \mathcal{R} \mapsto \mathbb{R} \times \mathbb{R}$$
$$\dot{l}(r) = -(\frac{\partial l(r)}{\partial x}, \frac{\partial l(r)}{\partial y})$$

5.2 Local computation of the pheromone

In a swarm robotics context, these functions need to be computed solely based on local information and in the face of finite information propagation speed. Therefore, the following simple update rule for a robots local pheromone state is used.

Definition 24 (Local Pheromone Update Rule)**.** *At any time step the local pheromone state $d(r)$ of a robot r is computed using the following formula.*

$$d^{t+1}(r) = \begin{cases} 0 & r \in \mathcal{E} \\ \min\limits_{n \in N(r)} d^t(n) + 1 & else \end{cases}$$

This update rule only relies on local information about a robot's neighborhood. Since the number of neighbor robots is bounded by $O(1)$, the update rule can be executed in constant time.

Now it is important to see that this update rule actually does what it is supposed to do.

Theorem 3. *Given a monotonous set of emitting robots $\mathcal{E}$ and an unchanging communication graph of the robot swarm, the local pheromone update rule will compute the pheromone function for all robots after $O(n)$ time steps.*

Proof. We prove by induction that $d(r)$ is correct at time step t for all $r \in \mathcal{R}$ with $l(r) \leq t$ and $d(r') = \infty$ for all $r' \in \mathcal{R}$ with $l(r') > t$.

$t = 0$ At time step $t = 0$ is $d^t(r) = \infty$ for $r \in \mathcal{R} \setminus \mathcal{E}$ and $d^t(r) = 0$ for $r \in \mathcal{E}$ $(l(r) = 0 \Leftrightarrow r \in \mathcal{E})$ according to the initialization.

$t \to t + 1$**:** We differ three cases:

- $l(r) < t + 1$**:** $d^{t+1}(r) = l(r) \wedge \forall n \in N(r) : (d^t(n) = l(n) \vee d^t(n) = \infty) \Rightarrow d^{t+1}(r) = d^t(r) = l(r)$
- $l(r) > t + 1$**:** $(\forall n \in N(r) : l(r) > t) \Rightarrow (\forall n \in N(r) : d^t = \infty) \Rightarrow d^{t+1}(r) = \infty$
- $l(r) = t + 1$**:** $(l(r) = t+1) \Rightarrow (\exists n \in N(r) : l(n) = t \wedge \forall n \in N(r) : l(n) \geq t) \Rightarrow (\exists n \in N(r) : d^t(n) = t \wedge \forall n \in N(r) : d^t(n) = t \vee d^t(n) = \infty) \Rightarrow d^{t+1}(r) = t + 1 = l(r)$

Because the maximal length of any shortest path is $|\mathcal{R}| - 1$, $d^{|\mathcal{R}|-1}(r) = l(r)$ is true for any robot r. □

5.3 Approximation of the pheromone gradient

To compute the pheromone gradient and thereby the direction towards the closest region of interest, the finite difference method for irregular grids is used as described in [39]. To compute the pheromone gradient at any robot r, the Taylor series expansion of degree 1 around r is used:

$$l(n) = l(r) + (p(n) - p(r))_x \frac{\partial l(r)}{\partial x} + (p(n) - p(r))_y \frac{\partial l(r)}{\partial y}$$

for all neighbors n of the robot r. This results in a (possibly overdetermined) set of linear equations with the partial derivatives along x and y as unknowns, whose least squares solution indicates the gradient of the pheromone function:

$$\begin{pmatrix} (p(n_1) - p(r))_x & (p(n_1) - p(r))_y \\ (p(n_2) - p(r))_x & (p(n_2) - p(r))_y \\ \vdots & \vdots \end{pmatrix} \begin{pmatrix} \frac{\partial l(r)}{\partial x} \\ \frac{\partial l(r)}{\partial y} \end{pmatrix} = \begin{pmatrix} l(n_1) - l(r) \\ l(n_2) - l(r) \\ \vdots \end{pmatrix}$$

Because the resulting gradient is pointing uphill, but the regions of interest are found at the local minima, the resulting vector is simply negated. Some configurations cause undetermined matrices or matrices with deficient rank. For these cases the local mean location of any neighbor robots that are closer to the next emitter is chosen as a fallback direction vector. An example of a pheromone gradient computed by this method can be seen in Figure 21.

5.4 Volatile Pheromones

After a robot ceases to emit a pheromone, the signal still remains within the swarm for a certain amount of time. It can take up to $O(d)$ time steps until the pheromone signal vanishes from the swarm, where d are the amount of hops a pheromone value can do till the maximum value is reached.

To overcome this problem, a volatile pheromone is used to remove the pheromone values more quickly: Every pheromone carries as a payload the unique ID of its emitting robot and a time stamp to encode when it was emitted. The time stamp is relative to the emitters local clock, that is not synchronized with other robots clocks.

5.4.1 Implementation

The implementation, which is given in Alg. 6, uses the following exposed variables.

value The local pheromone value (hop distance).

origin The robot who emitted this pheromone. Can be the robot itself.

originTS The local time of the emitter when it emitted this pheromone.

constrOrigin, contrTS A constraint for pheromones of *constrOrigin* only to accept them if they have a time stamp newer than *constrTS*.

First it is checked if the robot is in emitting state or in normal state. If it is emitting, then it is only setting the global values, as it is not influenced by the neighbors, and exits afterwards. Otherwise, we are calculating the values based upon the neighbors values. We check how old the last pheromone is, based upon the time it has been changed last. If it is too old, then a constraint on the origin is set, such that we are only accepting newer pheromones or pheromones of another

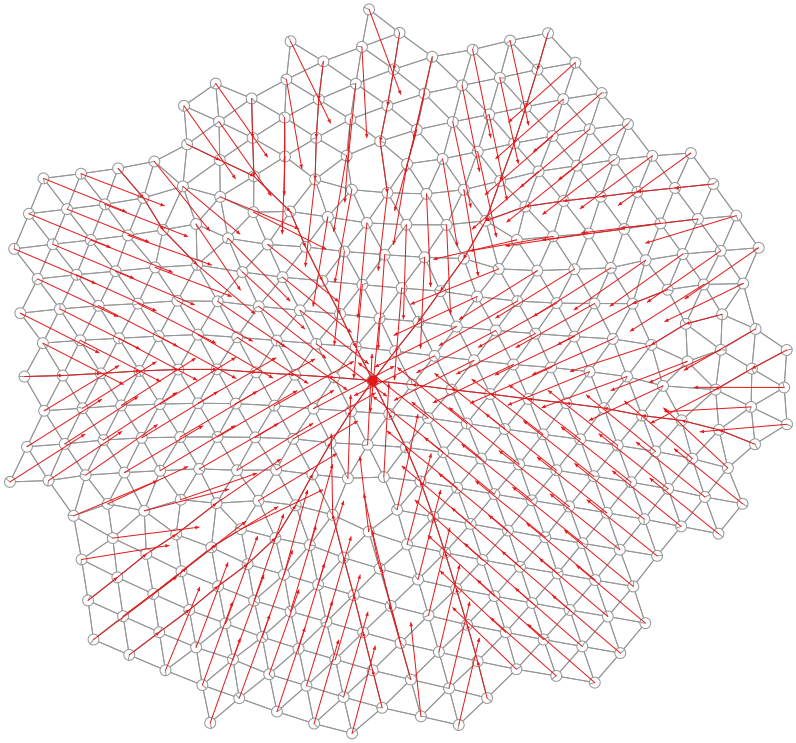


Figure 21: The pheromone gradient for a single emitting robot computed by the finite difference method.

origin. We can not always only take the newest one, as we do not have a global time and therefore we cannot compare the age of pheromones with different origin. Afterwards we are selecting the potential pheromones of the neighbors, such that they are defined, have no constraint, and no neighbor has a newer pheromone of the same origin. For checking the constraints not only the own previously mentioned constraint is checked but also those of the neighbors. This increases the robustness when dealing with multiple emitter. From the selected neighbors pheromones, the one with the lowest value (closest to leader) is taken and incremented by one. If the new pheromone is newer than the old or has a different origin, the time of the last change is set to the local time. If only the value changes, it is not touched as some fluctuations are usual.

Algorithm 6: Update of volatile pheromones

```
Algorithm Update(r)
    Input: Robot r
    if r ∈ ℰ then
        r.value ← 0
        r.origin ← id(r)
        r.originTS ← localtime(r)
        r.savedAt ← localtime(r)
    else
        if localtime() − r.savedAt ≥ MAX then
            r.constrtOrigin ← r.origin
            r.constrtTS ← r.originTS
        r.potNbrs ← {n ∈ N(r)|n.origin ≠
        r.origin ∧IsDefined(n) ∧ NotConstrained(r, n) ∧ Newest(r, n)}
        if r.potentialNeighbors = ∅ then
            r.value ← ⊥
            r.origin ← ⊥
            r.originTS ← ⊥
        else
            minNbr ← n ∈ potNbrs with n.value minimal, if not unique use the one with the
            lowest ID
            if minNbr.origin ≠ r.origin ∨ minNbr.originTS > r.originTS then
                r.savedAt ← localtime(r)
            r.value ← minNbr.value +1
            r.origin ← minNbr.origin
            r.originTS ← minNbr.originTS
Function NotConstrained(r, n)
    if n.origin ≠ r.constrOrigin ∨n.originTS > r.constrTS then
        return ∀n′ ∈ N(r) : (n.origin ≠ n′.constrOrigin ∨n.originTS > n′.constrTS)
    else
        return False
Function IsDefined(n)
    return n.value ≠ ⊥ ∧ n.origin ≠ ⊥ ∧ n.originTS ≠ ⊥
Function Newest(r, n)
    return ∀n′ ∈ N(r) : (n′.origin = n.origin ⇒ n.originTS ≥ n′.originTS)
```

The volatile pheromone can be modified to use other values as the hop distance. For example it can be used to determine the lowest ID of the robots.

5.5 Pheromone induced broadcast forest

In addition to the distance measurement towards the closest emitting robot, a pheromone can serve as a broadcast forest with a broadcast tree below each emitting robot. This allows emitting robots to supply their peers with additional information via a payload attached to the pheromone. This payload is passed along the broadcast tree below the emitter and is solely defined by the emitting robot. To design and prove algorithms based on the pheromones, some properties of these broadcast forests are now observed in detail.

Definition 25 (Pheromone Graph)**.** *The pheromone graph is a subgraph of S defined as $\mathfrak{P} = (\mathcal{R}, C_\mathfrak{P})$ where $C_\mathfrak{P} = \{(r, r') \in C | l(r) > l(r') \wedge \forall (r, r'') \in C : l(r) > l(r'') \Rightarrow \mathrm{id}(r') \leq \mathrm{id}(r'')\}$.*

In the following we show some interesting and useful properties of the pheromone graph. At first we want to ensure, that it is definite to prevent any ambiguity in algorithm definitions.

Theorem 4. *$\mathfrak{P}$ is definite for any fixed but arbitrary communication graph.*

Proof. $\mathfrak{P}$ is a subgraph of the communication graph $S = (\mathcal{R}, C)$. As it has the same node set, the nodes are definite. The arcs are selected from C by the definite l and the IDs of the robots. The boolean expression is independent of the order as it only references to the fixed communication graph S. □

Furthermore it is important, that each robot in the pheromone graph has not more than a single outgoing edge.

Theorem 5.

$$\forall r \in \mathcal{R} : |N^+_\mathfrak{P}(r)| = \begin{cases} 0 & r \in \mathcal{E} \vee l(r) = \infty \\ 1 & \textit{else} \end{cases}$$

Proof. We distinguish three cases.

$r \in \mathcal{E}$: If $r \in \mathcal{E}$ then $l(r) = 0$. As $\forall (r, n) \in C_\mathfrak{P} : l(r) > l(n)$ and $\forall r' \in \mathcal{R} : l(r') \geq 0$, there cannot be an outgoing arc $\Rightarrow |N^+_\mathfrak{P}(r)| = 0$

$l(r) = \infty$: As $\forall n \in N(r) : l(n) = \infty$ and thus $\forall n \in N(r) : l(r) \not> l(n)$, there cannot be any arc incident to r in $\mathfrak{P} \Rightarrow |N^+_\mathfrak{P}(r)| = 0$

$r \notin \mathcal{E} \wedge l(r) \neq \infty$: $r \notin \mathcal{E}(n) \Rightarrow l(r) > 0 \Rightarrow 0 < l(r) < \infty \Rightarrow$ there has to be a $n \in N(r)$ with $l(n) + 1 = l(r) \Rightarrow |N^+_\mathfrak{P}(r)| \geq 1$.

Assume $|N^+_\mathfrak{P}(r)| > 1$, then $\exists n, n' \in N^+_\mathfrak{P}(r)$ with $n \neq n'$ and $(\mathrm{id}(n) \leq \mathrm{id}(n') \wedge \mathrm{id}(n') \leq \mathrm{id}(n)) \Rightarrow \mathrm{id}(n) = \mathrm{id}(n')$ **Contradiction**: id is unique. $\Rightarrow |N^+_\mathfrak{P}(r)| \leq 1$

$\Rightarrow |N^+_\mathfrak{P}(r)| = 1$

□

This single outgoing edge defines a unique predecessor for each robot.

Definition 26 (Predecessor)**.**

$$\mathrm{pred}(r) = \begin{cases} r' \in N^+_\mathfrak{P}(r) & |N^+_\mathfrak{P}(r)| = 1 \\ \perp & \textit{else} \end{cases}$$

The swarm is partitioned by the emitting robots into one component for each emitter.

Theorem 6. *If the swarm is connected and $\mathcal{E} \neq \emptyset$ then $\mathfrak{P}$ has $|\mathcal{E}|$ components, where each component is a root tree with an $e \in \mathcal{E}$ as root.*

Proof. As the swarm is connected and $\mathcal{E} \neq \emptyset$: $\forall r \in \mathcal{R} : l(r) < \infty \Rightarrow (\text{pred}(r) = \perp \Leftrightarrow r \in \mathcal{E})$.
$\forall (r, n) \in C_{\mathfrak{P}} : l(r) > l(n)$ and $<$ is asymmetric on $\mathbb{N} \Rightarrow \mathfrak{P}$ is circle free.
$\Rightarrow$ pred defines for each component a root tree with a $r \in \mathcal{E}$ as root and every $r \in \mathcal{E}$ is a root of a root tree. □

Within the partitions of the swarm, all robots are connected via a shortest path to the corresponding emitter.

Theorem 7. *If a robot is connected to an emitter in the swarm then it is connected with exactly one emitter in the pheromone graph, which is closest in the communication graph. The connection in the pheromone graph is also a shortest path in the communication graph.*

Proof. As different disconnected components of a swarm cannot influence each other, we can bound the swarm and $\mathcal{E}$ to the component of the considered robot r. $\mathcal{E} \neq \emptyset$ as r is connected to an emitter. Therefore, we can use Theorem 6 and assume that r is part of a root tree in $\mathfrak{P}$ with a robot $e \in \mathcal{E}$ as root. $l(r)$ matches the amount of edges to the closest emitter in the pheromone graph. As for every $(n, n') \in C_{\mathfrak{P}} : l(n) = l(n') + 1$, $l(r)$ also matches to the amount of edges to the root $e \in \mathcal{E}$ of the tree. As every path of $\mathcal{E}$ is also a path in the communication graph, this is a shortest path to a closest leader. □

Each robot is assigned an emitter robot as the origin. Based on that origin different kinds of predecessor sets can be defined in addition to the unique predecessor defined earlier: either all robots that are closer to any emitting robot or all robots that are closer to the emitting robot within the same component. This distinction can be important for robots that lie on the boundary of one of the emitter induced components.

Definition 27. $\text{origin}_{\mathfrak{P}}(r)$ *defines the root of r in $\mathfrak{P}$. If it is a single node and no emitter, then it is undefined.*

We define the following two neighborhoods:

$$N'^{+}_{\mathfrak{P}}(r) = \{n \in N(r) | l(n) < l(r) \wedge \text{origin}(r) = \text{origin}(n)\}$$
$$N''^{+}_{\mathfrak{P}}(r) = \{n \in N(r) | l(n) < l(r)\}$$

6 Leader

In the swarm there are two kinds of robots: A small set of preselected and fixed leader robots, which have an external input, and the other robots, which we call mob robots. These leader robots have each an individual movement vector as external input, that denotes how the leader robots shall move. The leader robot does not have to adapt exactly this movement, but the more it does, the better is its efficiency.

Definition 28. *The robots in the set $\mathcal{L} \subset \mathcal{R}, |\mathcal{L}| \ll |\mathcal{R}|$ with $c : \mathcal{L} \to \mathbb{R}^2$ are called* leader robots *and c denotes their external movement vector. The robots in $\mathcal{R} \setminus \mathcal{L}$ are called* mob robots.

The mob robots are only working on local information and their task is to connect the leader robots. As the computation, memory, and communication complexity depends on the amount of leader robots, the amount has to be fixed to ensure scalability. For a simplification of the algorithm we assume that leader robots do not fail, but it would be no problem to extend the algorithm for this case. Also, we assume that the IDs of all leaders are known to all robots.

The leader robots are a possibility to control a swarm for exploration without losing the properties of a swarm. Leader robots can be moved out of the swarm to form pseudopodia, for example for parallel exploration or to check if an alley is secure with a small amount of robots before moving with the whole swarm into it. If a leader robot moves out of the swarm, mob robots have to follow it to ensure connectivity. For this reason each leader robot is distributing its control movement vector into the swarm. The mob robots are adapting the control movement vectors of the leader robots. As vectors become quickly inaccurate over multiple hops, due to local transformations based on inaccurate sensor measurements, and due to the fact that there are multiple leaders with different external movement vectors which are stretching the swarm, we smooth the value with the value of the *shortest path direction*.

Definition 29 (Shortest Path Direction)**.** *For a pheromone with $l \in \mathcal{L}$ as color and $\mathcal{E} = \{l\}$, the shortest path direction is defined as:*

$$\mathrm{SPD}_l : \mathcal{R} \to \mathbb{R}^2, r \mapsto \begin{cases} (0,0) & r = l \\ \mathrm{norm}(p_r(\mathrm{pred}(r))) & \mathrm{pred}(r) = l \\ (1 - \dfrac{1}{\min\{\chi, d(r,l)\}}) * \mathrm{norm}(p_r(\mathrm{pred}(r))) & \\ \quad + \dfrac{1}{\min\{\chi, d(r,l)\}} * \mathrm{SPD}_l(\mathrm{pred}(r)) & \textit{else} \end{cases}$$

where $\chi \in \mathbb{N} \setminus \{0\}$ is a tunable parameter and $\mathrm{norm} : \mathbb{R}^2 \to \mathbb{R}^2$ *is the vector normalization.*

The shortest path direction points in the direction of the shortest path to the leader. For larger χ distanced edges of the shortest path get a higher influence. We use an individual pheromone for each leader to distribute the external movement vector and thus each robot r also retrieves the external hop distance $d_l(r)$ for each leader l.

Definition 30 (Smoothed Distributed Movement Vector)**.** *The smoothed distributed movement vector for the leader robot l at the robot r is defined as*

$$\mathrm{SDMV} : \mathcal{L} \times \mathcal{R} \to \mathbb{R}^2, (l, r) \mapsto \phi(d_l(r)) * c(l) + (1 - \phi(d_l(r)))|c(l)| * \mathrm{SPD}(l, r)$$

where $\phi : \mathbb{N}_0^+ \to [0, 1] \subset \mathbb{R}$ is a tunable sigmoidal function.

The sigmoidal function ϕ interpolates smoothly between the external movement vector for small distances and the shortest path direction for greater distances, as shown in Fig. 22. It has to be chosen based upon the accuracy of the transformations, but also dynamic factors like proportion of the leader robots can influence the efficiency. Even for perfect sensing and transmission it can be advantageous if distanced robots move closer to the leaders.

The movement vector $c(l)$ for each leader robot $l \in \mathcal{L}$ is distributed via the pheromone of the leader. Each robot uses the average of the $N'^{+}_{\mathfrak{P}}$ pheromone neighborhood, since this is more reliable than only the value of the pheromone tree predecessor.

As we have multiple leaders, we have to merge multiple smoothed distributed movement vectors. We use the distances towards the leaders to weight each of them.

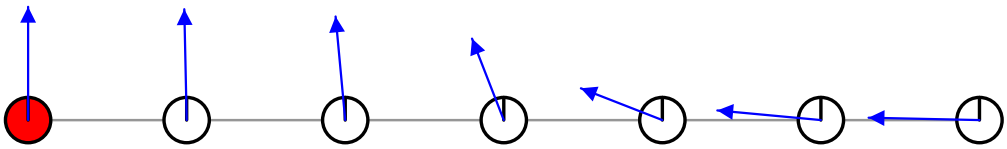


Figure 22: With increasing distance the movement determination goes from the global movement to the reliable shortest path direction.

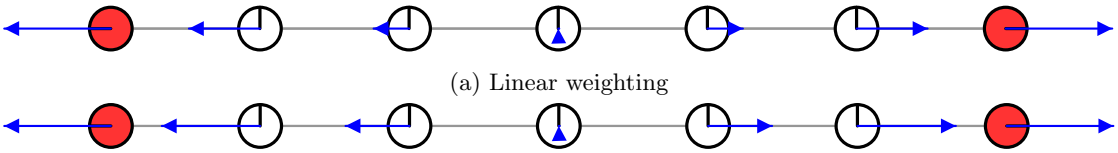
Algorithm 7: Leader movement force

The leader movement force for a set of leaders $\mathcal{L}$ is defined as

$$\text{leadermovement} : \mathcal{R} \to \mathbb{R}^2, r \mapsto \sum_{l \in \mathcal{L}} \text{SDMV}(l, r) * \frac{\delta(d_l(r))^{-1}}{\sum_{l' \in \mathcal{L}} \delta(d_{l'}(r))^{-1}}$$

where $\delta : \mathbb{N}_0^+ \to \mathbb{R}_0^+$ is a tunable weighting function.

The higher $\delta(d_l)$ becomes, the lower is the influence of the leader l. The faster this functions increases, the faster decreases the influence of a leader, but the higher is its influence on close robots. Fig. 23a shows two leaders marked in red steering into different directions while the other robots follow them linearly weighted. Fig. 23b shows the same configuration but quadratically weighted.



(a) Linear weighting

(b) Quadratic weighting

Figure 23: Leader following with different weighting functions δ. The two leaders are marked in red and steering into opposite directions. The other robots follow them weighted by their distance with δ.

7 Connectivity

The connectivity of a swarm is the most important aspect, as connectivity means the ability to communicate, navigate and sense. If the connectivity is lost, chances are small that the parts are able to rejoin again. As robots can fail, the swarm must be able to withstand the failure of some robots without losing connectivity between the leader robots. Also, it is important that leaders cannot split the swarm by moving into different directions. Therefore, we need a metric for connectivity between the leader robots, that indicates how many robots can fail before the leader robots are disconnected. As soon as the connectivity becomes too low according to this metric, the swarm will take countermeasures.

In this section we concentrate on how to determine the connectivity and how to react when it is in danger. First we propose a metric for measuring the connectivity between the leaders in Sec. 7.1. Then we define the concept of a n-core in Sec. 7.2, which helps to compute this connectivity metric in a distributed way. Finally, in Sec. 7.3 and Sec. 7.4 methods to maintain

the connectivity at a desired value are proposed.

7.1 Connectivity Metric

The most common metric for connectivity in graph theory is the n-connectivity.

Definition 31. *An undirected graph $G = (V, E)$ is n-connected if it has an order greater n and $G - V'$ is connected for each subset $V' \subset V, |V'| \leq n - 1$.*

Menger's Theorem says that in an n-connected graph, there exist n disjoint u-v-paths between all vertices $u, v \in V$. Two u-v-Paths are disjunct if they only share the vertices u and v. A short proof of the theorem can be found in [16]. This implies that there are n independent communication paths between each node. The amount of pairwise disjunct u-v-paths for given $u, v \in V$ can be computed using a Max-Flow-Algorithm with limited vertex capacity. A well known Max-Flow-Algorithm which runs in $O(|V| * |E|^2)$ is the algorithm of Edmonds and Karps. There are also distributed algorithms known, like [57], but they are not made for quickly changing topologies.

If the communication graph of a swarm is n-connected, $n-1$ robots can fail without destroying the connectivity of the swarm. However, the maximum n-connectivity of a robot in a regular fragmented swarm will be 6, as in a perfect grid each robot has at most six neighbors. If the expected failure rate of robots is high, this may be not sufficient. Therefore, we define a different metric, based upon the distance to the exterior boundary.

Definition 32 (Patched Communication Graph)**.** *The patched communication graph (PCG) is a variation of the communication graph, with additional connections between all robots on the same boundary.*

Definition 33 (Interior Value)**.** *The interior value* $\mathrm{IV}(r)$ *of a robot r is the shortest hop distance in PCG to a robot on the exterior boundary.*

Definition 34 (n-cyclic-T-enclosedness)**.** *A swarm with a denoted subset $T \subset \mathcal{R}$ is n-cyclic-T-enclosed if between all $r, r' \in T$ there exist a path P with $\forall x \in P : \mathrm{IV}(x) \geq n$.*

The metric provides a lower bound on how many robots need to fail, so that a robot of T can permanently loose connectivity. If the swarm is n-cyclic-T-enclosed, then there are at least n circles that enclose the set T. However, the metric is not perfect and needs a grid structure, since rows can be jumped over, as shown in Fig. 24.

The metric does only focus on complete failures of robots or communication links. Partly failed robots can be an even bigger problem, as [69] showed in an analysis. The detection and elimination of these robots can be a necessary but difficult task, since no robot will work perfect. This problem can however be considered separately and is not part of this thesis.

7.2 The swarms n-core

To maintain the swarm in a n-cyclic-T-enclosed state, a construct, that we call the *n-core* of the swarm is used. A n-core is a region at the center of the swarm.

Definition 35 (n-core robot)**.** *A robot r is a n-core robot if and only if* $\mathrm{IV}(r) \geq n$.

Definition 36 (n-core)**.** *A n-core is a connected set of n-core robots. A n-core is called a core of degree n.*

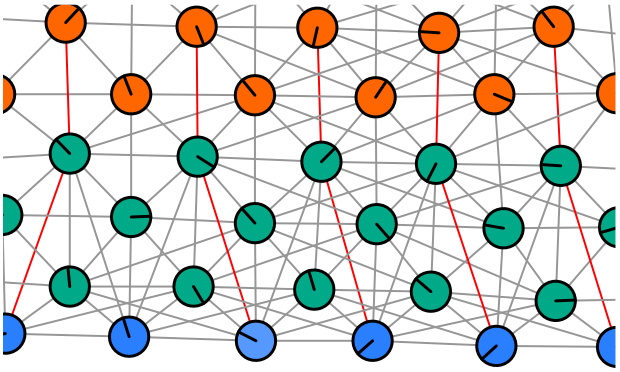
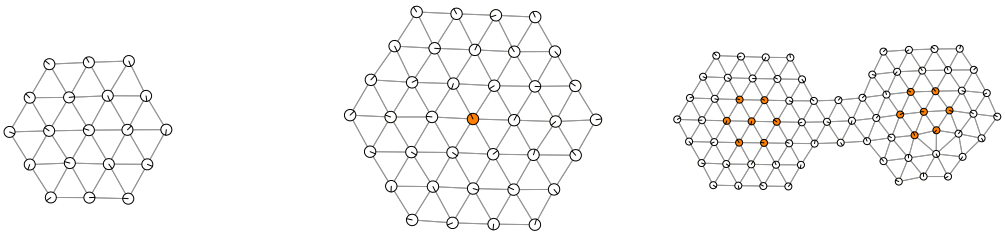


Figure 24: The orange robots have only an interior value of 2, as the red edges allow to jump over some rows. In a clean grid, this does not happen. The green robots haven an IV of 1 and the blue boundary robots an of 0.



a) no 3-core. b) a single 3-core. c) two disconnected 2-cores.

Figure 25: n-Core robots are highlighted in orange. The set of n-core robots does not need to be connected and for small groups of robots there might by no n-core robots at all.

Depending on the degree of the core and the size and shape of the swarm, there might be no n-core, a single n-core, or even multiple disconnected n-cores (see Fig. 25).

We can easily see that a robot swarm with one single n-core C and $\mathcal{L} \subset C$ is n-cyclic-T-enclosed where $T = \mathcal{L}$. In this case, at least $(2 * n) + 1$ robots need to fail to permanently disconnect two leader robots as shown in Figure 26.

If we want to keep the swarm n-cyclic-T-connected with the help of the n-cores, there are two different behavioral principles to adhere to:

1. Different n-cores need to rejoin.
2. Leader robots must stay within a n-core region.

In the following two sections we show how both behaviors can be implemented in a distributed way.

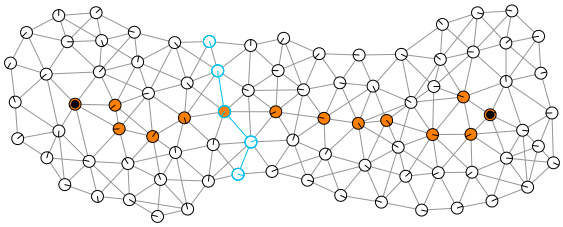


Figure 26: A swarm with a 2-core (in orange), that contains two leader robots (in black). Any cut, that would divide the 2-core in two connected components and thereby disconnect the two leaders consists of at least $5 = 2 * 2 + 1$ robots: two robots on each side of the 2-core and at least one 2-core robot. One possible cut is highlighted in blue.

7.3 n-Core rejoining

If there is more than one core containing leaders within a swarm, the n-cyclic-T-enclosedness is not given. The most obvious way to reestablish the n-cyclic-T-enclosedness in such a case is to join the disconnected n-cores back together.

7.3.1 Detecting n-cores

Before any disconnected n-cores can be rejoined, a distributed method to identify robots as n-core robots is necessary. To accomplish this, a pheromone is emitted by the robots sitting on the boundary to compute a lower bound for the interior value of the robots. This bound is then used to determine whether a robot is a n-core robot using the following routine.

Function isNCoreRobot
Input: Robot r **Output**: **True** if the robot is a n-core robot, **False** otherwise **return** $r.boundaryPheromone \geq n$

Because the boundary pheromone is only a lower bound of the interior value, this method can result in false negatives around holes in the swarm (see Figure 27). Since these false negatives result in a more conservative connectivity metric, and interior holes are usually closed quickly by boundary forces in practice they can be tolerated.

7.3.2 Detecting gaps between n-cores

In order to rejoin different n-cores, the gap regions between those n-cores need to be identified in a distributed way. A reliable way to detect gaps between n-cores is based on a Voronoi tessellation of the swarm. A Voronoi tesselation is a partition of a space based cells around seeds. In the swarm context, the space corresponds to the set of robots and the cores act as the seeds.

Definition 37 (Voronoi tesselation of a swarm)**.** *Let $\mathcal{R}$ be the space of robots with the distance function d representing the hop count distance within the communication graph. Let C_k be the k^{th} n-core in the swarm. The Voronoi cell V_k associated with the n-core C_k is the set of all robots in $\mathcal{R}$ whose distance to C_k is not greater than their distance to the other cores C_j, where $j \neq k$ or*

$$V_k = \{r \in \mathcal{R} | d(r, C_k) \leq d(r, C_j) \text{ where } j \neq k\}$$

If we want to detect gaps between n-cores based on the Voronoi tessellation, the following observation can be used.

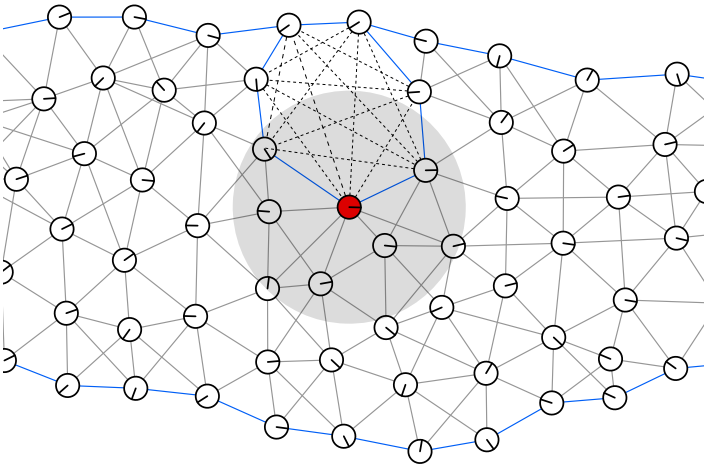


Figure 27: A swarm with a hole. Some of the additional edges in the patched communication graph are displayed as dashed lines. The boundary edges are colored in blue. The highlighted robot has an interior value of 1, while the value of the boundary pheromone is 0. The distributed n-core detection method would not identify this robot as a core robot, even though it is a 1-core robot.

Observation 2. *The edges of a Voronoi tessellation with the n-cores as seeds coincide with the center lines of the gaps between n-cores.*

We call this center line along the edge of the Voronoi tessellation a *critical cut* because its removal would result not only in the disconnection of the swarm but also in a disconnection of two n-cores containing leader robots (see Figure 28). Nonetheless, this center line does not have

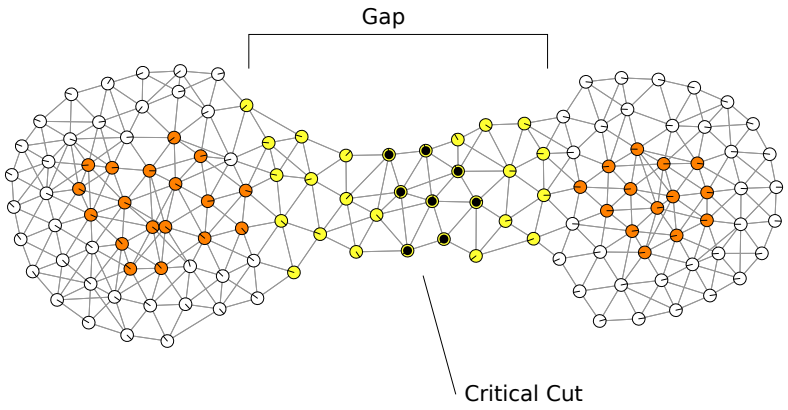


Figure 28: A gap between two 2-cores (in orange). The gap robots are marked in yellow, the cut is marked in black.

to be the shortest cut between two 2-cores (see Figure 29) Therefore, the notion of a gap has to be extended to an area around the critical cut, ranging from the border of one n-core to the other as shown in Figure 28. This gap area is characterized by the fact that all of its robots are not further away from the critical cut than the critical cut is away from the n-cores (as it lies on an edge of a Voronoi cell, the distance to both n-cores is equal by definition). We will now see,

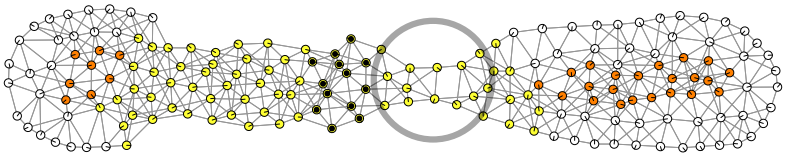


Figure 29: The border of the Voronoi cells does not have to be the smallest cut through the gap. The circle marks a cut of two robots, which is smaller than the critical cut.

how the above notion of a gap between two n-cores is detected in a scalable and distributed way. Several subtasks need to be solved to accomplish this.

First, the robots inside the different n-cores need to reach a consensus on their core ID. This is necessary for the distributed computation of the Voronoi tessellation, which is the second subtask to be solved. Third, the critical cuts need to be identified based on the Voronoi tessellation. Finally, the gap region has to be identified based on the critical cuts and their distances towards the n-cores.

Core ID consensus A core ID is seen as a subset of the set of all leader IDs. Namely, it is the set of those leader IDs which are member of the n-core. Since no leader can be a member of two n-cores at the same time, this ID is guaranteed to be unique amongst all n-cores, except for the case of n-cores that contain no leader and whose ID is therefore the empty set. This case of anonymous n-cores is treated separately and those n-cores are not considered when constructing the Voronoi tessellation. In order for a n-core robot to know its core ID, it has to know which leaders are contained within the same n-core. This is accomplished by a special intra-core-pheromone, that only propagates amongst n-core robots. Each of the leader robots emits its own color of this pheromone. As soon as a robot perceives one of the intra-core-pheromones, it adds the corresponding leader ID to its local core ID representation and removes it again after the pheromone vanished. To propagate these updates of the core IDs as fast as possible, a volatile pheromone is used. If the intra-core-pheromones are only propagated between n-core robots and they are constantly emitted by their corresponding leaders, any n-core robot can compute its core ID using the following routine.

Function computeCoreID

Input: Robot r
Output: Core id
id $\leftarrow \emptyset$
foreach $l \in \mathcal{L}$ **do**
  if $r.intraCorePheromone_l \neq \perp$ **then**
    id $\leftarrow$ id $\cup$ l
return *id*

Distributed Voronoi tessellation The Voronoi tessellation is again computed with the help of a pheromone in such a way, that in the end each robot knows the ID of its Voronoi cell, but not the whole tessellation.

The n-core robots emit a core pheromone with their core ID attached as a payload. These core IDs are then distributed within the swarm along the broadcast tree of the core pheromone. After the core pheromone has spread out in the swarm, each robot knows of exactly one core ID via this payload. Now it is easy to see – together with Theorem 7 – that this distribution of core IDs corresponds exactly to the Voronoi cells with the n-cores as seeds. It is important to

note, that anonymous n-cores with an empty set as core ID must not emit the core pheromone, as they are treated separately in the context of gap detection.

Critical cut detection After the Voronoi diagram has been computed and each robot knows what Voronoi cell it belongs to, it is rather simple to detect critical cut regions. We simply declare any robot as a critical cut robot, that sees a robot with a different Voronoi cell ID than its own within its neighborhood. For this, the Voronoi cell ID of the robots needs to be an exposed variable. A simple routine to detect critical cuts is then:

Function isCriticalCut

Input: Robot r
Output: **True** if the robot is on a critical cut, **False** otherwise
foreach $n \in N(r)$ **do**
  if $n.voronoiID$!= $r.voronoiID$ **then**
    return True
return False

Gap region detection The detection of the whole region of the whole region of the gap builds on top of the critical cut detection and makes use of another pheromone color. As mentioned before, all robots, that are not further away from the critical cut than the critical cut is away from any of the n-cores, are considered part of the gap.

This means, we need to first measure the distance between critical cut and n-core and then the distance between the critical cut and any other robot in the swarm. The first distance can simple be read of the pheromone value of the core pheromone at the site of the critical cut. The latter is computed by a help pheromone emitted by the critical cut.

This help pheromone serves multiple purposes at once. First it tells all other robots within the swarm the distance – and via its gradient – the direction towards the closest critical cut. Since every robot needs not only the distance towards the critical cut but also the distance between the critical cut and the n-core, this distance is attached as a payload to the help pheromone. The detection of the gap region can then simply be done using the following simple routine.

Function isGap

Input: Robot r
Output: **True** if the robot is in a gap, **False** otherwise
if $r.helpPheromone < r.helpPheromone.ccToCoreDist$ **then**
  return True
else
  return False

Note that even though these subtasks all depend on each other, there is no synchronization necessary because they are executed in parallel in practice. It is just a matter of emitting the right pheromones at the right locations with the right payloads. Table 1 gives an overview of the different pheromones used to detect gaps between n-cores.

Name	emitted if ...	Payload	Payload Value
boundaryPheromone	isExteriorBoundary()	—	—
intra-core-pheromone_I[3]	$r \in \mathcal{L}$	—	—
corePheromone	isNCoreRobot(r)	coreID	computeCoreID()
helpPheromone	isCriticalCut	ccToCoreDist	r.corePheromone

Table 1: Overview of the pheromones used for the detection of the gap region.

After the gap region and the direction and distance to the critical cut are detected, the robots can begin to take countermeasures against the gap.

Practical considerations Since the leader robots often tend to temporarily leave and rejoin a n-core, the core ID can change at high frequencies. While these changes propagate through the n-core, an erroneous cut can be detected along the border of the spreading intra-core-pheromone. This can simply be overcome, by ignoring cuts between n-cores whose core IDs are subsets of each other. Since one leader ID is already sufficient for a unique core ID, this mutual inclusion test will not result in false negatives. An adapted routine for the cut detection is then:

Function isCriticalCut

Input: Robot r
Output: **True** if the robot is a critical cut, **False** otherwise
for $n \in N(r)$ **do**
  if $!(n.voronoiID \subset r.vornonoiID$ **or** $r.voronoiID \subset n.voronoiID)$ **and** $n.voronoiID \neq r.voronoiID$ **then**
    return True
return False

7.3.3 Closing gaps with connectivity forces

After a gap has successfully been detected, the swarm needs to react appropriately. The most obvious solution for narrow areas of the swarm is to move other robots towards those areas. Therefore, the main principle behind the connectivity force is to pull robots along the gradient of the help pheromone towards the gap. However, using the raw vectors given by the pheromone gradient can cause new gaps to open and unattractive oscillations within the swarm. Consider

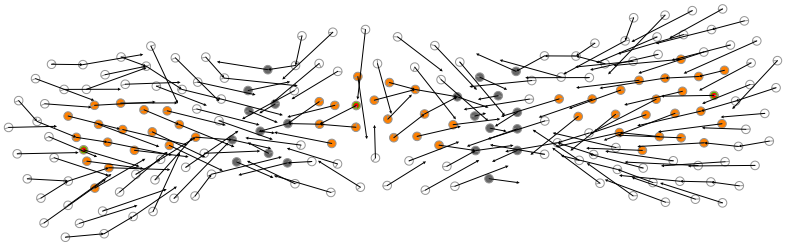


Figure 30: Three 2-cores in orange with two cuts in gray. Following the gradient of the help pheromone (black arrows) would open up a new gap in the center 2-core.

a swarm with three leader n-cores in a row as shown on Figure 30. If the robots would strictly follow the pheromone gradient, a new gap dividing the middle n-core in two halves would form immediately. At this point the two initial gaps might not be fixed yet, but the new central gap will already start to pull the robots away from them. This will close the central gap, but open up the initial gaps again or make them even thinner. This reconstitutes the initial state and the cycle starts anew.

To reduce these oscillating effects, the speed at which the robots react to the gradient is decreased with the distance towards the gap. In the above example this will let the robots at the critical central position react only slowly to the two gaps, so they are closes before a new gap can open in the center.

It is important to note, that the distance to a gap is not equal to the distance to a critical cut. While a critical cut always has a limited width, a gap can be stretched arbitrary wide (see

Figure 31). So if we would scale the connectivity force based on the distance to the cut, it might

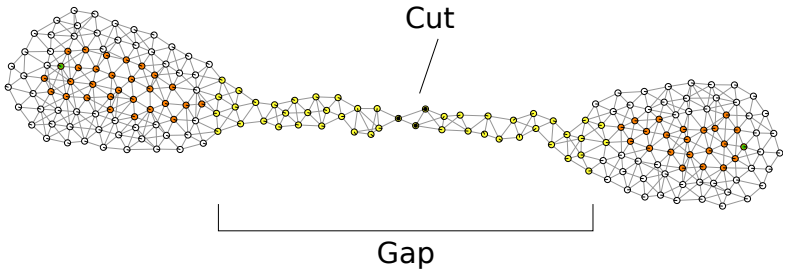


Figure 31: The gap (in yellow) can become arbitrary wide, while the cut between the two 2-cores stays at a constant width.

already have been diminished to a negligible amount when it reaches the two corresponding n-cores. Therefore, the distance to the gap region is more appropriate to pull the two n-cores closer together.

The distance can simply be computed based on the help pheromone by taking the maximum of zero and the difference between the help pheromones value and its distance to n-core payload:

Function distanceToGap

Input: Robot r
Output: The distance to the closest gat region
return $\max\{0, (r.helpPheromone - r.helpPheromone.cutToCutDist)\}$

The overall connectivity forces are then computed using the following routine.

Function getConnectivityForce

Input: Robot r
Cc **Output**: A force pushing robots towards gaps
if $r.helpPheromone \neq \perp$ **then**
  f $\leftarrow$ r.helpPheromone.gradient
  f $\leftarrow$ norm(f)
  f $\leftarrow$ f $* \frac{1}{1+\texttt{distanceToGap}(r)}$
  return f
else
  return $(0, 0)$

7.3.4 Dealing with anonymous n-cores

After the gaps between the leader n-cores have been detected, there still remains the problem of gaps between anonymous n-cores. Instead of detecting these gaps explicitly, the following simple rule is used to implicitly close these gaps. Any anonymous n-core robot drives along the gradient of the leader core pheromone towards the closest leader n-core. This will eventually result in the fusion of the anonymous n-core and the leader n-core thereby giving it an ID.

In the case of an anonymous n-core sitting in the same Voronoi cell to a leader n-core (see Figure 32) this closes the gap without pulling the leader n-core in a wrong direction. Only the less important anonymous n-core moves towards the leader n-core, but the leader n-core is not influenced by the anonymous n-core. In other cases, where the anonymous n-core is spread across multiple Voronoi cells, this behavior results in a new gap, because the anonymous n-core is split (see Figure 33). This might seem like a fatality, but since the n-core was per definition already

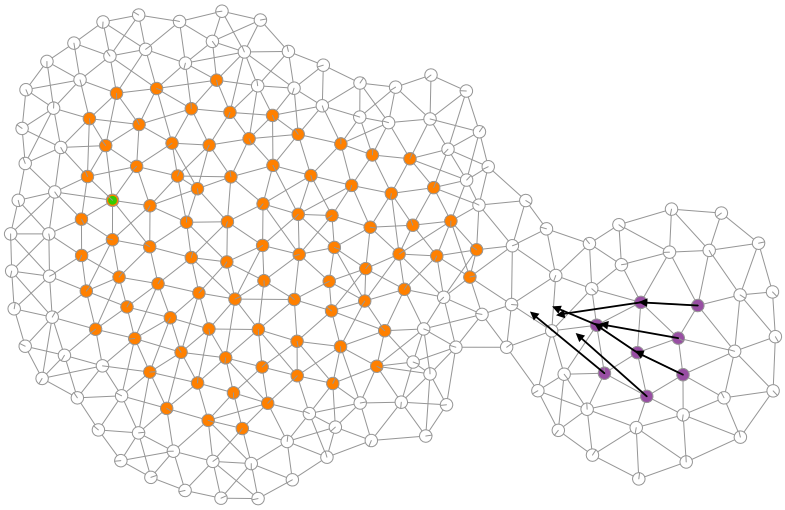


Figure 32: An anonymous n-core (in purple) rejoining the leader n-core (in orange).

sitting on the edges of a Voronoi cell, a gap between the surrounding leader n-cores has already been detected. The anonymous n-core behavior can be implemented in the following simple way:

Function getAnonymousCoreForce

Input: Robot r
Output: A force that pulls anonymous n-cores towards leader n-cores
if `isNCoreRobot(`r`)` **and** `computeCoreID(`r`)` $= \emptyset$ **then**
  return $r.leaderCorePheromone.gradient$
else
  return *(0,0)*

7.4 Moving leader robots back to the n-core

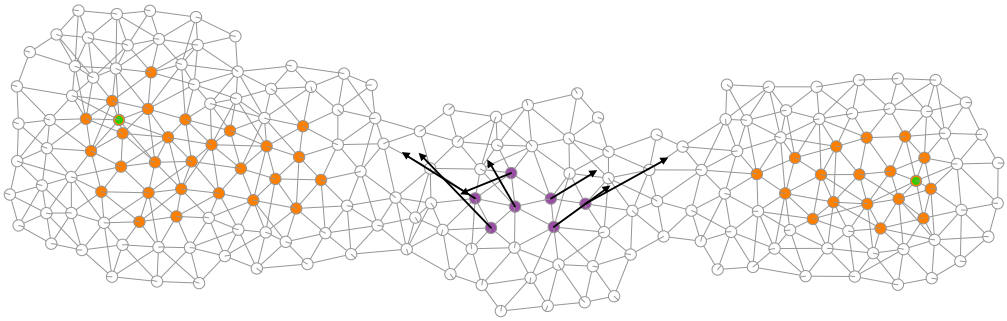
As soon as a leader leaves the n-core, it should return to the closest n-core, because otherwise the n-cyclic-T-connectivity is not given. This is as simple as following the gradient of the n-core pheromone. The following routine brings leader robots back into a n-core.

Algorithm 8: Return leaders to the n-core

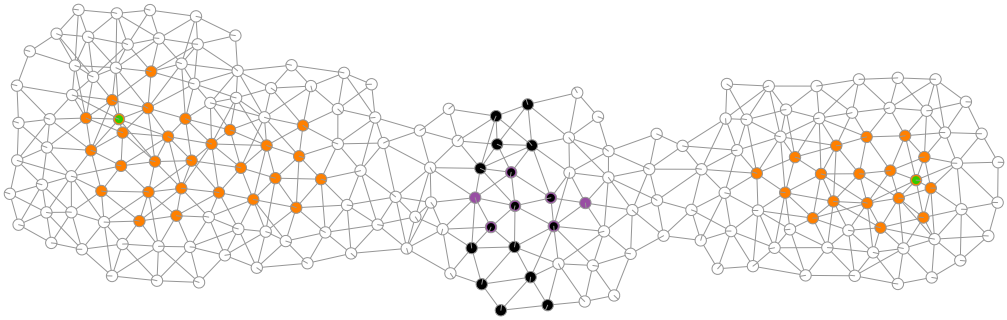
Input: Robot r
if $r \in \mathcal{L}$ **and** $\neg$`isNCoreRobot(`r`)` **then**
  r.setMovement(r.corePheromone.gradient)

8 Density

For swarms with a convex shape the flocking algorithm of Olfati-Saber (Sec. 3) and the dynamic boundary force are enough to keep the swarm tightly packed without holes. With multiple leader we are not always able to gain a convex shape as Fig. 19 shows. The dynamic boundary force would stretch the swarm to its maximum in attempt to remove the concave boundary parts, which is a very unstable state. The algorithm of Olfati-Saber does not care for this, as it only cares for a regular fragmentation and movement consensus. For this reason we need an additional



(a) Two 2-cores (in orange) with leaders (in green) on both sides and an anonymous 2-core (in purple) in the center. The anonymous n-core rejoining forces are shown in black.



(b) The detected cut between the two leader 2-cores is highlighted in black.

Figure 33: A splitting anonymous 2-core spread over two Voronoi cells.

force which cares for the density within the swarm and thus prevents the swarm to be stretched or compressed too much. Similar to substances like liquid or gas, we allow a firm density to be defined for the swarm such that a deviation from it results in a counterforce.

Definition 38 (Local Density)**.** *The local density of a robot in a swarm is the average density in its range area within the swarm. It is denoted with* $\rho : \mathcal{R} \to \mathbb{R}$*.*

The determination of ρ is made in Sec. 8.1. In Sec. 8.2 we introduce a force which enables a regular density, and in Sec. 8.3 a density sensitive weighting for algorithms (which is used by Alg. 9).

8.1 Local Density Determination

The main problem is to determine the local density of a robot, such that it can react to it. There are two difficulties with this:

1. The density of regular (exterior) boundary robots should not be lower only because of their boundary state.

2. Due to the line of sight constraint not the whole range area is visible.

We solve the first point by excluding all open sectors except of the largest exterior one. The second point is solved by calculating the visible area based upon the positions of the neighbors and their size.

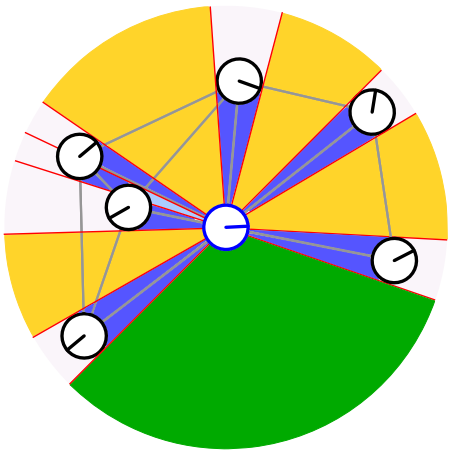


Figure 34: The exterior area is highlighted in green, the interstice areas in yellow and the vestibules areas in blue. The intersecting vestibule area is brighter. The segment lines are red.

For the calculation of the relevant area we divide the visible area into different subareas which are easy to calculate. All these areas are delimited by the segment lines of the neighbors, as shown in Fig. 34. Each neighbor has a clockwise and a counterclockwise segment line. Both lines start at the considered robot and end at the range margin. The clockwise segment line cuts the considered neighbor at it most clockwise point and the counter clockwise segment lines cuts it at its most counter clockwise point.

Definition 39 (Segment Line)**.** *The segment lines are defined as length sensitive directions, which start at the position of the associated robot r. Let* $\text{norm} : \mathbb{R}^2 \to \mathbb{R}^2$ *be the vector normalization,* $(x, y) \in \mathbb{R}^2$ *a vector representation, and* τ *the radius of a robot. The clockwise segment line of a robot r for a neighbor n is defined*

$$\text{ClkSgLine}_r(n) = \text{norm}(p_r(n) + \text{norm}((p_r(n).y, -p_r(n).x)) * \tau) * \Gamma$$

The counterclockwise segment line of a robot r for a neighbor n is defined as

$$\text{CtClkSgLine}_r(n) = \text{norm}(p_r(n) + \text{norm}((-p_r(n).y, p_r(n).x)) * \tau) * \Gamma$$

The area of a circle fragment can be determined by

$$A : [0, 2\pi] \to \mathbb{R}_0^+, (\theta, l) \mapsto \frac{1}{2} * \theta * l^2$$

We call the area in front of the neighbors the *vestibule area*. Single vestibule areas can intersect if the angle difference between the associated neighbors is small but that are only simple intersections.

Theorem 8. *Each point lies at most within two different vestibule areas.*

Proof. If two neighbors n and n' are intersecting each other, the intersected area is not intersectable by a further neighbor n'' without violating the line of sight constraint. If the neighbor's position $p(n'')$ is in the vestibule area of n or n', it would hide n or n' as closer neighbors hide a wider area. If it lies in one of the verging interstice areas, it can only be further away than n or n', for the same reason. On the other hand, if it lies further away than n or n', then its segment line does not range far enough, as further away neighbors cover less area. □

This means that we have to remove each intersection of interstice areas exactly once. One can check if the interstice areas of two neighbors intersect by their segment lines.

$$\begin{aligned}\text{IsIntersecting}_r(n,n') =& \measuredangle(\text{CtClkSgLine}_r(n'),\text{ClkSgLine}_r(n)) \\ &< \measuredangle(\text{CtClkSgLine}_r(n),\text{ClkSgLine}_r(n))\end{aligned}$$

For determining the union of all vestibule areas of a robot, we sum up all vestibule areas and remove the intersecting areas because they have been added twice.

$$\begin{aligned}\text{VestibuleArea}(r) =& \sum_{n\in N(r)} A(\measuredangle(\text{CtClkSgLine}_r(n),\text{ClkSgLine}_r(n)),|p(n)-p(r)|) \\ &- \sum_{\substack{n\in N(r),\\ n'=\text{ClkNxtNbr}_r(n)\wedge\\ \text{IsIntersecting}_r(n,n')}} A(\measuredangle(\,\text{CtClkSgLine}_r(n'),\text{ClkSgLine}_r(n)),\max\{|p_r(n)|,|p_r(n')|\})\end{aligned}$$

We call the area between two neighbors the *interstice area*. It is fully observable but may intersect with the exterior area. We first determine the whole interstice area and handle the exterior area afterwards.

$$\text{IntersticeArea}(r) = \sum_{\substack{n\in N(r),\\ n'=\text{ClkNxtNbr}_r(n)\wedge\\ \neg\,\text{IsIntersecting}_r(n,n')}} A(\measuredangle(\text{ClkSgLine}_r(n),\text{CtClkSgLine}_r(n')),\Gamma)$$

A robot should not have a different density only because it lies on the exterior boundary. Thus, we replace the interstice area of the largest exterior open sector with the average interstice area. This way the largest open sector does not influence the density, while having more than one open sector or lying on the boundary of a hole does. Otherwise, a shapely exterior boundary would lower the density and therefore attract other robots. Let $\text{EOS}(r)$ be the set of all open sectors of the robot r which are part of the exterior boundary and an open sector $(n,n')_r \in \text{EOS}(r)$ be maximal if $\measuredangle((n,n')_r) = \max\{\measuredangle((m,m')_r)|(m,m')_r \in \text{EOS}(r)\}$.

$$\text{ExteriorArea}(r) = \begin{cases} A(\measuredangle((n,n')_r),\Gamma) - \dfrac{\text{IntersticeArea}(r)}{|N(r)|} & \text{EOS}(r)\neq\emptyset \\ \quad \text{with } (n,n')_r \in \text{EOS}(r) \text{ maximal} & \\ 0 & \text{else}\end{cases}$$

The relevant visible area of a robot can now be determined by the sum of the interstice and vestibule area minus the exterior area, which is defined as zero for robots that are not part of the exterior boundary.

$$\text{RelevantVisibleArea}(r) = \text{VestibuleArea}(r) + \text{IntersticeArea}(r) - \text{ExteriorArea}(r)$$

Finally, we only have to divide the amount of visible neighbors by the area to obtain the local density of a robot:

$$\rho(r) = |N(r)|/\,\text{RelevantVisibleArea}(r)$$

8.2 Density Distribution

The density distribution is a force which tries to keep a specific local density at each robot. It creates an attraction to neighbors with low local density and a repulsion to neighbors with high local density. For smoothing the local density of a robot, it takes the mean value of the maximal local density in the neighborhood, the minimal local density in the neighborhood and the own calculation. This is important since the local density determination does not always represent the real density. On this way local minima and maxima can also propagate a bit further than the own neighborhood and thus speed up the conformation.

$$\rho'(r) = \frac{1}{3}\rho(r) + \frac{1}{3}\max\{\rho(n)|n \in N(r)\} + \frac{1}{3}\min\{\rho(n)|n \in N(r)\}$$

To determine the force we simply sum up all directions to the neighbors weighted by the difference to the optimal density. Let ϱ be the desired density for the swarm, then the density distribution force is defined by

$$\text{DensDist}(r) = \sum_{n \in N(r)} (\text{norm}(p_r(n)) * \phi(\varrho - \rho'(n)))$$

where $\phi : \mathbb{R} \to \mathbb{R}$ is a tunable function which determines the attraction/repulsion and norm : $\mathbb{R}^2 \to \mathbb{R}^2$ is the vector normalization. We set ϕ to

$$\phi : x \to \begin{cases} x^2 & x \geq 0 \\ -x^2 & \text{else} \end{cases}$$

but do not preclude that there may be better solutions. The density distribution force does not care for the positions or distances of the neighbors like the flocking algorithm of Olfati-Saber does, but only for the directions and densities. It leads to a kind of regular fragmentation but does not create a grid, as shown in Fig. 35.

8.3 Density Quality

Sometimes we need a function which scales from 1 for perfect density to 0 for the worst density to weight algorithms. This function weakens the force of an algorithm the worse the density gets and thus prevents it from influencing the swarm in low density regions which could destroy its structure. For this we use a bounded absolute difference to the desired density ϱ.

Definition 40 (Density Quality)**.** *The density quality for a specified desired density ϱ is defined by*

$$\text{DensQual} : \mathcal{R} \to [0,1], r \mapsto \phi(\min\{|\rho'(r) - \varrho|, b\}/b)$$

Where $b \in \mathbb{R}^+$ is a tunable constant and $\phi : [0,1] \to [0,1]$ is tunable function.

The constant b is the bound from which the absolute difference from the desired density is too high. The function ϕ maps the relative difference to the weighting. We chose this function to be quadratic, but if b is small the identity function may also be a reasonable choice.

9 Steiner Tree

The problem of connecting the leader robots with the least amount of mob robots is analog to the Euclidean Steiner tree problem (ESTP). The shorter the connecting network between

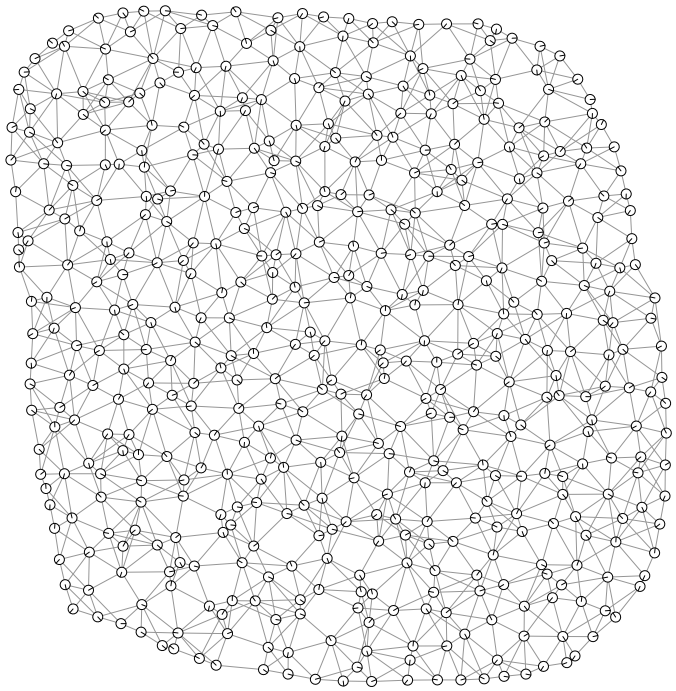


Figure 35: Density distribution and dynamic boundary forces. There is some kind of regular fragmentation but not a grid, like for Alg. 2

the leaders, the fewer robots we need to archive a base network, the more robots are left to strengthen it. Therefore, we aim to archive a Euclidean Steiner tree (EST) similar shape of the swarm between the leaders. One useful aspect of the ESTP is that it allows some optimization for a (not optimal) EST with simple local rules.

In Sec. 9.1 we give an introduction to the two Steiner tree problems ESTP and Steiner tree problem in graphs (STPG). In Sec. 9.2 we discuss an explicit determination of a Steiner tree in graphs (STG) on which we can do EST-optimization and why it is not feasible. As nature is able to do EST-optimization without knowing anything about Steiner trees, we consider some physical models and the appearance of ESTs in nature in Sec. 9.3. In Sec. 9.4 we propose a method for EST-optimization without determination in robot swarms. Finally, we mention in Sec. 9.5 that we are already obtaining an EST-optimization with flocking, boundary force and density distribution.

9.1 The Steiner Tree Problem

The Steiner tree problem (STP) dates back to the famous French mathematician Pierre de Fermat (1607-1665), who stated the problem of finding in the plane a point with minimal total distance to three given points. This problem definition corresponds the ESTP for $n = 3$. The generic form has first been defined by Kössler and Jarník, who searched for a minimal connecting network between n arbitrary points in the plane. The title 'Steiner Tree', after the Swiss mathematician Jakob Steiner (1769-1863), arose by a mistake of Courant and Robbins in their popular book 'What is mathematics?' [14]. An extensive review of the history of the ESTP has been made by [10].

The ESTP is the original problem of many derivations of the STP. All derivations have in common that they try to connect a given set of points or vertices with minimal costs, but they differ in constraints or graph class.

Definition 41 (Euclidean Steiner Tree Problem)**.** *Given a set of points (called terminals) in $\mathbb{R}^2$: find a network of minimal total Euclidean length which connects these points. Such a minimal network is called a minimal Steiner tree (SMT). A possibly not minimal solution is simply called EST.*

The network is allowed to use additional points, which are called *Steiner points*. These Steiner points have some known properties, which are helpful for finding the SMT. It is known, that every Steiner point has exactly three edges and that these edges have a mutual angle of 120°. Thus, a Steiner point is the *first Fermat point* of the triangle spanning the three adjacent points. Further, it has been shown by Courant and Robbins [14] that a SMT has at most $|V| - 2$ Steiner points. Nevertheless, Garey et al. [24] showed that the ESTP is $\mathcal{NP}$-hard, even for the discretized version (discrete Euclidean Steiner tree problem). The hardest part is the determination of the Steiner points, as for given Steiner points the problem is equal to finding the MST. The fastest known ([8]) exact serial algorithm for the ESTP is the *GeoSteiner* of Warme et al. [68], which has been used to solve instances with 10000 terminals. However, there are good heuristics to retrieve mostly close approximations of SMTs in only a fraction of the time, like [71] which archives a performance ratio of $1 + \ln(2/\sqrt{3}) \approx 1.1438$.

Another variation of the STP is the STPG. In it, we are only allowed to use connections and Steiner points out of a given graph.

Definition 42 (Steiner Tree Problem in Graphs)**.** *The STG has as input a weighted graph $G = (V, E)$ with the weighting function $c : E \to \mathbb{R}$ and a denoted set of nodes $S \subseteq V$, called terminals. We search a connected subgraph of minimal wight containing all terminals. Such a subgraph is called a minimal Steiner tree in graphs (SMTG).*

For non weighted graphs, the weighting function can be set to constant 1. If $S = V$ the problem is equivalent to the MST and polynomial solvable. The generic problem is $\mathcal{NP}$-hard.

9.2 Explicit Steiner Tree

There have been an amount of research for distributed heuristics for the STG, as it is for example important for efficient multimedia broadcasting. If we had determined a SMTG(or an approximation of it) in our swarm, we could afterward optimize it locally to an EST by differing between edge robots and Steiner point robots. The edge robots have two neighbors represent an edge in the ST and try to straighten themselves. The Steiner points have three neighbors and try to move towards the first Fermat point. It is possible, that a SMTG has non terminal robots with more than three neighbors, which have to be split up. Robots not within the STG are trying to surround the STG regular to ensure the n-cyclic-T-enclosedness of the swarm. As only local optimization of the EST can be done, which depends on the swarm configuration, the resulting ESTs for different configurations with non varying leader robots may vary heavily.

A survey of distributed algorithms and their approaches for the STP (mainly STG) has been made by [8]. Many of the distributed heuristics like [60] or [33] are based upon the exact distributed MST-algorithm of [23]. The general concept is to merge STG-fragments to larger fragments until there is only one fragment left which is a STG, starting with every terminal as a STG-fragment,. At every time the fragments are a vertex partition of the arising STG, where the edges within the fragments are already determined. This allows us to use the fragments as local STG before the complete STG has been computed. As the known algorithms need at least

$O(|V|log|V|)$ rounds and are therefore not scaling for large swarms, this is a necessary property. However, there are still some other drawbacks which make this approach unfeasible.

Swarms have a quickly changing topology and the most fragments are therefore obsolete before the global STG has been determined. Without a global STG, the connectivity is not ensured, as the optimization is only done within the fragments but not between fragments. We do not know if a robot can be used for the optimization or is needed for a not yet determined part of the STG. The connectivity between the fragments is therefore endangered. Additionally, the maintenance of a fragment becomes more and more difficult for larger fragments as there has to be a centralized robot which decides with which fragments it should merge. For doing so it has to collect the information of all robots inside the fragment. With a changing configuration, this is not feasible. The algorithm of Gatani et al. [25] considers a changing configuration, but only for small changes.

A naive approach could be to calculate an STG once while freezing the swarm and adjusting and repairing it afterwards. If the STG reaches a point, where it cannot be repaired anymore, the swarm is frozen again and a new STG is determined. However, this does not match our perception of a flexible swarm, as the swarm would frequently be unserviceable. An explicit determination of a STG followed by EST-optimization thus has been discarded by us.

9.3 Steiner Trees in nature and physics

ESTs can be found in nature, for example in soap films. Since the problem is very hard to solve with classical computation, there have been efforts to solve the problem by the construction of physical objects, which let nature solve a specific instance. If nature can solve a $\mathcal{NP}$-hard problem in polynomial time, it would imply that $\mathcal{NP}$-hard problems can be solved efficiently. Aaronson [4] discusses this issue and comes to the conclusion that none of the proposed methods is able to do so. However, since we do not want to really solve the ESTP, we may can get some inspirations from nature.

9.3.1 Physical Models

There are three different physical models known to us: The quite simple *string model*, the *soap film model* and the *membrane model*. They are all described in [29].

The *string model* can not actually solve the ESTP, but only find the positions of given Steiner points. For these Steiner points, the adjacency has to be known in advance. This model will move the Steiner points to their optimal positions by connected strings with weights.

The base is a board with low friction with holes representing the terminals. Through each holes goes a string of equal length with an equal weight attached to it. The Steiner points are sliding rings on the board connected with three strings. Due to the weights, the Steiner points should slide to the local optimal position.

Even if the concepts of pulling on strings should be a valid local optimization, we do not see how the physical implementation is able to optimize the edges between Steiner points.

The most popular model is the soap film model. In contrast to the string model, it can actually solve (or approximate) the ESTP and is also easy to construct. The physical construction is made by two parallel transparent plates, with posts at the positions of the terminals. This construction is filled with soapy water. As soon as we remove the water, there will be a thin soap film remaining between the posts, as can be seen in Fig. 36. Due to the boundary tension, this film is minimizing its surface and thereby minimizing the EST.

There have been made experiments by [17] with six terminals. He noticed that thinner pins are more likely to produce ESTs, while thicker often only produce spanning trees (no Steiner

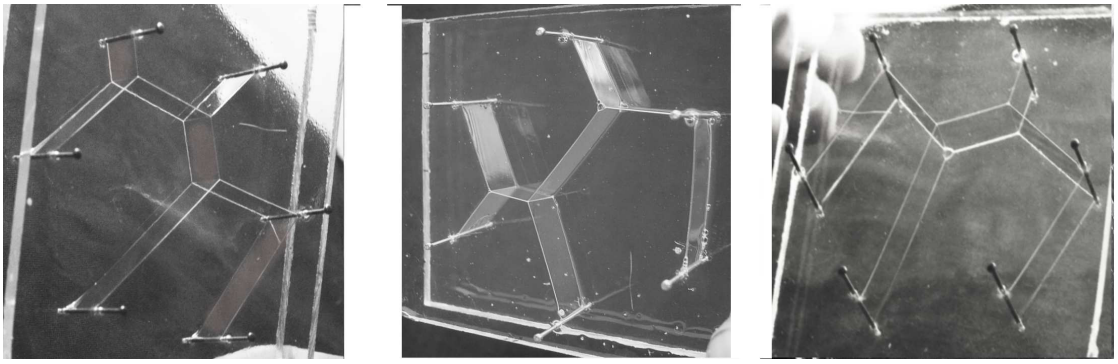


Figure 36: The soap films span networks close to the SMT between the pins. Taken from [17].

points). Neither ESTs nor spanning trees where global minimal for most of the time. It is notable, that he did not discover loops if the soap solution was bubble free.

The soap film model also has been adapted for a simulation based algorithm by [55]. It mostly produced results close to the SMT within a linear time in terms of the amount of terminals.

The third model is the *membrane model*, which puts rubber membranes between the terminals to estimate probably good positions for Steiner points by creating hills and valleys. It does not produce a concrete solution of a EST.

9.3.2 Natural Steiner Trees

In biology the slime mold *Physarum plasmodium* is known to span short networks between food sources. The reason for this is, that short and thick connections allow a more efficient extraction of nutrients. Connections with a high throughput become thicker while connections with low throughput vanish. These resulting networks are often EST similar but can contain cycles, as [46] have shown in their experiments. These additional cycles provide a higher fault tolerance against accidental disconnecting.

Another example are the wood ants *Formica aquilonia*, as studied by Buhl et al. [11]. The wood ants build cleaned trails, which sometimes even possess walls and tunnels. These 'highways' enable efficient traffic of large amounts of ants to travel towards the food resources. On the other hand, they need maintenance and thus more trails are more expensive. The tradeoff between direct highways and low maintenance costs results in networks similar to ESTs.

9.4 Implicit Steiner Tree

In a tree every edge connects two components. As the swarm is initially connected and holes are closed by the boundary force, we assume the swarm as tree shaped. Especially if there are much more robots than needed, this may not be true but for this case it would not matter as there is no need for optimization. Each robot tries to determine the two components that the edge it lies on connects and to optimize its position. We use the shortest path directions towards the leaders and cluster them based upon the angle difference into two clusters. The main advantage of the shortest path direction, beside its robustness, is that directions of different leaders become equal if they use the same edge. This also enables the existence of virtual Steiner points. For the optimization force, which tries to straighten the edges, we sum up the normalized mean of the two clusters. Additionally, we weaken this force if the robot density is becoming too high or

too low.

Algorithm 9: Implicit Euclidean Steiner Tree Optimization

The implicit EST-optimization force at a robot r for a set of leaders $\mathcal{L}$ is defined as

$$\text{IESTO} : \mathcal{R} \to \mathbb{R}^2, r \mapsto \begin{cases} \text{densityQuality}(r) * \sum_{v \in \text{Cluster}(\{\text{SPD}_l(r) | l \in \mathcal{L}\})} v & r \in \mathcal{R} \setminus \mathcal{L} \\ (0,0) & r \in \mathcal{L} \end{cases}$$

with $\text{densityQuality} : \mathcal{R} \to [0,1]$ is a weighting function, where higher values are closer to an optimal density, and $\text{Cluster} : 2^{\mathbb{R}^2} \to 2^{\mathbb{R}^2}$ a cluster algorithm which clusters in up to two vectors based upon the angle difference.

This force should only be active if the density is not too low or especially too high. The more robots there are, the less close a robot can move towards the minimum of the force, thus the swarm will be compressed. We discuss the determination of the density in section 9. The shortest path direction has been discussed in section 5. For clustering the directions we use the common k-Means-Algorithm of Lloyd [40] with $k = 2$ and the angle difference as metric. The clustered vectors are the averages of all vectors within the cluster sets.

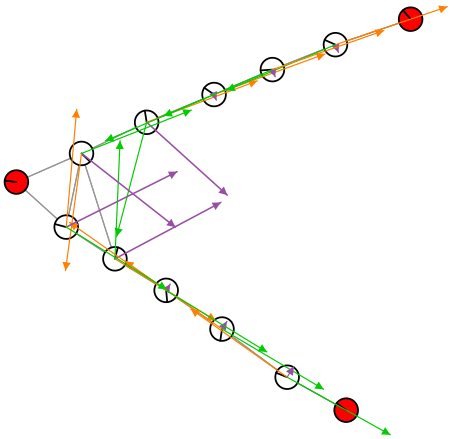
The k-Means-Algorithm is a simple heuristic for vector quantization. In every round it assigns a vector to its most similar cluster. The similarity is measured by the difference to the mean of the vectors inside the cluster. With the assignment the mean of the cluster changes. In the next round for each vector is checked if the assigned cluster is still the most similar cluster, if not it is moved to the most similar. This is repeated until no changes appear. In the beginning, each cluster contains an arbitrary vector.

Even if we only try to straighten the edges of the tree shape, we automatically do some EST-optimization.

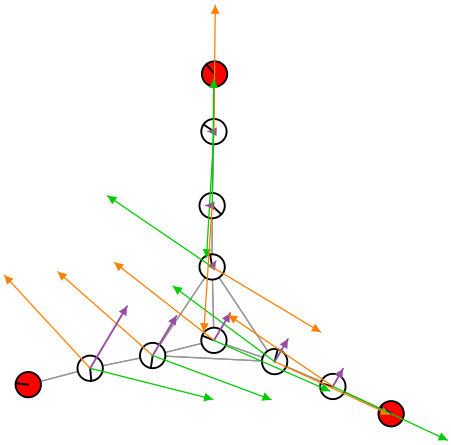
- The optimization creates Steiner points if the angle between adjacent edges in the shape is less than 120°, as Fig. 37a shows.
- The optimization supports the 120° rule of Steiner points in the shape, as Fig. 37b shows.
- The optimization removes Steiner points in the shape if they are not needed, as Fig. 37c shows.

9.5 Boundary Force and Steiner Tree

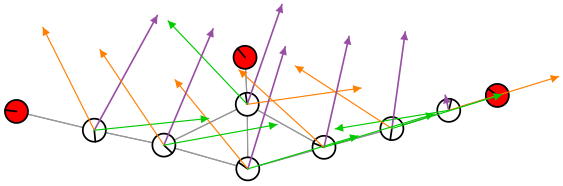
The soap film model in Sec. 9.3 creates good solutions using the boundary tension. However, the dynamic boundary force, described in Sec. 4.5, applied onto a swarm using the flocking algorithm (Alg. 2) is not sufficient. While it performs well if the boundary forces are only compressing, it fails for expansion. Instead of stretching the swarm it mostly splits the swarm to preserve the optimal distances, which cannot be obtained by stretching. The swarm first needs the properties of a fluid. We cannot mimic the molecules of a fluid, because there are much more molecules than robots. Nevertheless, the density distribution (Sec. 8.2) emulates the firm density of a fluid and overcomes some the issue of the flocking algorithm. In experiments a well tuned combination of flocking, boundary force and density distribution showed to be as good as the additional optimization of Alg. 9 for most of the time.



(a) The forces produce a Steiner point in the shape



(b) The forces optimize the position of the Steiner point in the shape



(c) The forces remove an unnecessary Steiner point

Figure 37: The optimization of Alg. 9. The pink arrows are the resulting forces and the other two are the clustered directions. The density discrimination is ignored for these examples.

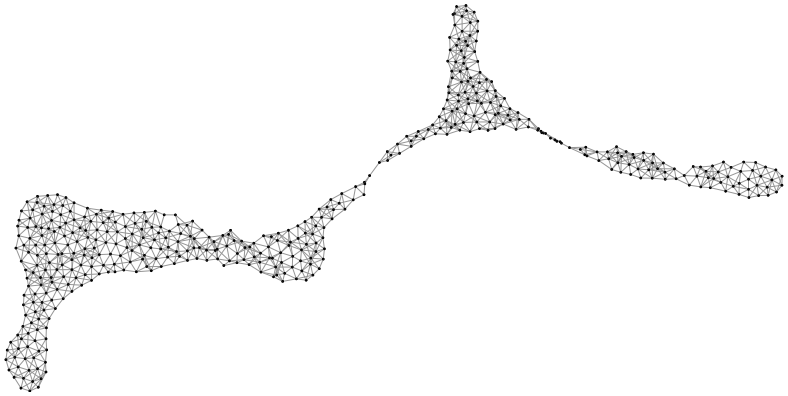


Figure 38: A swarm – moved by a naively integrated controller – is pulled apart by various leader robots.

10 Integration

All previously described control algorithms, that compute force vectors for the movement of the swarm robots, namely the flocking algorithm of Olfati-Saber, the boundary forces, the Steiner tree optimization, the connectivity protection, and the density distribution must be integrated into a single robot controller to unfold their full potential. Only then they result in a swarm, that is easily controllable while staying connected.

Because each of the algorithms can already show quite complex behavior, a mere vector sum of the control forces is not sufficient for integration into a single controller. Section 10.1 shows how the forces can be combined using different stages. Various problems with leaders, that tend to loose control over the swarm because they leave the core, are discussed and solved in Section 10.2. Finally, Section 10.3 gives the pseudocode for a robot controller that integrates the algorithms using the techniques proposed in the two preceding sections.

10.1 Balancing the forces

A common problem with this naive integration by vector addition is a very unbalanced spatial distribution of the robots along the swarm as shown in see Figure 38. This is a result of uncoordinated application of the connectivity forces and the leader forces. They often work in opposite directions, thereby canceling each other out which results in a swarm that neither follows any leaders nor fixes any arising gaps.

To improve upon those issues, an integrated controller is used, which applies the forces of the various algorithms in different stages, which are iteratively enabled based on various conditions given by the gap detection system. Each of the stages is activated if all previous stages are activated and its conditions are met.

Stage 0 This stage is always enabled without any preconditions. It activates the boundary forces, because they do not impose any threat on the connectivity of the swarm.

Stage 1 Only if the robot is not part of a gap, Stage 1 is enabled. Then it is considered safe to activate flocking forces, computed by Olfati-Sabers flocking algorithm as well as the density

distribution force, because there is no immediate risk of disconnecting the swarm at those robots.

Function stageOneConditions
Input: Robot r **return** ¬`isGap(`r`)`

Stage 2 The third stage activates the forces for the connectivity maintenance and the Steiner tree optimization. As these forces can be dangerous in fragile regions of the swarm, this stage is only enabled in regions close the leader cores. Specifically, the value of the leader core pheromone needs to be equal or less than n, where n is the degree of the core to be used.

Function stageTwoConditions

Input: Robot r
if $r.leaderCorePheromone < n$ **then**
  return True
else
  return False

Stage 3 The last stage activates the leader forces to steer the swarm. If these forces act in the opposite direction of the help pheromone gradient, they can cause dangerous elongations of the gap regions. To ensure, that the leader forces do not override the connectivity forces, this stage is only enabled, if a robot is twice as far away from the next gap than from the next leader core and if the angle between connectivity force and leader force is less than 0.7π

Function stageThreeConditions

Input: Robot r, connectivty force c, leader force l
if `distanceToGap(`r`)` $\geq 2 * r.leaderCorePheromone$ **and** `smallestanglebetween(`c, l`)` $< 0.7 * \pi$ **then**
  return True
else
  return False

10.2 Keeping the leader in the core without oscillations

The strategy to only allow leader robots to act as leader, while they are inside a core and to immediately return to the closest core in all other cases, causes two problematic oscillating behaviors.

Single pseudopodium oscillation In the first case, a leader is pulling a pseudopodium out of the swarm (1). If there is a scarcity of robots, this pseudopodium will become thinner and thinner as it is stretched by the leader (2). Eventually the core inside the philopodium disappears, thereby withdrawing any control over the swarm from the leader (3). The leader immediately starts to return to the main body of the swarm together with the rest of the retracting pseudopodium whose moment is now dictated by the boundary forces (4). As soon as the leader arrives at the core of the main body of the swarm, it starts to pull out the pseudopodium again, thereby starting the oscillating cycle anew. See Figure 39 for an illustration of the whole cycle.

Control stealing oscillation In the second case two leaders are involved, that try to pull the swarm in opposite directions. At first they might stretch the swarm (1), but as soon as one of the leaders leaves the core, because it was driving slightly faster than the other core robots

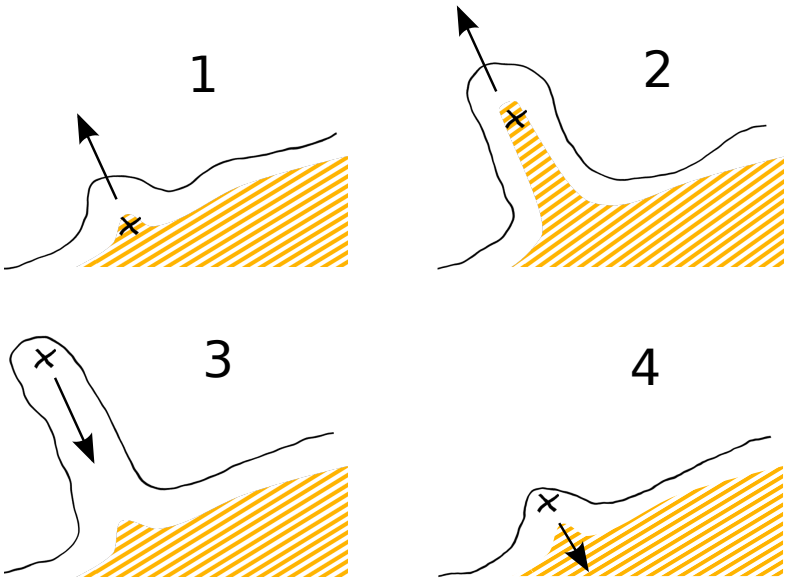


Figure 39: A leader is pulling a pseudopodium out of the swarm. The leader is indicated by the cross, the core is shown in orange.

(2), its opponent takes control over the whole swarm. It will pull the core away from the first leader, giving him no way to easily rejoin the core again (3). After the core completely reached the second leader, it slows down so that the first leader can catch up (4) to regain control over the swarm. Because the length of the external steering vector of the first leader is now longer (if the steering vector is based on a global goal position) its leader forces become stronger (5,6,7), eventually overtuning the second leader (8) thereby starting the cycle anew in the opposite direction. This way two leader robots can steal each other the control over the swarm causing severe oscillations. See Figure 40 for an illustration of the whole cycle.

To overcome the oscillating movement patterns, a strategy to keep the leaders within the core is used, that is more aware of the leaders relative position to the core. With this strategy, leaders can guide the mob no matter if they are within the core or not. To return to the core they either follow the core gradient, or – if they happen to sit in front of the core – just wait until the core catches up (see Fig. 41 for an illustration).

This lazy waiting strategy is only applied though, if the conditions for stage 2 are met. The final routine to bring leaders back into the core, which overrides all other forces if necessary, is described by the following algorithm.

Function leadersToCore

```
Input: Robot r
if r ∈ ℒ and ¬isNCoreRobot(r) then
    if smallestanglebetween(leaderForce, r.corePheromone.gradient) < 0.7 * π or
    ¬stageTwoConditions(r) then
        r.setMovement(r.corePheromone.gradient)
    else
        r.setMovement(0, 0)
```

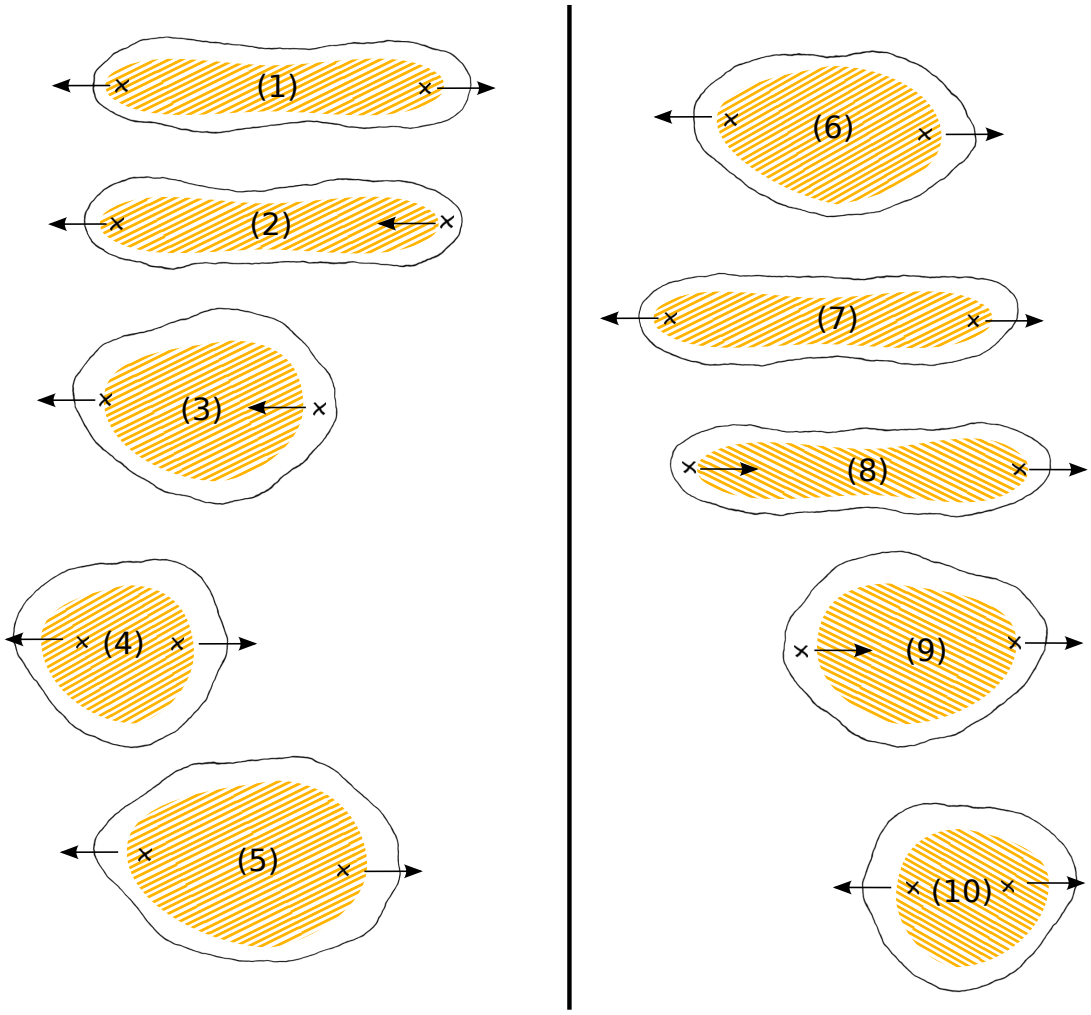


Figure 40: A whole cycle of an oscillation, where two leader robots steal each other the control over the swarm.

In this case the leader is sitting behind the core. Therefore, it must move to catch up.

Relative to the external movement vector, the core sits behind the leader. It is therefore sufficient to wait for the core.

Figure 41: Different relative positions of a leader robot towards the core.

10.3 Integrated control loop

Balancing the forces as described in 10.1 and keeping the leader robots inside the cores as described in 10.2 finally results in the following control loop for the integrated controller:

Algorithm 10: Control loop of the integrated controller

```
Input: Robot r
movemetDir ← dynamicBoundaryForce(r)
if stageOneConditions(r) then
    movementDir ← movementDir + flockAlg(r) + DensDist(r)
    if stageTwoConditions(r) then
        movementDir ← movementDir + getConnectivityForce(r) + IESTO(r)
        if stageThreeConditions(r) then
            movementDir ← movementDir + leadermovement(r)
r.setMovement(movementDir)
leadersToCore(r)
```

11 Simulator

From simple local behavior much more complex global behavior can emerge. Thus, it is often difficult to predict how even a simple local algorithm behaves at a global level. The most efficient way to test it is then to implement the algorithm and feed it into a simulation. There already exist a bunch of different simulators, but as we did not find any which met our requirements, we designed a new 2D swarm simulator called *Platypus3000*.

The custom *Platypus3000* simulator, written in Java, was specifically developed to simulate robots modeled after the R-Ones. This lack of generality helped to reduce development as well as execution time of the final software because specific features of the R-Ones like directed IR communication could be efficiently implemented in higher abstraction layers instead of the physical layer.

The main features of the *Platypus3000* are the following:

- Simulating swarms with up to 3000 robots on a modern computer without getting too slow.

- Simple and fast implementation of robot controllers for rapid prototyping.
- A simple graphical user interface (GUI) to inspect and manipulate the swarm.
- Tuning parameters during execution.
- Dynamic custom visualization overlays for the local state of a robot.
- Headless execution and creation of vector graphics of the swarm.

We describe the basic simulation in Sec. 11.1, the GUI in Sec. 11.2, the overlay system in Sec. 11.3 and the basic programming of robot controller and algorithms in Sec. 11.4.

11.1 Robot Simulation

To simulate a robots physical properties like mass, acceleration and friction and its interaction with the environment like collisions, the physics library JBox2D [1] is used. This physics library is optimized for speed, but its accuracy is sufficient since the robot's geometry is as simple as a round disk. JBox2D allows the simple integration of static and dynamic obstacles within the environment. Furthermore, its ray-casting, collision detection and spatial querying features allow for an efficient implementation of the robots sensors and communication capabilities. A robot within the simulator has the following sensors, communication capabilities and actuators:

- Sense of the neighbors relative position and bearing, where neighbors are robots within line if sight and not further away than 1 m.
- Sense for collisions between their own body and other bodies with in the simulation environment. This includes the position on their body where the collision occurred.
- Ability to send and receive directed and broadcast messages to and from their neighbors.
- Ability to control their LED's colors.
- Ability to control the robot's movement.

The features are presented to the user of the simulator using an interface. To gain control over a robot, a robot controller has to be implemented consisting of an initialization and a control loop function. The robot interface is passed as an argument to the control loop where it can be used to implement the robot's behavior. In each step of the simulation, the physics engine is advanced, the neighborhood graph is updated, the communication messages are propagated amongst the robots, robot collisions are computed, the steering vectors of the robots are transformed into physical forces and the control loops of the robot controllers are executed once. Each of these steps simulates $\frac{1}{60}$ of a second.

11.1.1 Neighborhood

The real R-One robots sense their neighbor's orientation, bearing and distance based on their directed IR communication interfaces. In the simulation this relation is reversed: The IR communications simulation is based on the neighborhood set of a robot. It is faster to send messages within a neighborhood graph that was extracted from the swarm configuration using the physics engine than to simulate the properties of modulated IR lights. In each simulation step the swarm's communication and neighborhood graph is computed based on the distance and line of sight constraints discussed in Section 2.2.2. To test for line of sight between any two robots, a

virtual ray is cast from one robot to the other. If the ray hits any third object, line of sight is not given. To check the distance constraint for all n^2 pairs of robots with complexity less than $O(n^2)$, a rectangular range query selects only a small subset of potentially close robots. The query is posed to a spatial data structure provided by the physics engine at a cost of $O(\log(n))$. Since the query is a square and the robot's sensor range is of circular shape, some robots in

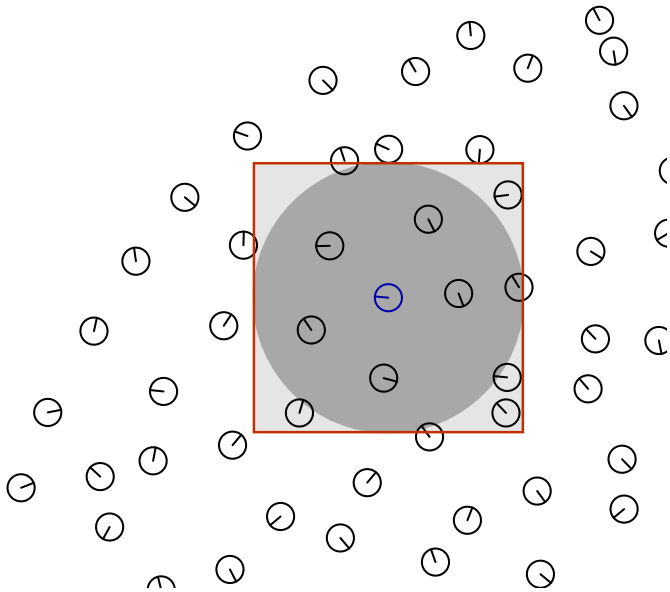


Figure 42: Some robots might fall into the range query (red square) but are still not in the range of the robot highlighted in blue

the queries corners might still fail the distance constraint as shown in Figure 42. Therefore, the robots within the query result still need to be filtered based on the precise distance. This can be done in $O(1)$, because only a constant number of robots fit inside the range query. This way the unit disc graph of the robot swarms configuration can be found in $O(n \log n)$ time. The unit disc graph is then thinned out by checking every edge for line of sight using a ray-cast.

Function computeNeighborhood

```
Input: Robot r
neighborhood ← rangeQuery(rect)
forall the n ∈ neighborhood do
    if dist(r, n) > range then
        neighborhood ← neighborhood \ n
    else if ¬lineOfSight(r, n) then
        neighborhood ← neighborhood \ n
return neighborhood
```

The swarm's communication graph is computed by combining the neighborhoods into a single graph. To avoid duplicate ray casts, the ray-cast results are cached during each time step. The information about a robot's neighborhood is then presented through the robots interface as a list of neighbor views. Each neighbor view contains the neighbor's ID and its pose relative to the observer.

11.1.2 Communication

As already mentioned, the messaging system is build on top of the neighborhood graph. The robot interface provides two functions to send messages: `void send(message, id)` and `void send(message)`. The former is used to send messages to specific robots in the neighborhood; if the robot with the given ID does not lie within the neighborhood, the message is lost.

The latter is used for broadcasts, which means that the message is sent to all neighbor robots. Incoming messages are stored within a queue of fixed length to simulate the limited storage space of the R-One robots. If the incoming message queue is full, any further messages are discarded. This queue is presented via the robots interface as a list of messages to the controller. After each time step the message queue is cleared. As a consequence the robot controllers need to process the received messages immediately. Each message consists of the message payload, the sender's ID and the destination's ID (if it is not a broadcast message).

11.1.3 Collisions

JBox2D already supports the detection of collisions, so that we only have to process the position of the collision and the disappearance of the collision. Collisions are saved in a queue and only the oldest collision is readable for the robot controller. If a collision disappears, it is removed from that queue and if it was the oldest collision, the sensed collision is the next one in the queue.

The constraint of only sensing the oldest collision has been made because the R-Ones sense collisions through the shifting of a ring around it. However, it is very easy to modify the simulation to allow the controllers to sense all collisions.

11.1.4 Movement

As implementing a two wheel movement in JBox2D needs a lot of overhead, we wrapped the movement control to prohibit e.g. too high velocities or driving sidewards. Each 2D velocity vector is first transformed into a constrained 1D velocity and angular velocity, as described in Sec. 2.2.3. The velocity of the robot object in the physic engine is then overwritten by a velocity vector into its actual orientation and the angular velocity of the previous step.

11.2 Visualization

11.2.1 GUI

Visualization Since the robots are only simulated in a 2D world, the visualization is two dimensional as well. For this the Processing library[52] has proven easy to use and fast enough because of its OPENGL [56] based backend. Few lines of code provide drawing functions for robots, obstacles and the robots' neighborhood graph. For convenience features like drag n' drop for robots and obstacles, pinning robots to their current position as well as rotation and deleting robots were implemented. The world can be navigated with the mouse using a simple zoom and pan scheme. Screen-shots of the current view can be made using a PDF export feature to obtain a vector graphics representation.

Parameter Playground To quickly and intuitively explore the vast parameters space spanned by the many tweaking and tuning possibilities of swarm algorithms, a parameter management system was developed, that requires minimal changes to existing algorithm implementations.

Any float member variable can be registered with its name, its surrounding object (or class for static variables), a group name and a range at the parameter playground. A slider together

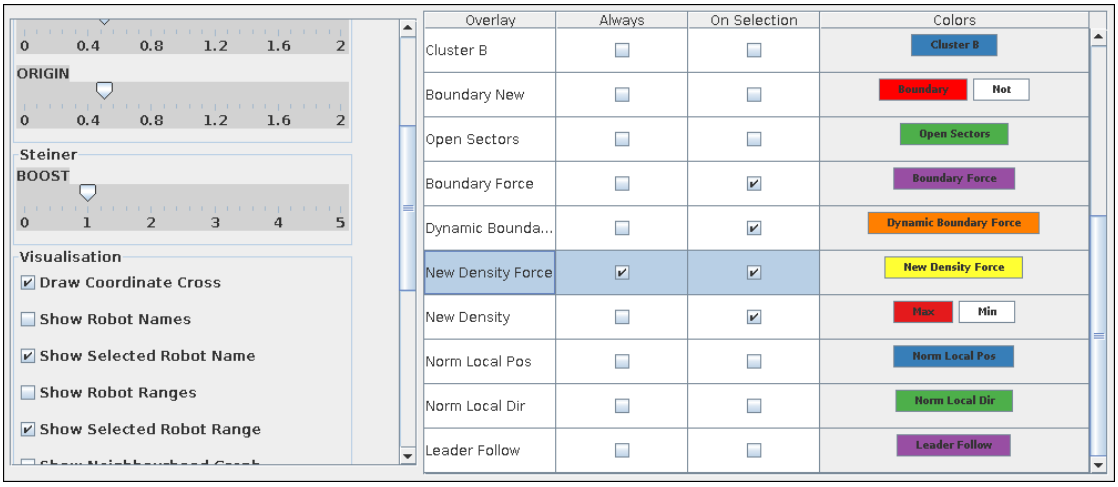


Figure 43: The GUI with the parameter-playground on the left and the overlay management on the right. Please notice that it only contains so many entries because the controller is very complex.

with its name will automatically be created to control the variables value. Similarly, boolean variables can be attached to labeled checkboxes to control their state. This works purely based on Javas reflection and introspection features.

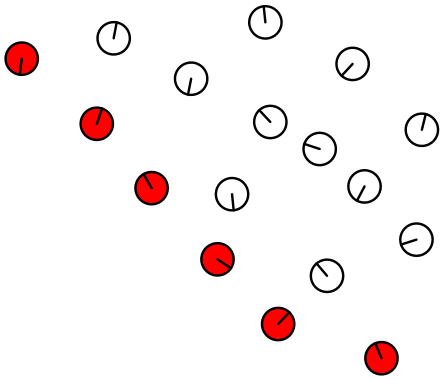
11.3 Overlays

It is not only difficult to infer global behavior out of local behavior but also the reverse direction can be challenging. Therefor a tool to visualize the internal states of the individual robots, which cause the global behavior of the swarm, is a necessary addition to the classical debugger. The overlay system allows a look not only into a single robots states but gives an overview over the whole swarm to quickly identify errors, that can barely be found with a normal debugger. The main idea is to dynamically visualize the state of the swarm robots, e.g. by drawing different forces as arrows on top of the robots or by colorizing the robots to symbolize different states.

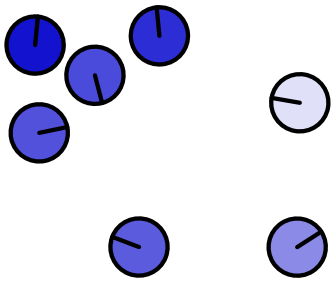
A simple GUI (see Fig. 43) allows the configuration of those overlays. In the GUI the overlays can be individually activated for all robots or only the select robots and the colors can be changed. The colors are either picket automatically from a predefined palette[4] or are assigned manually in the robot controllers. Most of the figures in this thesis were created by activating an appropriate set of overlays and printing the visualization to a file.

There is already a set of generic overlays to visualize vectors, float values and discrete states. The *vector overlay*, which can be seen in Fig. 44c, visualizes a vector by an arrow. It is also possible to visualize a set of vectors with the *multi vector overlay*, shown in Fig. 44d. To visualize a float value within a specified range, like the density, the *fading color overlay* can be used (Fig. 44b). It fades between two robot colors for an upper and a lower bound. If only discrete states, like the boundary state, have to be visualized, the *discrete color overlay*, which is shown in Fig. 44a is a valid choice. For a finite set of states, it colors the robots in a specific color, depending on the state. If there are different overlays active which color the robots, each overlay only colors a ring of the robot, as shown in Fig. 44e.

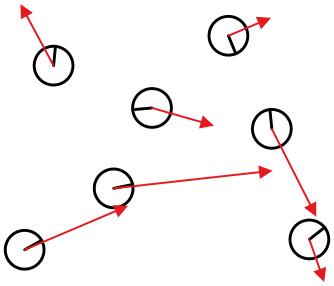
[4]The schemes on `http://colorbrewer2.org/` by Cynthia Brewer proved to be very helpful.



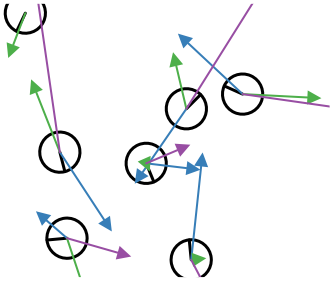
(a) Discrete color overlay



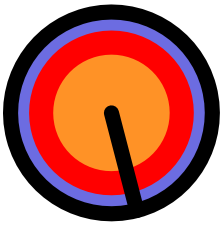
(b) Fading color overlay.



(c) Vector overlay



(d) Multi vector overlay



(e) If multiple color overlays are active, they are split into rings

Figure 44: An overview over different overlays

Own overlays can be implemented by extending the abstract `LocalOverlay` class. It requires the implementation of a foreground and a background drawing loop, which can be left empty if not needed. The coordinate system is local as described in Sec. 2.2.3. A simple overlay which tracks a vector and visualizes it by a line can be implemented as follows:

```
public class VectorAsLineOverlay extends LocalOverlay {
  public Vec2 drawnVector;
  DynamicColor color;

  public VectorOverlay(RobotController controller, String name, Vec2 vector) {
    super(controller, name);
    this.drawnVector = vector;
    this.color = getColor(name); //obtain the color from a palette
  }

  @Override
  public void drawBackground(PGraphics pGraphics, ColorInterface robot) {
    //Nothing to do
  }

  @Override
  protected void drawForeground(PGraphics pGraphics) {
    if(drawnVector != null && drawnVector.lengthSquared() > 0) { //check vector
      pGraphics.stroke(color.getColor()); //set stroke color
      pGraphics.line(0,0, drawnVector.x, drawnVector.y); //draw line
    }
  }
}
```

Overlays for a robot are efficiently managed with a hash map and only executed if they are active.

There are not only robot specific overlays but also global overlays, with access to all robots. They can for example be used to calculate and visualize global metrics or algorithms.

11.4 Robot Programming

There are two main points we found to be important when implementing elegant robot controllers:

1. A controller must be easy and familiar to write.
2. Algorithms must be reusable and easy to combine.

For the first point we decided to orientate the robot controller style on the `init`/`loop`-style of Processing. The implementation is done by extending an abstract class which requires the implementation of the `init()` and the `loop()` methods. Both methods have a *robot interface* as parameter to control the robot and access the sensor values.

Algorithms are implemented similarly but as an algorithm may depend on others or shall be combined with others, an *algorithm manager* and a *force tuner* have been introduced. While the *algorithm manager* cares for the correct order of execution of the algorithms, the *force tuner* allows algorithms, which return a velocity, to be activated and weighted dynamically during the execution via a simple GUI.

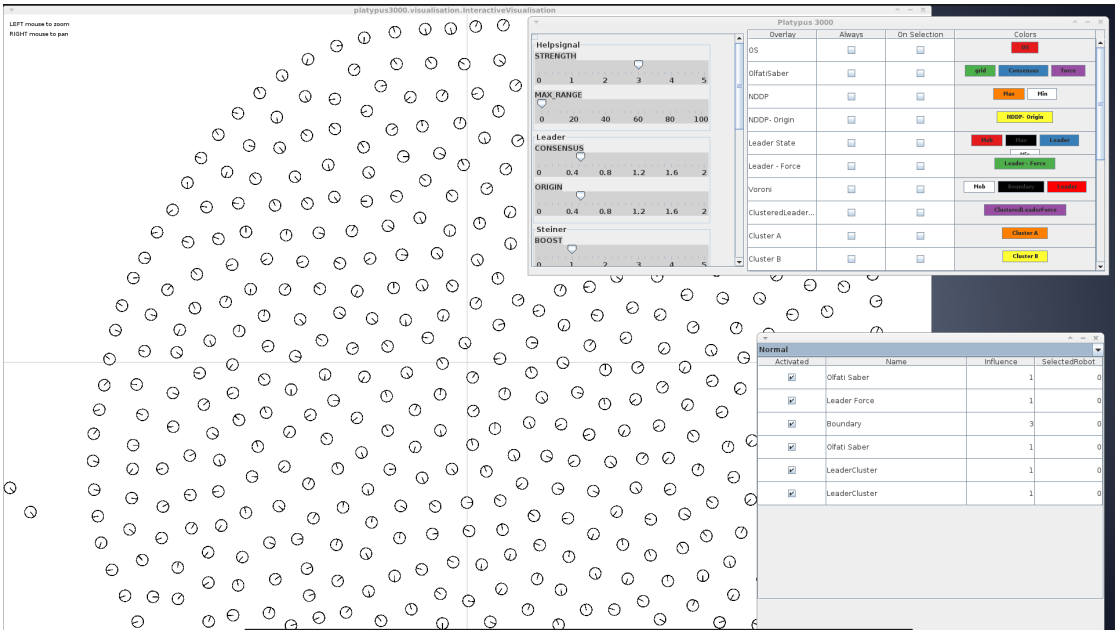


Figure 45: The visualization and the other elements are each executed in a separate window.

11.4.1 Robot Controller

To implement a robot controller, the abstract class `RobotController` has to be extended. It has the method `void init(RobotInterface robot)` which is called once in the beginning, and `void loop(RobotInterface robot)`, which is called in every simulation step. The simulation can only proceed if all controllers finished their work, thus robot controllers have to be implemented to finish within one time step and save their values needed for the next time step. Continuously running algorithms are not possible. A robot controller which does nothing at all looks as follows:

```
import platypus3000.simulation.*;

public class DumbController extends RobotController {

  @Override
  public void init(RobotInterface robot) {
    //Setup controller
  }

  @Override
  public void loop(RobotInterface robot) {
    //Read sensors and adjust movement
  }
}
```

Even if the robot interface is passed as an argument, it always stays the same for each controller. Thus, every robot in the simulation has its own instance of the robot controller.

Nonetheless, it is possible to replace the controller of a robot during the execution of the simulation. This is for example used to implement a manual controller via the mouse in the interactive visualization, which is swapped in for selected controllers.

The robot interface provides the following methods:

`int getID()` Returns the unique id of the robot.

`void send(MessagePayload message, int address)` Sends a message to a specific neighbor.

`void send(MessagePayload msg)` Sends a message to all neighbors.

`Iterable<Message> incomingMessages()` Returns an iterator over all received messages.

`LocalNeighborhood getNeighborhood()` Returns the neighborhood.

`boolean hasCollision()` True if the robot is colliding with something.

`float getCollisionSensor()` The bearing of the collision.

`Vec2 getLocalPositionOfCollision()` The local position of the collision.

`void setSpeed(float speed)` Sets the velocity into the direction of the robot's orientation.

`void setRotation(float rotation)` Sets the angular velocity.

`void setMovement(Vec2 direction)` Sets a 2D-Velocity vector which is automatically transformed into speed and rotation.

`Vec2 getLocalMovement()` Returns the actual local 2D-Velocity.

`void say(String text)` Prints a text below the robot in the Visualisation.

`long getLocalTime()` An increasing local time.

11.4.2 Algorithms

When an algorithm should be reused, its implementation as a robot controller is not useful. Inheritance may allow some kind of reuse but as Java does not allow multi-inheritance, combining different algorithms is difficult. Therefore, algorithms are implemented separately and managed by an *algorithm manager* which executes them in the right order. If a controller wants to use an algorithm, it creates an instance of it in the init-phase and adds it to an algorithm manager. The algorithm manager loops all algorithms in the beginning of the loop-method to update their results. A controller using the flocking algorithm and the boundary algorithm can look as follows:

```
public class FlockingAndBoundaryController extends RobotController {
  AlgorithmManager algorithmManager = new AlgorithmManager();

  OlfatiSaber_Alg1 flockingAlgorithm;
  StateManager stateAlgorithm; //Exposed Variables needed for boundary
  BoundaryDetection boundaryAlgorithm;

  @Override
  public void init(RobotInterface robot) {
    flockingAlgorithm = new OlfatiSaber_Alg1(this);
```

```
    algorithmManager.addLoopAlgorithm(flockingAlgorithm);
    stateAlgorithm = new StateManager();
    algorithmManager.addLoopAlgorithm(stateAlgorithm);
    boundaryAlgorithm = new BoundaryDetection(this, stateAlgorithm);
    algorithmManager.addLoopAlgorithm(boundaryAlgorithm);
  }

  @Override
  public void loop(RobotInterface robot) {
   algorithmManager.loop(robot);
   robot.setMovement(flockingAlgorithm.getForce().add(
      boundaryAlgorithm.getBoundaryForce()));
  }
}
```

Algorithms for the controller implement the interface *Loopable* which has the methods `Loopable[] getDependencies()` and `void loop(RobotInterface robot)`. The method *getDependencies* returns those algorithms which have to be executed before and the method *loop* executes the actual algorithm. The *algorithm manager* then automatically executes all algorithms in the right order[5].

Integrating multiple algorithms which produce forces like the flocking algorithm, is often a matter of tuning the influence of each force. For this reason we developed a force tuner which allows dynamic activation and weighting of those forces via a GUI (Fig. 46). It also shows the current influences of different forces acting on the selected robot. The forces are differed by a name given as a string. As we mostly want to tune the algorithms of not only a single but a group of robots, the weightings are for groups of robots. The GUI switches between different groups via a *JComboBox* at the top of the window.

11.5 Configurations

To simulate a swarm we do not only need a robot controller but also the initial positions of the robots. We call this a configuration. First a simulator object has to be created. Afterwards we can create robots by passing them a controller, the simulator object, and an initial pose. Finally, we can either execute single simulation steps manually or start it within an interactive visualization. For simple controller and a swarm of only three robots the initialization could look like this.

```
Simulator sim = new Simulator();
Robot r1 = new Robot(new OwnController(), sim, 1, 2, 0);
Robot r2 = new Robot(new OwnController(), sim, 1.2f, 2, 0);
Robot r3 = new Robot(new OwnController(), sim, 1.4f, 2, 0);
InteractiveVisualisation.showSimulation(sim);
```

12 Experiments

Predicting the exact global behavior, that arises from a swarm algorithm, can be very hard. When multiple algorithms act together within a single swarm, this prediction can be even more complex.

[5] Circular dependencies are resolved using a special time-travel technique that we can not elaborate any further at this point because of concerns about the national security.

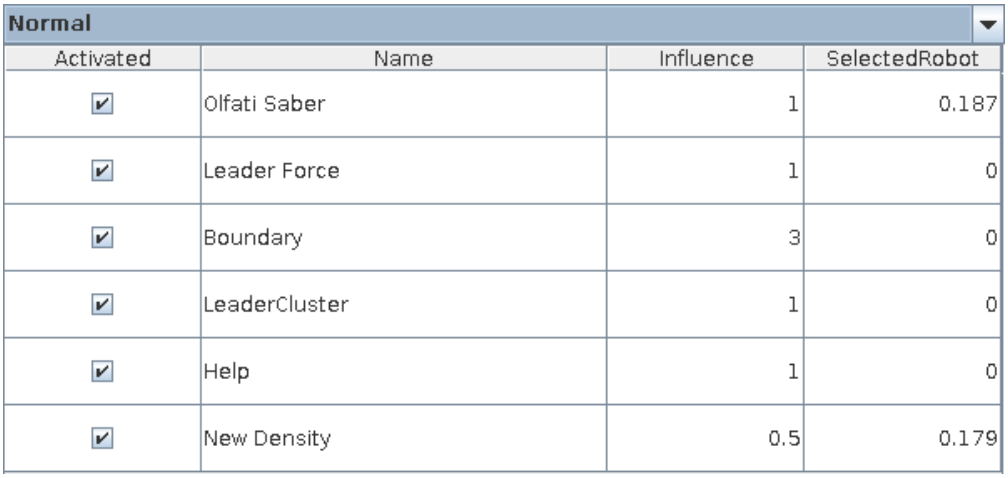


Figure 46: The GUI of the force tuner. In the first column the force can be activated or removed. The name of the force is shown in the second column. In the third column the influence can be changed and the fourth column shows the strength of the force at the selected robot.

Therefore, the only way to assess the different algorithms and their interplay is to go beyond theoretical analysis by conducting a series of experiments within a simulation environment.

In Section 12.1 the specific experimental setup is outlined followed by a description of the different robot controllers, that were tested within the experimental setup, in Section 12.2. The metrics, that were used to asses the robot controllers are introduced in Section 12.3. Finally, the results of the experiments are discussed in Section 12.4.

12.1 Experimental Setup

For ease of comparison a similar experimental base setup is chosen for all experiments. The swarm robots are randomly placed inside a circle with a radius of 7 m with approximately three robots per square meter. To ensure an equal distribution over the whole circle, the robot positions are determined by random polar coordinates where the angle is chosen by using a regular random number and the radius is chosen by using the square root of a random number:

$$r = \sqrt{\text{rand}()} * \text{radius}$$
$$\alpha = \text{rand}() * 2\pi$$

where rand() yields a random number between 0 and 1. These randomly placed robots act as mob robots. They are guided by leader robots, which are placed in various different configurations among the mob robots. For the leader configurations the star constellations *Ursa Minor*, *Taurus*, *Cancer*, *Orion*, *Cassiopeia* and a simple *line*[6] are used. These configurations are scaled and translated in such a way, that they fit well inside the circles (see Figure 47).

For all connectivity constraints regarding the n-cyclic-T-enclosedness of the swarm, a $n = 1$ is used. An experiment ends, if the leader robots are not connected by the maximal connected component of the swarm any more.

During the experiment the leader robots external movement vectors are set to the coordinates of their corresponding star in the constellation.

[6]Even though the line is not an official star constellation, it is easy to spot on both hemispheres during the whole year given a cloudless sky.

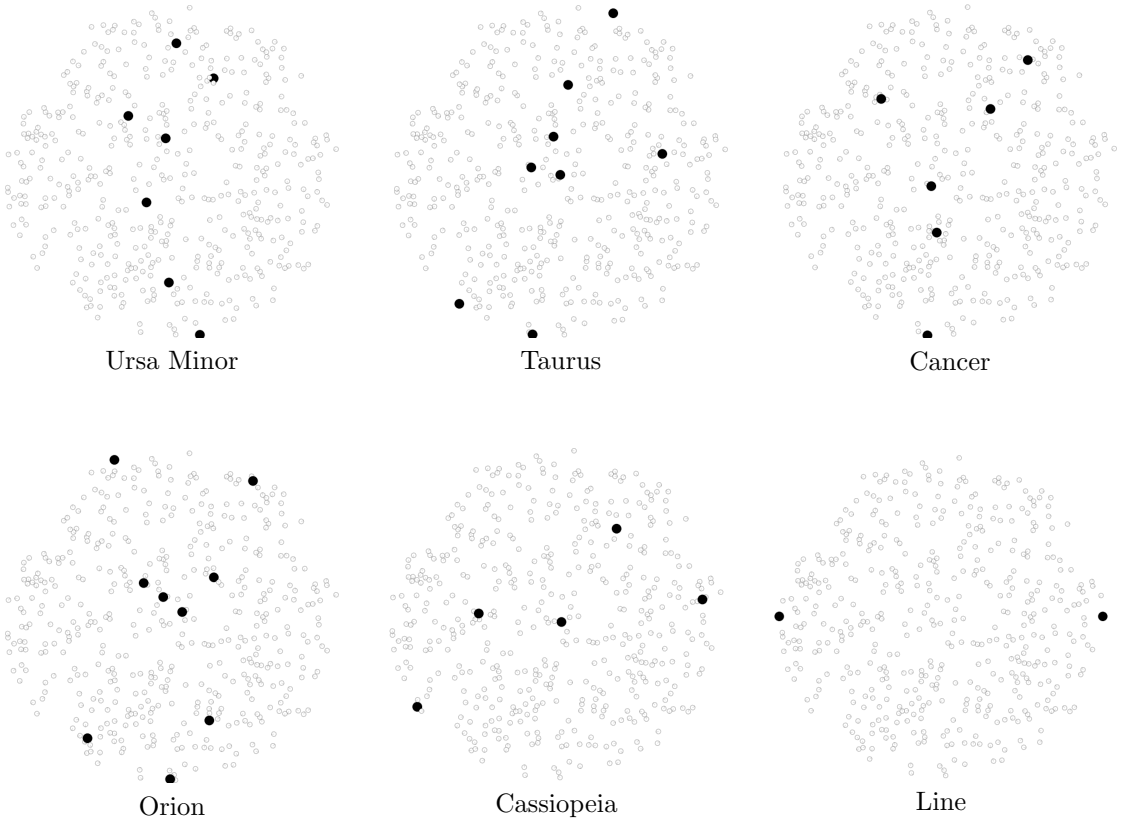


Figure 47: The five initial configurations at a radius of unit7m.

Over the course of the experiments, the constellations are artificially scaled up at a constant rate. This means that initially the leaders do not move because the constellation is not scaled yet. Later they are gradually pulled apart in different directions at different speeds depending on the distance to the center (see Figure 48).

12.2 Observed Controllers

Five different robot controllers were tested, that each combine the proposed algorithms in different ways.

Olfati-Saber controller The first and simplest controller just combines the dynamic boundary forces with Olfati-Sabers flocking algorithm. Both algorithms compute a direction vector, whose sum is used as the steering vector for the robots. Leaders are implemented by just adding the external movement vector c to this vector sum.

Base controller The second controller is used as a reference for comparison of all controllers and simply combines the steering vectors of the boundary, Olfati-Sabers flocking algorithm, the leader algorithm and the density forces. Again all steering vectors are added and the sum is used as a steering vector for the robot.

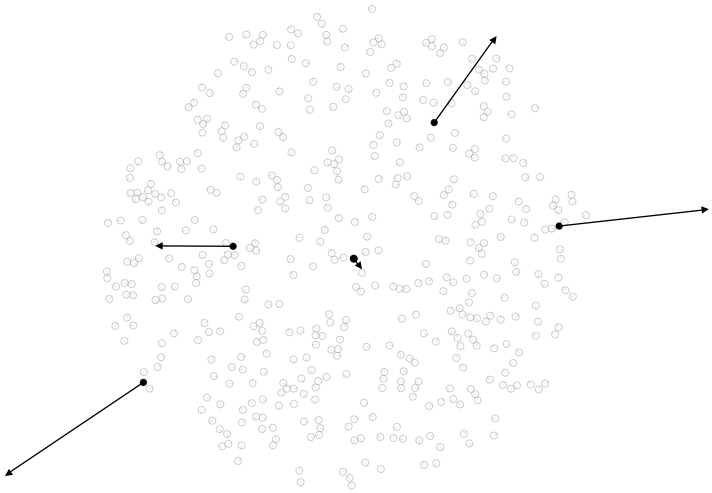


Figure 48: Cassiopeia constellation with the external movement vectors controlling the leaders. The speed at which the leader robots are pulled outwards depends on their position in the constellation. Because the constellation is scaled up artificially, leaders further away from the center are pulled apart faster.

Steiner controller The third controller extends the base controller by adding the direction vector computed by the Steiner tree optimization algorithm to the vector sum. This direction vector is weighted by the density quality as described in Definition 40.

Connectivity controller The fourth controller extends the base controller by adding the connectivity forces as described in Section 7. Leader robots are treated separately; they act as normal leader robots when inside a core, but strictly drive towards the closest core without leading the swarm when they are outside the core.

Integrated controller The last controller integrates all the algorithms in a more sophisticated way than a mere vector sum. It is described in detail in Section 10.

For each of those controllers and each of the star constellations 200 experiments are conducted which result from different seeds of the random number generator that is used for the initial placement of the mob robots. This results in a total of 6000 experiments to be conducted.

12.3 Metrics

The examined controllers are ranked based on four different performance metrics.

The *total runtime*, which designates the number of seconds a controller was simulated before the leader robots are not connected any more by the maximal connected component of the swarm. Assuming perfect robots, that do not fail, this performance metric is best to asses the ability to maintain the swarm's connectivity. For an ideal controller, this value could be infinity, and therefore not measurable.

The *well connected runtime* on the other hand, is the time it takes the n-cyclic-T-enclosedness to fall below n, where n is the degree of the core. For this, the n-cyclic-T-enclosedness during

the first 3⅓ seconds is ignored, because the initial random configuration of the swarm contains many holes, which result in bad connectivity values during the first seconds of the operation of the swarm. As discussed in Section 7, this performance measure is more appropriate when we consider fault tolerant swarms.

The *well controlled runtime* is used as an indicator for how long the leaders – and therefore the swarm – can keep up with the external steering commands they receive. Specifically it is computed by the time it takes the maximal distance between any leader and its goal position to exceed 3 m. Anything above 3 m is considered as not well controlled any more.

Finally, the *well connected and well controlled runtime* is examined, which is not explicitly measured but rather computed as the minimum of the well connected and the well controlled runtime of the swarm. It indicates how long a swarm maintains the ideal state, where it is controllable in a well connected way.

12.4 Results

Because different initial configurations – caused by different seeds for the randomly generated configurations – tend to lead to a whole range of different runtimes, for each of the star constellations and each of the four metrics, a boxplot to compare the different algorithms was created (see Figures 49, 50, 51 and 52). In addition, the generated charts, for each of the constellations and each of the controllers, a series of pictures of the simulation was taken to make it easier to compare the shapes produced by the different controllers over time (see Table 2, 3, 4, 5, 6 and 7). To ensure that each of those series is representative, they were extracted from the experiment run with the seed, that later turned out to be the median experiment concerning the total runtime.

If we look at the *total runtimes* (see Figure 49), there is a clear ranking amongst the controllers, except for the nearly identical results for the base controller and the Steiner controller. The Olfati-saber controller shows the shortest runtimes – as expected – followed by the base and Steiner controller. Finally, the integrated controller outperforms the connectivity controller in most constellations.

The boxplots for the *well connected runtimes* (see Figure 50) show the importance of the connectivity algorithm. While the Olfati-Saber, base and Steiner controller all show negligible well connected runtimes, the connectivity and integrated controller perform equally well. Nonetheless, their median well connected runtimes are only about half of their total runtime medians.

The base and the Steiner controller excel at the *well controlled runtimes* (see Figure 51) with values similar to their total runtimes, which can be explained by their missing connectivity awareness. The performance of the connectivity controller varies greatly probably due to the fact that it tends to "neglect" some leaders (see Table 6 where the upper left leader seems to have no influence after 33 seconds). The integrated controller has slightly worse performance than the base and the Steiner controller. This is expected, because it needs to show a "stalling" behavior, where it ceases to further expand because the connectivity is threatened.

Finally, the *well connected and well controlled runtimes* (see Figure 52) show, that only the integrated controller can really improve the connectivity and control the swarm at the same time. Nonetheless, it is not perfect, because the runtimes show great variance and for example the mean for the line constellation is very low.

Table 2: Cancer experiments with median total runtimes

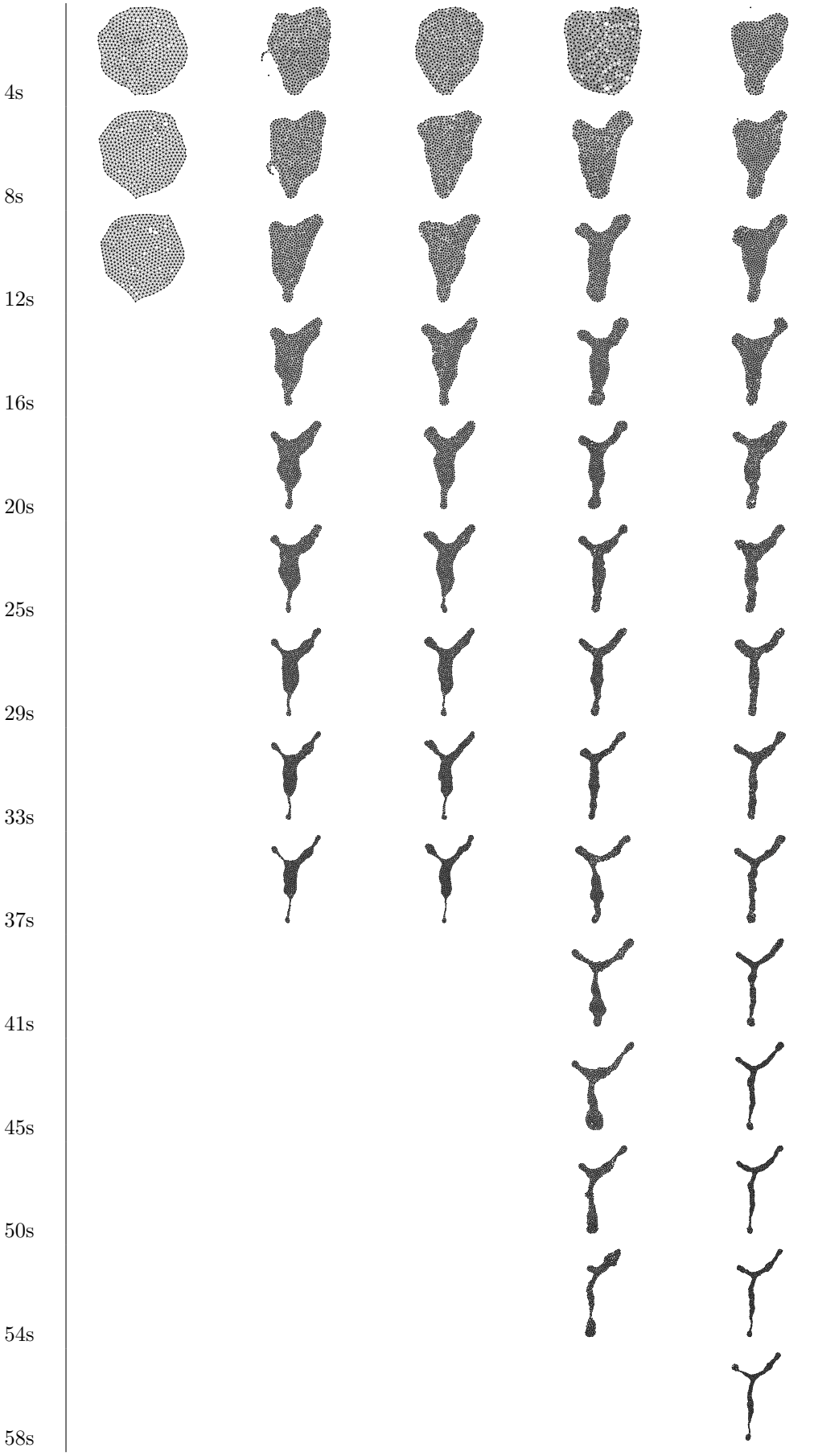


Table 3: Cassiopeia experiments with median total runtimes

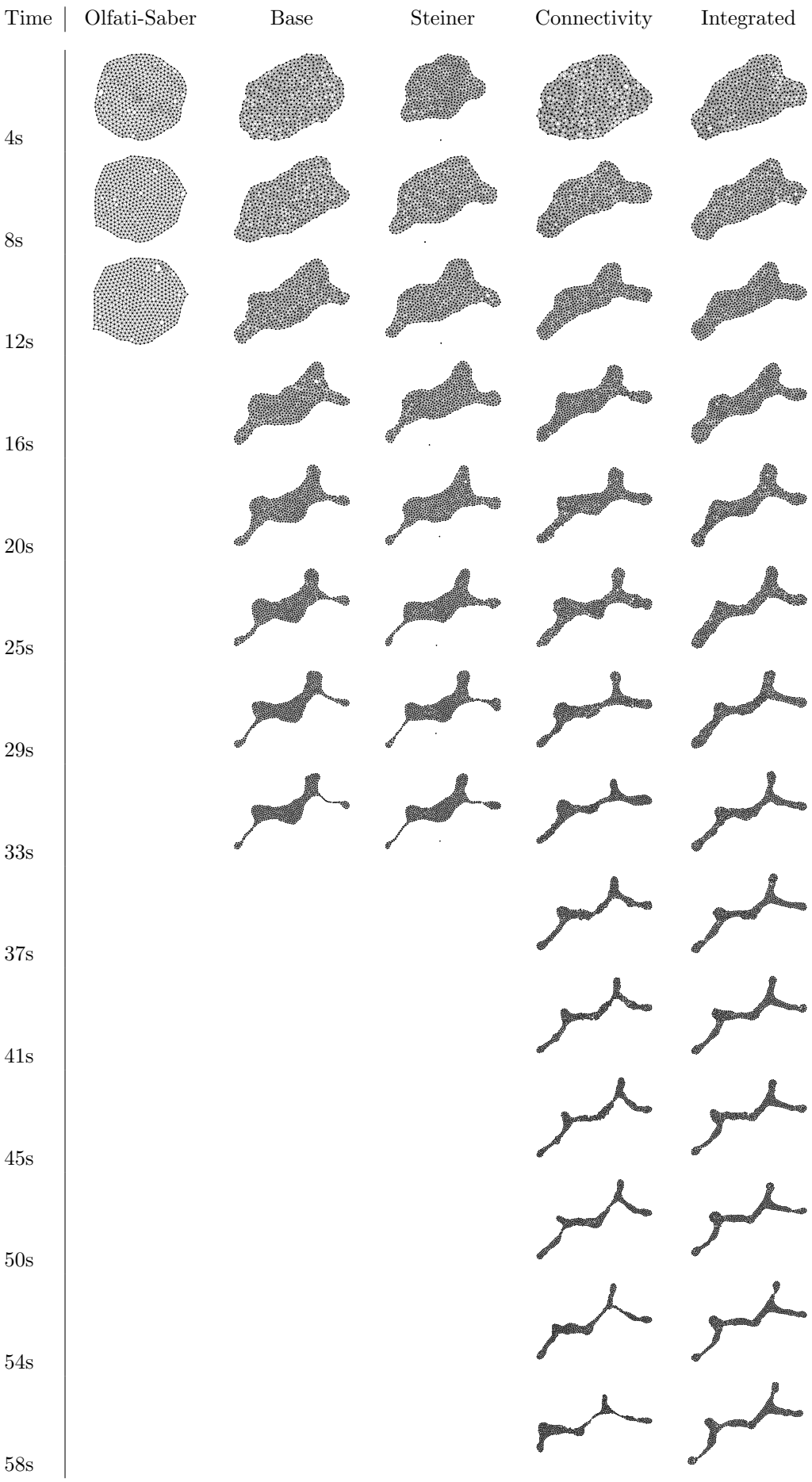


Table 4: Line experiments with median total runtimes

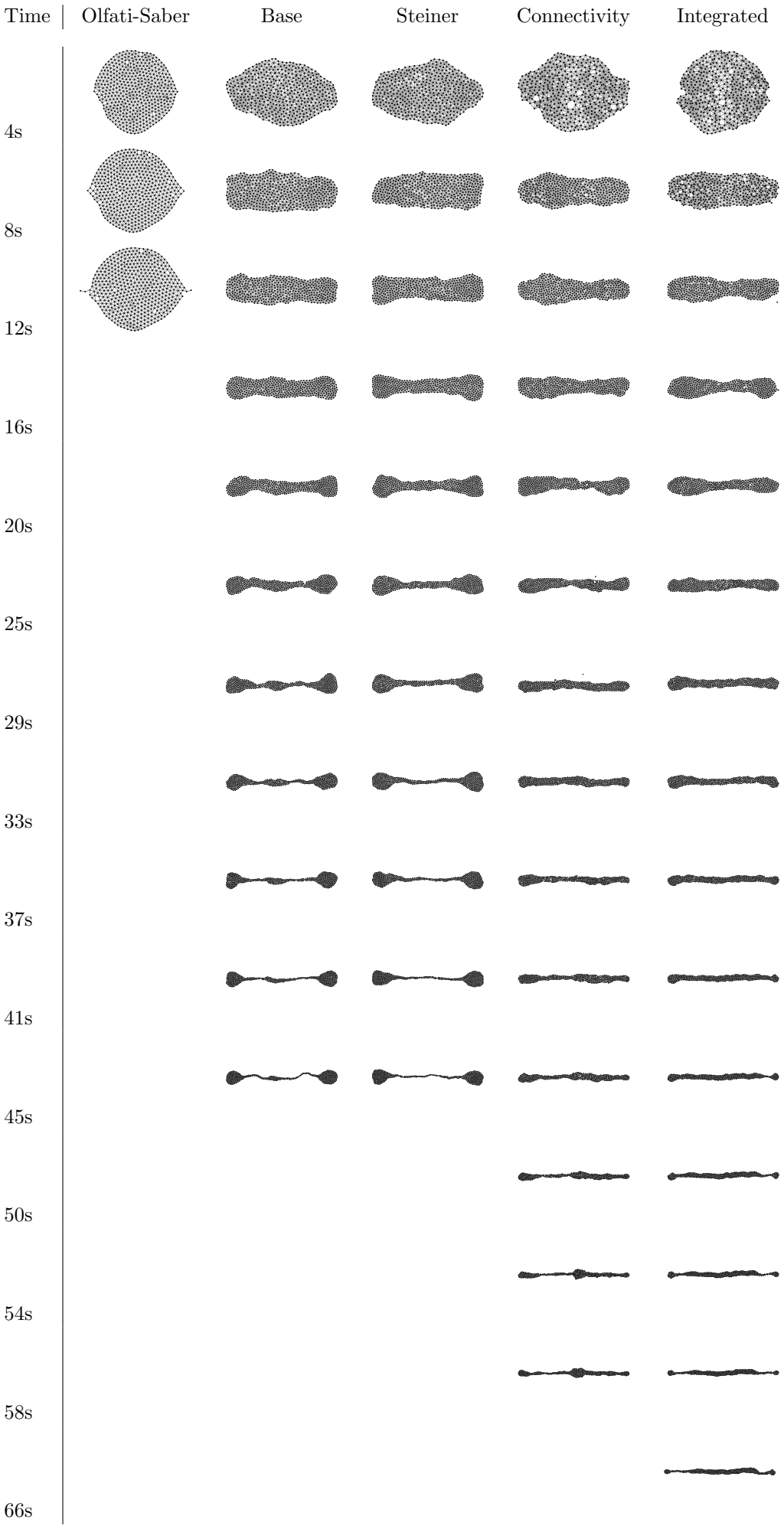


Table 5: Taurus experiments with median total runtimes

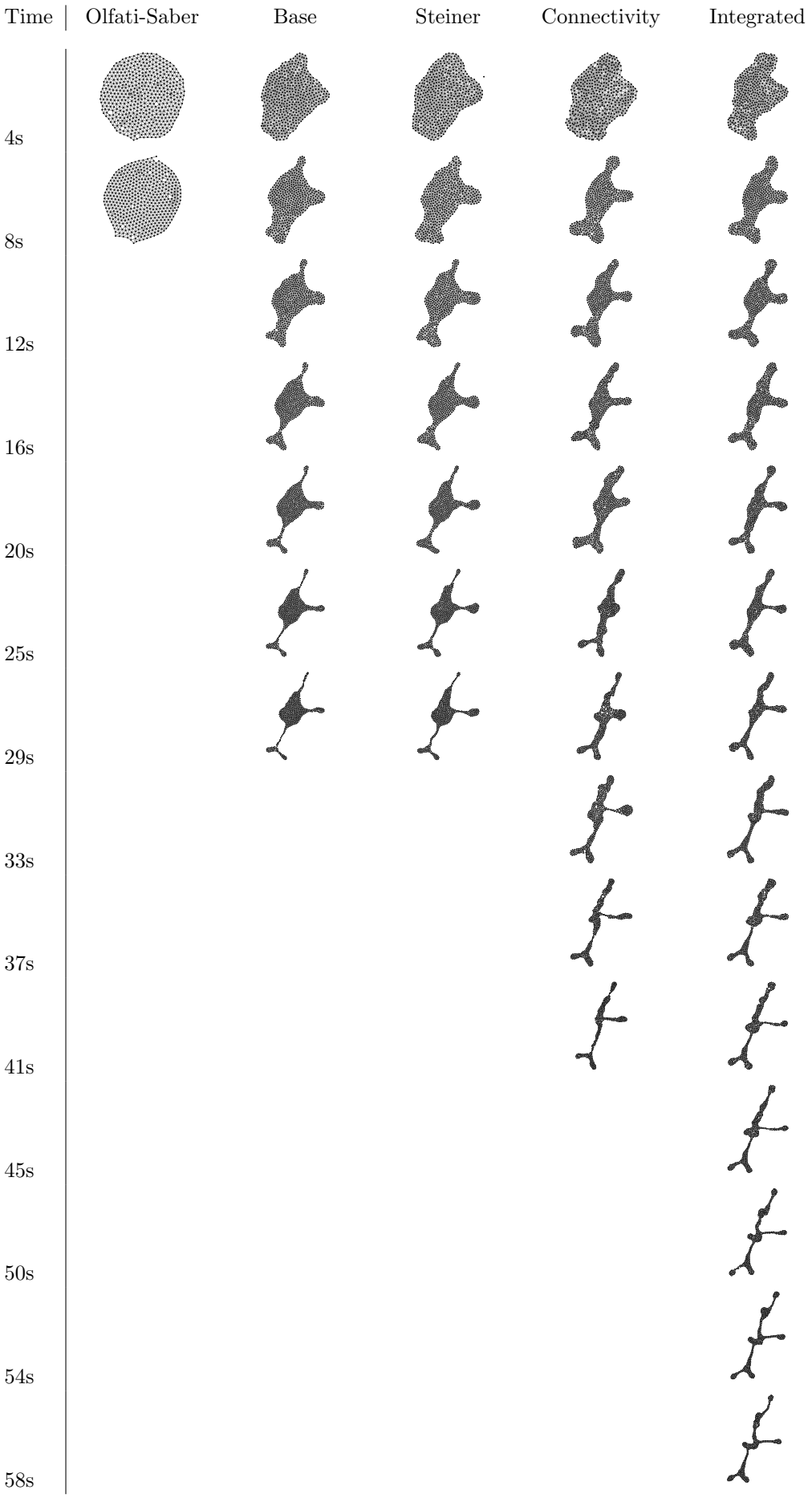


Table 6: Orion experiments with median total runtimes

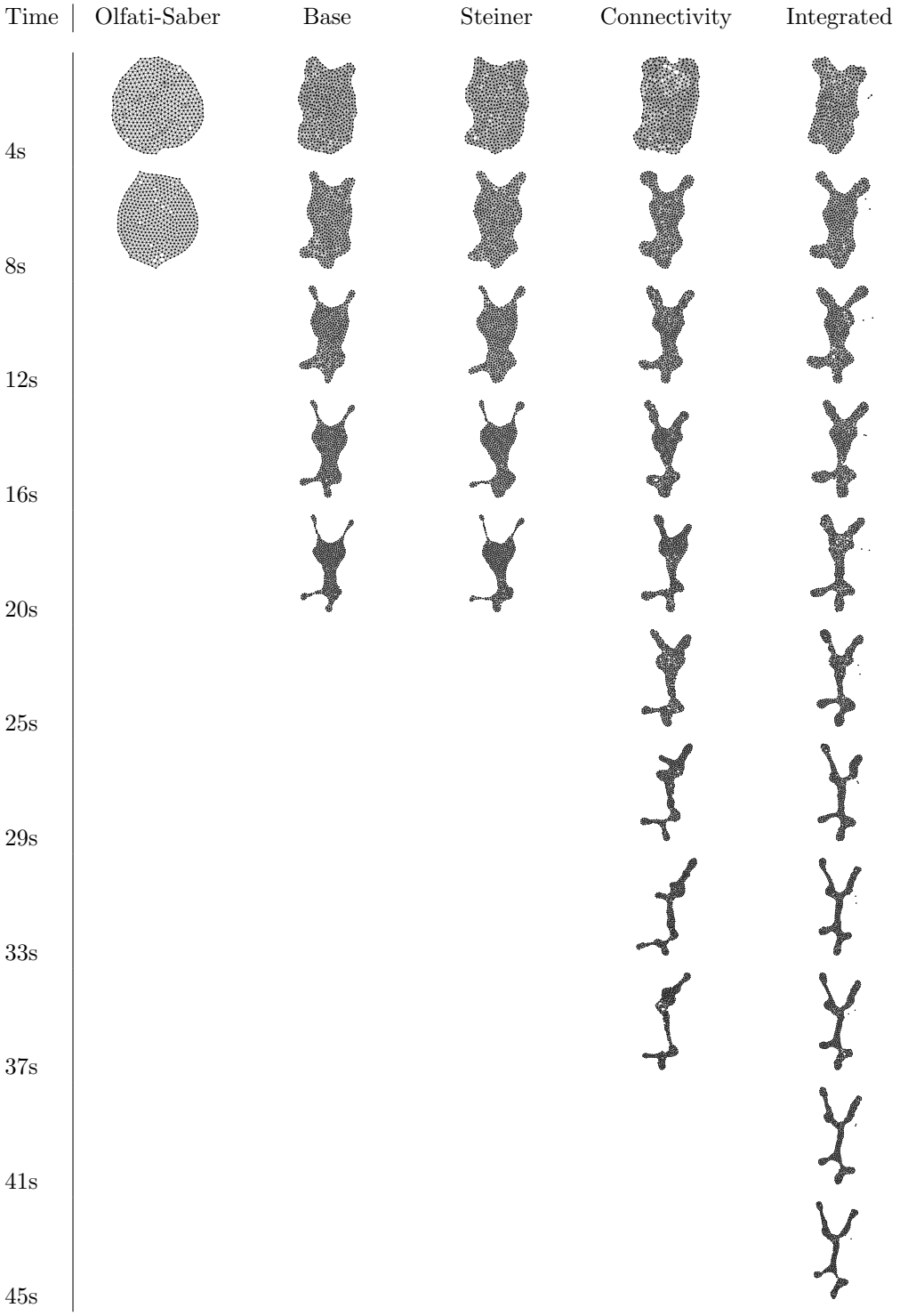
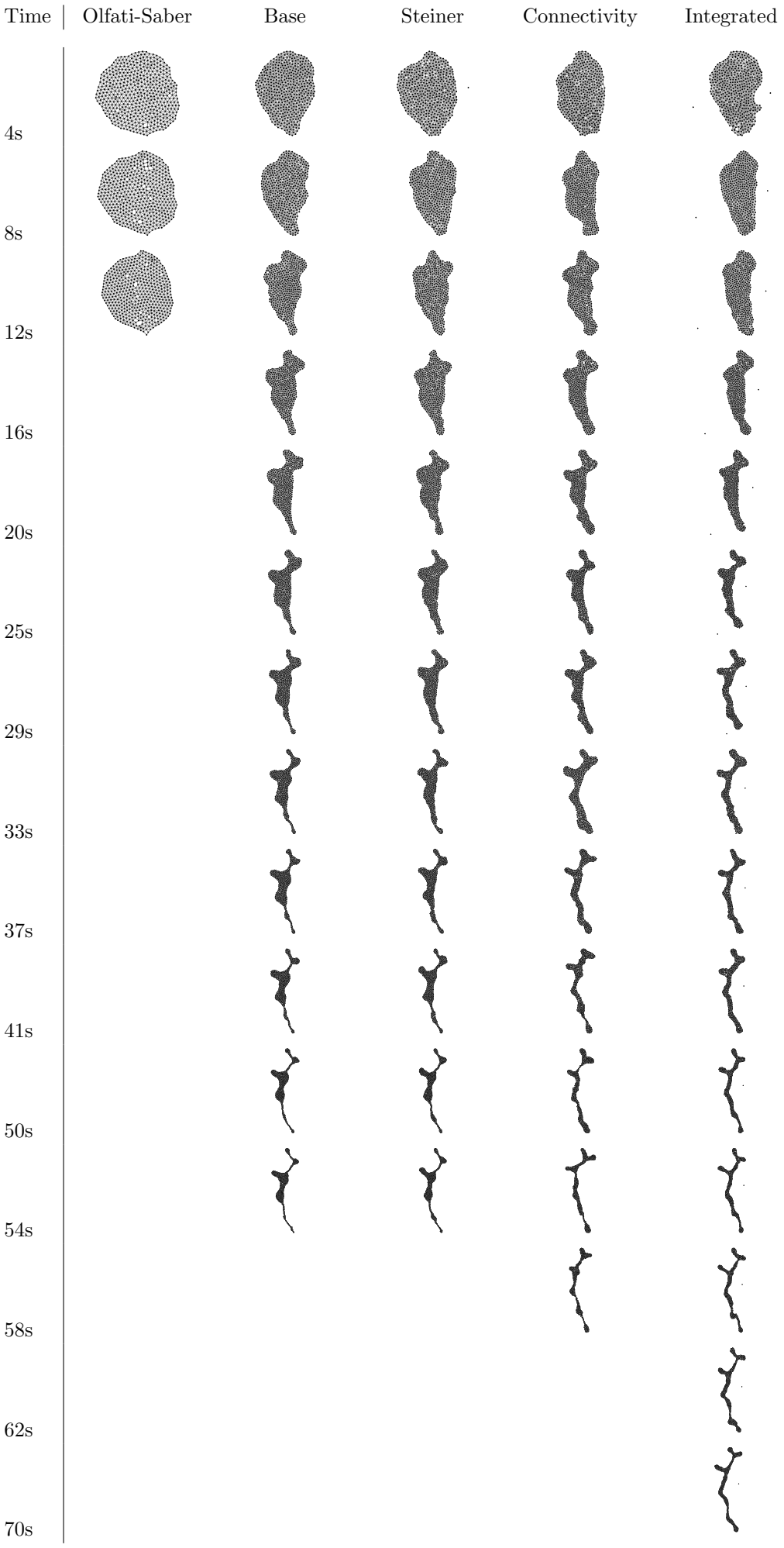


Table 7: Ursa Minor experiments with median total runtimes



13 Conclusion

In this explorative study about scalable algorithmic methods for robot swarms, we could present the following results.

First a simulator for robots based on the robot model of the R-Ones with features for rapid prototyping was created to allow for a qualitative and quantitative analysis of the swarm behaviors emerging from our algorithmic methods.

To perform a robust boundary detection while considering the line of sight constraint, the algorithms by McLurkin and Demaine [44] were extended. To allow for size aware boundary forces and the discrimination between external and internal[7] boundaries, a boundary size estimation heuristic was developed.

A heuristic method to ensure uniform robot density throughout the swarm was proposed, which introduces the characteristics of two dimensional water to the global movement of the swarm.

For communication within the swarm, Paytons pheromone model [51] was extended by a volatile pheromone model for quick invalidation of outdated pheromones. Furthermore, some proofs about the local implementation of the pheromone model and a pheromone induced broadcast forest are given.

To control the swarm externally, a leader system is introduced that allows for multiple independent leaders to manipulate the shape of the swarm.

The connectivity of the swarm is guarded by a pheromone based heuristic to detect the violation of connectivity constraints and thin regions of the swarm.

A series of experiments were conducted within the simulator to analyze the performance of the described algorithmic methods. Thereby we could show, that using the algorithmic methods, a robot swarm can be well controlled while the tolerance for single robot failure is improved and the disconnection of the swarm is delayed.

13.1 Lessons learned

After our first contact with the field of swarm robotics, several informal conclusions can be drawn.

We often looked into the seemingly similar field of distributed algorithms as a source for inspiration, but found, that those algorithms are hard to adapt to swarm robotics, because the changing topology of robot swarms poses an additional challenge. For example the distributed counting algorithms we found are based on the assumption of invariable sets to be counted, which is not always valid in swarm robotics, where e.g. the set of boundary robots is constantly changing. Generally, algorithmic methods with no defined start or termination, that constantly try to approximate a solution value worked best, because it allows the solution value to change over time.

Inspirations from nature seem to result in good algorithmic methods, although those concepts often have to be slightly adapted to working within a digital environment. Blindly copying nature was not sufficient. For example the initial pheromone model we used was based on floating point values to represent the continuous nature of chemical pheromone concentrations. This turned out to be hard to debug and finally had no advantages over a much simpler discrete model.

When trying out new strategies or new variants of a strategy, an interactive simulation and visualization environment with extensive debugging features specially tailored for the use in swarm robotics was found to be very important. It allows a qualitative analysis of the algorithmic methods and is useful to gain an intuition of the various parameters.

[7] holes

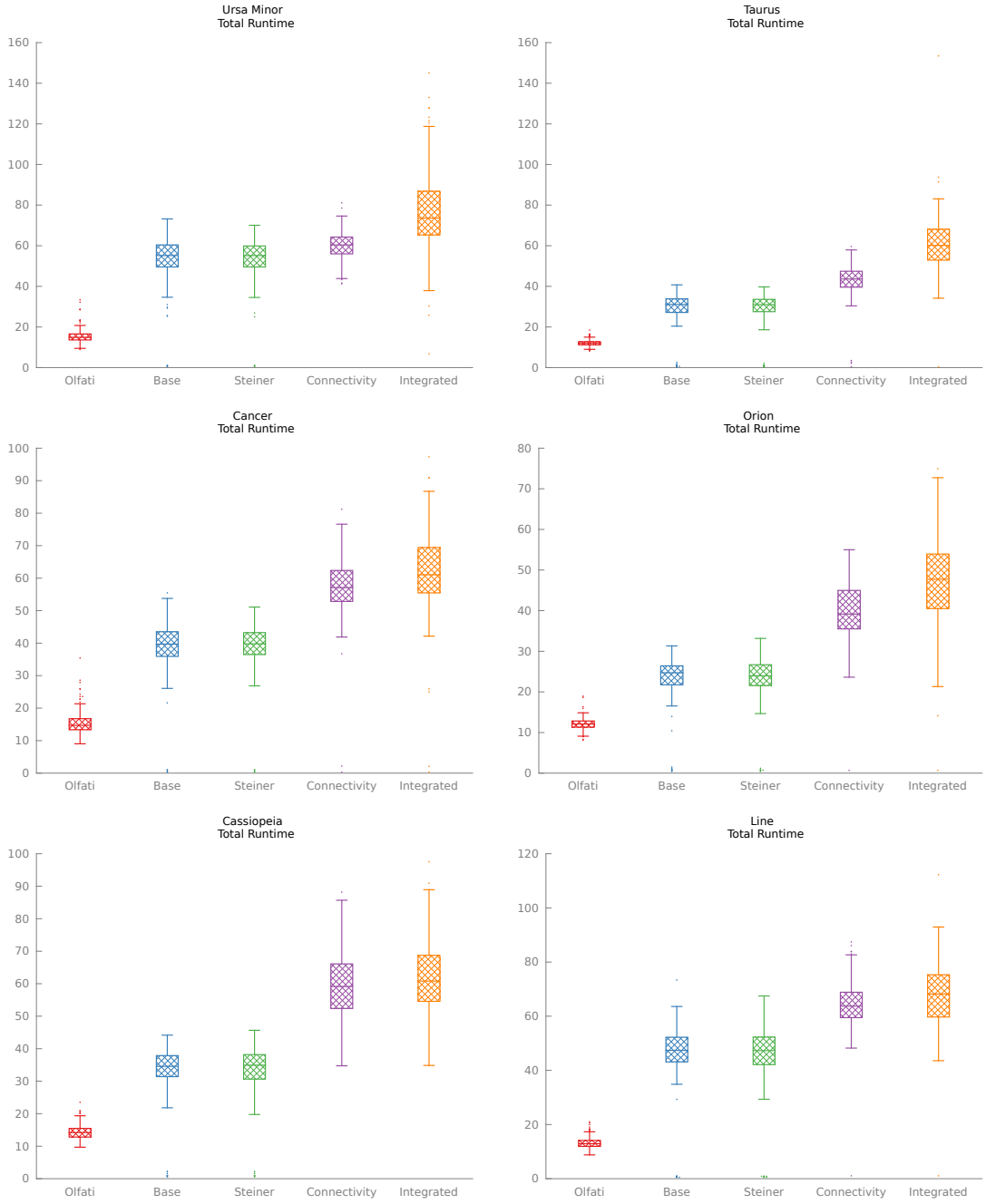


Figure 49: Total runtimes

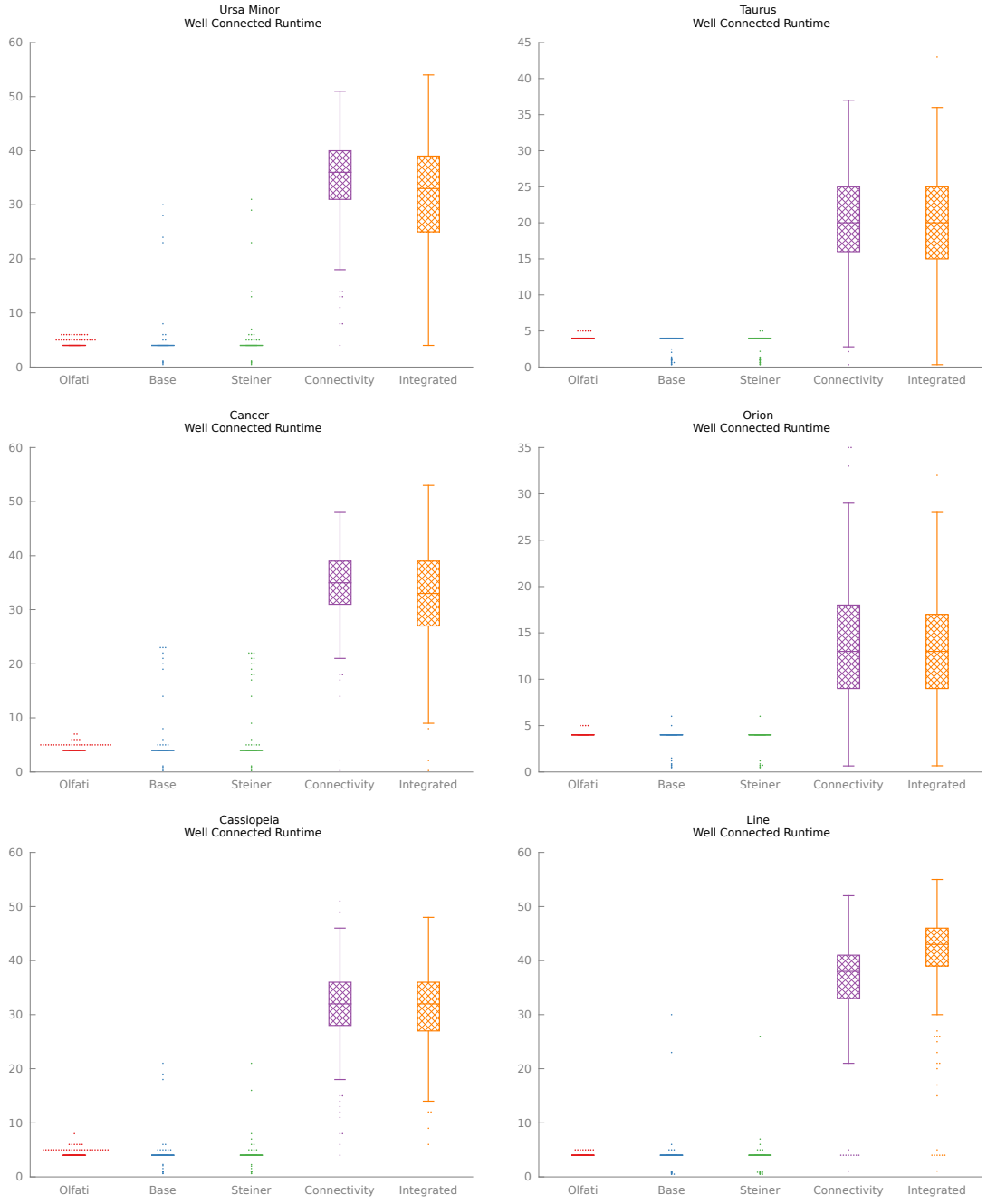


Figure 50: Well connected runtimes

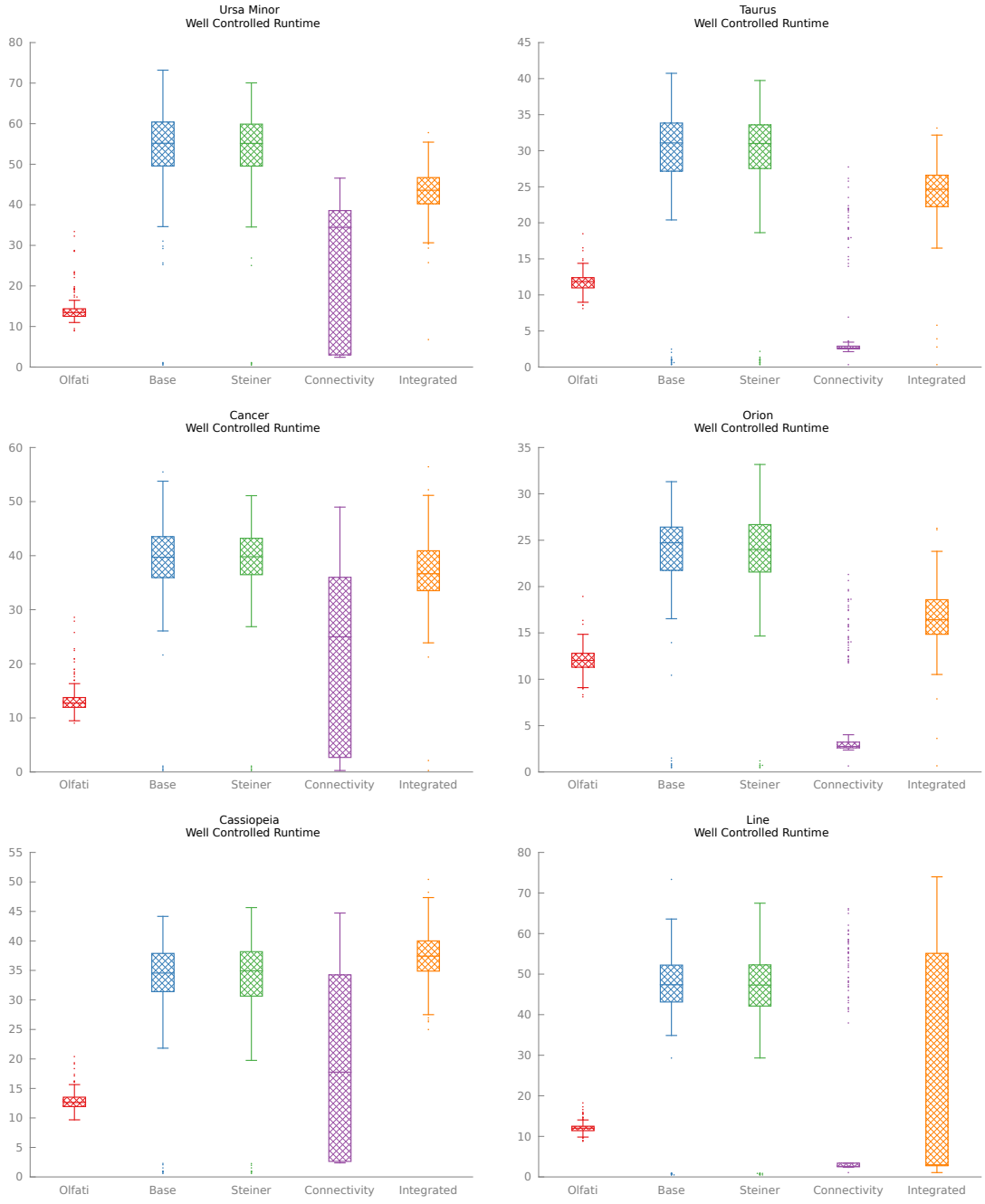


Figure 51: Well controlled runtimes

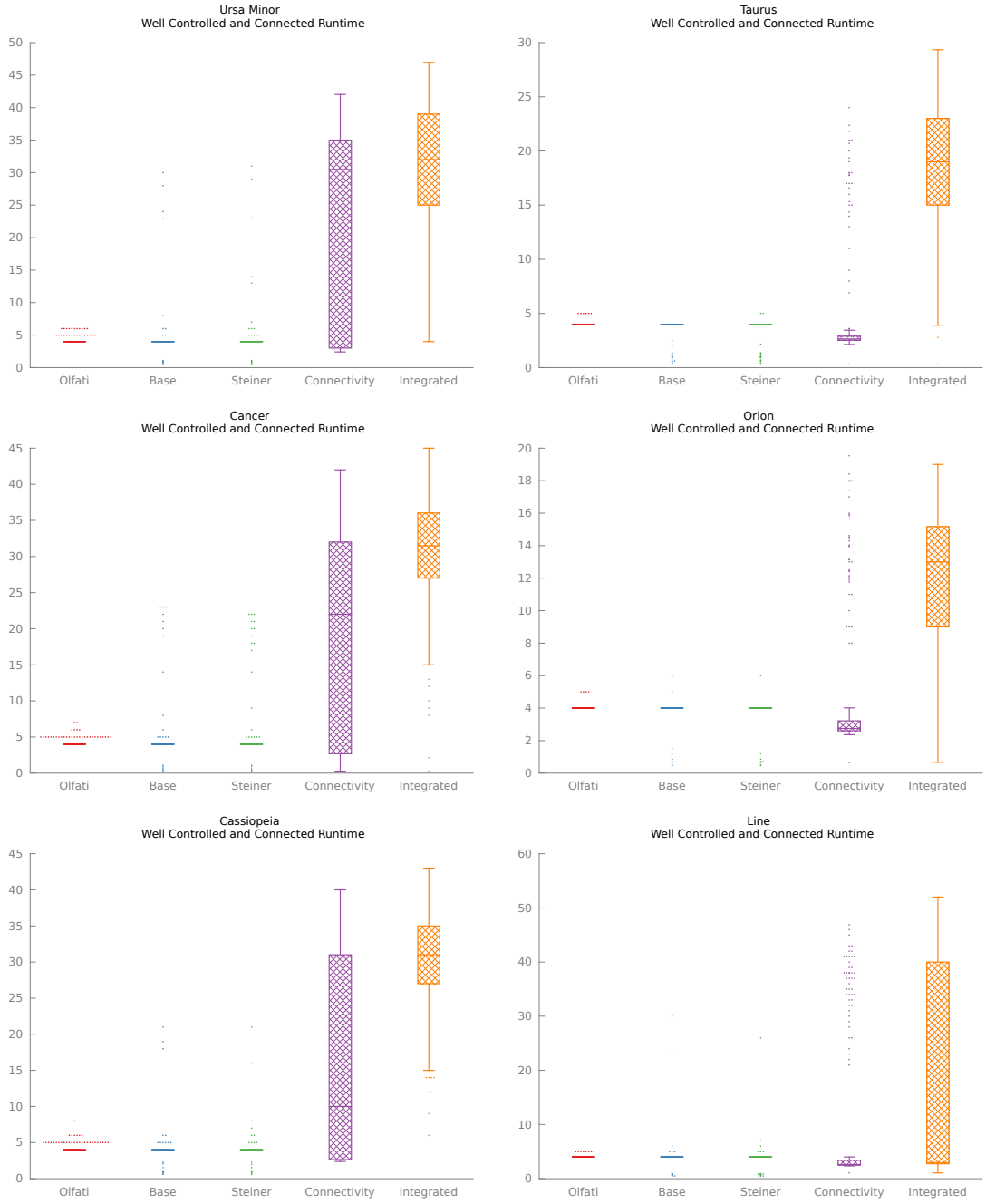


Figure 52: Well controlled and connected runtimes

When debugging robot controllers, those which relied on package based communication were so cumbersome to deal with, that we eventually switched to the method of exposed variables as a means of communication.

13.2 Future Work

While the results of the simulated experiments might look promising, tests on real hardware robots are necessary to asses the feasibility of our proposed methods. Since it is still rather expensive to maintain a large swarm of R-Ones, simulated experiments with a noise model could be a compromise. For this, the current noise model would need to be improved and verified against real hardware robots.

As the current debugging and analysis tools have proven very useful, more advanced tools, which might need less intrusive modifications of the robot controller implementations, should be considered. Still, before any new features are added, the FLOSS release of the simulator is of higher priority, so others can benefit from our work. For this, a cleaner API for writing robots controllers and debugging overlays as well as more throughout documentation of the code is necessary.

The experiments conducted so far all involve a swarm of about 460 robots. To assess the upwards and downwards scalability, experiments using smaller and bigger swarms should be considered. The local runtime complexity and behaviors will of course stay the same, but the emerging global behaviors of the swarm could vary greatly.

Only some of the proposed algorithms were proven to converge towards correct results and therefore we still have to consider them as heuristics. Nonetheless, we suspect, that with enough constraining assumptions, for many of them at least some bounds and approximation factors can be found.

For other parts of our work no proper proofs can be found, because we discovered some imperfections in hindsight. There are still some edge cases where the current boundary definition can give false positives and some assumptions about the swarm topology made by the connectivity detection heuristic can lead to false positives as well, if the swarm contains very large holes.

For the proposed connectivity metric, the additional patching edges could be equipped with negative weights based on their euclidean length to incorporate the geometrical properties of the holes into the metric.

We assume that especially the dynamic boundary force and the density distribution can still be improved. Due to time constraints, the tuning of those algorithms for the final integrated controller was not very intensive. For some experimental controllers and implementations, we observed better behaviors, which got for unknown reason lost in further integrations.

The method of robot manipulation used in this thesis was based on velocity vectors to be applied to the robots. Changing this to acceleration vectors could result in interesting other behaviors, e.g. because small forces can accumulate, when applied over a longer period of time.

After having hand-crafted all the algorithms and their integration ourselves, a design method based on genetic algorithms for robot controllers seems tempting. Instead of evolving a new controller from ground up, we think that only evolving a controller, that integrates our existing algorithms and heuristics could be interesting. For this an artificial neural network with access to all the steering vectors of the proposed algorithms, information about exposed variables of neighbors and other sensory input could be evolved together with the many tunable parameters to be found in our algorithms.

Finally, a much better integration of the algorithmic methods is possible, if only core robots react to connectivity and leader forces. In some explorative tests, no configurations resulted in a disconnected swarm even after about 175 seconds of simulation. Experiments with this modified

integrated controller could show vastly improved experimental results, but were left out due to time constraints.

References

[1] Jbox2d, 9 2014. URL http://www.jbox2d.org/.

[2] R-one website, 2014. URL http://mrsl.rice.edu/projects/r-one.

[3] Robobees, 2014. URL http://robobees.seas.harvard.edu/.

[4] Scott Aaronson. Guest column: Np-complete problems and physical reality. *SIGACT News*, 36(1):30–52, March 2005. ISSN 0163-5700. doi: 10.1145/1052796.1052804. URL http://doi.acm.org/10.1145/1052796.1052804.

[5] Xiaole Bai, Santosh Kumar, Dong Xuan, Ziqiu Yun, and Ten H Lai. Deploying wireless sensors to achieve both coverage and connectivity. In *Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, pages 131–142. ACM, 2006.

[6] Tucker Balch and Maria Hybinette. Social potentials for scalable multi-robot formations. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 73–80. IEEE, 2000. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=844042.

[7] Jan Carlo Barca and Y Ahmet Sekercioglu. Swarm robotics reviewed. *Robotica*, 31(03): 345–359, 2013. doi: 10.1017/.

[8] Mitja Bezenšek and Borut Robič. A survey of parallel and distributed algorithms for the steiner tree problem. *International Journal of Parallel Programming*, 42(2):287–319, Apr 2014. ISSN 1573-7640. doi: 10.1007/s10766-013-0243-z. URL http://dx.doi.org/10.1007/s10766-013-0243-z.

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

[10] Marcus Brazil, Ronald L. Graham, Doreen A. Thomas, and Martin Zachariasen. On the history of the euclidean steiner tree problem. *Archive for History of Exact Sciences*, 68(3): 327–354, 2014. ISSN 0003-9519. doi: 10.1007/s00407-013-0127-z.

[11] Jerome Buhl, Kerri Hicks, Esther R. Miller, Sophie Persey, Ola Alinvi, and David J. T. Sumpter. Shape and efficiency of wood ant foraging networks. *Behavioral Ecology and Sociobiology*, 63(3):451–460, Dec 2008. ISSN 1432-0762. doi: 10.1007/s00265-008-0680-7. URL http://dx.doi.org/10.1007/s00265-008-0680-7.

[12] Hande Çelikkanat and Erol Şahin. Steering self-organized robot flocks through externally guided individuals. *Neural Computing and Applications*, 19(6):849–865, 2010. URL http://link.springer.com/article/10.1007/s00521-010-0355-y.

[13] Brent N. Clark, Charles J. Colbourn, and David S. JOHNSON. Unit disk graphs. In *Discrete Mathematics 86 (1990)*, 1990.

[14] R. Courant and H. Robbins. *What is Mathematics?* Oxford Univ. Press, New York, 1941.

[15] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2010. ISBN 3642142788.

[16] GA Dirac. Short proof of menger's graph theorem. *Mathematika*, 13(1):42–44, 1966.

[17] Prasun Dutta, S. Pratik Khastgir, and Anushree Roy. Steiner trees and spanning trees in six-pin soap films. *CoRR*, abs/0806.1340, 2008.

[18] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans. Inf. Theor.*, 29(4):551–559, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1983.1056714. URL http://dx.doi.org/10.1109/TIT.1983.1056714.

[19] Joel M. Esposito. Maintaining wireless connectivity constraints for robot swarms in the presence of obstacles. *Journal of Robotics*, 2011:1–12, 2011. ISSN 1687-9619. doi: 10.1155/2011/571485. URL http://dx.doi.org/10.1155/2011/571485.

[20] Marwan Fayed and Hussein T. Mouftah. Localised alpha-shape computations for boundary recognition in sensor networks. *Ad Hoc Networks*, 7(6):1259?1269, Aug 2009. ISSN 1570-8705. doi: 10.1016/j.adhoc.2008.12.001. URL http://dx.doi.org/10.1016/j.adhoc.2008.12.001.

[21] Sándor P. Fekete, Alexander Kröller, Dennis Pfisterer, Stefan Fischer, and Carsten Buschmann. Neighborhood-based topology recognition in sensor networks. *CoRR*, cs.DS/0405058, 2004. URL http://arxiv.org/abs/cs.DS/0405058.

[22] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, (1), 2008.

[23] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983. ISSN 0164-0925. doi: 10.1145/357195.357200. URL http://doi.acm.org/10.1145/357195.357200.

[24] M. R. Garey, R. L. Graham, and D. S. Johnson. The complexity of computing steiner minimal trees. *SIAM Journal on Applied Mathematics*, 32(4):835–859, 1977. ISSN 00361399. URL http://www.jstor.org/stable/2100193.

[25] Luca Gatani, Giuseppe Lo Re, and Salvatore Gaglio. An efficient distributed algorithm for generating and updating multicast trees. *Parallel Computing*, 32(11-12):777–793, Dec 2006. ISSN 0167-8191. doi: 10.1016/j.parco.2006.09.002. URL http://dx.doi.org/10.1016/j.parco.2006.09.002.

[26] Simon Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques Marie Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989. URL http://www.springerlink.com/index/U14278K0L8K43177.pdf.

[27] Heiko Hamann and Heinz Wörn. Aggregating robots compute: An adaptive heuristic for the euclidean steiner tree problem. In *From Animals to Animats 10*, pages 447–456. Springer, 2008. ISBN 978-3-540-69133-4. doi: 10.1007/978-3-540-69134-1_44. URL http://dx.doi.org/10.1007/978-3-540-69134-1_44.

[28] Adam T. Hayes and Parsa Dormiani-Tabatabaei. Self-organized flocking with agent failure: Off-line optimization and demonstration with real robots. In *ICRA*, pages 3900–3905. IEEE, 2002. ISBN 0-7803-7273-5. URL http://dblp.uni-trier.de/db/conf/icra/icra2002.html#HayesD02.

[29] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem (Annals of Discrete Mathematics)*. North-Holland, 1992. ISBN 044489098X. URL http://www.amazon.com/Steiner-Problem-Annals-Discrete-Mathematics/dp/044489098X%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D044489098X.

[30] Hu Jiang-Ping and Yuan Hai-Wen. Collective coordination of multi-agent systems guided by multiple leaders. *Chinese Physics B*, 18(9):3777, 2009. URL http://iopscience.iop.org/1674-1056/18/9/027.

[31] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98, 1986. URL http://ijr.sagepub.com/content/5/1/90.short.

[32] Andreas Kolling, Steven Nunnally, and Michael Lewis. Towards human control of robot swarms. In *Proceedings of the seventh annual ACM/IEEE international conference on human-robot interaction*, pages 89–96. ACM, 2012.

[33] Vachaspathi P Kompella, Joseph C Pasquale, and George C Polyzos. Two distributed algorithms for multicasting multimedia information. In *Proc. Second International Conference on Computer Communications and Networks*. Citeseer, 1993.

[34] Alexander Kröller, Sándor P. Fekete, Dennis Pfisterer, and Stefan Fischer. Deterministic boundary recognition and topology extraction for large sensor networks. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 1000–1009, New York, NY, USA, 2006. ACM. ISBN 0-89871-605-5. doi: 10.1145/1109557.1109668. URL http://doi.acm.org/10.1145/1109557.1109668.

[35] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956. URL http://www.jstor.org/stable/2033241.

[36] Zhang Kun, Qi Yong, and Zhang Hong. Dynamic multicast routing algorithm for delay and delay variation-bounded steiner tree problem. *Knowledge-Based Systems*, 19(7):554–564, Nov 2006. ISSN 0950-7051. doi: 10.1016/j.knosys.2006.04.012. URL http://dx.doi.org/10.1016/j.knosys.2006.04.012.

[37] Seoung Kyou Lee. Distributed space coverage for exploration, localization, and navigation in unknown environment. 2014.

[38] Naomi Ehrich Leonard and Edward Fiorelli. Virtual leaders, artificial potentials and coordinated control of groups. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 3, pages 2968–2973. IEEE, 2001. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=980728.

[39] Tadeusz Liszka and Janusz Orkisz. The finite difference method at arbitrary irregular grids and its application in applied mechanics. *Computers & Structures*, 11(1):83–95, 1980. URL http://www.sciencedirect.com/science/article/pii/0045794980901492.

[40] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28(2):129–137, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489. URL http://dx.doi.org/10.1109/TIT.1982.1056489.

[41] Pieta K Mattila and Pekka Lappalainen. Filopodia: molecular architecture and cellular functions. *Nature reviews Molecular cell biology*, 9(6):446–454, 2008.

[42] Ralf Mayet, Jonathan Roberz, Thomas Schmickl, and Karl Crailsheim. Antbots: A feasible visual emulation of pheromone trails for swarm robots. In M. Dorigo et al., editors, *ANTS 2010*, volume 6234 of *LNCS*, pages 84–94. Springer, 2010.

[43] J. Mclurkin. Algorithms for distributed sensor networks. 1999.

[44] James McLurkin and Erik D. Demaine. A distributed boundary detection algorithm for multi-robot systems. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009. doi: 10.1109/iros.2009.5354296. URL `http://dx.doi.org/10.1109/IROS.2009.5354296`.

[45] Ciamac C. Moallemi and Benjamin Van Roy. Consensus propagation. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 52(11):4753–4766, 2006.

[46] T. Nakagaki, R. Kobayashi, Y. Nishiura, and T. Ueda. Obtaining multiple separate food sources: behavioural intelligence in the physarum plasmodium. *Proceedings of the Royal Society B: Biological Sciences*, 271(1554):2305?2310, Nov 2004. ISSN 1471-2954. doi: 10.1098/rspb.2004.2856. URL `http://dx.doi.org/10.1098/rspb.2004.2856`.

[47] Shervin Nouyan, Alexandre Campo, and Marco Dorigo. Path formation in a robot swarm. *Swarm Intelligence*, 2(1):1–23, 2008.

[48] R. Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Trans. Automat. Contr.*, 51(3):401–420, Mar 2006. ISSN 0018-9286. doi: 10.1109/tac.2005.864190. URL `http://dx.doi.org/10.1109/TAC.2005.864190`.

[49] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, November 2005. ISSN 1387-2532. doi: 10.1007/s10458-005-2631-2. URL `http://dx.doi.org/10.1007/s10458-005-2631-2`.

[50] Lynne E. Parker. Designing control laws for co op erative agent teams. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1993.

[51] David W Payton, Michael J Daily, Bruce Hoff, Michael D Howard, and Craig L Lee. Pheromone robotics. In *Intelligent Systems and Smart Manufacturing*, pages 67–75. International Society for Optics and Photonics, 2001.

[52] Processing. Processing, 2014. URL `http://processing.org/`. Processing, an open source programming language.

[53] Craig Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Published in Computer Graphics and 21(4) and July and pp and (ACM SIGGRAPH '87 Conference Proceedings and Anaheim and California and July 1987.)*, 1987. URL `http://www.cs.toronto.edu/~dt/siggraph97-course/cwr`.

[54] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. *2012 IEEE International Conference on Robotics and Automation*, 2012. doi: 10.1109/icra.2012.6224638. URL `http://dx.doi.org/10.1109/ICRA.2012.6224638`.

[55] Vydunas Saltenis. Simulation of wet film evolution and the euclidean steiner problem. *Informatica, Lith. Acad. Sci.*, 10(4):457–466, 1999. URL http://dblp.uni-trier.de/db/journals/informaticaLT/informaticaLT10.html#Saltenis99.

[56] Mark Segal and Kurt Akeley. The opengl graphics interface. Technical report, SILICON GRAPHICS COMPUTER SYSTEMS, 1994.

[57] Alexander Shekhovtsov and Václav Hlaváč. A distributed mincut/maxflow algorithm combining path augmentation and push-relabel. *International Journal of Computer Vision*, 104(3):315–342, 2013. ISSN 0920-5691. doi: 10.1007/s11263-012-0571-2. URL http://dx.doi.org/10.1007/s11263-012-0571-2.

[58] Wei-Min Shen, Cheng-Ming Chuong, and Peter Will. Simulating self-organization for multi-robot systems. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2776–2781. IEEE, 2002. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.124.1827.

[59] Wei-Min Shen, Peter Will, Aram Galstyan, and Cheng-Ming Chuong. Hormone-inspired self-organization and distributed control of robotic swarms. *Autonomous Robots*, 17(1): 93–105, 2004. URL http://link.springer.com/article/10.1023/B:AURO.0000032940.08116.f1.

[60] Gurdip Singh and Kusuma Vellanki. A distributed protocol for constructing multicast trees. In *OPODIS*, pages 61–76. Citeseer, 1998.

[61] Onur Soysal and Erol Sahin. Probabilistic aggregation strategies in swarm robotic systems. In *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, pages 325–332. IEEE, 2005.

[62] William M Spears, Diana F Spears, Jerry C Hamann, and Rodney Heil. Distributed, physics-based control of swarms of vehicles. *Autonomous Robots*, 17(2-3):137–162, 2004. URL http://link.springer.com/article/10.1023/B:AURO.0000033970.96785.f2.

[63] William M Spears, Diana F Spears, Rodney Heil, Wesley Kerr, and Suranga Hettiarachchi. *An overview of physicomimetics*. Springer, 2005.

[64] Yuan Gong Sun, Long Wang, and Guangming Xie. Average consensus in networks of dynamic agents with switching topologies and multiple time-varying delays. *Systems & Control Letters*, 57(2):175–183, 2008. ISSN 0167-6911. doi: 10.1016/j.sysconle.2007.08.009. URL http://dx.doi.org/10.1016/j.sysconle.2007.08.009.

[65] A. M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 237(641):37–72, Aug 1952. ISSN 1471-2970. doi: 10.1098/rstb.1952.0012. URL http://dx.doi.org/10.1098/rstb.1952.0012.

[66] JP Vite and GB Pitman. Bark beetle aggregation: Effects of feeding on the release of pheromones in dendroctonus and ips. 1968. URL http://www.nature.com/nature/journal/v218/n5137/abs/218169a0.html.

[67] Yue Wang, Jie Gao Joseph S.B. Mitchell, Dept. of Computer, Science Dept. of Applied, Math., Statistics, Stony Brook, University Stony, Brook University, Stony Brook, NY Stony Brook, and NY. Boundary recognition in sensor networks by topological methods. 2006.

[68] David M Warme, Pawel Winter, and Martin Zachariasen. Exact algorithms for plane steiner tree problems: A computational study. In *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, 1998.

[69] Alan F. T. Winfield and Julien Nembrini. Safety in numbers: fault-tolerance in robot swarms. In *30 Int. J. Modelling, Identification and Control, Vol. 1, No 1, 2006*, 2006.

[70] Lin Xiao, Stephen Boyd, and Seung-Jean Kim. Distributed average consensus with least-mean-square deviation. *Journal of Parallel and Distributed Computing*, 67(1):33–46, Jan 2007. ISSN 0743-7315. doi: 10.1016/j.jpdc.2006.08.010. URL `http://dx.doi.org/10.1016/j.jpdc.2006.08.010`.

[71] Alexander Zelikovsky. Better approximation bounds for the network and euclidean steiner tree problems. Technical report, Charlottesville, VA, USA, 1996.