Project Thesis

Structured Triangulation in Multi-Robot-Systems

Dominik Krupke

March 15, 2016

Institute of Operating Systems and Computer Networks
Prof. Fekete

Supervisor:
Prof. Dr. Sándor Fekete

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, March 15, 2016

_____

Abstract

Simple robots are not able to map their environment due to missing sensors or limited computational capabilities. This strongly limits their capabilities, e.g. in exploration as it cannot remember if it has visited a place before. With a swarm of such simple robots, many problems efficiently can be solved by cooperation. This thesis considers robots only knowing the distances to neighbors as well as sensing collisions. These robots build an infrastructure by triangulating their environment (using robots as nodes and communication links as edges) to support other robots. This thesis builds upon and extends work of Lee et al. [1] that implemented this for robots with bearing sensors. Further, we consider the removal of robots from this triangulation such that the created areas are of limited size. This creates a street network with a specific density that can be reached quickly from any point. We show that it is NP-hard to remove a maximum amount of robots such that the size constraints are still fulfilled and give a practical heuristic. The methods in this thesis have been implemented and tested in a simulator with simulation experiments provided at the end of the corresponding chapters.

Contents

List of Figures

List of Tables

1 Introduction

There are some important scenarios such as the exploration of a large unknown environment or its surveillance where it can be advantageous to use a large amount of very simple robots instead of a single complex robot. Such a swarm of simple robots can not only exploit parallelism but can also robustly perform in dangerous environments where the destruction of some of its members is likely. Further, the swarm can be scaled in size to adapt to the size of the environment. However, to make a swarm of robots affordable the capabilities of its robots have to be seriously limited. While an advanced mobile robot is equipped with powerful sensors and computational capabilities that allow it to map the environment and localize itself in it (SLAM), a swarm robot might only sense collisions and neighbors (two swarm robot platforms are presented later in this section). Swarm robots can overcome these strong limitations by cooperation.

Imagine you want to implement a surveillance for an unknown environment. A single robot would need to create a map of the environment, localize itself in it, and remember when it has visited any position last. Now assume you have a swarm of very simple robots that can built a triangle mesh in the environment (see Fig. 1.1). A localization in a triangle of such an artificial structure can be implemented using only distance information to the corresponding three vertex robots. Further, the static vertex robots can save when this triangle has been visited the last time and also give information about the last visit of adjacent triangles. A set of simple robots only equipped with surveillance sensors and neighbor distance estimation can now perform the surveillance task by moving from triangle to the adjacent triangle with the oldest visitation timestamp.

In the first part of this thesis we consider constructing such a triangulation with robots that only know the distances to their neighbors and sense collisions with the environment. Further, the robots do not have real odometry and can only guess how much they moved or rotated based on time. In the second part of the thesis we consider how to remove robots from this triangulation such that a street network remains where the distance of any point in the environment to it is limited. This can be used if e.g. the surveillance sensor range exceeds the communication range of the simple robots and the amount of robots is limited. See Fig. 1.2 for a visualization of a surveillance robot with long view that can reach every point of the environment from some triangle as the holes are of limited size. Further motivations for such a network are provided in the introduction of the corresponding chapter.

Our robot model is motivated by the Kilobots developed by Rubenstein et al. [2] in Harvard that can be seen in Fig. 1.3a. The developer claim that it has only 14$ costs in parts and can be assembled in 5 minutes. The Kilobot has a radius of less than 2 cm and slap-

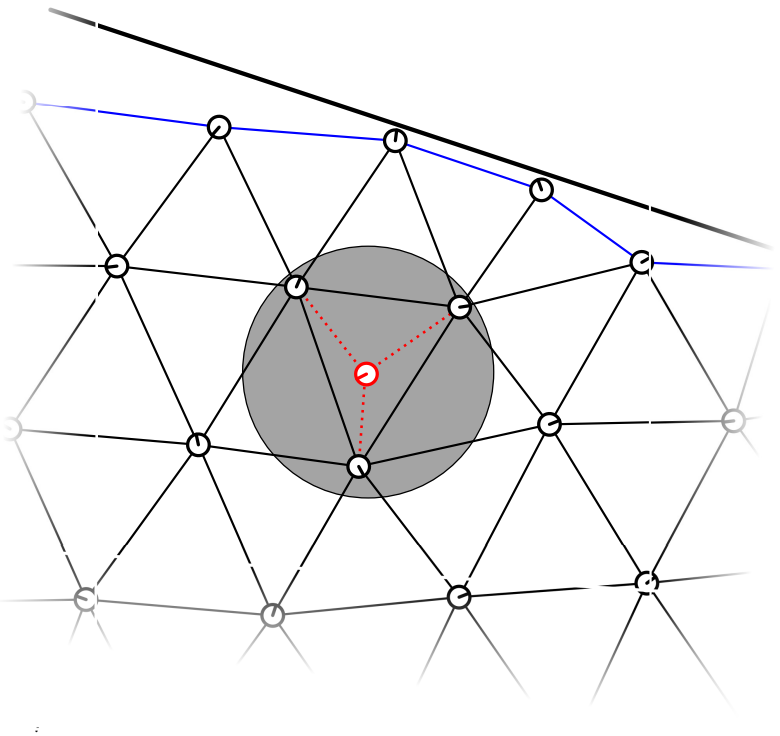


Figure 1.1: The surveillance robot (red) using a triangulation. The blue edges mark the boundary, the gray circle the surveillance range.

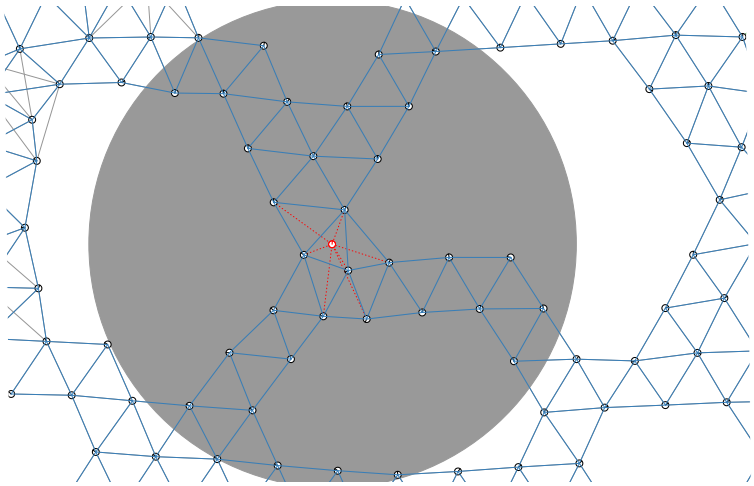
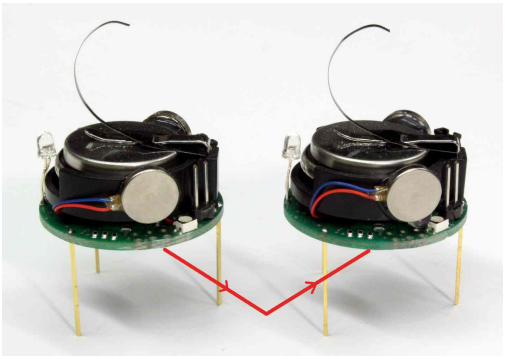


Figure 1.2: A triangulation with holes of limited size such that every point is still in range (gray) of the surveillance robot from some point of the network.



(a) Kilobots — Photo taken from the corresponding paper [2]

(b) R-One

Figure 1.3: Swarm robot platforms

stick based locomotion (no odometry) with a velocity of $1\,\mathrm{cm\,s^{-1}}$ and 45 degrees/s. Its communication is based on infrared and provides a datarate of $30\,\mathrm{kbit\,s^{-1}}$ with a range of 10 cm combined with a 10bit resolution distance sensing. It is powered by an Atmega328 with 8 MHz, 2 kB SRAM, and 32 kB flash memory. The 3.4 V 160 mAh lithium-ion battery is sufficient for 3-24 hours of operation. Further, collective power, charging, and programming is possible. The low communication range (compared to size) of the Kilobot and its missing collision sensor make it impractical for our method but these limitations are easy to overcome.

There are other swarm robot platforms as the R-One [3], the e-puck [4], Colias [5], and the i-Swarm [6]. Their capabilities depend on the used techniques and are very heterogeneous.

There has been previous work by Lee et al.[1] that considered constructing such triangulations with another robot model. Instead of distance knowledge, the robots have only low resolution bearing sensors. They implemented their approach using the R-One platform. The R-One swarm robot of the Rice University [3] (see Fig. 1.3b) is more powerful than the Kilobot but also has a higher price of 220$. Its Texas Instruments LM3S8962 Stellaris Processor with 50 MHz, 64 kB flash, and 256 kB SRAM provides a multiple of the computational power of the Kilobot. Instead of a single infrared receiver and transmitter, the R-One has 8 with a range of 1.2 m that are used for communication, neighbor localization, and obstacle detection. The different headings of the infrared receiver and transmitter allow the R-Ones to differ between 16 directions. Its 8 bump sensors allow it to also detect collisions and their direction. The battery provides 4 hours of operation time.

1.1 Overview

In Chapter 2 we state the notation and the used robot model. We also briefly describe the usage of public variables and virtual pheromones, as well as mention the simulator and the implementation. In Chapter 3 we give an overview of previous and related work. Of special importance is the description of the triangulation algorithm of Lee et al. [1] as it lays the foundation for our algorithm. In Chapter 4 we first describe how to perform the elementary tasks of localization and motion. Then we describe our algorithm for triangulation with only distance information. Afterwards we consider how to parallelize the triangulation process and how to detect redundant robots. In Chapter 5 we consider the removal of robots in the triangulation such that the emerging holes are of limited size. We show that this problem is NP-hard and provide a heuristic that can be executed parallel to the triangulation. Chapters 4 and 5 both end with an experiment section. Finally, in Chapter 6 we give a short conclusion and mention future work.

The code of the implementation in a simulator and a video of the simulation can be found at `https://github.com/SwarmRoboticResearch/Structured-Triangulation-in-Multi-Robot-Systems`. It is highly recommended to watch the video to get a quick impression of the result of this thesis.

2 Preliminaries

In this chapter a short overview of the robot model, notations, and techniques is given.

2.1 Robots

Robots do have a disk-shape of a positive radius and thus can not only collide with the environment but also with each other. They can move forwards and rotate but only can make a vague guess on how much they actually moved or rotated based on the elapsed time. A robot can send messages that are received by all robots within a specific range *MAX_RANGE* and whose line of sight to the robot is not blocked by a wall. By receiving a message, a robot can sense the existence of the sender and the distance to it. A robot determines its neighborhood only by the messages it receives and in especially cannot distinguish if a neighbor has left the range or all messages of it have gone lost.

We denote the set of all robots by $\mathcal{R}$. $|ab|$ is the distance between the centers of the robots a and b. $N(r) = \{n \in \mathcal{R} \mid |nr| \leq \mathit{MAX\_RANGE}\}$ provides all robots in the neighborhood of r. As the robots are not aware of the actual values, in algorithms distance and neighborhood refer to the local estimations. Each robot has a unique id and we use it as representative for a robot, i.e. for a robot r we use r to denote the robot as well as its id.

2.2 Definitions and Notation

Angles are measured counterclockwise and in radian. E.g. $\measuredangle_{-\pi,\pi}(a,b)$ denotes the counterclockwise angle from vector a to vector b normalized to the range $(-\pi, \pi]$.

The hop distance between two robots refers to the amount of messages that are needed to forward a message between the two robots. Thus, it equals the amount of edges of the shortest path in the connectivity graph. The connectivity graph is an undirected graph $G = (V, E)$ with the vertex set $V = \mathcal{R}$ and E defined by the neighborhood relation $N(\cdot)$ of above.

2.3 Public Variables and Virtual Pheromones

Many algorithms involve the internal state of the neighbored robots that goes beyond what sensors can perceive. For this, communication between the robot and its neighbors is needed. Instead of exchanging the state via messages in the algorithms, the robots provide their states as public variables that can be read by their neighbors. This can easily be implemented by letting all robots broadcast their state (all public variables) frequently. Old neighbor states are dismissed as soon as a newer one of the corresponding neighbor is received (the public variables are updated) or they reach a specific age (the corresponding neighbor is assumed to be vanished and thus deleted).

We denote the access of a public variable x from a neighbor n by the point notation $n.x$ as used for member variable access in programming languages like $C++$ and Java.

This allows us to write simple algorithms as determining the hop distance to a specific robot g by the following algorithm:

$$r.\text{hop\_dist} = \min\{n.\text{hop\_dist} \mid n \in N(r)\} + 1$$

The goal robot g has the constant value $g.hop\_dist = 0$ while all other robots initially start at infinity. Note that this equals a distributed and continuous version of the common Bellman-Ford algorithm. Also, the id of a robot can be assumed to be a public variable $r.id$.

Public variables such as the hop distance implementation provide a gradient that can be used to move towards the goal. This has similarities to the evaporating pheromone traces of insects like ants. For this reason we denote such a kind of public variables as (virtual) pheromones. Of course this is just an example of a general technique that is based on emitting and weakened passing.

Virtual pheromones are a common technique in swarm robotics and there are many implementations, applications, and notations. McLurkin [7] describes a simple model for wireless sensor networks similar to the above one. Another model has been given by Payton et al. [8]. Virtually emitting pheromones in tile world (not robot bound) is done by Shen et al. [9, 10] to e.g. form patterns and to perform tasks like aggregating around a goal. Hrolenok et al. [11] use beacons to distribute the pheromones.

There are also physical implementations that are closer to nature such as using heat traces [12], odour traces [13], alcohol traces [14], phosphorescent glowing paint [15], RFID [16], and a display on the ground [17].

In this thesis we stick with the simple Bellman-Ford variant and leave e.g. dynamic adaption of the weakening factor to future work.

2.4 Simulator and Implementation

To validate the proposed methods, they have been implemented and tested in a simulator that has been implemented in a previous work [18]. The simulator is written in Java and uses internally the JBox2D physics engine [19]. The simulation is performed in discrete time steps. In every time step the robots can perform arbitrary calculations but the actual actions and updates are only performed between two time steps. Messages are received in the next time step. The provided neighborhood model of the simulator matches the actual state and provides not only distances but the relative positions. However, it is wrapped by a more realistic and limited neighborhood as described below.

We use the following robot model: Robots have a radius of 5cm and a range of 1.2m (center to center). The communication is blocked by obstacles intersecting the line of sight between the centers of the two robots. The internal neighborhood model updates the state of a neighbor (especially the distance) with every received message. If no messages from a robot have been received for a specific time (measured in time steps), the neighbor is deleted. A robot sends a broadcast with its public variables in every round with

a probability of 50%. This emulates an unreliable communication channel and induces asynchrony in the information distribution. The velocities (straight forward and rotation) are randomly varied with up to 20% but with a maximal velocity of $0.2\,\mathrm{m\,s^{-1}}$ and a time step length of $1/60\,\mathrm{s}$ this is a rather low distortion. The measured distances are randomly distorted by up to 1%. These measurements are *not* further smoothed/improved by the algorithms (e.g. by averaging them over multiple time steps).

The link to the material and code of this thesis has been given in Chapter 1.

3 Previous Work

3.1 Triangulation

There already has been some work on building and using static robot triangulations. The process of building such a triangulation is primarily focussed on setting the points (robots) such that only a small amount of robots is used, the triangulated area is maximized and/or the resulting triangles are of high quality. The edges are communication links and thus are of limited length.

Fekete et al. [20] give some theoretical foundations of building such triangulations. They showed that the minimum relay triangulation problem (MRTP) (the term relay and robot are used synonymously) in which the polygonal environment has to be triangulated by a minimum amount of robots is NP-hard even for the offline version in which the environment is known in advance. They also show that the offline version of the maximum area triangulation problem (MATP) in which a maximum area has to be triangulated with a limited amount of robots is NP-hard, too. An online algorithm (only local environment within range of a robot known) for the MRTP that achieves a competitive ratio of 3 is given. For the online version of MATP (OMATP) no such algorithm can be given. They also provide polynomial approximation scheme algorithms for both offline versions. The limitation on orthogonal polygonal environments with integer length edges is considered in [21].

For the above mentioned 3-competitive online algorithm for MRTP of Fekete et al. [20] the robots will enter the environment through a gate of two static robots. First the boundary is covered with robots, then the interior by unit size triangles, and last these two are connected. An example can be seen in Fig. 3.1.

Lee et al. [1] continued the work on a more practical level. They provide a greedy algorithm for OMATP that covers the area in a breath first pattern and gives a completeness

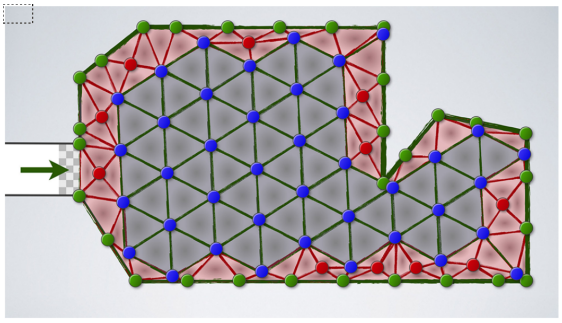


Figure 3.1: Online triangulation for MRTP by Fekete et al. [20].

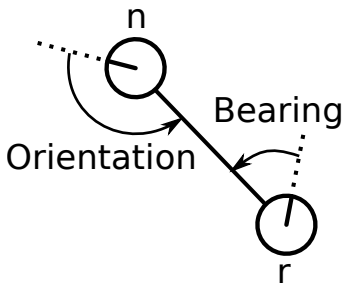


Figure 3.2: Bearing and orientation from the view of r on neighbor n.

guarantee. It is validated in experiments with real robots using only low resolution bearing sensors. They also elect a corresponding robot for each triangle which can be used to store information about it. These corresponding robots are also adjacent to the corresponding robot of each adjacent triangle and thus build a dual graph. They prove that this dual graph can be used for navigation and yields paths that are only longer by a constant factor than the direct euclidean distance (the constant relates to the longest/shortest edge ratio and the minimum angle). The general approach of the algorithm is described in Section 3.2

There is a video [22] that gives a short overview of the previous mentioned work. It can also be found at `https://www.youtube.com/watch?v=V5vpwVFMPqs`.

3.2 OMAT-Algorithm of Lee et al.

We now briefly describe the online maximum area triangulation algorithm of Lee et al. [1]. For a more detailed textual description, pseudocode, and experiments, we refer to the original paper.

The robots can only measure the bearing and orientation (see Fig. 3.2) of their neighbors (limited range) as well as detect collisions with the environment (walls and robots can be discriminated). Distances to neighbors can not be sensed. The bearing sensor is able to discriminate the neighbors by a unique id which is also used for messaging. Robots share their measurements, providing a 2-hop neighborhood.

Initially, there is only one edge with between robots which act as an entry to the environment and their edge length as optimal edge length reference. The other robots enter the environment through this edge.

There are three types of edges:

Frontier edges These edges are a margin to the to-be-triangulated-area. Thus, they have a triangle on one side and are still missing a triangle on the other side. In progression of the algorithm, a robot will come and build the missing triangle. Initially, the base edge is such a frontier edge. The algorithm terminates as soon as there are no frontier edges or robots to extend them left.

Internal edges These edges once have been a frontier edge but have been extended and now have a triangle to both sides.

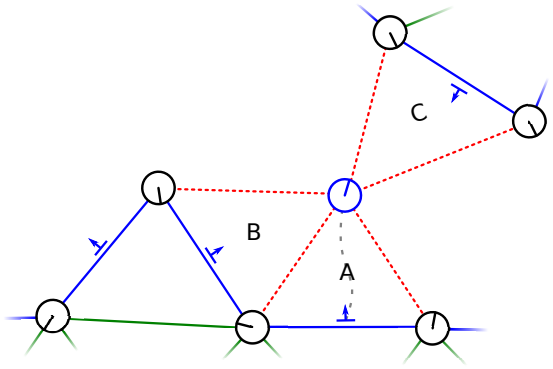


Figure 3.3: During the extension not only the triangle A of the extended frontier edge has to be built but also the triangle B of one of the neighbored frontier edge and the opposite triangle C. The blue edges mark the frontier edges and the green edges the internal edges.

Wall edges Wall edges are similar frontier edges but cannot be extended as they are at a wall.

The basic idea now is very simple: If there is a frontier edge and free robots left, let a free robot enter the environment via the base edge and move to the closest frontier edge (thus the triangulation grows breadth first). This can be implemented by letting the frontier edge robots emit a virtual pheromone and let the free robots move to the frontier edge using the dual graph of the triangulation and the gradient of the virtual pheromone. Extending a frontier edge is (neglecting problems with low sensor resolution[1]) simple as the robot only has to move to the right position and tell the two robots of the frontier edge the success (so it becomes internal).

However, the problematic point is that more triangles than only the second triangle for the frontier edge may need to be created. An example can be seen in Fig. 3.3 where not only the just extended triangle A has to be added but also the triangle B and C. For this, the neighborhood is scanned for matching frontier edges and if the corresponding is considered as good, the triangle is added. Fig. 3.4 shows a case where the triangle C has to be added that initially does not have any frontier edge.

After the extension, multiple new frontier edges can emerge. If not both its incident robots are colliding with the wall and thus the edge is actually a wall edge, it has to send a signal to attract a free robot for expansion.

3.3 Usage of Triangulation

The triangulations provide a 'smart environment' that can be used by other more advanced robots to achieve higher level tasks. Lee et al. [23] use them to distribute guard robots into the controlled area such that the response time is minimized. This is achieved by applying a discrete Voronoi tesselation on the triangles and let the guard robots move to the

[1] See the original paper [1] for a tested controller.

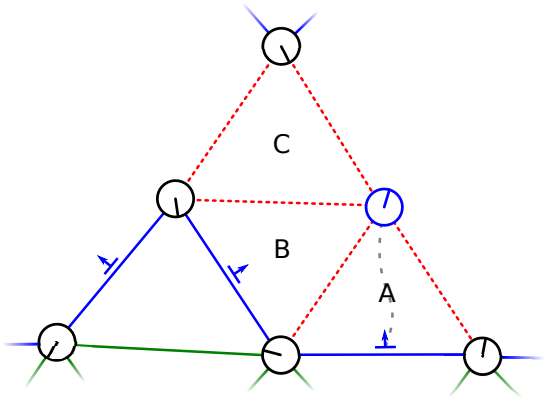


Figure 3.4: Another case of an additional expansion triangle *C*. While in Fig. 3.3 the additional triangle had a frontier edge, this time it does not.

center of their Voronoi cell. The guard robots use the triangles also for movement and navigation, the triangle robots might be able to detect an event and send a signal. In further work, Maftuleac et al.[24] use the triangulation for patrolling guard robots. Coverage frequency and visitation time are to be optimized. For this, they research the two simple policies *Least Recently Visited* and *Least Frequently Visited*.

Algorithms in which robots save information in the environment (as the previous two works did in triangles) are often referred to as Ant-Algorithms. There has been other (and earlier) work which uses such a 'smart environment' but is not focused on triangulations or the construction of the environment.

Koenig et al. [25] consider coverage and repeated coverage (visiting frequency) in a tile world. A robot can read and write data of the tile it is in and also read data of the four adjacent tiles. They consider two methods: *Learning Real Time A** where initially all unvisited tiles have the value 0 and a robot updates its own tile to the minimal value of the neighborhood plus one (compare to pheromones). The robot then moves to the adjacent tile with the smallest value. The second method is *Node Counting* where each robot increases the value of its tile by one (thus it contains the count of visits) and also moves to the adjacent tile with the smallest value. In a theoretical analysis they show that LRTA is polynomial while NC can be exponential. In experiments, however, they perform similar. Wagner et al. [26] researched three methods for coverage with evaporating values but not limited to the tile world. They proved two of them to be polynomial. Andries and Charpillet [27] propose an Ant-Algorithm for a limited amount of search robots in a tile world that visit every reachable tile at least once. After every tile has been visited, the search robots return to the origin tile. In later work [28] they extended the algorithm for robots that have a view of more than one tile. A tile has only to be in view of a search robot once and a search robot can read and write all tiles within its range.

A tile world has been built in hardware by Pepin et al. [29] using tiles equipped with weight sensor and processing unit.

3.4 WSN Supported Swarms

There is also other work that uses static robots or wireless sensor nodes placed in the environment to support dynamic robots. Some of this work can also be seen as Ant-Algorithms.

Searching: Stirling et al. [30] consider exploration/searching with flying robots that can attach to the ceiling and sense the relative positions of their neighbors. While in this thesis we distribute the robots breadth first, they do it depth first. As soon as a branch is explored, the robots are recovered and go into the next branch. This allows ranging much deeper into the environment with a limited amount of robots but also reduces the advantage of parallelism. They do not form triangles as we do but place the robots on a square grid utilizing the knowledge of the relative positions of the neighbors.

Batalin and Sukhatme [31] research coverage and exploration of a large environment via a single robot that can deploy radio beacons. Each radio beacon recommends a direction (north, east, south, west) a robot in its range should go. The approach can be adapted to multi robot systems.

Ghoshal and Shell [32] span a room by building a rapidly-exploring-random-tree-like structure of robots. From randomly placed robots, a robot is randomly selected and starts cycling until it hits the tree. It then either continues an edge of it or builds a new branch.

Foraging: Hrolenok et al. [11] propose a foraging method where robots deploy, move, and update wireless sensor nodes. The nodes do not communicate with each other but only with the robots. After the food source has been found, the nodes first provide a long entwined path to it, later it becomes shorter and smoother.

Hoff et al. [33] split the swarm into walkers that actually perform the foraging and beacons that only support the walkers.

O'Hara and Balch [34] support their swarm by a randomly distributed WSN that guides the robots to unexplored areas, to discovered attractor caches, as well as back to the nest. They also research the influence of the WSN's density and precision on the swarm's performance.

Navigation: Li et al. [35] use a WSN for guidance one to a goal while avoiding dangerous areas. The approach is based on artificial potential fields. This method could be used in our algorithm for guiding the free robots on a way to the frontier that avoids areas with obstacles as these often have irregular triangulations.

Barth [36] uses wireless sensor nodes as markers in an environment (e.g. a maze) that can communicate with each other and as soon as the goal has been found provides the shortest path to reach it via dynamic programming. The markers are placed by the robots themselves.

Other The WSN not supporting the robots but the robots supporting the WSN has been considered by Winfield et al. [37] where robots distribute information between disconnected parts of the WSN.

3.5 Dispersion and Coverage

There has also been other work that places robots or wireless sensor nodes into the environment to achieve a coverage but without a triangulation. Howard et al. [38] propose a centralized online algorithm to maximize the visible area of an unknown environment by iteratively placing wireless sensor nodes to the frontier of the already explored area. This is related to the OMRTP-algorithm of Fekete et al. [20]. Bai et al. [39] provide an optimal pattern to build a sensor network of minimal node count that provides a full coverage while also being 2-connected. The pattern is dependent on the communication-range/sensing-range ratio and does assume an open space without obstacles. A triangulated coverage as considered in this thesis also provides a 2-connected graph but we assume communication range to equal the sensing range and also have a polygonal environment with obstacles.

Very related to achieving a coverage is the dispersion of robots where also has been a considerable amount of work. McLurkin and Smith [40] propose an algorithm that achieves a dispersion by using two states. One disperses interior robots uniformly by applying a repulsion on the closest neighbors and the other for expanding the frontier. The approach guarantees connectivity to allow homing behavior for self-charging. Howard et al. [41] use potential fields that apply repulsion forces to the robots that repel them from other robots as well as walls. This presses robots out of dense areas into sparse/free areas.

The potential field work for flocking of Olfati-Saber [42] or for pattern generation of Spear et al. [43] also provide some kind of dispersion. They can even produce triangular/hexagonal grids but the robots are not locally aware of these. Also, the potential fields use the relative positions of the neighbors to calculate the strength and direction of the forces. In this thesis, however, we only have distances and the previous algorithm of Lee et al. [1] only has bearings and orientations.

3.6 Placement and Selection Problems

From a theoretical point of view, the two problems related to this thesis can be classified as a placement problem and a selection problem. In the triangulation problems MRTP and MATP the main question is how to place the robots in the plane. Thus, this kind of problem is considered as placement problem and is of mainly geometric nature. Examples in the swarm robotic and WSN context are for example the Relay Placement Problem [44] also considered with k-connectivity [45], a distributed version of the Art Gallery Problem [46], as well as the already mentioned work of Bai et al. [39] and Howard et al. [38].

The second problem in this thesis considers the selection of robots in the triangulation. This kind of problem is considered as selection problem and is of mainly graph theoretic nature. Some other of such problems with (possible) relevance for swarm robotics are the Maximum Leave Spanning Tree [47], the Steiner Tree in Graphs problem [48]. Bulsu et al. [49] select a subset of nodes from a dense WSN to provide localization.

3.7 Localization

A survey on mobile localization in wireless networks is given by Gustafsson and Gunnarsson [50]. An introduction to cooperative localization in wireless sensor networks is given by Patwari et al. [51].

4 Triangulation using Distance Sensors only

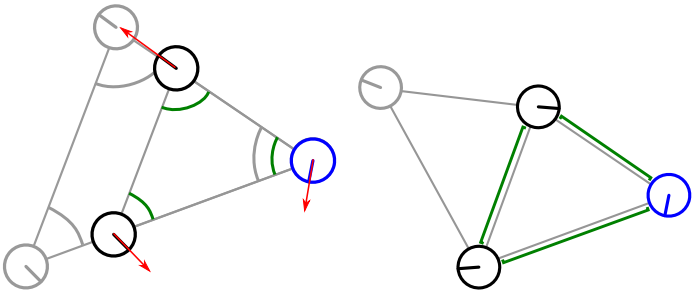
The OMATP algorithm of Lee et al. [1] builds a triangulation using only the neighbors' bearings/orientations in low resolution. We now consider how to build triangulations with only the neighbors' distances (but in a higher resolution). While our proposed approach has been influenced and motivated by the work of Lee et al., the two algorithms differ in most parts. This is mainly raised by ambiguity due to the missing orientation but this work also extends the work of Lee et al. [1]. For example we consider parallelism in Sec. 4.3 or the detection of redundant robots in Sec. 4.4. We also put a stronger focus on robustness and reparation. On the other hand, we assume a much higher accuracy of the sensors and have only tested the approach in a simulator with slight perturbations.

Let us take a brief look at the difference of knowing bearings or distances. We assume that we have a 2-hop neighborhood thus each robot does not only know its own measurements but also the ones of its neighbors. In Fig. 4.1 we see an example where a robot builds a triangle with two other robots. If the robot knows the bearings, it can calculate the exact shape of the triangle except for its scale. If the robot knows only the distances, it can calculate the size of the triangle but the shape is ambiguous. It cannot differ between the actual triangle and the triangle mirrored on the two other robots (thus does not know on which side of the edge it is). It also has no knowledge of its orientation and thus does not now where it is heading and moving. While a robot with bearing sensors can rotate a specific angle to adapt the heading to the desired goal, a robot with distance sensor only can only rotate for a guessed time and has to move and relocate in order to check if it is heading in the right direction.

This chapter is structured as follows: First, we provide the tools for localization and controlled movement using static triangles as reference in Sec. 4.1. This allows us to move to a specific point relative to a triangle. In Sec. 4.2 we then discuss the actual algorithm and in Sec. 4.3 the parallelization method. In Sec. 4.4 we show how to detect the redundancy of a robot and in Sec. 4.5 we provide simulation experiments.

4.1 Localization and Movement

One of the most basic elements we need for the implementation of the algorithm is the localization relative to a triangle. This is needed to move within the triangulation towards the frontier edges as well as to expand a frontier edge. Note that we do not have any directions and thus a robot does not know in which direction it will move. It has to make a



(a) Bearing (b) Distance

Figure 4.1: The robot (blue) and the two other robots in black. Ambiguous alternative perceptions in gray. Measurements in green. Fig. (a) shows that the scale is ambiguous having only bearing information but the directions of all robots is unique. Fig. (b) shows that having only distance information the size of the triangle is unique but the triangle might be perceived mirrored and the orientations are unknown.

second localization after a small movement (we can only guess the actually moved distance due to missing odometry) and derive the direction from the difference to the previous localization. We assume that the accuracy of the localization (directly depending on the accuracy of the sensors) is high enough such that the needed distance to be moved to obtain a reliable difference is relatively small. If the robot is moving in the wrong direction, it has to rotate in the corresponding direction. The robot does not get feedback during the rotation and thus can only guess how long it has to rotate from the last rotations. There are some problems to be solved:

- Build a uniform coordinate system relative to a triangle such that the two localizations have coordinates in the same coordinate system. Also, different robots can exchanged their believed position on this way. The coordinate system has to be robust against small changes in the triangle.
- Transform a point to the coordinate system of a triangle neighbored to the current one. Robots move from triangle to triangle and thus need to frequently change the coordinate system. Not being able to transform coordinate system dependent information would mean a partial reset of the robot during every triangle transition.
- Determine the triangle the robot is in, in case the robot doesn't know due to an error or if it just has been introduced. This has to be a weighted value and not just a boolean as there are inaccuracies in the measurements and if a robot is close to an edge it could detect to be in none or both of the incident triangles. With weighted values, always the triangles with the best (highest) value can be chosen.

Note that localization via triangular beacons (with known positions) and distances is by no means new and from a mathematical point of view quite simple. Inaccurate sensors and inferences make this more difficult and some references to other work have been given in Sec. 3.7. However, short range infrared distance measurements (as the Kilobots use) in our

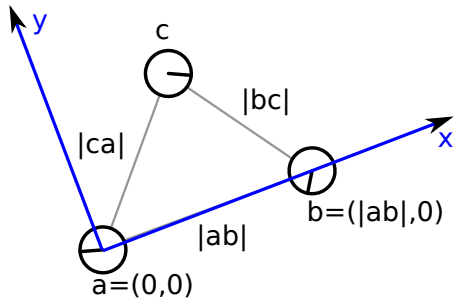


Figure 4.2: The triangle (a, b, c) with a has minimal ID defines a non-ambiguous coordinate system. The position of c is $(\cos(\alpha) * |ca|, \sin(\alpha) * |ca|)$ with $\alpha = \arccos(\frac{|ca|^2+|ab|^2-|bc|^2}{2*|ca|*|ab|})$

application can usually be made much more accurate than the distance measurements in other applications.

4.1.1 Coordinate System

First let us define the triangle relative coordinate system. Let (a, b, c) (counterclockwise) be the given triangle with IDs and pairwise distances known. We always place the origin of the coordinate system at the robot with the lowest ID and orientate the x-axis into the direction of the counterclockwise successor. W.l.o.g. let a be the minimal robot then the robots have the coordinates as given in Fig. 4.2. The resulting coordinate system is independent of the sensing robot and small changes of the distances in the triangle will only slightly change it.

The coordinate system only depends on the IDs and distances of the three triangle robots. As we assume a 2-hop neighborhood that allows a robot to also know the neighbors' neighbors and their distances, a robot in contact with at least two of the triangle's robots can determine the coordinates of all three triangle robots.

4.1.2 Localization

We have a unique coordinate system relative to a given triangle defined. Now we want to localize a robot within this coordinate system. Obviously, this is not always possible, in especially if only one or none of the triangle's robots are in range. If we have contact to two of the triangle's robots, we can determine the triangle robots positions relative to the coordinate system due to the 2-hop neighborhood. Using a simple circle intersection, we can determine two possible coordinates for our robot. If all three triangle's robots are available, we can determine the coordinates unambiguously.

One thing that has to be considered is that the distances are not accurate. Also, we don't want ambiguous positions but automatically derive the most probable one. In the following we first provide an ambiguous method that can cope with slightly inaccurate distances and delivers a unique position if all three triangle's robots are in range. Then we provide a method to obtain the most probable position by using the previous position, the absence of robots as information, and also the information of its neighbors.

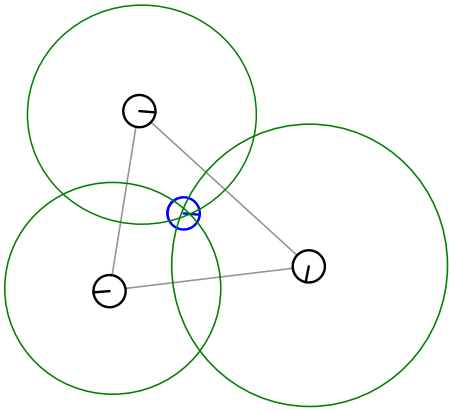


Figure 4.3: The average of the three circle intersection around the blue robot estimates its position. The circles equal the measured distances of the blue robot.

Ambiguous Localization

Every two robots of the triangle together with the distances to them provides two possible locations. If all three robots are known, we obtain $3 * 2$ locations. With perfect measurements, there would be one matching location for all three robot pairs. However, with inaccurate measurements we have to take the most similar location of the robot pairs and average them. An example can be seen in Fig. 4.3 where three circle intersection are close to the actual position but do not match exactly. As there are only $2^3 = 8$ possibly combinations we can simply calculate all means and take the one with the minimal maximal difference to any of its points.

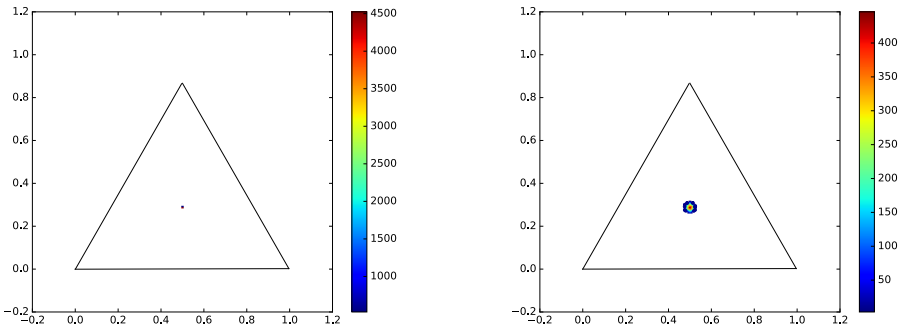
Obtaining the two locations given by a robot pair is done by simple circle intersection, CIRC-INTERSECT. For the case that the circles are too small to intersect, we scale the circles until they do (this is actually simply the point on the line between the two centers relative to the radii). This is also done similarly if one of the circles is too big as long as the necessary scaling is within some bounds. Otherwise, the CIRC-INTERSECT returns the empty set $\emptyset$.

The concrete localization procedure is given in Alg. 1. Note that the maximal difference of the return coordinates is actually also a precious information as it provides the robot a feedback on how reliable its current movement is. A localization with a high difference probably contains an outliner measurement that has to be omitted. However, we omit this for simplicity.

In Fig. 4.4 the density distribution of localizations with different distortions but all three triangle robots in range can be seen.

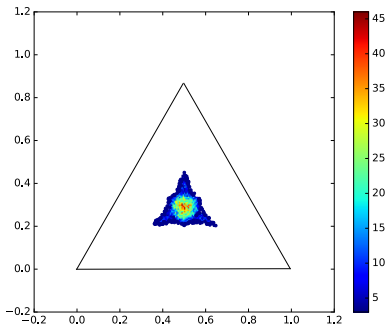
Unique Localization

The previous localization method provides us up to two positions. If all three triangle robots are in range and the distances are accurate enough, it returns a unique position. If there are only two triangle robots in contact, it returns two possible locations. This can happen if the robot is outside the triangle and the third robot is simply out of range or if



(a) 1%

(b) 10%



(c) 30%

Figure 4.4: The distribution for 10,000 localizations with random noise in the measured distance (up to 1/10/30% deviation, uniform distribution). The located robot is placed in the center of an equilateral triangle with robots at $(0,0)$, $(0,1)$, $(0.5, 0.866)$.

Algorithm 1 Ambiguous Localization

```
1:  procedure ambiguousLocalize(Triangle T(a,b,c), Robot r)
2:      if N(r) ∩ T < 2 then
3:          return ∅
4:      if N(r) ∩ T = 2 then
5:          Let {c1, c2} = N(r) ∩ T
6:          return circ-intersect(c1, c2, |c1r|, |c2r|)
7:      bestAvg ← ⊥, minDiff ← ⊥
8:      for l1 ∈ circ-intersect(a, b, |ar|, |br|) do
9:          for l2 ∈ circ-intersect(b, c, |br|, |cr|) do
10:             for l3 ∈ circ-intersect(c, a, |cr|, |ar|) do
11:                 avg ← (l1 + l2 + l3)/3
12:                 max_diff ← max{|l − avg| | l ∈ {l1, l2, l3}}
13:                 if bestAvg = ⊥ ∨ max_diff < min_diff then
14:                     bestAvg ← avg; min_diff ← max_diff
15:     return bestAvg
```

the contact is interrupted by an obstacle or noise.

Algorithm 2 states the selection process. In case there are only one or no positions returned by the ambiguous localization, the procedure only forwards the result. Otherwise, if there are two possible positions, the algorithm first tries to use the one with the minimal distance to its previous position. The previous position (denoted by *prev_pos*) can be unknown, too old, or the distance to the possible current positions too large (denoted by *max_exp*) to be feasible. If there are other robots in the neighborhood, the ambiguous localization algorithm can be used on them using the 2-hop neighborhood. If there are possible neighbor localizations, the position that matches best to any of these localizations is returned. This of course does not work if the neighbor also has only contact to the same two triangle robots. If everything fails or is simply not available, we assume that the third robot is simply out of range and we use the position with the maximal distance to it.

A combination of all possible selections is also possible but omitted for simplicity.

4.1.3 Closest and Encompassing Triangle

In the triangulation algorithm the robots move from triangle to triangle. For this they need a metric to decide how close to or how deep in a triangle they are. This is in especially important for robots close to the margin of a triangle. A boolean inclusion decision might conclude that the robot is also in the adjacent triangle or in no triangle at all.

We simply use the closest distance to the boundary of the triangle as value. If the robot is assumed to be inside the triangle, the value is positive otherwise it is negative. Thus, the higher the value, the closer or deeper the robot is in the triangle. Also, the higher the absolute value, the more reliable is the inclusion decision.

Algorithm 2 Unique Localization

```
1:  procedure uniqueLocalize(Triangle T, Robot r)
2:      P ← ambiguousLocalization(T, r)
3:      if P = ∅ then
4:          return ⊥
5:      else if |P| = 1 then
6:          return p ∈ P
7:      else
8:          if prev_pos ≠ ⊥ ∧ min{dist(p, prev_pos) | p ∈ P} ≤ max_exp then
9:              return p ∈ P with dist(p, prev_pos) minimal
10:         else
11:             Let P = {p1, p2}
12:             d1 ← diff(r, T, p1), d2 ← diff(r, T, p2)
13:             if d1 = ⊥ ∨ d2 = ⊥ then
14:                 Let x = T \ N(r)
15:                 return p ∈ P with dist(p, x) maximal
16:             else
17:                 return p1 if d1 < d2 else p2
18: procedure diff(Robot r, Triangle T, Position p)
19:     diff_sum ← 0
20:     diff_n ← 0
21:     for n ∈ N(r) with N(r) ∩ T ≠ N(n) ∩ T do
22:         P ← ambiguousLocalization(T, n)
23:         if P ≠ ∅ then
24:             diff_sum ← diff_sum + min{|dist(p, p′) − d(n)| | p′ ∈ P}
25:             diff_n ← diff_n + 1
26:     if diff_n = 0 then
27:         return ⊥
28:     else
29:         return diff_sum/diff_n
```

The calculation can be simply performed by first doing a localization and checking if the position is inside the triangle. As a triangle is convex, this is a trivial operation. The minimal distance to its margin can also be simply calculated by the minimal distance to its three edges.

4.1.4 Transformation

If a robot transits to a neighbored triangle, it has to transform its saved positions and directions to the new triangle. Otherwise, every triangle transition (which are frequent) would partially reset the robot and strongly reduce its efficiency.

We assume that the robot can sense all three robots of the new triangle. Thus, it also can sense at least two robots of the old triangle and localize the other robot of the new triangle in the old coordinate system. With the coordinates of the new triangle in the old coordinate system we can build a simple transformation. Let o_{new} be the minimum robot of the new triangle, o' its successor, and α the counterclockwise angle between the x-axis and $o' - o_{\text{new}}$.

$$\text{transformed\_pos} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & -\cos\alpha \end{pmatrix} * \begin{pmatrix} x - o_{\text{new}}.x \\ y - o_{\text{new}}.y \end{pmatrix}$$

4.1.5 Moving to a point

For expanding a triangle or simply moving to a neighbored triangle, we need to be able to move to a specific point (or at least sufficiently close to it). The problem is that we neither have orientation sensors nor real odometry. Thus, we neither know in which direction the robot will move nor can we execute accurate rotations/movements. However, we can obtain the direction by making a localization, moving straight forward, and doing a further localization. The difference of these two localizations provides the direction. Some kind of odometry can be implemented by providing the average velocity (forward and rotation) of the robot model and adjusting it using the last localizations. The adjustment is necessary as average rotation velocity strongly depends on the local terrain.

The Algorithm 3 has four tunable parameters:

goal_tolerance Maximal distance to goal when finished.

angle_tolerance Angle difference to goal that triggers rotation.

avg_rotation_velocity Will be adjusted during execution. The initial value is robot model specific and can be estimated in previous experiments.

min_move_dist The distance two localizations should have for a usable direction estimation.

If the robot does not know its last position or direction it executes a localization, moves a sufficient distance, and does another localization. The last localization becomes the last position and the difference of the two localizations the direction. Now the algorithm executes a simple loop until it is within the specified range of the goal position. The loop

calculates the difference between the current heading direction and the direction of the goal. If the difference is too large, it calculates a rotation time based on the estimated rotation velocity to correct the heading. After executing the rotation the robot moves straight forwards and does another localization to calculate the new heading direction. It adapts the estimated rotation velocity based on the rotated time and the actual performed rotation difference. If the heading is within some bounds, the robot simply moves straight forward until it has to adapt the heading again or the goal is reached. The heading direction is also recalculated during the straight movement phases as the longer the moved distance is, the more reliable is the estimated heading direction.

The 'move until' command needs to make multiple localizations. As localizations are often expensive (not computationally but the distance measurements are made with a low frequency), a practical solution should also use an estimated velocity for the straight movement.

Algorithm 3 MoveTo

```
1:  procedure MoveTo(Δ : Triangle, goal: ℝ², direction: ℝ²)
2:      last_pos ← Locate(Δ)
3:      if direction= ⊥ then
4:          Move until dist(Locate(Δ), last_pos) > min_move_dist
5:          new_pos ←Locate(Δ)
6:          direction ← new_pos -last_pos
7:          last_pos ← new_pos
8:      while dist(last_pos, goal) > goal_tolerance do
9:          α ← ∡_{−π,π}(direction, goal − last_pos)
10:         if |α| > angle_tolerance then
11:             Rotate (α ≥ 0:left; α < 0:right) for |α|/avg_rotation_velocity
12:             Move until dist(Locate(Δ), last_pos) >min_move_dist
13:             new_pos ←Locate(Δ)
14:             new_direction ← new_pos-last_pos
15:             Calculate the last average rotation velocity and adjust avg_rotation_velocity
16:             direction ← new_direction, last_pos ← new_pos
17:         else
18:             Move until dist(Locate(Δ), last_pos) >min_move_dist
19:             last_pos ←Locate(Δ)
20:             Adjust direction using last localizations without rotation
```

The used triangle can be swapped at any time by simply transforming the positions and directions as explained in Sec. 4.1.4.

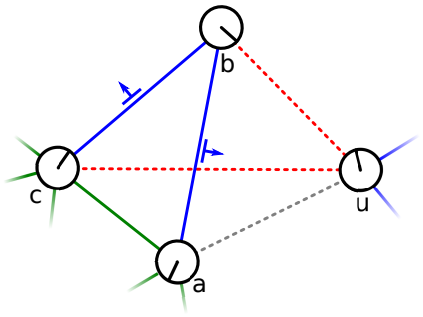


Figure 4.5: Robot u in triangle discovery might discover the frontier edge cb (possibly the robot a is hidden) which however would result in an invalid triangle. Robot u can not deduce the direction of the frontier edge cb without seeing robot a.

4.2 Algorithm

The methods of the previous section allow us to navigate within the triangulation and even move to specific points in range, e.g. for expanding a frontier edge as in the algorithm of Lee et al. (see Sec. 3.2). The triangle discovery that is executed after the edge expansion and finds other triangles that have implicitly been created is not possible with only distance measurements. This can be seen in Fig. 4.5 where the robot u might detect the edge cb (and misses ab because it hasn't noticed robot a) which would lead to a corrupt triangle that overlaps with abc.

In our approach the robots move to a frontier edge in a breadth first manner and extend this edge similarly to the algorithm of Lee et al.. After the expansion, however, there is no triangle discovery. Instead, all frontier edges remain except the just expanded one but a robot checks during the expansion if there is already an implicit triangle. If there is such one, the robot notifies the corresponding robots and moves to another frontier edge. While the algorithm of Lee et al. starts with an edge on which new robots are introduced, we start with a triangle as the robots otherwise might expand into the wrong direction.

The now discussed algorithm consists of fives states (see Fig. 4.6): Lost, FrontierMoving, Expanding, CheckingCandidate, and Static. The Lost-state tries to find a reference triangle by trying to find a static robot via random walk and then try to get closer to it to get in range of its triangle. Having found a reference triangle, the robot has a relative coordinate system for controlled movement and transits to the state FrontierMoving. In this state it moves from triangle to triangle to the next frontier edge. If it has reached a frontier edge, it tries to expand it in the Expanding-state. If it finds a static robot that could also expand this edge, it checks the validity of this candidate in the CheckingCandidate-state. Otherwise, it builds a new triangle and becomes static. A static robot can also be deleted again if e.g. it has a corrupted triangle or is redundant. Then it will transit to the Lost-state.

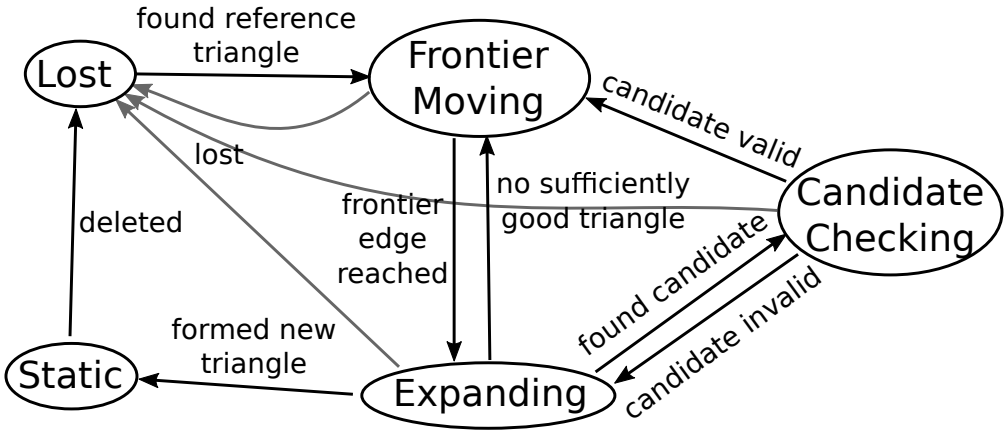


Figure 4.6: The five states of the algorithm

4.2.1 Public Variables and Messages

We now discuss the used public variables and messages of the algorithm. While the actual use of some elements might only become clear in later sections, it is important to know the provided information of each robot.

The following public variables are possessed by all robots (static and free). They mainly provide the ID and the 2-hop neighborhood.

ID $\in \mathbb{N}$**:** The ID of the robot

distances $\subset \mathcal{R} \times \mathbb{R}$**:** Provides the 2-hop neighborhood by publishing the own 1-hop neighborhood (the distance to each adjacent robot). As the proposed approach tries only to have one free robot in each triangle, the size of this variable is usually still less than 10. Some filtering is possible but not considered in this thesis.

is_static $\in \mathbb{B}$ True if the robot is static (part of the triangulation), false otherwise.

The free robots need some public variables to synchronize between each other in order to prevent collisions and inconsistencies.

in_triangle $\in \mathcal{R} \times \mathcal{R} \times \mathcal{R}$**:** Distributes the triangle the robot currently is in to prevent other robots to move into it.

expansion_value $\in \mathbb{R} \cup \{\perp\}$**:** Used for synchronization during edge expansion. If two adjacent robots execute an edge expansion during the same time, they compare this value to decide who has to abort to prevent corrupted triangulations. The value equals the distance to the expanding edge and is $\perp$ for all non expanding robots (which also allows identifying expanding robots).

Static robots have to provide information of the triangulation.

triangles $\subset \mathcal{R} \times \mathcal{R} \times \mathcal{R}$**:** Each static robot has a list with all its incident triangles. A triangle is decoded by the IDs of its three robots in counterclockwise order. Even so

a static robot is only expected to have around 6 incident triangles, due to some irregularities there also can be some more. This implementation is not optimal as it has a lot of redundant information (e.g. the robot's own ID in every triangle) but it is simple to use. Frontier edges are identified by checking in how many triangles of its incident robots it is in.

wall_edges $\subset \mathcal{R}$**:** If an edge is not expandable because of a wall, it is marked by one of its incident robots (the one with the smallest id). It only contains the id of the other incident robot.

frontier_pheromone $\subset \mathcal{R} \times \mathbb{N}$**:** For every edge one of its incident robots (smallest id) calculates and distributes its frontier pheromone value.

evasion_pheromone $\subset \mathcal{R} \times \mathbb{N}$**:** Analog to the frontier pheromone but instead to frontier edges it leads to edges that have an incident triangle with no free robot in it. It will be discussed in Sec. 4.3

Not for all information, the best way to distribute it are public variables. Sometimes a directed message is better since one can wait until the receiver has acknowledged the information. However, messages can be lost and the robots should be able to cope with that.

DeletionDemand: Sent to a static robot if it has been discovered to be part of a corrupted triangle. The static robot then becomes free and a hole in the triangulation arises that is rebuilt later.

DeletionNotification: Sent by the deleted static robot to all its adjacent robots to make sure all triangles containing it are deleted.

TriangleNotification: Notifies static robots of a new incident triangles by attaching the three ids of the triangle in counterclockwise order.

WallMessage: Sent by an expanding robot to the corresponding incident robot of the expanded edge to notify it that the expansion failed due to a wall. The edge is then marked as wall edge.

ExpansionTrial: Sent by an expanding robot to the corresponding incident robot of the expanded edge that it tries to expand the edge. If there are too many trials where the robot failed or vanished, the frontier edge is deactivated (blacklisted).

4.2.2 State: Static

Robots in the *Static*-state are part of the triangulation and thus do not move or have any concrete procedures. Their main task is to provide their public variables to the free robots and update them (especially the pheromone values). An implicit task is of course to also act as static beacon of the triangles such that the free robots can localize themselves. The provided public variables (namely *triangles*, *wall_edges*, *frontier_pheromone*, and *evasion_pheromone*) have already been mentioned in the previous section.

Triangles: When a robot becomes static, it just has finished an edge expansion and thus knows exactly one triangle. As we do not have a triangle discovery phase as the algorithm of Lee et al., this is the only triangle it identifies itself. It is notified by *TriangleNotification*-messages of all other triangles. If it receives such a message, the triangle is added to the set.

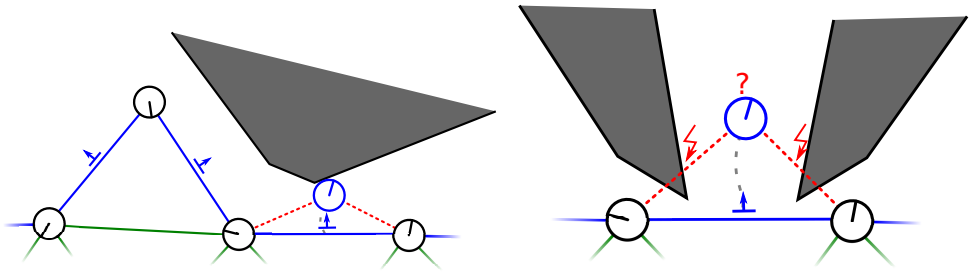
The triangles also provide the set of adjacent robots (and incident edges) in the triangulation. An edge that is only in one of the triangles is either a wall or a frontier edge (or a blacklist edge). If an edge is in more than two triangles, there is obviously an inconsistency. There is also an inconsistency if the robot finds a triangle in its neighborhood in which it does not participate but is located in. Note that the set of triangles only contains the triangles on the lowest level which should not contain any other static robot. In both cases, the static robot will in every time step delete itself from the triangulation with a specific probability. This randomization helps to keep the created hole small (there are mostly more than one static robot involved in an inconsistency) and there is also a greater hope to work around problems induced by the environment.

Besides the own discovery of inconsistencies, inconsistencies can also be discovered by free robots traversing through one of its triangles. In this case, they will send a *DeletionDemand*-message. If a static robot deletes itself, it sends a *DeletionNotification*-message to all its triangulation neighbors. It uses an acknowledgement based protocol to ensure the receipt of this message. A static robot receiving such a *DeletionNotification*-message removes all triangles containing the sender from its set. This is necessary as the just freed robot with a very high probability will participate in the rebuilding of the just created hole. Old triangles could be inconsistent with the new triangulation and again trigger a deletion. However, a lost *DeletionNotification*-message might create some additional chaos but will not create harm in the long run.

A further reason for a robot to delete itself is of course if all its adjacent triangles have been removed. Sometimes it can also be reasonable to delete all static robots that are no longer connected to the entrance. This however is not mandatory and most of the time the connection will be rebuilt.

Frontier edges: As already mentioned, a static robot detects its incident frontier edges by counting for each edge in how many triangles it is in (and checking that it is not marked as wall or blacklisted, see next paragraph). The robot of an edge with the lowest id is responsible for the management and provision of the edges' values. Thus, a static robot only has to provide values for a subset of its incident edges. Free robots are guided to the frontier edges by a virtual pheromone induced on the edges. The frontier edges have the value 0, all other have the minimal value of another edge in the two triangles it is in plus one.

Let $\mathrm{fph}_r(n)$ be the frontier pheromone value of edge (r,n) saved in robot $r = \min\{r,n\}$, T_r are the triangles saved in robot r. Then we can formally state the frontier pheromone



(a) Wall edge (b) Robots get lost during expansion

Figure 4.7: Two reasons an edge is not expandable: In the first example a wall prevents a sufficiently large triangle and in the second example the expanding robot loses contact during expansion and gets lost.

value as

$$\mathrm{fph}_r(n) = \begin{cases} 0 & \text{if } (r,n) \text{ is frontier edge} \\ \min\{\mathrm{fph}_{\min\{u,v\}}(\max\{u,v\}) \mid (u,v) \in t, \\ (r,n) \in t, (u,v) \neq (r,n), t \in T_r\} + 1 & \text{else} \end{cases}$$

Each static robots constantly evaluates this function for all its incident edges it is responsible for (those where it has the smallest id). To lower the attraction of a frontier edge, the value each edge increases with each step can be increased. This can be useful if the of free robots is high but is not considered in this thesis as mentioned in Sec. 2.3.

An example of the boundary pheromone can be seen in Fig. 4.8.

Wall and blacklisted edges: Edges that cannot be expanded have to stop emitting the frontier pheromone. There can be different reasons that an edge cannot be expanded, the most frequent is that there is a wall shortly behind it, see Fig. 4.7a. Another reason is that due to obstacles the expanding robot loses connection to the static robots during extension and becomes lost, see Fig. 4.7b. There can also be irregularities in the triangulation that forbid to triangulate a small hole (a good metric to position robots and rate triangles should prevent this but is hard to find).

If there is a wall, the expanding robot will send a *Wall*-message and retreat. The static robot receiving this *Wall*-message then marks the corresponding edge as wall. In case, somehow, the wall edge is in two triangles, the wall marker is removed. Edges that cannot be expanded due to e.g. communication loss as in Fig. 4.7b are harder to deal with. Each expanding robot will notify the responsible robot about its trial. If there has been a specific amount of trials without a success, the edge is blacklisted for a longer time. This prevents further robots to get lost or be attracted to the edge while there are other frontier edges with better prospects of expansion.

There are obvious some problems with some walls and obstacles. The proposed algorithm can fail to triangulate these specific parts of the environment but will succeed in the rest of the environment. For simplicity we do not cope with these problems and assume

that most part of the environment is simple and also easily accessible (and not e.g. only through a gate like in Fig. 4.7b).

4.2.3 State: Lost

If a free robot has no triangle to refer to in order to use the tools of Sec. 4.1, it transits to the *Lost*-state. The goal of this state is to find a reference triangle that provides the robot localization and control movements. There are two possible scenarios: No static robot is in range and at least one static robot is in range but no triangle. If there is a reference triangle in range, the robot immediately transits to the FrontierMoving-state. In the first case, the robot can only do a random walk. It moves forward for a random time and then rotates randomly. This is repeated until a static robot comes in range. The random times should be tuned for the specific robot model but without odometry and different velocities for different terrains, this is only possible to a limited degree. If there is a collision, the forward movement time should be smaller since already a small movement will solve the collision.

If there is a static robot in range, the robot tries to get closer to it. As we only have the distance to it, we only get feedback for if the last movement increased or decreased the distance. We use the following heuristic: If the last straight forward movement got us closer, we move again forward with a high probability. With a low probability, we execute a random rotation in the hope to improve the orientation. If the last forward movement increased the distance, we do a random rotation. After each rotation, we do a forward movement. If by some chance, we got very close to the static robot but still see no reference triangle, we move either to the second closest static robot or do a random walk.

As soon as a reference triangle is found, we transit to the *FrontierMoving*-state.

4.2.4 State: FrontierMoving

Robots in this state move from triangle to triangle closer to a frontier edge. The adjacent triangle to go to is selected by choosing the edge with the minimal boundary pheromone value (random selection if multiple possibilities). To reduce the risk of colliding with a static robot, a free robot moves from the triangle center to the center of the edge to the center of the targeted neighbored triangle. This can also be seen in Fig. 4.8. The prevention of collision between free robots is considered in Sec. 4.3. If a robot is initially in no triangle, it searches the closest center of an edge of a triangle in range. It enters the triangle via this point and moves to the triangles center before continuing the normal procedure. During the whole time, a free robot searches for inconsistencies in the triangulation.

The main inconsistencies a free robot can detect are overlapping triangles. It notices them if it senses that it is in two triangles at the same time and has validated the reliability of the measurements. There are three forms of overlapping triangles. The first one is displayed in Fig. 4.9a where two overlapping triangles share exactly one robot. This can happen due to obstacles that prevent the sight on the two other robots of A. It is dealt with in more detail in the *CandidateChecking*-state in Sec. 4.2.6. The second case of overlapping triangles are triangles that share a common edge as can be seen in Fig. 4.9b. This might be

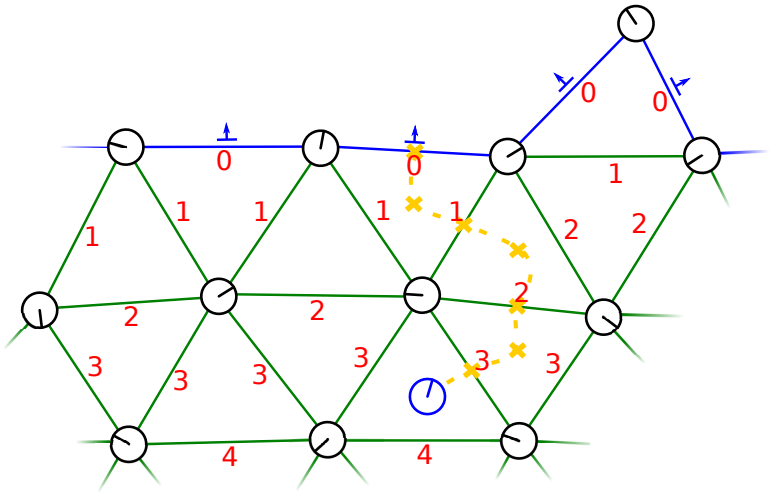


Figure 4.8: A free robot in FrontierMoving-state moves from triangle to triangle center via the edge center following the frontier pheromone gradient

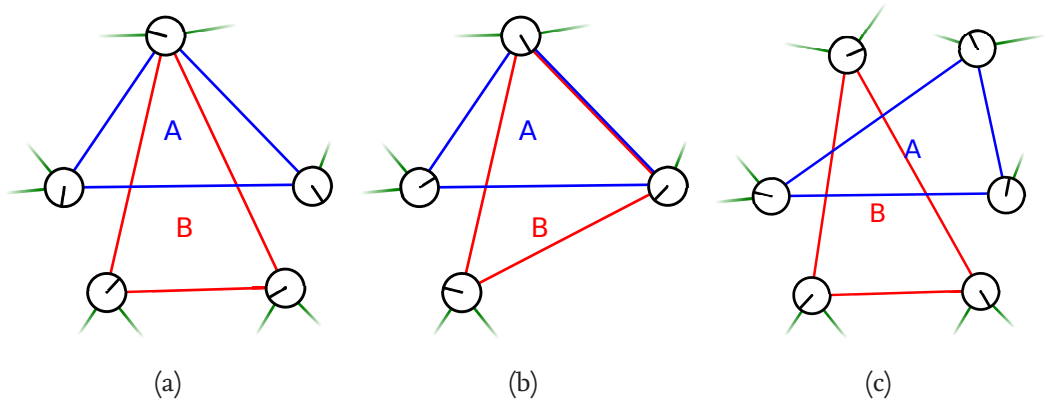


Figure 4.9: Different possibilities of corrupt triangles

an edge extension gone wrong. In some cases (when there is also a triangle on the other side of the edge), the static robots can detect the inconsistency themselves (see Sec. 4.2.2). The third case are triangles that do not share any robot as in Fig. 4.9c. This problem also mainly occurs due to obstacles that prevent the view of the expanding robot onto the other triangle. In all cases, the free robot that detected the inconsistency first makes sure that it hasn't been a measurement error. Then it sends a *DeletionDemand*-message to involved robots. It is also possible to only send it to the intersection of the two triangles in the first two cases or only the robots of one triangle in the third case.

4.2.5 State: Expanding

This state tries to expand a frontier edge and thus builds a new triangle. The *FrontierMoving*-state already placed the robot on the corresponding frontier edge. It now has to move to a matching expansion point that creates a well formed triangle and notify the

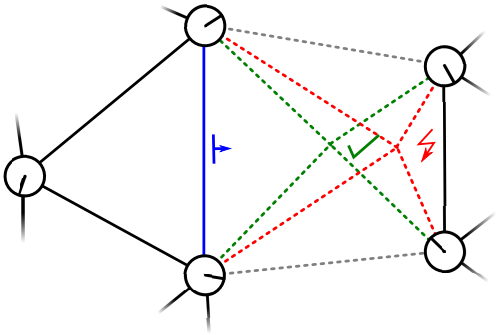


Figure 4.10: The red expansion optimizes the quality of the just created triangle but enforces other bad triangles. The green position would have been better.

corresponding neighbors on success. In case it detects an already static robot that could also extend this frontier edge, the robot transits to the *CandidateChecking*-state. If there is a wall, it notifies the robot of the frontier edge that it is a Wall-Edge. In the *Static*-state we already discussed how to prevent endless losses of expanding robots in special cases.

Triangle Quality

The first open question is: What are 'good' triangles? We want triangles to be large in order to efficiently cover large areas and want robots to be able to move in them without collisions. We cannot prevent all collisions but by placing the static robots as far away of their paths as possible, we reduce the risk. Of course the static robots themselves should be close enough to communicate reliable and have only few static neighbors. A further point is, that the corresponding metric should be simple enough to be evaluated locally by the expanding robots. Also, it should not only concentrate on the current triangle but also on the adjacent triangles still to be built. A perfect triangle is useless if all adjacent triangles are of poor quality (see Fig. 4.10). This becomes specially important for repairing holes in the triangulation.

Fortunately, the goals mostly strongly correlate but there are still many possible quality metrics. Naive metrics as only focusing on maximizing the edge lengths or the angles performed poorly in our implementations. While performing well as long the triangulation was regular, the triangles became quickly misshaped as soon as the environment disturbed induced irregularities. The final metric used by us is the following:

$$\begin{aligned}\texttt{quality}(\Delta) = \max\{&\\ &\{\min\{dist(p,(u,v)) \mid p \notin (u,v) \text{ is position of a static robot}\} \mid (u,v) \in \Delta\}\cup\\ &\{\min\{dist(u,(p,p') \mid (p,p') \notin \Delta \text{ is edge in triangulation} \mid u \in \Delta\}\}\end{aligned}$$

Thus, we try to maximize the minimal distance between an edge of a triangle and all other static robots. Obviously, it is sufficient to only focus on static robots in range. This metric does not only focus on the considered triangle by maximizing the minimal triangle height but also the distance of the triangle to the neighbored static robots and thus the

quality of the implicit triangles (that will later be created in the *CandidateChecking*-state). To prevent long unreliable edges, we introduce a maximal edge length after which the quality is set to 0.

Even so this metric performed well with carefully chosen bounds on the minimal quality and maximal edge length, this metric is not perfect. In especially, it does only maximize the distance to other static triangles but on this way may enforce bad triangles as the created implicit triangles all have too long edges.

A further problem is the question if every incomplete triangulation that fulfills quality constraints given by this metric can also be completed without breaking the quality constraints. For the given metric, degenerated cases can be created where the triangulation can only be completed by replacing already placed robots. By relaxing the quality metric, this might can be fixed but also create triangles that the free robot have problems to use. Therefor, the final triangulation might contain some holes as the corresponding frontier edges get deactivated by the static robots if too many expansions failed.

Expansion Point

Doing advanced optimization to find the optimal (or near optimal) expansion point is beyond the computational capabilities of the robots. Additionally, the knowledge of the robot is potentially incomplete in the beginning as some important static robots might not be visible from the initial position. If the robot does the optimization in the initial position and then moves to the calculated expansion point, it might notices that there are some other static robots drastically reducing the rating of the point.

Most of the times, there are no other static robots to consider during expansion. This is due to the breadth first growing of the triangulation. For this case, the optimal expansion point can easily be calculated: Expand the frontier edge by an isosceles triangle with the two new edge lengths being the maximal still reliable edge length. This point can be calculated by simple circle intersection.

To keep the computational complexity low, the robot selects a few points of potential expansion points, see Fig. 4.11. This includes in especially the previously mentioned optimal point if there are no other static robots and some evasion points to the left and right of this point. The robot calculates the rating of all these points using its current knowledge and heads for the current best one. These ratings are upper bounds as the robot can only know about possibly yet unseen static robots that might lower the rating if it is at the corresponding position. During moving, the robot also frequently rates the current position and saves the best of these positions. It is added to the potential extension points and thus becomes the expansion point if the other points show do be worse than it.

The important knowledge for an expanding robot are the positions of the adjacent static robots. Initially, this knowledge consists of all static robots in range that can be localized using only their 2-hop neighborhood. As soon as the expanding robot builds a sufficiently large triangle with the frontier edge, it can also localize further static robots relative to this triangle and transform the position to the actual triangle. However, some static robots can not be localized and only their distance to the current position of the expanding triangle

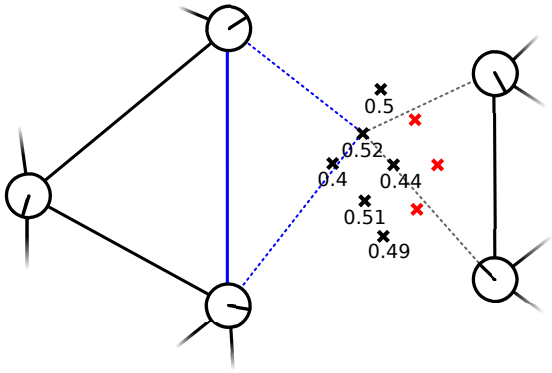


Figure 4.11: A small set of expansion points is rated and the point with the highest rating is chosen. The red points are infeasible because they are to close to an edge. The rating not only depends on the own directly created triangle (blue) but also on the distance to other static robots and edges.

is known. These distances are used as an upper bound (multiplying it by 0.8 provides a better estimation) for the rating of the current position and can lead to a sudden decrease of the rating of an extension point once it is reached. This again might change the favored extension point and the expanding robot possibly visits multiple potential expansion points before deciding for one. As mentioned, the expansion is most of the time simple and the first favored extension point or a point found on the way to it becomes the actual extension point.

Note that the expanding robot could also use the distances to the static neighbors from different positions to localize them. This would need a more advanced movement of the expanding robot which is due to the low frequency of these cases not reasonable.

During the whole time, the expanding robot checks that the environment it is expanding into is consistent. If it notices e.g. that it is within another triangle, the expansion is aborted.

4.2.6 State: CheckingCandidate

In contrast to the algorithm of Lee et al., our approach does not have a triangle discovery phase but robots in *Expanding*-state notice that the triangles they are trying to build can also be built with already existent static robots. Such an already existent static robot that can build a (quality constraint fulfilling) triangle for a frontier edge is called *candidate*. The metric can be evaluated the same as for regularly expanding robots as the 2-hop neighborhood provides the checking robot with all necessary data. However, not every candidate yields a valid triangle as it might be on the wrong side or overlapping with other triangles (see Fig. 4.12). While a regularly expanding robot automatically expands into the correct direction by having the corresponding triangle of the frontier edge as reference and checking for overlapping during expansion, the checking robot has to ensure this for the candidate.

A robot in expansion state begins with the expansion of the triangle until it is suffi-

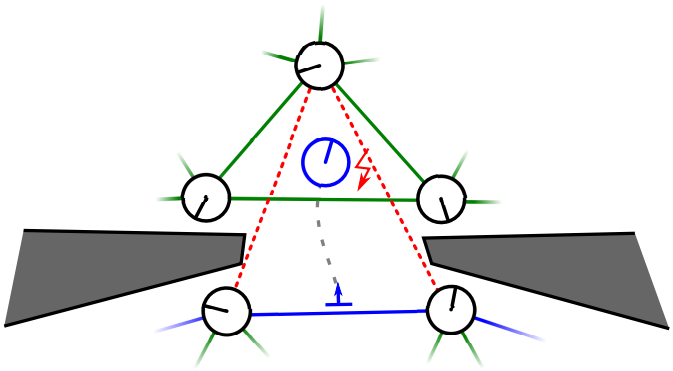


Figure 4.12: Due to hidden robots, it is not immediately seen that the candidate would induce an overlapping triangle. If the free robot moves towards the candidate, it will notice the overlapping.

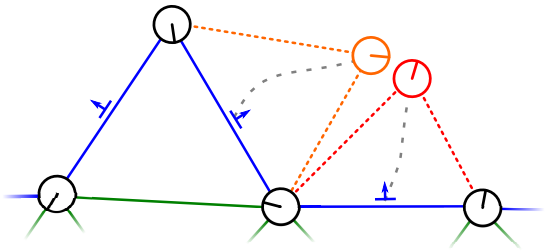


Figure 4.13: The red and the orange robot execute a colliding expansion. With no additional synchronization, there can either be a degenerated slim triangle or overlapping triangles.

ciently outside of origin triangle. Now it can detect candidates on the right side of the expansion edge and rate their resulting triangles. If there is such a candidate, the robot switches to the candidate checking state. In this state, it choses the candidate with the best rating and validates the triangle by moving sufficiently close to the candidate. If the robot notices any inconsistencies, the candidate is dismissed. If there are no other candidates, the robot switches back to the expansion state and tries to create a triangle on its own.

4.3 Parallelization

The OMAT-Algorithm of Lee et al. [1] only deploys one robot at a time, thus there are no concurrency problems. However, parallelization is not only needed for speed up but also fix inconsistent parts of the triangulation where after detection multiple robots might become free. There are two problems to be considered: Firstly, interior robots should keep distance but also allow inducing new robots from multiple sources without blocking. Secondly, there should be no overlapping edge expansions (note that this is expansion point and not frontier edge dependent, see Fig. 4.13).

4.3.1 Interior Synchronization and Evasion

The main idea of synchronizing interior free robots is straight forward and has also been used in the later work of Lee et al.: We allow only one robot per triangle at a time. This can easily be implemented by distributing the current triangle for each free robot per public variable. Before transiting to an adjacent triangle, a robot checks if there is already a robot in range that claims this triangle. If so, the robot waits, otherwise it transits and changes the claimed triangle. The triangle are can be uniquely identified by the ids of their three triangle robots and when on the edge between the two triangles, the corresponding robot should be able to see all robots in the triangle. A localization of the neighbors is not necessary. Further, as all robots follow the boundary pheromone there should be no blocking cycles.

Obviously, the one-robot-per-triangle-rule will often be violated during the execution. Not only because of sensing error but two robots might enter a triangle at the same time from different edges or for frontier triangles an exterior robot might want to enter. While in the first case one of the robots might simply return to its previous triangle, in the second case all adjacent triangles might be occupied. For this case multiple robots need to move up in order to provide a free triangle for the new robot. Of course we want to do this in as few transitions and also with as less obstruction as possible.

We implement this 'evasion' via a simple pheromone on the edges that leads to empty triangles equal to the frontier pheromone. Each edge that is incident to an empty triangle (easily observable by the corresponding robots of the edge as it is incident to both triangles) gets the value zero. All other edges get in the same manner as the frontier pheromone the minimal edge value of the edges of the two incident triangles plus one. It is easy to see that this provides the shortest path of edges (transitions) to a free triangle, see Fig. 4.14. Note that this evasion pheromone can also be used to check valid transitions (the corresponding value has to have the pheromone value zero).

In a triangle that has more than one robot, one of the robots now can transit over the incident edge with the minimum evasion pheromone value and thus propagate the multiple occupied triangle to the boarder of the occupied triangles until it gets resolved by an empty triangle. Now, there only needs to be a selection of which robot should leave. Each robot can calculate the minimal distance to a center of an edge with minimal pheromone value. Obviously, the most rational decision is to let the robot with the smallest distance evade (in especially as this one should be able to go on a straight collision free way to it). This can be done either by a short message exchange or by localizing the neighbors by the 2-hop neighborhood and calculating the foreign distances by themselves.

This system fails if there are more free robots than triangles.

4.3.2 Edge Expansion Synchronization

To synchronize the edge expansion in order to prevent overlapping triangles or too close static robots, we use a similar method. By enforcing the one robot per triangle rule as explained previously, there won't be two edge expansions from the same edge. However,

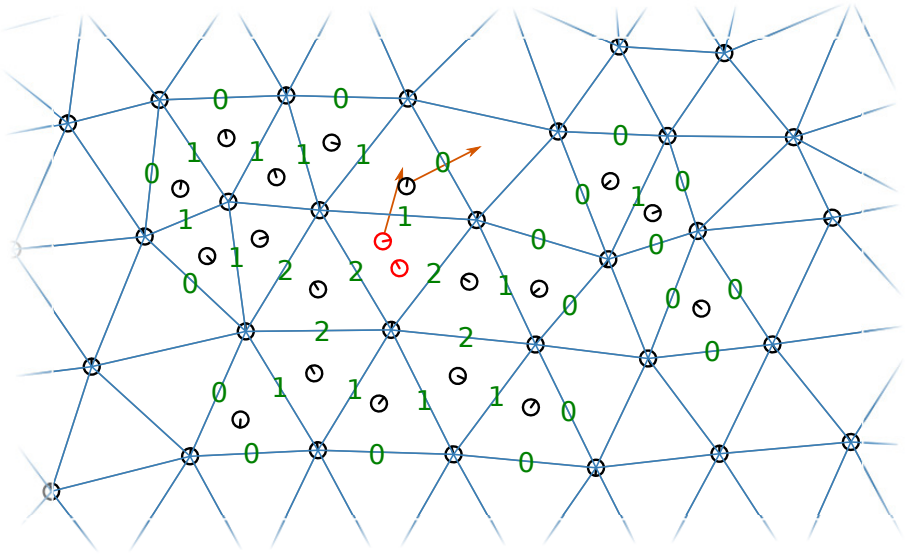


Figure 4.14: The evasion pheromone in green (only relevant part). The two red robots induce an evasion marked by orange arrows.

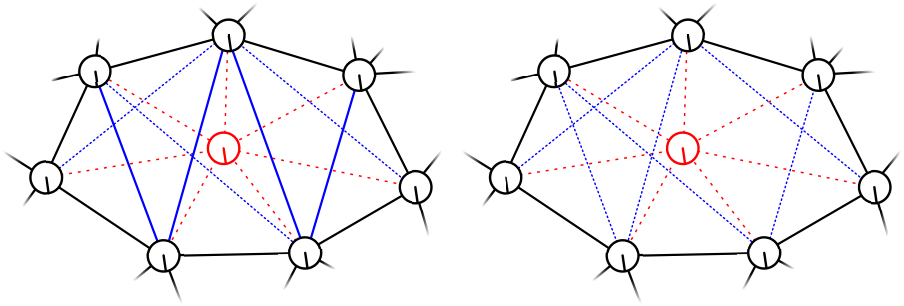
there might still be one from opposite triangles or as in Fig. 4.13 where two robots are competing for the same position. Even if the two competing robots can see each other, they can not immediately see that they are competing as they cannot exchange their goal positions (they are in different reference coordinate systems that are not necessarily available for the corresponding other robot). Allowing only one robot to be in Expanding-state at a time (e.g. by letting the robot with the minimal id retreat as soon as they sense each other) will drastically reduce the parallelism.

We solve the problem by two triggers. The first one is triggered if two expanding robots are closer than 0.8 times the maximal triangle edge length. It results in a retreat (abort expansion) of the robot that is closer to its origin frontier edge. This value is distributed by the public variable *expansion_value*. The second trigger is triggered if they are very close and it can be assumed that the first trigger for some reason failed. This trigger lets both robots abort the expansion but should only very seldom actually be used.

The general idea behind the first trigger is that nearly finished expansions are not aborted by some just started expansions. If a random robot or even both abort the expansion, long lasting loops can be created that possibly also block the expansion of the triangulation in other parts.

4.4 Redundant Robots

A problem of the OMAT-algorithm of Lee et al. is that due to the low resolution of the bearing sensors and missing distance sensors, the triangles might become smaller until the robots are infeasible dense. A simple heuristic would be to simply virtually remove a



(a) (b)

Figure 4.15: A robot is redundant for the triangulation if a triangulation can be built without it. The regarded robot is marked in red. The possible edges for the triangulation in blue. Fig. 4.15a has a valid triangulation marked in bold. Fig. 4.15a doesn't have one.

robot and check if the created hole can be triangulated without additional robots. If it can, we free the selected robot and obtain larger triangles as well as an additional free robot for further expansion. A positive example can be seen in Fig. 4.15a and a negative one in Fig. 4.15b. Note that the selected robot has all the necessary knowledge and can thus execute the calculation on its own.

The problem that arises is to calculate for a given polygon and a set of edges, if these edges contain a triangulation of the polygon.

Problem 4.1. *Given a simple polygon P and an edge set $E \subset V(P) \times V(P)$. Does E contain a polygon triangulation of P.*

A similar problem for a point set triangulation was already shown to be NP-hard by Llyod [52].

Theorem 4.1 (Lloyd 1997 [52])**.** *For a given set of edge segments, it is NP-hard to decide whether they contain a triangulation of their endpoints or not.*

Our problem can easily be reduced to this problem by adding the polygon edges, all connectivity edges that lay inside the polygon, as well as a trivial non-ambiguous triangulation of the outer.

Note that this does not say anything about the complexity of our problem except that it is at most as hard as the one of Theorem 4.1, thus is in NP (this, however, is trivial to see). Indeed, the problem is polynomial solvable by the following algorithm.

4.4.1 Algorithm

Let $P = v_0, v_1, v_2, \ldots, v_{n-1}, v_0$ be a polygon with the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_0)$. The inclusion operator $\in P$ is used for its vertices as well as its edges (for edges: $(u, v) = (v, u)$). To check that an edge (v, w) is in the interior of the polygon we use $(v, w) \subset P$. We denote the index for a vertex w in this polygon with $idx(w)$. Note that it is of no interest which vertex is chosen to be of index 0 but it has to be fixed. The index corresponding

successor respective predecessor of a vertex v is denoted by v_{+1} resp. v_{-1} (the successor of the last vertex is of course the first vertex, etc.).

We assume that E is filtered such that for all $e \in E : e \subset P$. This can be done with a simple line intersection test. Further, we assume that no three points are collinear. The procedure hasTriangulization that checks if a polygon has a triangulation in an edge set is stated in Alg. 4. The triangulation itself can be recovered in linear time from the markers made by Alg. 4.

Algorithm 4 PolygonTriangulateability

```
1:  procedure hasTriangulization(Polygon P, Edge Set E)
2:      for (u, v) ∈ E do
3:          if ∃w : (u, w) ∈ P ∧ (v, w) ∈ P then recEatTriangle((u, v), w, E)
4:      for (u, v) ∈ E do
5:          if (u, v) marked on both sides then
6:              return true
7:      return false
8:  procedure recEatTriangle(Edge (u, v), Vertex w, Edge Set E)
9:      if is_marked((u, v), w) then
10:         return
11:     mark((u, v), w)
12:     u′, v′ ←getSuccessors((u, v), w)
13:     if (u, v′) ∈ E then
14:         recEatTriangle((u, v′), v, E)
15:     if (u′, v) ∈ E then
16:         recEatTriangle((u′, v), u, E)
17:     for s ∈getEdgesBetween((u, v), (v, v′), w, E) do
18:         if (u, s) ∈ E ∧ (v, s) ∈ E∧is_marked_op((u, s), v) then
19:             recEatTriangle((v, s), u, E)
20:     for s ∈getEdgesBetween((u′, u), (u, v), w, E) do
21:         if (v, s) ∈ E ∧ (u, s) ∈ E∧is_marked_op((v, s), u) then
22:             recEatTriangle((u, s), v, E)
```

The following subroutines are used:

mark$((u, v), w)$ Marks the edge (u, v) to the side on which w (not collinear to (u, v)) lies as feasible. This means, there is a triangulation to this side. w is saved for later reconstruction of the triangulation. If an edge is marked from both sides, there exists a triangulation of the full polygon. Checking on which side of an edge a point lies is an elementary operation.

is_marked$((u, v), w)$ Checks if the edge (u, v) is marked to the side on which w lies. Note that this has to be the same result as *is_marked*$((v, u), w)$ which however is trivial

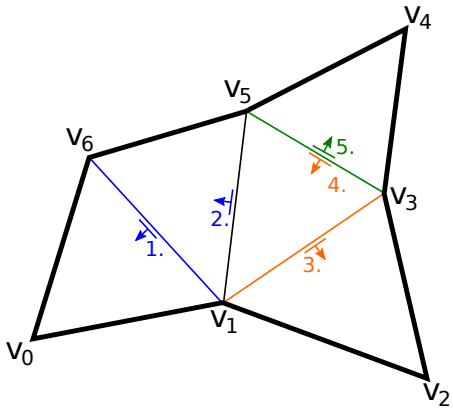


Figure 4.16: First the recursion is started with the 'ear-edge' (v_1, v_6) and stops with (v_1, v_5) as neither (v_1, v_3) nor (v_3, v_5) are marked. Next the recursion is started on the next 'ear edge' (v_1, v_3) and finishes with (v_3, v_5) as (v_1, v_5) marks the other polygon half as triangulateable. With the next 'ear-edge' (v_3, v_5) an edge is marked from both sides and thus a triangulation has been found.

to implement.

is_marked_op$((u,v),w)$ Analogous to *is_marked* but this time it is checked if the edge (u,v) is marked to the side on which w does *not* lie.

getSuccessors$((u,v),w)$ Returns the successor vertices to u and v in the polygons-half that does not contain w. With successor is not meant the ones with the next higher index, but the two vertices adjacent to the edge (u,v). This can easily be implemented by using the indices, e.g. if $idx(u) < idx(w) < idx(v)$ than return u_{-1} and v_{+1}.

getEdgesBetween$((u,v),(v,w),s,E)$ The edges of a given edge set E incident to v are split into two by the two edges regarding their orientation. The edge subset not containing (v,s) is returned (w.l.o.g. one of it contains (v,s)). The order of the edges is irrelevant but they have to share one vertex which is assumed to be v here. This can be implemented in $O(|E|)$.

An example is given in Fig. 4.16.

Theorem 4.2. *Algorithm 4 has a runtime of* $O((|P|+|E|)^2)$ *and a memory usage of* $O(|P|+|E|)$.

Proof. For simplicity we denote $n = |P| + |E|$. The memory usage is obvious as we only need to save a constant amount of data for each edge in E and also the recursion depth is in $O(|E|)$.

The procedure hasTriangulization has a loop of runtime $O(n)$ (without recEatTriangle) as it only has to run over all consecutive vertex triples v_i, v_{i+1}, v_{i+2} and check if $(v_i, v_{i+2}) \in E$. This results in $O(n)$ calls of recEatTriangle. The final loop has obviously a runtime of $O(n)$.

We now only have to analyse recEatTriangle. For each edge in E the function only goes further if it hasn't already been executed from this side (thus except for two times, the runtime for an edge is $O(1)$). If it goes further it has two if-conditions that can be executed in $O(1)$ (without recEatTriangle calls) and two for-loops with $O(n)$ runtime (also without recEatTriangle calls). The overall runtime of recEatTriangle is thus $O((|E|)^2)$ plus the amount of calls. The amount of calls is easy to limit as it is called $O(n)$ times in hasTriangulization and also $O(n)$ times in recEatTriangle for the $O(n)$ times it does not directly return. Thus, it is called at most $O(n^2)$ times and we have an overall runtime of $O(n^2)$. □

4.4.2 High Quality Triangulation

Algorithm 4 can not only check if an edge set contains any triangulation but also if it contains a triangulation where each triangle fulfills a local quality constraint. With *local* we mean that only properties of the triangle itself but not e.g. from neighbored triangles are allowed to be use. We can for example enforce a minimal edge length constraint on the triangles or a minimal area constraint.

For this we simply have to abort recEatTriangle if the passed triangle u, v, w (from edge (u, v) and the vertex w) does not fulfill the quality constraint. It is easy to see that neither correctness nor runtime of Algorithm 4 is harmed by this modification. The corresponding high quality triangulation can be reconstructed as for the unmodified version by the saved vertices in the markers.

4.5 Simulation Experiments

To validate the triangulation method, simulation experiments in three different environments have been made. All three environments have a size of $8\,\text{m} * 12\,\text{m}$ with the entrance placed in the middle of the lower horizontal boundary. While the first environment does not have any further obstacles, the second environment has a large $2\,\text{m} * 2\,\text{m}$ obstacles placed in the middle and the third environment has four small $1\,\text{m} * 1\,\text{m}$ obstacles. The three environments can also be seen in the triangulation example in Fig. 4.18, 4.19, and 4.20. For each environment, 10 runs have been made.

As written in Sec. 2.4, the robots have a radius of 5 cm and a range of 1.2 m. The communication is blocked by obstacle but not other robots. A simulation step has a length of $1/60\,\text{s}$ and the robots move with a velocity of up to $0.2\,\text{m}\,\text{s}^{-1}$. The movement is slightly disturbed. Neighbors can only be sensed via receiving messages that only have a chance of 50% in every simulation step. The distance measurements (also updated after receiving a message of the corresponding neighbor) is noised with up to 1% deviation.

A new robot is introduced as soon as the previous robot has a distance of at least 0.8 m from the insertion point (center of entrance triangle). The optimal edge length is set to 0.95 m while the maximal reliable edge length is set to 1.05 m.

The tables 4.1, 4.2, and 4.3 show the results in means of time until no frontier edge was left (measurements have been made in 50 step intervals), number of robots in the fi-

Instance	Steps	#Robots	#Triangles	Area m^2	#Wall	#Blacklist
0	200300	189	320	91.19	116	0
1	225100	206	349	90.21	124	3
2	199900	193	327	90.83	114	1
3	217950	190	321	89.73	113	1
4	221300	205	351	90.42	116	0
5	211300	201	343	90.73	112	4
6	219800	201	342	90.83	119	0
7	200550	184	311	90.59	108	1
8	210750	193	324	90.15	117	1
9	206050	188	318	90.91	115	0
AVG	211300	195.0	330.6	90.56	115.4	1.1

Table 4.1: The results of ten runs in the first environment. The environment and a triangulation of one of the runs can be seen in Fig. 4.18.

nal triangulation, number of triangles in the final triangulation, covered area of the final triangulation, as well as numbers of edges marked as wall or blacklisted in the final triangulation. It can be seen that the first environment has in averaged been covered to 93.9%, the second to 91.7%, and the third to 89.8%. While this might seem to be low values, the Figures 4.18, 4.19, and 4.20 show that the practical coverage is probably still sufficiently good for most use cases.

The distribution of the areas (quality) of the triangles can be seen in Fig. 4.17. For all three environments the distributions are similar.

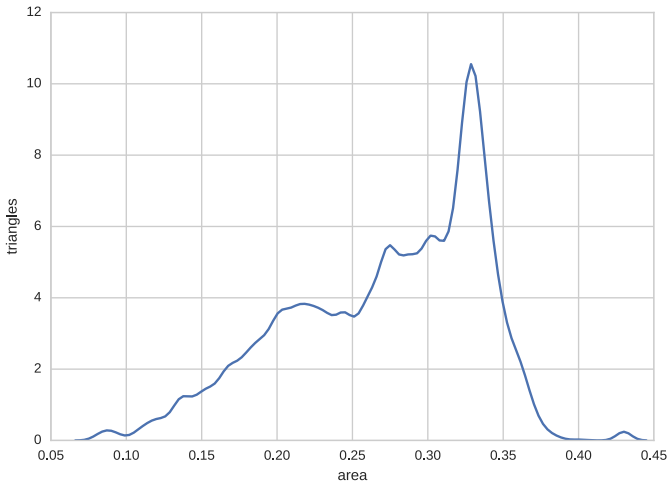
Some observations can be made: In Fig. 4.18 a blacklisted triangle can be seen. This equals a hole in the triangulation and is a problem of finding an expansion point that not only increases the quality of the newly created triangle but also considers future adjacent triangles. In Fig. 4.20 it can be seen that the outer boundary of the triangulation can have blacklisted edge that are not the fault of a bad expansion point. Further, the convex vertices of the obstacles can be seen as one of the primary reasons for blacklisted edges.

Instance	Steps	#Robots	#Triangles	Area m^2	#Wall	#Blacklist
0	211300	192	314	84.85	133	5
1	192700	186	303	84.49	133	3
2	192850	190	308	85.64	135	5
3	208800	189	308	84.25	138	2
4	216100	194	315	84.81	135	7
5	197600	179	288	83.10	134	7
6	223900	196	320	85.41	140	5
7	216900	197	323	85.00	135	4
8	197350	190	307	84.91	136	8
9	207350	199	328	84.79	134	4
AVG	206485	191.2	311.4	84.73	135.3	5

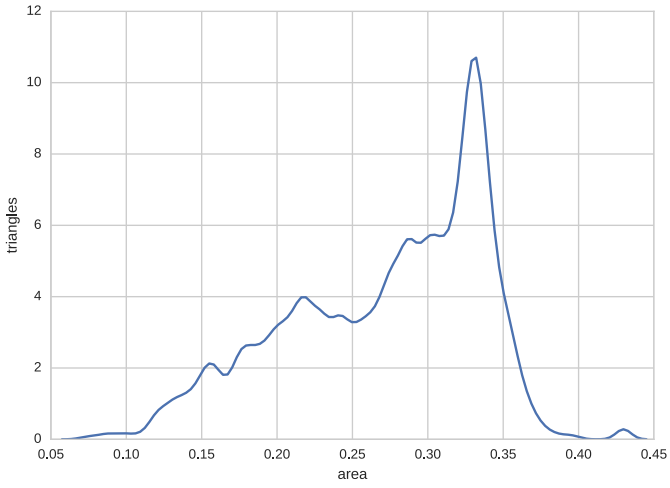
Table 4.2: The results of ten runs in environment 2. The environment and a triangulation of one of the runs can be seen in Fig. 4.19.

Instance	Steps	#Robots	#Triangles	Area m^2	#Wall	#Blacklist
0	228350	212	329	83.25	168	18
1	234750	202	313	83.47	164	16
2	227800	208	325	82.34	170	13
3	226850	200	312	83.02	167	19
4	232450	205	324	83.36	161	13
5	232300	213	334	82.49	176	16
6	234000	207	325	83.36	164	18
7	214250	197	307	82.52	173	10
8	216650	198	311	83.09	156	17
9	242800	209	326	82.78	176	18
AVG	229020	205.1	320.6	82.97	167.5	15.8

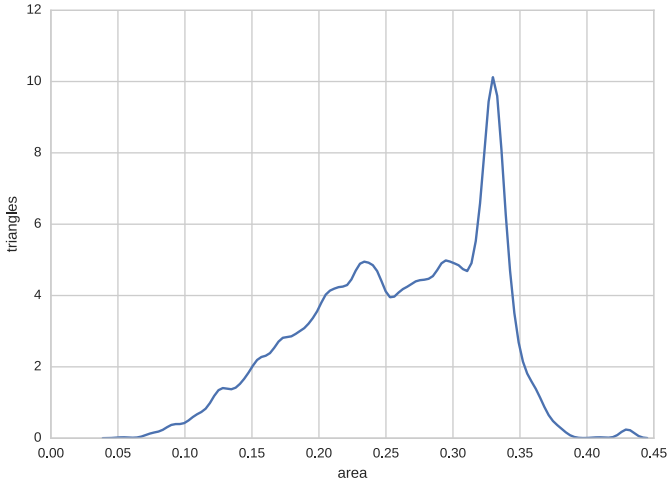
Table 4.3: The results of ten runs in environment 3. The environment and a triangulation of one of the runs can be seen in Fig. 4.20.



(a) Areas of triangles in environment 1



(b) Areas of triangles in environment 2



(c) Areas of triangles in environment 3

Figure 4.17: The areas of the triangles

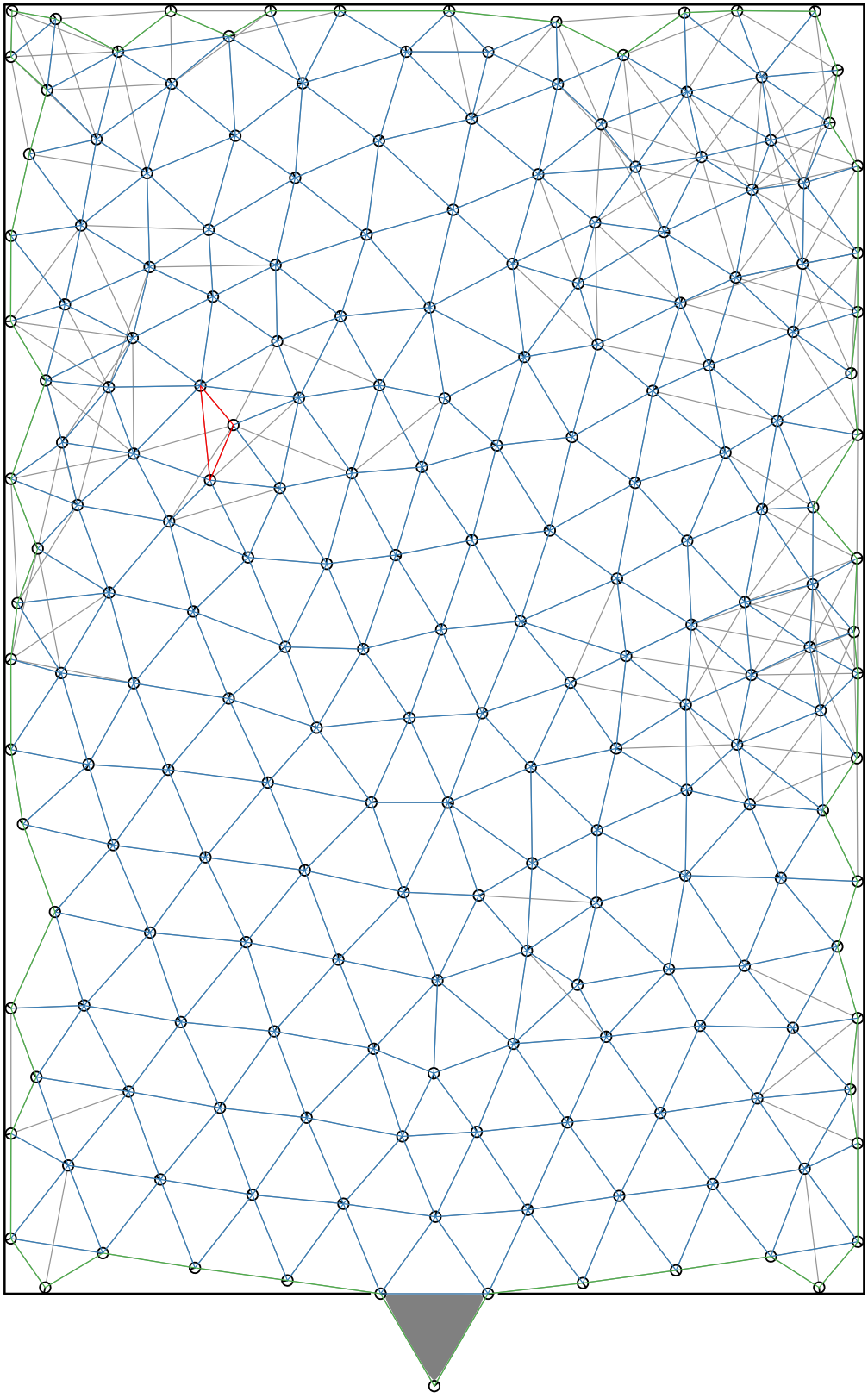


Figure 4.18: Example of a triangulation of the first environment. Initial triangle in grey, interior triangulation edges in blue, wall edges in green, and blacklisted edges in red.

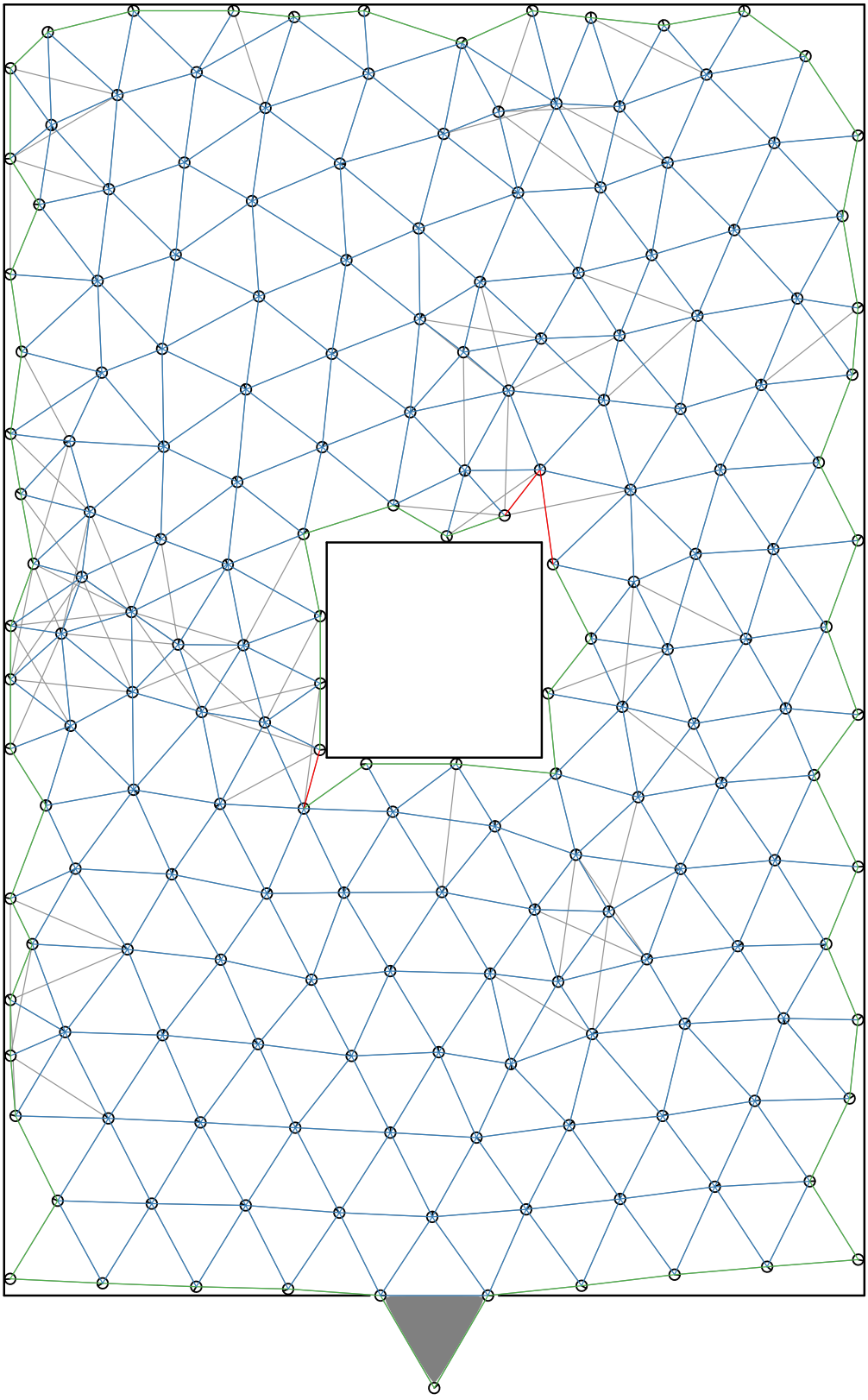


Figure 4.19: Example of a triangulation of the second environment. Initial triangle in grey, interior triangulation edges in blue, wall edges in green, and blacklisted edges in red.

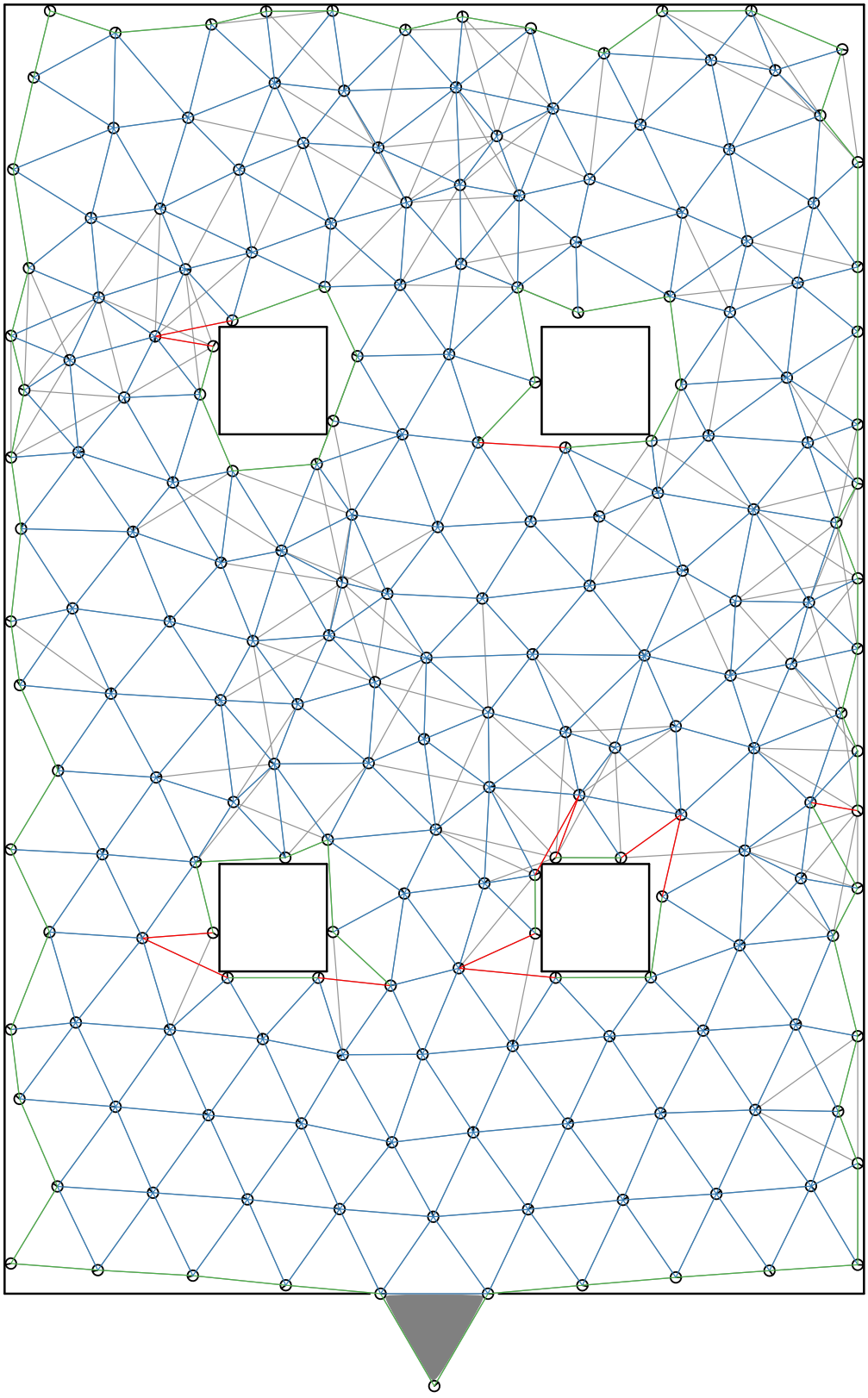


Figure 4.20: Example of a triangulation of the third environment. Initial triangle in grey, interior triangulation edges in blue, wall edges in green, and blacklisted edges in red.

5 Area Partition

While the previous chapter considered how to build a triangulation in the environment, this chapters considers how to remove robots out of it such that a sufficiently dense 'street network' remains. Such a network can be seen in Fig. 5.1. Sufficiently dense network refers to a network where the free areas/holes are of limited size. We refer with size limitation to a limited radius in hops of a removed connected component of robots. The wall robots of the triangulation are not allowed to be removed as this is considered as a decrease of the coverage.

There are many examples where such a network can be useful. The most obvious might be the provision of a 'highway'.

Example 5.1 (Highway)**.** *Due to the limited radius of the fields, every robot not in range of the network can reach it in limited time by moving straight forward in any direction. If a robot has found something that it wants to carry home or needs to recharge its batteries, it can simply move in any direction and will quickly find the 'highway' that leads it home. Of course the highway can also be used to lead robots to other positions, e.g. a robot can call for support on it and other robots can move to it via the network.*

You may also focus on the fields instead of the network. The triangulation creates a partition of the environment into triangular fields. If the robots have sufficiently accurate odometry to move on their own within small environments, this might be not necessary.

Example 5.2 (Surveillance with odometry)**.** *A robot with odometry might be able to scan a limited area bounded by static robots without constant localization via beacons. In an environment with a partition into fields of limited size, surveillance robots can move from field to field an in each field perform such a scan. This environment can be built with much fewer robots than a full triangulation on the cost that instead of just visiting a field, the field has to be scanned.*

Another useful example might be the following.

Example 5.3 (Capturing)**.** *A set of catcher robots want to catch a moving target. We have a network of static robots that can sense but not capture the target. This allows them to identify the field transitions of the target and thus the field the target is currently in. The catcher robots now can quickly move to it using the 'highways' and surround the corresponding field.*

The highways can be of different widths. It can be a simple path of static robots or a chain of triangles. Possible are also highways with multiple lanes. However, in this thesis we only focus one the size of the holes. The NP-hardness proof uses simple robot paths the heuristic on the other hand produces chains of triangles. This is done for two reasons: First, for moving along paths of robots a new movement controller would have

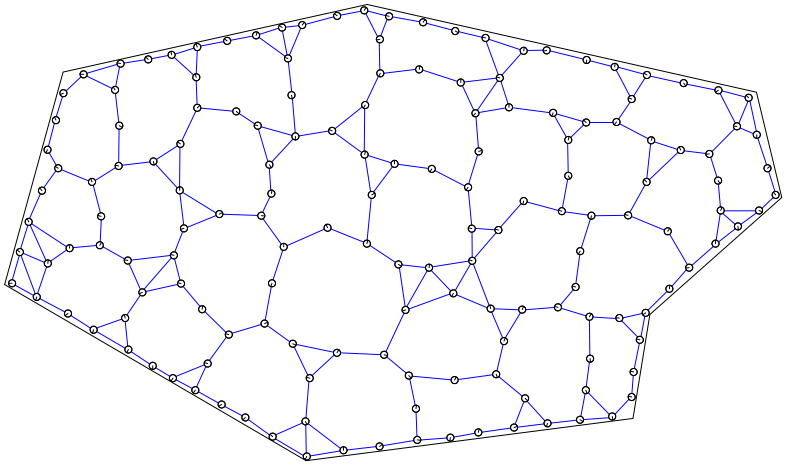


Figure 5.1: A partition in larger fields (blue). It uses much fewer robots than a full triangulation. The fields are of limited size.

been needed. Second, a chain of triangles is easier to obtain than a connected path with the proposed approach.

In the following we first prove that the removal of a maximal amount of robots under given constraints on the size of the holes is NP-hard in Sec. 5.1. Then we propose a heuristic that can produce such a network parallel to the triangulation (thus the freed robots can directly be reused to extend the triangulation) in Sec. 5.2. The heuristic never creates too large holes but the performance might vary strongly. This can be seen in Sec. 5.3 where simulation experiments are provided.

5.1 NP-hardness

In this section we prove that it is NP-hard to remove the maximum amount of robots where each removed cluster is allowed to have only one robot. Thus, the radius of the removed components is limited to 0. This reduces our problem to a subset of the NP-hard independent set problem as we want to find the maximum independent set in the set of all robots that are allowed to be removed (not wall robots). Note that we need to reduce the problem to a superset and not a subset of an NP-hard problem to show its own hardness.

The proof itself is a reduction on Planar-3SAT and does not use any novel techniques. It is mainly inspired by the NP-hardness proof of dispersion by Baur and Fekete [53]. Hence, we only discuss the general idea without going too much into details.

Our proof constructs an environment and triangulation for an arbitrary Planar-3SAT formula that allows the removal of a specific amount of robots if and only if there is a valid variable assignment that makes the formula true. The construction consists of three elements:

1. Variable gadgets that ensure a variable is only true or false. A violation results in a reduced number of removed robots.

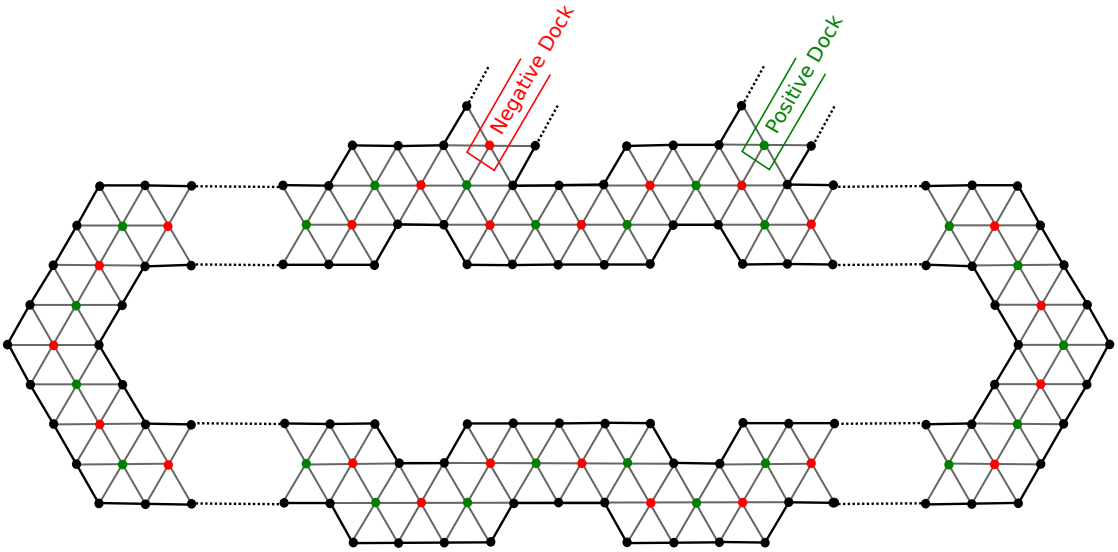


Figure 5.2: A variable gadget. There are only two possible selections in the circle highlighted in red and green. The thick black edges denote an edge between the robots but also a wall. The docks are already part of the connectors.

2. Clause gadgets that ensure each clause is satisfied (at least one of its three literals is true). If none of the literals is true, the number of removed robots is less than with at least one true literal. More than one true literal does not increase the number.

3. Connectors that propagate the variable assignment to the clauses.

We discuss the elements in the stated order.

Variable Gadgets: An example of a variable gadget can be seen in Fig. 5.2. The removal candidates are an even circle. Obviously, there are exactly two maximum independent sets of half the circles size. One of it will be the true assignment and the other the false assignment. Besides the circle it has positive and negative docks that are connected to the clause gadgets by connectors. If a clause has a negation of this variable, it uses the negative dock and the positive otherwise. A variable is true if and only if the robot in front of the positive dock is not removed thus the first robot of the connector can be removed.

Clause Gadget: A clause gadget can be seen in Fig. 5.3. If and only if one of the red robots is not removed, we are able to remove one of the triangle's robots. Each true atom will remove its corresponding green robot and the red otherwise. Thus, a satisfied clause allows removing four robots while an unsatisfied only three. The negation of variables is done by using the negative dock instead of the positive dock of the variable gadget.

Connectors The candidates of the connectors are paths with an even amount of robots and connect a positive or negative dock of a variable to the corresponding dock of a clause. The size of the maximum independent set is half the path's robots. If and only if the first robot of the path (the one adjacent to the variable gadget) can be removed the last robot of the path can remain while still having a maximum independent set. If the last robot (the

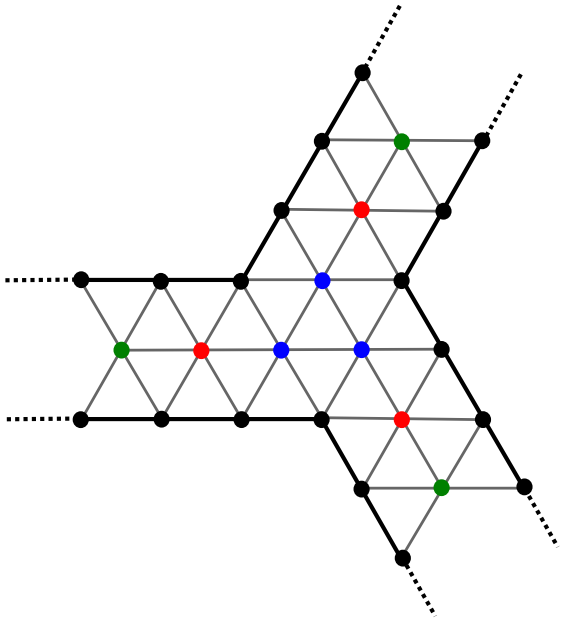


Figure 5.3: A clause gadget. One of the blue robots can be selected if and only if one of the red ones is not. The thick black edges denote an edge between the robots but also a wall. The red robots are the last robots of the connectors.

robot adjacent to the triangle in the clause gadget) is not removed, a robot of the triangle can be removed. In order to be able to connect the clause gadgets and variable gadgets, it has to be flexible and is thus not in the triangle grid. It is easy to see that stretching and bending of the connectors is possible to a sufficient degree.

As variables gadgets, clause gadgets, and connectors are all disjunct and a satisfying variable assignment obviously allows each of the elements to choose one of its maximum independent sets and the selection of these again imply a specific assignment.

5.2 Heuristic

The proposed heuristic does not focus on holes but on clusters. A cluster refers to a connected set of robots with a limited radius. An interior robot of the triangulation is allowed to be in exactly one cluster, wall robots are never in a cluster. Only the robots adjacent only to robots of the same cluster are to be removed. Thus, the margin robots of the cluster remain and the created hole has a smaller radius than the cluster. E.g. a cluster of radius 3 can create at most holes of radius 2.

The hardness proof already lessens the hope to find an optimal algorithm for the ('simple') centralized offline version. The distributed online version provides two further difficulties that complicate the problem:

1. The robots have only local knowledge of the environment that is far below the range of a potential cluster.

2. The triangulation is not yet finished but in the finished parts clusters should already

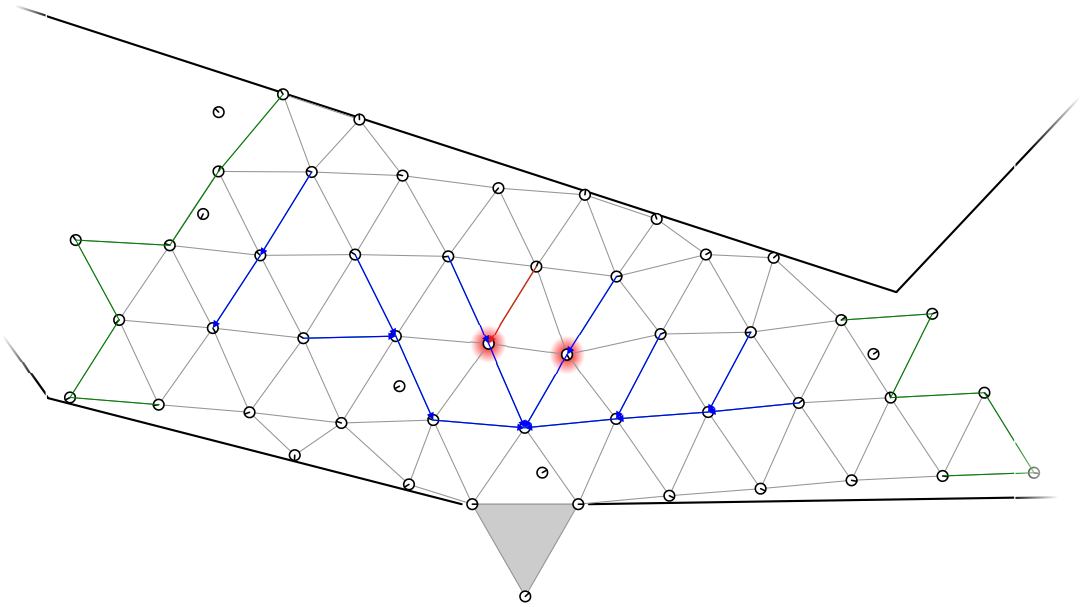


Figure 5.4: A partial triangulation with two clusters with MAX_RADIUS = 3. The green edges in the triangulation highlight the frontier. The blue arrows display the parent relationship in a cluster tree for non-stable robots and the red respective for stable robots. There is only one such stable robot as the frontier is still very close. The red highlighted robots of the large cluster are the only robots that are not on its margin and thus the only robots that would be freed if the cluster becomes frozen. The grey triangle is the entrance triangle.

been built and robots of them freed to speed up the triangulation process.

In the following, we provide a simple heuristic for the distributed online scenario in Sec. 5.2.1 and evaluate its performance in simulation experiments in Sec. 5.3.

5.2.1 Algorithm

The general idea of the heuristic is that every static robot joins the largest adjacent cluster (and possibly leaves the current smaller one) if its distance to the center does not exceed the maximum radius. If it cannot join any a cluster, it creates a new one with itself as center. The clusters are organized as rooted trees (the root as center) which allows to easily calculate the size as well as limit the radius. The root can also decide when to freeze the cluster and free the robots. This happens if the cluster can no longer grow, no robot wishes to leave, and the frontier is far away.

We now describe the details: First the used public variables and constants, then how the size is calculated, then how a new cluster emerges, then how a robot joins an existent cluster, and finally how a cluster is frozen and its robots freed. An example can be seen in Fig. 5.4 and the concrete procedure is stated in Alg. 5.

Algorithm 5 Clustering

1: **procedure** CLUSTER_ITERATION(Robot r)
2: **if** r is not interior robot in triangulation **then**
3: **return**
4: **if** r.frozen **then**
5: **if** $\exists n \in N_\Delta(r) : (n.\text{clust\_id} \neq r.\text{clust\_id} \vee n \text{ not interior})$ **then**
6: r.is_margin $\leftarrow$ true
7: **return**
8: **if** $\forall n \in N_\Delta(r) : ((n.\text{parent} \neq r.\text{id} \vee n.\text{is\_margin}) \wedge (n.\text{clust\_id} = r.\text{clust\_id} \rightarrow n.\text{frozen})$ **then**
9: Free robot
10: **return**
11: **if** r.stable = false $\vee$ r.parent.clust_id $\neq$ r.clust_id **then**
12: cand $\leftarrow \{n \in N_\Delta(r) \mid n.\text{hop\_dist} < \text{MAX\_RADIUS}\}$
13: **else**
14: cand $\leftarrow \{r.\text{parent}\}$
15: cand $\leftarrow \{n \in \text{cand} \mid \text{SIZE}(r, n.\text{clust\_id}) = \max\{\text{SIZE}(r, n'.\text{clust\_id}) \mid n' \in \text{cand}\}\}$
16: cand $\leftarrow \{n \in \text{cand} \mid n.\text{hop\_dist} = \min\{n'.\text{hop\_dist} \mid n' \in \text{cand}\}\}$
17: cand $\leftarrow \{n \in \text{cand} \mid n.\text{clust\_id} = \min\{n'.\text{clust\_id} \mid n' \in \text{cand}\}\}$
18: $p \leftarrow \text{maxarg}_n\{n \in \text{cand} \mid n.\text{id} = \min\{n'.\text{id} \mid n' \in \text{cand}\}\}$
19: **if** $p = \perp \wedge r.\text{hop\_dist} > 0$ **then**
20: r.clust_id $\leftarrow$ random new id
21: r.parent $\leftarrow \perp$
22: r.subsize $\leftarrow 1$; r.supersize $\leftarrow 1$
23: r.hop_dist $\leftarrow 0$
24: r.stable $\leftarrow$ false; r.frozen $\leftarrow$ false
25: **else if** $r.\text{hop\_dist} = 0 \wedge (p = \perp \vee r.\text{clust\_id} = p.\text{clust\_id} \vee r.\text{supersize} > p.\text{supersize} \vee (r.\text{supersize} = p.\text{supersize} \wedge r.\text{clust\_id} < p.\text{clust\_id}))$ **then**
26: $r.\text{subsize} \leftarrow 1 + \sum_{n \in N_\Delta(r), n.\text{parent} = r.id} n.\text{subsize}$
27: r.supersize $\leftarrow r$.subsize
28: r.stable $\leftarrow$ STABLE(r)
29: r.frozen $\leftarrow r$.stable
30: **else**
31: r.clust_id $\leftarrow p$.clust_id
32: $r.\text{subsize} \leftarrow 1 + \sum_{n \in N_\Delta(r), n.\text{parent} = r.id} n.\text{subsize}$
33: r.supersize $\leftarrow p$.supersize
34: r.hop_dist $\leftarrow p$.hop_dist $+ 1$
35: r.stable $\leftarrow$ STABLE(r)
36: r.frozen $\leftarrow p$.frozen
37: r.parent $\leftarrow p$

Variables

The following public variables and constants are used by the algorithm:

MAX_RADIUS The maximum radius a cluster is allowed to have. The radius is measured in hops. This value is constant and known in advance by all robots.

clust_id The ID of the cluster a robot is in. The initial value is $\perp$.

parent The parent in the cluster tree. For the root of the cluster this is $\perp$. This is also the initial value.

subsize The size of the subtree. The initial value is zero.

supersize The size of the cluster (tree). The initial value is zero.

hop_dist The hop distance to the root of the cluster. The initial value is $+\infty$.

stable True if the subtree has reached its maximal size either because the depth has reached the MAX_RADIUS or there is no way to expand due to walls. The initial value is false.

frozen Set to true if the cluster is ready to free its interior robots. The initial value is false.

is_margin Set to true if the robot is frozen but not interior to the cluster, i.e. it is at the margin of the cluster. The initial value is false.

Size

In a rooted tree, the size can easily be calculated in a distributed manner. For this three public variables are introduced: *parent*, *subsize*, and *supersize*. The variable *parent* simply contains the ID of the parent in the tree of the cluster (thus implements the tree). The variable *subsize* is the size of the subtree, thus the sum of all *subsize*s of all neighbors with the considered robot as parent plus one for the robot itself. The variable *supersize* is the size of the whole tree and thus the result of the size calculation. For every non-root robot, this is the *supersize* of its parent. For the root it is the *subsize*. This size determination is dynamic but the time for changes to propagate correlates linearly to the tree's radius.

The size of an adjacent cluster is determined as follows

$$\texttt{size}(r, \text{clust\_id}) = \max\{n.\text{supersize} \mid n \in N_\Delta(r) \wedge n.\text{clust\_id} = \text{clust\_id}\}$$

This makes the parent selection more stable compared to simply using the *supersize* of the corresponding neighbor.

Root

If a robot is not able to join any cluster, either because none of its adjacent robots are in one yet or the distance to the center is too large, it creates a new cluster on its own. For this, it randomly choses a cluster id (*clust_id*, possibly its own ID or a random number that is large enough to prevent collisions) and sets its hop distance to the root (*hop_dist*) to zero. The initial size (*subsize* and *supersize*) is obviously one and it does not have a parent in the resulting tree of the cluster. If a root robot can join a cluster larger than its own, it discards its own cluster (which then will vanish).

Joining a cluster

A robot joins the largest cluster of the clusters of adjacent robot with a *hop_dist* less than the maximal radius. If there are multiple choices, it choses the cluster with the smallest hop distance and if this is still ambiguous, the one with the smallest cluster id. Of this cluster, it choses the adjacent robot with the smallest hop distance to the root (and ID if ambiguous) as parent in the cluster tree. The hop distance of the robot is set to the parent's hop distance plus one. As long as a robot is not stable, it can freely switch its parents and clusters. On this way robots will leave small cluster and enter large cluster which then extrude the small clusters. If a robot is stable, it sticks to its parent as long as the parent doesn't change the cluster. A not-stable robot continuously joins the best cluster.

Freezing and Freeing

The previous part dynamically builds clusters and during the whole time clusters will arise and vanish as well as grow and shrink. Changes in one cluster can also propagate over multiple clusters. As the sizes of clusters grow lexicographically, the algorithm will converge after finite time in finite triangulations (assuming no external instabilities) but this might be a long time. Thus, we need a component that at some point freezes a cluster and allows us to free its interior robots. This point of time should be early as the freed robots can directly move to the frontier and extend the triangulation but the cluster should also wait as long as it can grow as the proportion of interior robots in the cluster usually grows with its size.

There are simple properties to recursively check if a subtree has reached its maximal size. First, all subtrees of this subtree have to be fully grown, too. To be sure, we demand that not only the subtrees but all robots with a higher hop distance have to be stable, too. Second, if the hop distance to the root of a robot equals the maximal radius it can not have any children and thus cannot grow. Third, if there are no adjacent robots that could join the cluster. This can be because they are margin robots of the triangulation or part of an already frozen clusters. As the frontier potentially induces a lot of changes its close environment, we further demand that all robots of a cluster have a distance to the frontier of at least the maximal radius.

We implement the state of a subtree to be fully grown and distanced to the frontier via the public variable *stable* and the above mentioned local evaluation by the function `stable`(*r*: Robot). If the root of the cluster becomes stable, it sets the variable *frozen* to true which is adapted recursively by all children. This freezes the cluster and allows freeing its interior robots. To ensure that the state change propagates to the cluster, a robot is only allowed to remove itself if all its children have removed themselves or marked themselves as margin or the cluster. Further, all adjacent robots have to be frozen, too. A robot marks itself as margin if it is frozen and it has adjacent robots that are either on the margin of the triangulation or belong to another cluster.

Instance	Steps	#Robots	#Triangles	Area m^2	#Freed
0	342550	319	450	118.08	47
1	338400	302	426	115.55	49
2	372000	312	458	121.58	44
3	328400	311	452	122.71	41
4	342000	308	449	119.99	49
5	328450	299	416	112.36	52
6	355300	318	454	116.50	48
7	334750	313	449	119.25	48
8	333950	301	428	118.59	46
9	321400	302	439	122.04	40
AVG	339720	308.5	442.1	118.67	46.4

Table 5.1: The results of ten runs using clusters with a maximal radius of 2 (radius of holes ≤ 1). An example of one of the runs can be seen in Fig. 5.5.

5.3 Simulation Experiments

The approach has been tested in a rectangular environment of size 12 m $*$ 15 m and three different cluster radii (2,3, and 4). For each radius, ten runs have been made with the same robot model as in the previous chapter. The results are given in means of number of time steps until no frontier edge is left and all static robots are margin, amount of static robots at this point, the number of triangles, the area covered by triangles, and the number of robots freed. They are given in Table 5.1 for radius 2, in Table 5.2 for radius 3, and in Table 5.3 for radius 4. Blacklisted triangles in the triangulation as they appeared in the previous chapter can significantly reduce the number of freed robots. As the problem of blacklisted triangles belong to the previous section and in the simple environment blacklisted edges are not needed, a deletion is triggered on occurrence of one. This rebuilds the triangulation at this part until no blacklisted edge is left and thus the final triangulation does not have any blacklisted edges. Examples for the final triangulations for the different radii can be seen in Figures 5.5, 5.6, and 5.7.

The results show that with radius 2 on average 46.4 robots are freed. For radius 3 the number strongly increases to 82.3 freed robots. For radius 4 the number increases to 109.7. It is surprising that the increase from radius 3 to radius 4 is relatively small as the volume grows quadratically with the radius. This, however, could be the fault of the small environment.

In the Figures 5.5, 5.6, and 5.7 it can be seen that with radius 2 the probability that a 'highway' is wider than one triangle is higher than for the radii 3 and 4. This observation agrees also with the non-displayed runs. Further, there are sometimes small clusters that theoretically could have been enlarged (e.g. see the small cluster in Fig. 5.7 and the robot on its lower right). An explanation for this could be larger slim clusters that are not able to free any robots and badly placed centers.

Instance	Steps	#Robots	#Triangles	Area m^2	#Freed
0	306250	261	350	99.16	74
1	335650	267	355	96.10	78
2	314400	275	366	97.49	81
3	330900	282	369	94.38	84
4	341700	285	375	96.69	83
5	302200	253	332	93.22	85
6	325700	276	357	90.02	91
7	312350	268	359	100.04	69
8	301700	266	343	89.10	94
9	300750	256	329	90.55	84
AVG	317160	268.9	353.5	94.68	82.3

Table 5.2: The results of ten runs using clusters with a maximal radius of 3 (radius of holes ≤ 2). An example of one of the runs can be seen in Fig. 5.6.

Instance	Steps	#Robots	#Triangles	Area m^2	#Freed
0	303600	223	275	76.75	120
1	330200	241	301	82.70	116
2	315000	237	290	77.23	117
3	312350	249	318	88.65	95
4	312450	239	312	87.89	110
5	348700	254	318	82.60	111
6	328750	252	317	81.84	112
7	353650	258	336	86.75	103
8	335800	236	296	79.41	116
9	363400	278	357	90.74	97
AVG	330390	246.7	312.0	83.46	109.7

Table 5.3: The results of ten runs using clusters with a maximal radius of 4 (radius of holes ≤ 3). An example of one of the runs can be seen in Fig. 5.7.

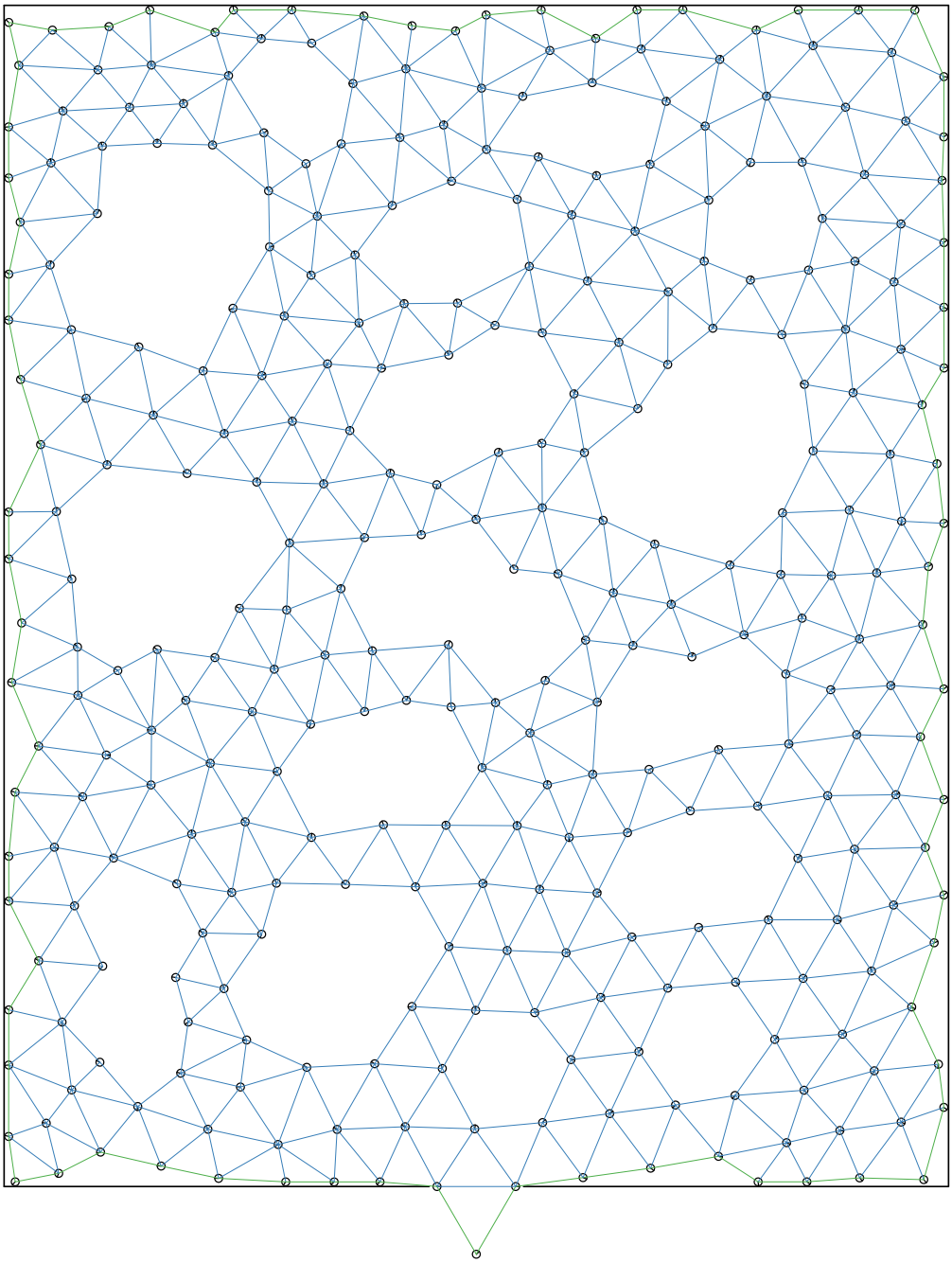


Figure 5.5: Instance 0 of partition experiments with cluster radius 2 (hole radius 1).

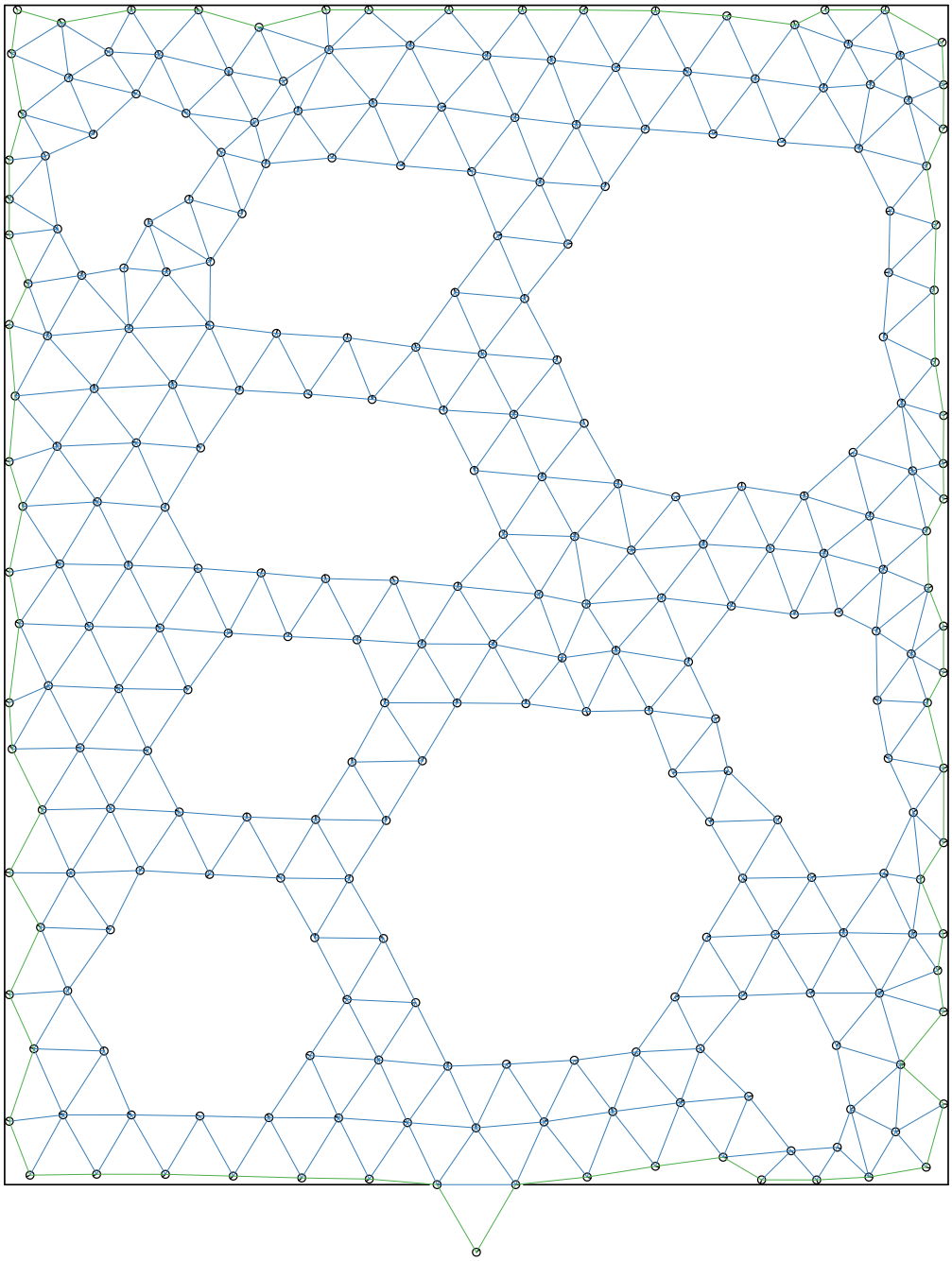


Figure 5.6: Instance 0 of partition experiments with cluster radius 3 (hole radius 2).

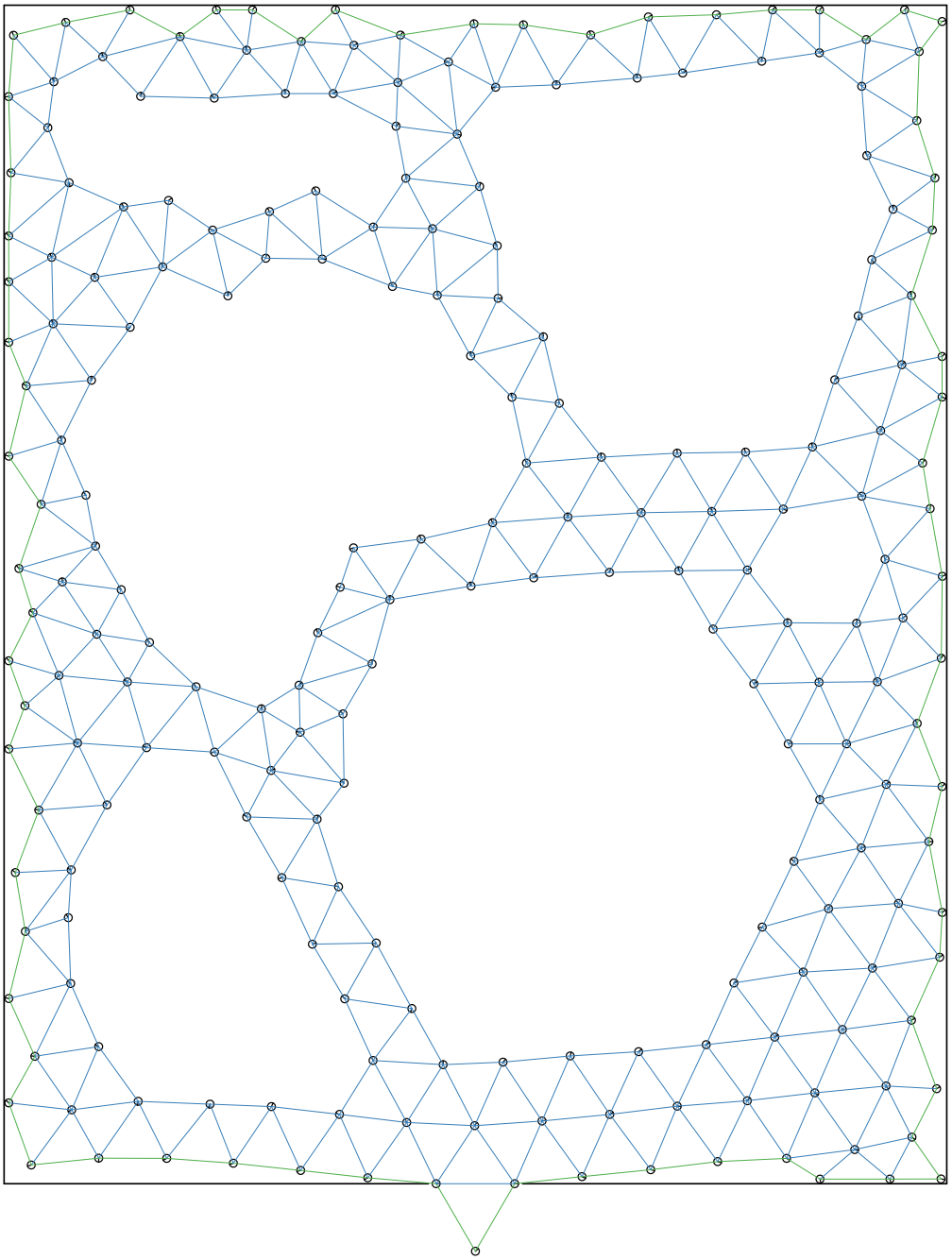


Figure 5.7: Instance 0 of partition experiments with cluster radius 4 (hole radius 3).

6 Conclusion

In this thesis we considered two problems: First, how to build a triangulation with robots that only have neighbor distance information and collision sensing. Second, how to remove robots of this triangulation such that a sufficiently dense street network remains. We stated an algorithm to build a triangulation that grows breadth first and allows a high degree of parallelism. A metric for the placement of robots such that only good triangles are created has shown to be problematic. We also showed that a robot can calculate in polynomial time if it is redundant in this triangulation. This problem equals the geometric question if an edge set contains a triangulation for a simple polygon. For the second problem, we showed that it is NP-hard to remove the maximum amount of robots. We gave a heuristic that can be executed parallel to the triangulation but in experiments it performed poorly with a cluster radius of 2. For higher radii it performed better but the environment in the simulations was simple.

There is much future work remaining. First, there are some aspects to be considered for practical implementations. The localization has been implemented rather naïve and would probably fail on real hardware where the measurements can be far more erroneous. There are many techniques that can be used to remove outliers and improve the measurements by combining more information. Further, the localization frequency was relatively high in the simulations. The minimal localization frequency and accuracy needed for the movement controller to perform reliable should be researched.

Second, the metric to rate expansion points still produced some bad triangles. The performance of other metrics should be evaluated. A perfect local metric that does only creates good triangles that provides enough space for navigation is probably impossible but this remains to be proven. Another approach might be to replace some robots if it is not possible to create a good triangle.

Third, a frontier edge is marked as wall if there has been a collision with an object that is no robots during expansion. This creates a lot of uncovered area around the walls which could be improved by retrying the expansion with different angles. The positions of the collisions could be saved and a local map of the wall created.

Fourth, the frontier pheromone currently does not consider how many robots are already available or currently following the trace. It might be possible to increase the value the pheromone value gets increased with every hop if it passes areas with many free robots. This could also improve the distribution of free robots and reduce the risk of collisions.

Fifth, the heuristic for partition creates many thick ‘highways’. If the roots of the clusters are able to move, this could possibly be improved by implementing attraction and repulsion between the roots comparable to flocking. Further, experiments with larger en-

vironments and obstacles have to be made to evaluate the performance of the heuristic more accurate. For now, the partition also does not have a reliable reparation mechanism as the triangulation. This can easily implemented by refilling a hole as soon as its margin has been damaged. Another interesting point may be to purposely create wider highways leading to important areas with a high demand of robots.

Bibliography

[1] Seoung Kyou Lee, Aaron Becker, Sándor P Fekete, Alexander Kroller, and James McLurkin. Exploration via structured triangulation by a multi-robot system with bearing-only low-resolution sensors. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 2150–2157. IEEE, 2014.

[2] 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*, May 2012.

[3] James McLurkin, Andrew J Lynch, Scott Rixner, Thomas W Barr, Alvin Chou, Kathleen Foster, and Siegfried Bilstein. A low-cost multi-robot system for research, teaching, and outreach. In *Distributed Autonomous Robotic Systems*, pages 597–609. Springer, 2013.

[4] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stephane Magnenat, Jean-Christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on autonomous robot systems and competitions*, volume 1, pages 59–65. IPCB: Instituto Politécnico de Castelo Branco, 2009.

[5] Farshad Arvin, John Murray, Chun Zhang, Shigang Yue, et al. Colias: an autonomous micro robot for swarm robotic applications. *International Journal of Advanced Robotic Systems*, 11(113):1–10, 2014.

[6] Raimon Casanova, A Dieguez, Andreu Sanuy, Anna Arbat, Oscar Alonso, Joan Canals, Manel Puig, and Josep Samitier. Enabling swarm behavior in mm 3-sized robots with specific designed integrated electronics. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 3797–3802. IEEE, 2007.

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

[8] 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.

[9] 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.

[10] 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.

[11] Brian Hrolenok, Sean Luke, Keith Sullivan, and Christopher Vo. Collaborative foraging using beacons. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 3-Volume 3*, pages 1197–1204. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[12] R Andrew Russell. Heat trails as short-lived navigational markers for mobile robots. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 4, pages 3534–3539. IEEE, 1997.

[13] Andrew Russell, David Thiel, and Alan Mackay-Sim. Sensing odour trails for mobile robot navigation. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 2672–2677. IEEE, 1994.

[14] Titus Sharpe and Barbara Webb. Simulated and situated models of chemical trail following in ants. In *Proc. 5th Int. Conf. Simulation of Adaptive Behavior*, pages 195–204, 1998.

[15] Ralf Mayet, Jonathan Roberz, Thomas Schmickl, and Karl Crailsheim. Antbots: A feasible visual emulation of pheromone trails for swarm robots. In *Swarm Intelligence*, pages 84–94. Springer, 2010.

[16] Vittorio A Ziparo, Alexander Kleiner, Bernhard Nebel, and Daniele Nardi. Rfid-based exploration for large robot teams. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 4606–4613. IEEE, 2007.

[17] Ken Sugawara, Toshiya Kazama, and Toshinori Watanabe. Foraging behavior of interacting robots with virtual pheromone. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 3074–3079. IEEE, 2004.

[18] Platypus3000 Swarm Robot Simulator. `https://github.com/SwarmRoboticResearch/platypus3000`.

[19] JBox2D: A Java Physics Engine. `http://www.jbox2d.org/`.

[20] Sándor P Fekete, Tom Kamphans, Alexander Kröller, Joseph SB Mitchell, and Christiane Schmidt. Exploring and triangulating a region by a swarm of robots. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 206–217. Springer, 2011.

[21] Sándor P Fekete, Sophia Rex, and Christiane Schmidt. Online exploration and triangulation in orthogonal polygonal regions. In *WALCOM: Algorithms and Computation*, pages 29–40. Springer, 2013.

[22] Aaron Becker, Sándor P Fekete, Alexander Kröller, Seoung Kyou Lee, James McLurkin, and Christiane Schmidt. Triangulating unknown environments using robot swarms. In *Proceedings of the twenty-ninth annual symposium on Computational geometry*, pages 345–346. ACM, 2013.

[23] Seoung Kyou Lee, Sándor P Fekete, and James McLurkin. Geodesic topological voronoi tessellations in triangulated environments with multi-robot systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 3858–3865. IEEE, 2014.

[24] Daniela Maftuleac, Seoung Kyou Lee, Sándor P Fekete, Aditya Kumar Akash, Alejandro López-Ortiz, and James McLurkin. Local policies for efficiently patrolling a triangulated region by a robot swarm. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 1809–1815. IEEE, 2015.

[25] Sven Koenig, Boleslaw Szymanski, and Yaxin Liu. Efficient and inefficient ant coverage methods. *Annals of Mathematics and Artificial Intelligence*, 31(1-4):41–76, 2001.

[26] Israel Wagner, Michael Lindenbaum, Alfred M Bruckstein, et al. Distributed covering by ant-robots using evaporating traces. *Robotics and Automation, IEEE Transactions on*, 15(5):918–933, 1999.

[27] Mihai Andries and François Charpillet. Multi-robot exploration of unknown environments with identification of exploration completion and post-exploration rendezvous using ant algorithms. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 5571–5578. IEEE, 2013.

[28] Mihai Andries and Francois Charpillet. Multi-robot taboo-list exploration of unknown structured environments. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sept 28 - Oct 03, 2015, Congress Center Hamburg, Hamburg, Germany*, 2015.

[29] Nicolas Pepin, Olivier Simonin, and François Charpillet. Intelligent tiles-putting situated multi-agents models in real world. In *ICAART*, pages 513–519, 2009.

[30] Timothy Stirling, Steffen Wischmann, and Dario Floreano. Energy-efficient indoor search by swarms of simulated flying robots without global information. *Swarm Intelligence*, 4(2):117–143, 2010.

[31] Maxim A Batalin and Gaurav S Sukhatme. Coverage, exploration and deployment by a mobile robot and communication network. *Telecommunication Systems*, 26(2-4):181–196, 2004.

[32] Asish Ghoshal and Dylan A Shell. Being there, being the rrt: Space-filling and searching in place with minimalist robots. In *2011 AAAI Spring Symposium Series*, 2011.

[33] Nicholas R Hoff III, Amelia Sagoff, Robert J Wood, and Radhika Nagpal. Two foraging algorithms for robot swarms using only local communication. In *Robotics and Biomimetics (ROBIO), 2010 IEEE International Conference on*, pages 123–130. IEEE, 2010.

[34] Keith J O'Hara and Tucker R Balch. Pervasive sensor-less networks for cooperative multi-robot tasks. In *Distributed Autonomous Robotic Systems 6*, pages 305–314. Springer, 2007.

[35] Qun Li, Michael De Rosa, and Daniela Rus. Distributed algorithms for guiding navigation across a sensor network. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 313–325. ACM, 2003.

[36] Eric J Barth. A dynamic programming approach to robotic swarm navigation using relay markers. In *American Control Conference, 2003. Proceedings of the 2003*, volume 6, pages 5264–5269. IEEE, 2003.

[37] Alan FT Winfield. Distributed sensing and data collection via broken ad hoc wireless connected networks of mobile robots. In *Distributed Autonomous Robotic Systems 4*, pages 273–282. Springer, 2000.

[38] Andrew Howard, Maja J Matarić, and Gaurav S Sukhatme. An incremental self-deployment algorithm for mobile sensor networks. *Autonomous Robots*, 13(2):113–126, 2002.

[39] 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.

[40] James McLurkin and Jennifer Smith. Distributed algorithms for dispersion in indoor environments using a swarm of autonomous mobile robots. In *Distributed autonomous robotic systems 6*, pages 399–408. Springer, 2007.

[41] Andrew Howard, Maja J Matarić, and Gaurav S Sukhatme. Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. In *Distributed Autonomous Robotic Systems 5*, pages 299–308. Springer, 2002.

[42] R. Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Trans. Automat. Contr.*, 51(3):401–420, Mar 2006.

[43] 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.

[44] Alon Efrat, Sándor P Fekete, Poornananda R Gaddehosur, Joseph SB Mitchell, Valentin Polishchuk, and Jukka Suomela. Improved approximation algorithms for relay placement. In *Algorithms-ESA 2008*, pages 356–367. Springer, 2008.

[45] Jonathan L Bredin, Erik D Demaine, Mohammad Taghi Hajiaghayi, and Daniela Rus. Deploying sensor networks with guaranteed fault tolerance. *IEEE/ACM Transactions on Networking (TON)*, 18(1):216–228, 2010.

[46] Subhash Suri, Elias Vicari, and Peter Widmayer. Simple robots with minimal sensing: From local visibility to global geometry. *The International Journal of Robotics Research*, 27(9):1055–1067, 2008.

[47] Sayaka Kamei, Hirotsugu Kakugawa, Stéphane Devismes, and Sébastien Tixeuil. A self-stabilizing 3-approximation for the maximum leaf spanning tree problem in arbitrary networks. *Journal of Combinatorial Optimization*, 25(3):430–459, 2013.

[48] 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, 2014.

[49] Nirupama Bulusu, Deborah Estrin, Lewis Girod, and John Heidemann. Scalable coordination for wireless sensor networks: self-configuring localization systems. In *International Symposium on Communication Theory and Applications (ISCTA 2001), Ambleside, UK*, 2001.

[50] Fredrik Gustafsson and Fredrik Gunnarsson. Mobile positioning using wireless networks: possibilities and fundamental limitations based on available wireless network measurements. *Signal Processing Magazine, IEEE*, 22(4):41–53, 2005.

[51] Neal Patwari, Joshua N Ash, Spyros Kyperountas, Alfred O Hero III, Randolph L Moses, and Neiyer S Correal. Locating the nodes: cooperative localization in wireless sensor networks. *Signal Processing Magazine, IEEE*, 22(4):54–69, 2005.

[52] Errol Lynn Lloyd. On triangulations of a set of points in the plane. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 228–240. IEEE, 1977.

[53] Christoph Baur and Sándor P Fekete. Approximation of geometric dispersion problems. *Algorithmica*, 30(3):451–470, 2001.