On Distributed Cooperative Mobile Robotics: Decomposition of Basic Problems and Study of a Self-stabilizing Circle Formation Algorithm

By Samia SOUISSI

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in the partial fulfillment of the
degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

September, 2004

Master thesis

On Distributed Cooperative Mobile Robotics: Decomposition of Basic Problems and Study of a Self-stabilizing Circle Formation Algorithm

By Samia SOUISSI (210207)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the
degree of
Master of Information Science

Written under the direction of
Professor Takuya Katayama

and approved by
Professor Koichiro Ochimizu
Professor Takuya Katayama
Associate Professor Xavier Défago

August, 2004 (Submitted)

Abstract

Enabling mobile robots to work in cooperating teams holds great promises as an efficient and reliable way to solving tasks autonomously. However, addressing the coordination and control of autonomous mobile robots within a team remains a difficult task. Many people have addressed this issue by studying how a complex global behavior can emerge from the interactions of many robots exhibiting a simple local behavior. This approach, called behavior-based, can provide us with an interesting insight on these issues, but it also gives the wrong impression that they are solved. This is however very far from reality, since these heuristics can provide no assurance that a given problem will actually be solved, let alone any proof of correctness.

In contrast, we look at the problem from a computational standpoint, in the sense that we try to determine the local behavior of robots, given a desired global behavior. In particular, our work focuses on basic recurring problems of cooperation.

The main contributions of this dissertation are as follows. First, we outline a specification framework to define basic problems for cooperative autonomous mobile robots. The framework consists of four generic properties that can be combined to define various different problems, including many surveyed in the literature. We see this as a necessary step toward a better understanding of the problems and their relationships. Second, we take a closer look at a specific coordination problem whereby robots must coordinate themselves to form a circle. More specifically, we study the convergence of a self-stabilizing circle formation algorithm using computer simulation. This leads us to propose a simpler algorithm, also self-stabilizing, that has a faster convergence.

Key words : Cooperative Mobile Robots, Basic Recurring Problems, Circle Formation, Self-Stabilizing Algorithm, Simulations Results

....to my parents, my brothers and my sisters for their everlasting love.

Acknowledgments

This work has been supported by the Japanese government scholarship (MONBUKAGAKUSHO).

I first of all wish to express my sincere gratitude to my principal advisor Professor Takuya Katayama for his support, kind supervision and encouragements during my work.

I also want to extend my gratitude and appreciation to my advisor Associate Professor Xavier Défago for his kind guidance, support and for his helpful discussions and valuable suggestions which kept me on the right track.

I am very grateful to Dr. Adel Cherif who recommended for me JAIST for his advices and kind assistance before and after my entering to JAIST.

I gratefully recognize all of my colleagues in the laboratory and the secretary officer Misato Morita for their continuous help and sympathy. In particular, I am grateful to Rami Yared and Péter Urbán for their insightful comments to my work.

Very special thanks to my dear friend Ahlem Ben Hassine, with whom I very much enjoyed living and discovering Japan.

I am very grateful to my friend Nebil Achour for being an excellent brother to me , amazingly helpful and always trying to create Tunisian atmosphere in Japan and his wife Caroline for her hospitality and lovely acquaintance.

I would like also to thank Dr. Yasser Kotb for all his advices and support during my first days at JAIST.

I am grateful to my friend Dr. Shafique Ansari and his wife Dr. Zubaida Ansari for their hospitality and their nice friendship. Similarly, I wish to thank my friend Mohamed Mustapha Azim and his wife Maha for hosting me several times.

I also want to devote my thanks to my previous professors and friends in Tunisia.

Last and by no means least, I'm forever grateful and in dept to my dear parents, my brothers and their wives, my sisters and their husbands, my nephews, my nieces and all my relatives for all the love, support and encouragements.

Contents

List of Figures

List of Tables

Chapter 1

Context and Motivation

The more cooperative the group, the greater is the fitness for survival which extends to all of its members. (Ashley Montague).

1.1 Motivation

The continuous advances in technology have facilitated the use of large number of robots in order to carry out a large variety of cooperative tasks. In fact, recently the interest has shifted from the design and deployment of few, rather complex and expensive robots towards the design and use of a large number of robots, which are simple, relatively inexpensive, but capable together of performing rather complex tasks. Several reasons motivate this shift, including reduced costs, faster computation, fault tolerance capabilities, the possibility of expendability of the system and the reusability of the robots in different applications. Hence, the motivation for using distributed teams of multiple robots.

The main motivation behind this work is to address the large number of tasks arising from real-life which are too dangerous or difficult for humans. For instance, travelling in the space, in the depth of the oceans, inside volcanoes, into burning buildings and others. The participation of distributed teams of robots in such tasks may evade human risks.

One of the most challenging and difficult aspect in cooperating mobile robots is the control and coordination of robots' motion. This aspect still remains a difficult task. This is a likely consequence of the fact that basic problems of coordination between robots has not been clearly identified and expressed in a rigorous way. Moreover, there is a lack of algorithmic solutions to these problems.

1.2 Approach

We approach the problem of cooperating mobile robots from a computational standpoint, in the sense that we try to determine the local behavior of robots, given a desired global behavior. Suppose for instance, that we want to gather robots at some location on the plane. Then, how should we program such a team so its members can cooperate to gather at the same location?

Among other things, it is essential to properly identify and specify the problems that are central to the field. To draw a parallel, problems such as Leader Election and Consensus are central to parallel and distributed systems. In contrast, currently no agreement has been reached on the exact definition of fundamental problems in cooperative mobile robotics. This is a likely consequence of the fact that problems such as gathering[1] or flocking,[2] are notably expressed in a way that depends on details of the system model, or restrict their implementation to specific solutions. For instance, the definition of the flocking problem, as proposed by Prencipe [19], requires the existence of a single leader robot to determine the path that the group must follow. Hence, fully symmetrical or fault-tolerant solutions of the problem are unfortunately ruled out by their definition. In addition, recurring problems in cooperative mobile robotics have not been defined in a consistent way. This makes it particularly difficult to study their relationships and develop algorithmic solutions to these problems.

1.3 Contribution

The major contributions of this work are:

1. First, we study the convergence of a self-stabilizing circle formation algorithm using computer simulation (Chapter 3, Chapter 4).

2. Second, we propose a novel and arguably more simple algorithm for circle formation problem, which is self-stabilizing. The algorithm deterministically forms a circle and converges toward a configuration wherein all robots are located evenly on the boundary of the circle (Chapter 5).

3. Third, we propose a consistent specification framework, whereby several fundamental problems in cooperative mobile robotics can be expressed. More specifically, we define

[1]Gathering: all robots must eventually be at the same location.
[2]Flocking: all robots must move in such a way that their respective locations always forms a given shape.

two safety properties and two liveness properties, which can be combined in several ways to define different problems. We discuss some problems presented informally in the literature, and express their specification in terms of the four proposed properties (Chapter 6).

1.4 Related work

1.4.1 Multi-robot Motion Coordination

A vast amount of researches have been conducted in the field of cooperative mobile robotics (see [5] for a survey). One of the topics with particular interest was multi-robot motion coordination. There are several approaches to multi-robot motion coordination and control reported in the literature including centralized and decentralized ones.

Totally decentralized ones are in general behavior-based [17, 4]. In behavior-based approaches (also known as artificial intelligence approaches) some behavior are prescribed for each agent and the final control is derived by weighting the importance for each behavior. With this methodology, the goal is to predict what complex global behavior can emerge from the interaction of many agents exhibiting a very simple local one. We call it as a ”bottom-up” approach. In a behavior-based approach, algorithms are designed using mainly heuristics. The main problem with this approach is the lack of formal proofs of correctness and guarantees of completion and stability.

Recently the complexity of coordinating multi-robot teams and the desire to deal with real world applications such as Mars’s ground preparation [20], multi-robot mapping and surveillance [15], cooperative search and rescue missions, cooperating autonomous vehicles [3], etc. have motivated algorithmic contributions or computational approaches, in which we find our interest. With this approach, the question can be as follows: given a desired global behavior, what local control behavior shall we give to robots so that they can coordinate their actions. Typically, we start from the application requirements, we decompose into sub-problems and then we try to find algorithmic solutions that fulfill these requirements . This approach (that we call a ”top-down” approach) is based on algorithms, which enables proof and guarantee results for controller design.

Unfortunately, only few studies address the problem from a computational standpoint (e.g., [10, 22]), which means that much remains to be done to develop its theoretical foundations. We aim at improving the current situation by providing a consistent specification framework for problems involving mobility among cooperative autonomous robots. We think that this will contribute to a better understanding of the problems, and pave the ground for further developments. We believe that the field will reach its maturity only once its theoretical foundations have been clearly established; similar to the progress of conventional distributed systems over the last three decades or so.

1.4.2 Survey of Fundamental Problems

In this section, we survey the most recurring problems found in the literature on cooperative mobile robotics. For each problem, we give a brief description and present important known results. Whenever possible, we try to illustrate the problem with a concrete example.

Gathering

With the gathering problem, the robots, initially located at arbitrary positions, are required to gather in a not predetermined point. This problem has been studied extensively in the literature, among which only few addresses the problem from a computational standpoint.

Notable exceptions are studied by Suzuki and Yamashita [22] where, along with the definition of their model (see Sect. 2.1), they propose an algorithm to solve the gathering problem deterministically, in the case where robots have unlimited visibility. Ando et al. [1] propose an algorithm to address the gathering problem in systems wherein robots have limited visibility. Their algorithm converges toward a solution to the problem, but it does not solve it deterministically.

Prencipe [19] study similar issues in his CORDA model. Among other things, one feature the robots must have in order to solve this problem, is the ability to detect multiplicity. The gathering in the case of limited visibility is also discussed by Flocchini et al.[11], and by Cielibak and Prencipe [6]. In the limited visibility setting, the proposed algorithm requires that the robots agree on the direction and orientation of both x and y axis.

Shape formation

The shape formation problem, also known as geometric pattern formation problem, is a generalization of the gathering problem mentioned earlier. It is defined as follows: given a group of n robots $\{r_1, r_2, \dots, r_n\}$ with distinct positions and located arbitrarily on the plane, the robots are required to form a specific geometric shape, where the shape is a set of points (given by

Cartesian coordinates) in the plane. The target shape is known beforehand by all the robots in the system, and it can be for instance a point, a circle, a regular polygon, or some other arbitrary pattern. However, the location of the shape, as well as its size and orientation are not specified.

This problem has been investigated extensively in the cooperative robotics literature, with many ad hoc solutions being proposed. From a computational point of view, this problem has only been studied by a few authors.

Suzuki and Yamashita [22] studied the formation of geometric patterns in the plane. They propose an non-oblivious algorithm for gathering, as well as circle formation. They show that asymmetric patterns cannot be formed as they consider that robots are anonymous.

In the same model, Défago and Konagaya [7] propose a self-stabilizing algorithm for the circle formation problem. With that algorithm, robots deterministically make a circle, albeit not uniformly. The algorithm converges asymptotically toward a solution whereby robots are uniformly distributed along the circle boundary.

Flocchini et al. [9] discuss the problem of arbitrary pattern formation, of which they give an informal definition. They show several important results about this problem, depending on what common knowledge the robots are assumed to share about the coordinate system. The authors give a more formal definition of the problem in their later work [11].

Flocking

Flocking is the problem where robots are required to move in formation like a flock of birds. More rigorously, given set of n robots $\{r_1, r_2, \dots, r_n\}$, the robots are required to keep a given shape while moving.

In the literature, the flocking problem is generally expressed in a ”leader-followers” model [19, 12, 13]. One robot (the leader) is chased by the other robots (the followers). The motion of the leader is not constrained by the problem. In contrast, the followers must follow the leader in such a way that the relative positions of the robots always form a given shape.

To the best of our knowledge, the flocking problem has been studied computationally only in the CORDA model [19, 12, 13]. The authors presented an oblivious algorithm that allow the robots to keep formations that are symmetric with respect to the direction of movement of the leader. Gervasi and Prencipe [13] simulate their algorithm and present interesting results.

Solutions to the flocking problem are useful primitives for larger tasks. For instance box pushing or cooperative manipulation, where robots can be asked to move heavy loads.

Motion planning

Motion planning problem refers to the computational process of moving from one place to another in the presence of obstacles. In other words, motion planning must be performed taking into consideration other robots and the global environment; this multiple-robot path planning is an intrinsically geometric problem in configuration space-time. It is also known as multi-robot path planning. Motion planning has a historical importance in the literature. A noteworthy distributed approach is the one of Yeung and Bekey [23], where each robot initially attempts a straight-line path to the goal; if an interfering obstacle is seen, then the robot will scan the visible vertices of the obstacle and move towards the closest one. Dynamically varying priorities are given to each robot to resolve path intersection conflicts. Conflicting robots can either negotiate among themselves or allow a global blackboard manager to perform this function. Some recent works have addressed some non traditional planning problems. For example, Hert and Lumelsky [14] propose an algorithm for path planning in tethered robots, and Ota et al [18] consider the problem of moving while grasping large objects.

1.5 Structure

This thesis is organized in seven chapters (including this one) and two appendices.

- Chapter 2 presents the system model, some basic definitions and describes an existing distributed algorithm for forming a circle by a set of mobile robots.
- Chapter 3 describes the implementation of the algorithm presented in chapter 2.
- Chapter 4 evaluates the algorithm presented in chapter 2.
- Chapter 5 proposes a new distributed circle formation algorithm for mobile robots.
- Chapter 6 outlines a specification framework for defining recurrent coordination problems for cooperative mobile robotics.
- Chapter 7 concludes and discusses future work.

Chapter 2

System Model and Definitions

This chapter presents first a short review of the system model and basic definitions used in this thesis and then describes a self-stabilizing distributed algorithm proposed by Défago and Konagaya [7], whereby a group of mobile robots eventually form a uniform circle.

2.1 System Model

The system model we consider is defined by Suzuki and Yamashita [22] in which a collection of autonomous mobile robots evolve asynchronously in two-dimensional space. They interact indirectly through their actions on the environment. The robots are anonymous in the sense that they are unable to uniquely identify themselves, neither with a unique identification number nor with some external distinctive mark (e.g., color, flag). They share no common sense of direction, unit distance, or location of the origin of their coordinate system.

Each robot is modelled as a mobile processor with infinite memory and a sensor to detect the instantaneous position of all robots. Each robot has its own local $x - y$ coordinate system (origin, orientation, distance) and has no knowledge neither of the local coordinate system of the other robots, nor of a global coordinate system.

It is assumed that the robots are mathematical points never collide, and that two or more robots may simultaneously occupy the same physical location. Robots are oblivious in the sense that they are unable to remember any past actions or observations. Besides that, all the robots execute the same deterministic algorithm. In addition, there is no central control and no direct communication between them.

Time is represented as an infinite sequence of time instants during which each robots can be either *active* or *inactive*. Each time instant during which a robot becomes active, the robot observes its environment, computes a new position, and moves toward that position. The activation of robots is unpredictable and unknown to robots, with the guarantees that (1) every robot becomes active at infinitely many time instants, and (2) at least one robot is active during each time instant.

Depending on the context, the model can be further restricted by assuming limited visibility, or that robots are oblivious (i.e., keep no memory of past actions).

2.2 Basic Definitions

2.2.1 Smallest Enclosing Circle

The Smallest Enclosing Circle also called Minimal Enclosing Circle, is the problem of finding the smallest circle that completely contains a set of points. More formally, given a set S of n points in the plane, find the circle C of smallest radius with the property that all points in S are contained in C or its boundary.

The Smallest Enclosing Circle is defined by two opposite points or at last three different points. It can be computed in $O(n)$ time by a simple algorithm but with big constant or in $O(n \log n)$ (e.g. Skyum [21]).

2.2.2 Voronoi Diagram

Let S a set of n sites on the plane. For each site *p* of S, the Voronoi cell $V(p)$ of p is the set of points that are closer to *p* than to other sites of S. The Voronoi diagram $V(S)$ is the space partition introduced by Voronoi cells (see Figure 2.1).

Formally, for any point p of the plane let $close(p)$ be the set of sites that realize the closest distance between p and the sites in S, i.e., $s \in close(p)$ if $dist(s,p) \leq dist(t,p)$ for all $t \in S$.

The complexity of computing Voronoi diagram is $O(n)$. Voronoi diagrams have a variety of uses such as nearest neighbor search, facility location, motion planning, etc. More details about Voronoi diagrams, their properties and their use are discussed by Aurenhammer [2].

2.3 Défago and Konagaya Algorithm

The algorithm proposed by Défago and Konagaya [7] solves the problem of circle formation in two steps, each of which is solved by a different algorithm. The first algorithm takes a config-

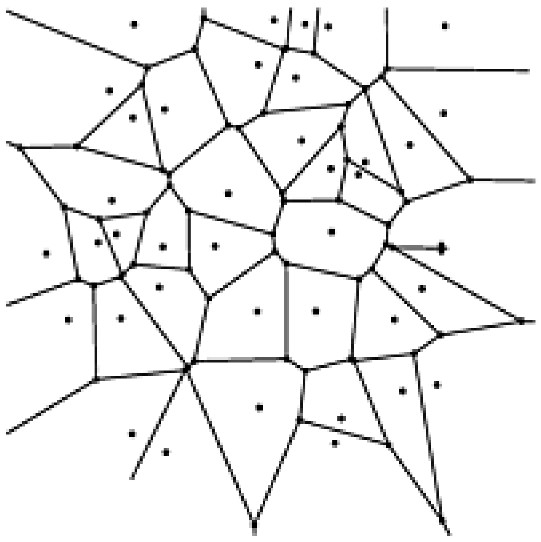


Figure 2.1. A Voronoi diagram.

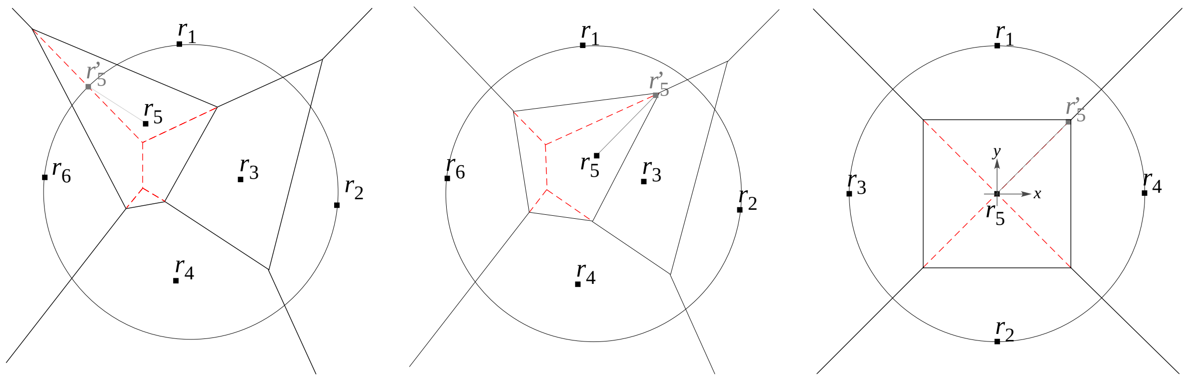
uration where in the robots are spread arbitrarily on the plane, deterministically arranges them to form a non degenerate circle (*circle formation*). The second algorithm, takes a configuration where the robots are already located on the boundary, eventually converges toward a situation wherein all robots are uniformly distributed on the circumference of the circle (*uniform transformation*). The system model considered is defined by Suzuki and Yamashita [22] as described in Section 2.1.

Each algorithm consists of a deterministic function φ that is executed by every robot r_i each time it becomes active. φ_{circle} is the algorithm that solves the circle formation problem and $\varphi_{uniform}$ is the algorithm that solves the uniform transformation problem.

The arguments of φ consist of the current position of the robot (p_i), and a multiset of points (P) containing the observed position of all robots at the corresponding time instant (all positions are of course expressed in terms of the local coordinate system of r_i). The value returned by φ is the new position for r_i.

Restrictions on movements In order to solve the given problems, the authors impose some restrictions on the movements of robots in addition to those inherent to the system model. Doing so ensures that (1) no two robots occupy the same position simultaneously (Restr. 1), and that (2) the smallest circle enclosing all robots remains invariant (Restr. 2–4).

Restriction 1. *A robot always moves toward a point that is inside its Voronoi cell.*



(a) The smallest enclosing circle is reachable.

(b) The smallest enclosing circle is unreachable.

(c) Symmetry is broken by using the local coordinate system.

Figure 2.2. Illustration of Algorithm 3, as executed by one robot (in each case, r_5 moves toward r'_5).

Restriction 2. *No robot ever moves beyond the boundary of the smallest circle enclosing all robots.*

Restriction 3. *All robots located on the boundary of the smallest enclosing circle remain on that boundary.*

Restriction 4. *Robots located on the circumference of the smallest enclosing circle do not move unless there are at least three such robots with distinct positions.*

2.3.1 Algorithm1: Circle Formation

The algorithm presented in this section solves the circle formation problem. It takes a configuration in which all robots have distinct positions and regardless of the activation schedule, eventually brings the system toward a configuration in which all robots are located on the boundary of the circle.

The principle of the algorithm is as follows: as mentioned earlier, the smallest enclosing circle is kept as an invariant. So, all robots are made to move toward the boundary of this circle. In other words, robots that are already on the boundary do not move, and robots that are in the interior of the circle are made to move toward the boundary of the circle.

A robot located in the interior of the circle can find itself in either one of three types of situations. First, as illustrated for robot r_5 on Figure 2.2(a), the simplest situation occurs when the circle intersects the Voronoi cell of the robot. In this case, the robot selects a point on the intersection of the circle and the Voronoi cell, and moves toward it (r'_5 on Fig. 2.2(a)).

The second situation arises when the Voronoi cell of the robot and the smallest enclosing circle do not intersect (Fig. 2.2(b)). In this case, the robot selects a point in its Voronoi cell, such it is the nearest point to the boundary of the circle (or farthest its center).

The third situation arises when, due to symmetry, there exist several such points (Fig. 2.2(c)). In this case, all solutions being the same, one is selected arbitrarily. This is done, for instance, by selecting the solution with the highest x-coordinate (and then y-coordinate) according to the local coordinate system of the robot.

Algorithm 1 Formation of an (arbitrary) circle (code executed by robot r_i)

```
function $\varphi_{circle}(P, p_i)$
 1: if $p_i \in \mathrm{SEC}(P)$ then {$r_i$ already on the boundary.}
 2:     stay still.
 3: else if $\mathrm{Vcell}_{p_i}(P) \cap \mathrm{SEC}(P) \neq \emptyset$ then
 4:     {c.f. Fig. 2.2(a)}
 5:     target := $\mathrm{Vcell}_{p_i}(P) \cap \mathrm{SEC}(P) \cap \mathrm{Voronoi}(P - \{p_i\})$
 6:     move toward $target$. (e.g., $r'_5$ in Fig. 2.2(a))
 7: else {Voronoi cell of $p_i$ inside circle}
 8:     {c.f. Fig. 2.2(b)}
 9:     compute points in $\mathrm{Vcell}_{p_i}(P)$ closest to $\mathrm{SEC}(P)$.
10:     if exactly one candidate exists then
11:         move toward that point. (e.g., $r'_5$ in Fig. 2.2(b))
12:     else {Several candidates exist}
13:         {c.f. Fig. 2.2(c)}
14:         select candidate with greatest $x$-coordinate, and then $y$-coordinate (i.e., first in lexical order).
15:         move toward that point. (e.g., $r'_5$ in Fig. 2.2(c))
16:     end if
17: end if
```

2.3.2 Algorithm2: Uniform Transformation

The algorithm presented in this section solves the uniform transformation problem. It takes a configuration in which all robots are located on the circumference of a circle and regardless of the activation schedule, eventually converges the system toward a configuration in which all robots are spread evenly on the boundary of the circle.

As illustrated on Figure 2.3, whenever a robot r_i becomes active, it considers its two direct

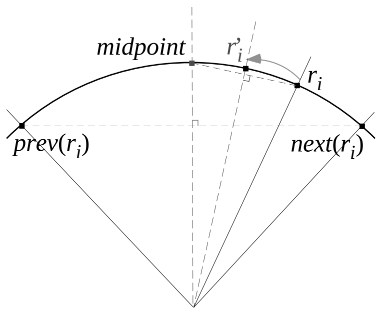


Figure 2.3. Robot r_i moves halfway toward the midpoint (Algo. 2).

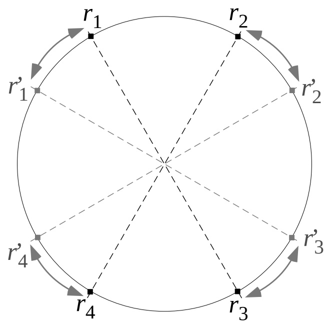


Figure 2.4. Moving to the midpoint can result in oscillations.

neighbors $prev(r_i)$ and $next(r_i)$, and computes the midpoint between them, $midpoint$. Then, r_i moves halfway toward $midpoint$, as permitted by the restrictions on the movement of robots.

The reason for moving halfway toward the midpoint rather than toward the midpoint itself is to prevent situations such as the one illustrated in Figure 2.4, where the system oscillates endlessly between two different configurations if robots are perfectly synchronized. The system would get stuck into an infinite cycle, and hence be unable to progress toward an acceptable solution.

Algorithm 2 Convergence toward a uniform configuration

```
function $\varphi_{uniform}(P, p_i)$
1: {Assumes all robots are on the boundary of $\mathrm{SEC}(P)$.}
2: $prev(p_i)$ := direct neighbor of $p_i$ counterclockwise.
3: $next(p_i)$ := direct neighbor of $p_i$ clockwise.
4: $midpoint$ := midpoint of arc $prev(p_i)$, $p_i$, $next(p_i)$.
5: $target$ := midpoint of arc $midpoint, p_i$.
6: move toward $target$
```

The algorithm is memoryless (oblivious) in the sense that the next position of a robot is determined based only on the current positions of the other robots independently on the previous history of the system.

In their SYm model, Suzuki and Yamashita [22] have observed that any oblivious algorithm developed in their model has the desirable property of *self-stabilization*. Self-stabilization is the property of a system which, started in an arbitrary state, always converges toward a desired behavior [8]. In particular, a self-stabilizing system is able to tolerate any number of transient faults and the state immediately after the occurrence of an error can be regarded as initial state.

The algorithm proposed by Défago and Konagaya [7] solves the problem of circle formation in the SYm model, where robots are oblivious. Although they have proven the correctness of the algorithm, they did not study the actual rate at which the convergence occurs. Among our work, we have implemented and simulated their algorithm. In this chapter, we describe the mechanisms of the algorithm. In Chapters 3 and 4, we respectively present the actual implementation of the algorithm and the results of the simulations.

Chapter 3

Implementation of the Circle Formation Algorithm

In this chapter, we describe the implementation of the two algorithms; circle formation and uniform transformation presented in the previous chapter.

3.1 Overview Implementation

Figure 3.1 describes the programs developed for the implementation of both algorithms; *circle formation* and *uniform transformation*.

- *Program*1 *(Initial configuration)*: generates a random configuration of the initial positions of the robots (Appendix A.1). The implementation of the algorithm is done using the programming language *Ruby*.

- *Program*2 *(Activation schedule)*: generates a set of different activation schedules based on the probability of activation of robots (Appendix A.2).

- *Program*3 *(Circle formation)*: is the implementation of the circle formation algorithm. The program takes as input an initial configuration file generated by Program1 and an activation schedule file generated by Program2 and generates a history file of the positions of the robots at each activation step.

  The implementation of the algorithm is done using the programming language *C++* and the geometric library LEDA Library of Efficient Data algorithms (Appendix A.3).

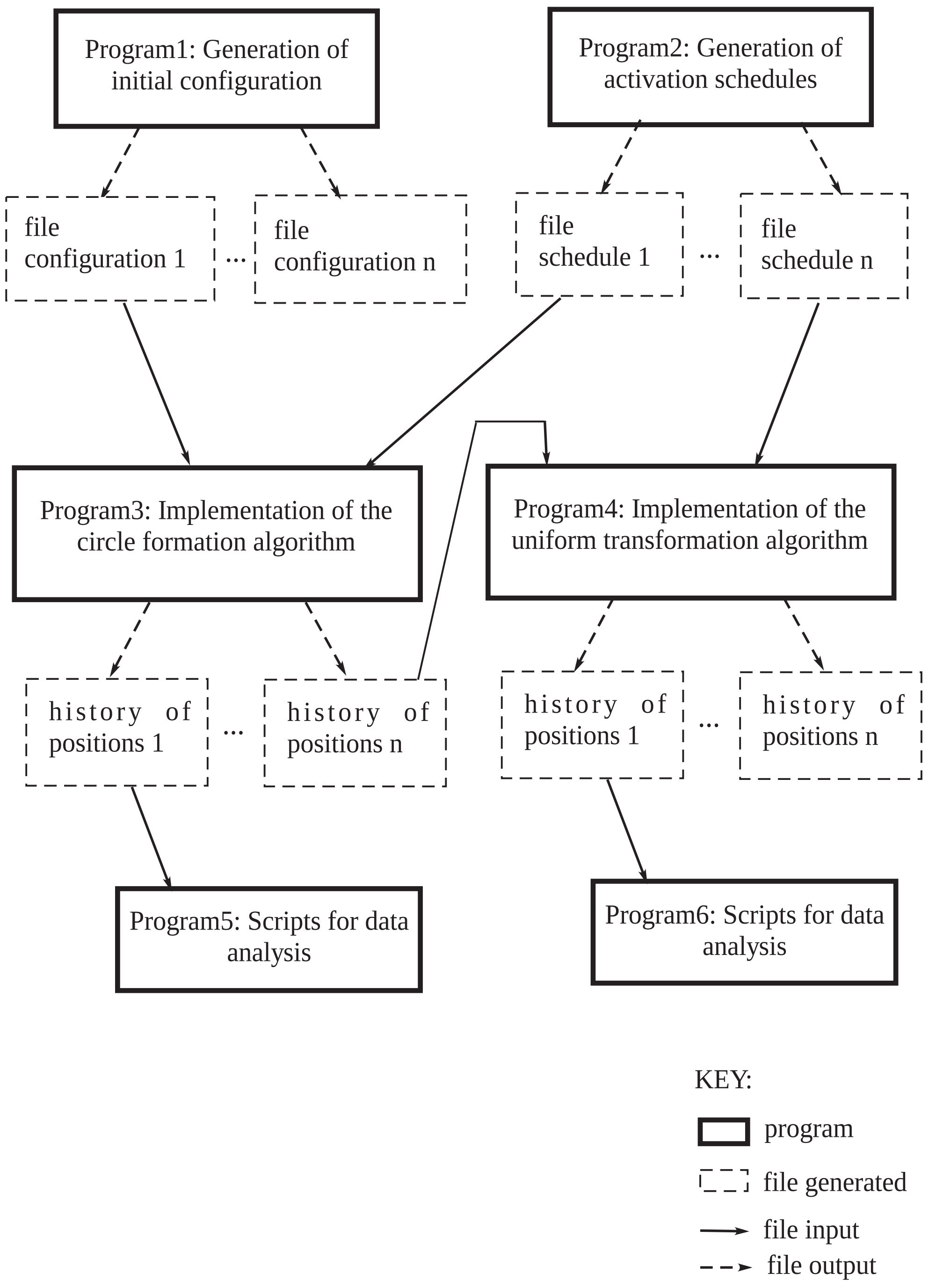

Program1: Generation of initial configuration
Program2: Generation of activation schedules
file configuration 1
...
file configuration n
file schedule 1
...
file schedule n
Program3: Implementation of the circle formation algorithm
Program4: Implementation of the uniform transformation algorithm
history of positions 1
...
history of positions n
history of positions 1
...
history of positions n
Program5: Scripts for data analysis
Program6: Scripts for data analysis
KEY:
program
file generated
file input
file output

Figure 3.1. Overview implementation.

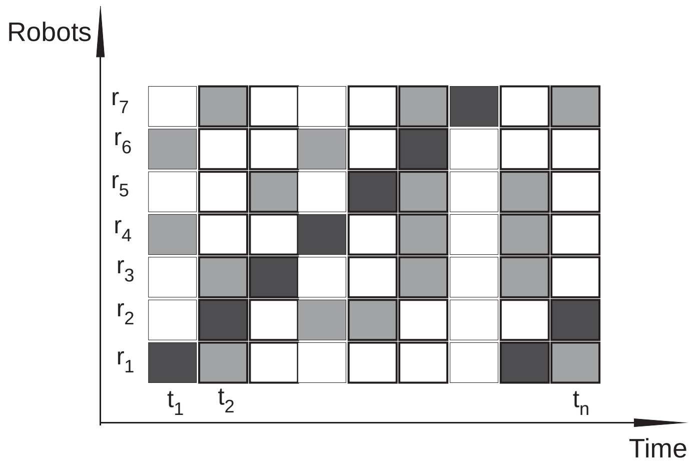


Figure 3.2. An example of activation schedule ($p = 0.5$).

- *Program*4 *(Uniform Transformation)*: is the implementation of the uniform transformation algorithm. The implementation is done on the same manner as for the implementation of the circle formation algorithm (Appendix A.4).
- *Program*5 *and* 6: are program scripts that takes as input the history files generated by program3 and 4 and generates data files to analyze.

3.2 Activation Schedule

As we mentioned in the system model, the activation of robots is determined by an activation schedule which is unpredictable and unknown to robots with the guarantees that:

1. Each robot becomes active at infinitely many instants.
2. At least one robot is active during each time instant.

The idea of the implementation of the activation schedule is basically, the robots are activated in a *round-robin* manner which means that at each time instant, only one robot is active.

In addition, there is a probability p for each robot that it gets activated even though it is not its turn. This probability introduces some simultaneity of activation between robots. Figure 4.1.1 shows an activation schedule with $p = 0.5$. The black color characterizes the robot activated by the round robin and the grey color represents the subset of robots activated simultaneously in addition to the one activated by the round robin.

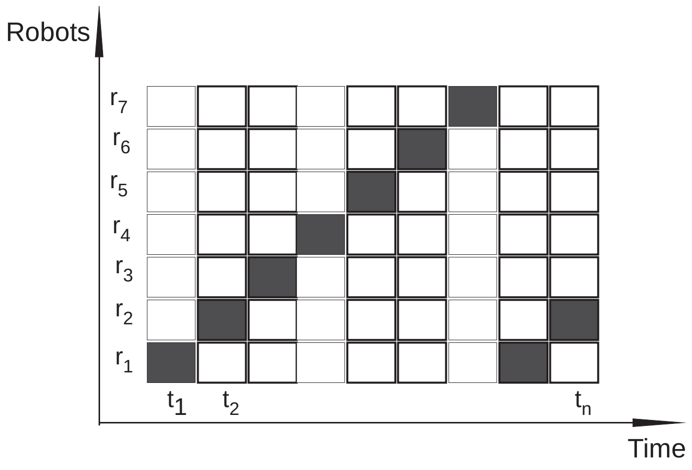


Figure 3.3. Activation schedule ($p = 0$).

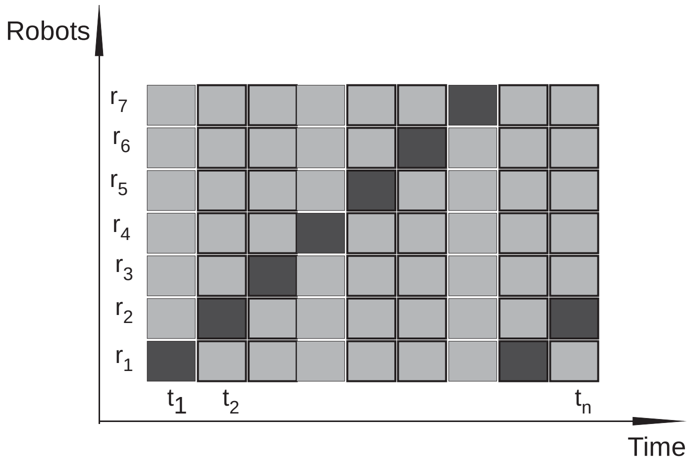


Figure 3.4. Activation schedule ($p = 1$).

There are also two special cases; if $p = 0$, then robots are activated in mutual exclusion and according to a round-robin fashion; if $p = 1$, then all robots are always activated together and we say that they are synchronized. It is important to understand that the activation schedule and its parameters are unknown to the algorithm.

- if $p = 0$, only the robot activated by the round- robin is active; none of the other robots in the team is active (Figure 3.3).
- if $p = 0.5$, at each time instant, in addition to the robot activated in the round robin, a subset of robots are activated; each other robot has probability $p = 0.5$ to be activated (Figure 4.1.1).

It is noteworthy that it is not possible to generate all possible activation schedules programmatically. The method described here generates a smaller subset of the activation schedules that are allowed by the model.

- if $p = 1$, in addition to the robot activated by the round robin, each other robot has a probability equal to 1 to be activated. Thus, all the robots are activated together. (Figure 3.4)

With this implementation, we guarantee that each robot becomes active at infinitely many instants (due to round robin) and at least one robot is active during each time instant (according to the probability of activation). However, the activation schedules generated by our implementation are a strict subset of those allowed by the model in the case when the probability of activation is not equal to 1.

The program code of the implementation of the activation schedule is presented in Appendix A.2.

3.3 Circle Formation Algorithm

The circle formation algorithm is based on the manipulation of several non trivial geometrical notions, including Voronoi diagram and the smallest enclosing circle. In our implementation of the algorithm, in addition to the programming langauge C++, we included the library of computational geometry LEDA (Library of Efficient Data Algorithms) to alleviate the task of programming of the algorithm. Nevertheless, it remains a huge programming work.

One of the most difficult parts to implement is how to identify the Voronoi cell of each robot knowing only its position. One solution that we proposed to this problem, consists on finding the nodes (robots) which are neighbors to the active robot by computing the Delaunay triangulation [1] of the list of the robots' positions of and then determining the adjacent edges to the robot that forms its Voronoi cell.

The adjacent edges of the robot are the edges in Voronoi diagram which are perpendicular to the edges formed by the robot itself and its node neighbors. After identifying the different edges forming the Voronoi cell of the robot, we need to check if these edges intersect with the SEC. If among the edges there are ray, then obviously these rays intersect with the SEC. Therefore, in this case, the Voronoi cell of the robot is not inside the SEC, but intersect with it. If the edges

[1]The Delaunay triangulation is the geometric dual of the Voronoi diagram. If one draws a line between any two points whose Voronoi domains touch, a set of triangles is obtained, known as the Delaunay triangulation.

are all segments and at least one of the segments intersect with the SEC, then the Voronoi cell of the robot intersect with the SEC. In the other case, the Voronoi cell of the robot does not intersect with SEC.

For computing the new position in which an active robot will move, we proposed the following solution; in the case when the Voronoi cell of the robot is inside the SEC, we compute the distance between the current position of the robot and all the sites of the Vornoi diagram which belong to its cell and then we take the maximum distance (maximum point inside the robot's cell). In the case, when the Voronoi cell of the robot intersect with the SEC, we calculate the next position of the robot as follows; we exclude the active robot from the list of robots, we compute a new Voronoi diagram, and then we find the intersection of the SEC with the new Voronoi diagram. However, we will have as a result a lot of points including useless ones, i.e. points not inside the Voronoi cell of the robot. For filtering useful solutions, we need to take points which have minimal distance to the position of the robot.

The complete program code describing the implementation of this algorithm is presented in Appendix A.3. A simplified C++ version of the program code is depicted as follows: First, we illustrate the function $main$, which include the handling of several experiments (runs) and then the function $main\_processing$, which represents the main implementation of the algorithm.

Function main

```
1:  int main (int argc, char$*argv[]$)
2:  // Iteration of the different configurations (samples)
3:  **for** (int $k = 0$;$k < samples$;$k++$) **do**
4:      // Iteration of the different schedules (runs)
5:      **for** (int $run = 0$; $run < n\_runs$; $run++$) **do**
6:          // Call function $read\_file\_configuration$ to get initial positions of the robots
7:          positions = $read\_file\_configuration(file\_config, n\_robs)$;
8:          // Call function $activation\_schedule$ to read file activation schedule
9:          $activation\_schedule(sched)$;
10:         // Write parameters in history file: call function $produce\_headers\_file\_history$
11:         $produce\_headers\_file\_history(hist)$;
12:         // Write initial configuration in history file: call function $produce\_history\_file$
13:         $produce\_history\_file(0, hist)$;
14:         // Call function $main\_processing$ which represents the main implementation of the algorithm circle
            formation
15:         $main\_processing(file\_schedule, file\_history)$;
16:     **end for**
```

```
17: end for
18: return 0;
19: // end main
```

Function $main\_processing$

```
 1: int $main\_processing(char * sched, char * hist)$
 2: // Compute the Smallest Enclosing Circle
 3: SEC = $SMALLEST\_ENCLOSING\_CIRCLE(list\_points)$;
 4: // Call function $VORONOI$ to compute Voronoi diagram
 5: VORONOI($list\_points$,VD);
 6: // Call function $Draw\_Voronoi$ to draw Voronoi diagram
 7: $Draw\_Voronoi$(VD);
 8: // Call function $all\_on\_boundary$ to check if all robots are on the boundary
 9: if ($all\_on\_boundary(table\_points)$) then
10:     cout<<"All on the boundary";
11: else
12:     // Iteration of the different generations in an activation schedule
13:     for ($counter\_g$=0; $counter\_g < n\_gens$;$counter\_g$++) do
14:         // Iteration of the active robots by generation
15:         while (activation [$counter\_g$][$counter\_rob$] && $counter\_rob$ ¡= n$\_robs)$ do
16:             // Check if the active robot is on the boundary
17:             // Call function $test\_boundary$
18:             if ($test\_boundary$ ==1) then
19:                 // If the robot on the boundary do no thing: Just write in history file
20:                 // Call function $produce\_history\_file$
21:                 $produce\_history\_file(counter\_g$, hist);
22:             else {not on the boundary}
23:                 // Compute a new position
24:                 // Call function $node\_neighbors$ to compute the robot's neighbors
25:                 $node\_neighb$ = $node\_neighbors(list\_points, pos\_active)$;
26:                 // Determine the edges forming the Voronoi cell of the active robot
27:                 $list\_adj\_edges$ = $DT.adj\_edges(v)$;
28:                 // Check if the Voronoi cell of the robot intersect or not with the SEC
29:                 // Call function $rays\_on\_cell$ to check if the robot's Voronoi cell contains rays
30:                 $res\_r$ = $rays\_on\_cell(pos\_active)$;
31:                 // if the Voronoi cell contains rays then obviously it intersects with the SEC
32:                 if ($res\_r$ == 1) then
33:                     // Determine a new position for the robot in the case when its Voronoi cell intersects with SEC.
```

```
          // Call the function $Robot\_next\_position$
          $new\_pos$ = $Robot\_next\_position(list\_points, pos\_active)$;
        **else**
          // Check if the Voronoi cell of the robot contains segments that intersect with the SEC
          // Call function $segments\_on\_cell$
          $res\_s$ = $segments\_on\_cell(pos\_active)$;
          **if** ($res\_s$ == 1) **then**
            // Compute a new position for the robot in the case when its Voronoi cell intersect with SEC.
            // Call the function $Robot\_next\_position$
            $new\_pos$ = $Robot\_next\_position(list\_points, pos\_active)$;
          **else** {the Voronoi cell of the robot does not intersect with the SEC}
            // Compute a new position for the robot in the case when its Voronoi cell is inside the SEC.
            // Call the function $Robot\_inside\_next\_position$
            $new\_pos$ = $Robot\_inside\_next\_position(list\_points, pos\_active)$;
          **end if**
        **end if**
        // Update of the new position of the robot.
        $table\_points[id\_active$-1] = $new\_pos$;
        // Write in file history
        // Call function $produce\_history\_file$
        $produce\_history\_file(counter\_g$, hist);
        // Check if all on the boundary
        **if** ($all\_on\_boundary(table\_points)$) **then**
          cout<<" All on the boundary ";
          return 1;
        **end if**
      **end if**
      $counter\_rob$++;
    **end while**
  **end for**
**end if**
return 1;
//End function $main\_processing$
```

3.4 Uniform Transformation Algorithm

The implementation of the uniform transformation algorithm is less complicated than the implementation of the circle formation algorithm. It is based on the computation of the angles. The complete program code describing the implementation of the this algorithm is given in Appendix A.4. A simplified C++ version of the program code is illustrated as follows:

Function main

```
int main (int argc, char$*argv[]$)
// Iteration of the different configurations (samples)
**for** (int $k = 0$;$k < samples$;$k++$) **do**
  // Iteration of the different schedules (runs)
  **for** (int $run = 0$; $run < n\_runs$; $run++$) **do**
    // Get the history of positions of the robots produced by the the algorithm circle formation
    // Call function $read_file_configuration\_all\_on\_boundary$
    $positions\_in\_circle$ = $read_file_configuration\_all\_on\_boundary$("history");
    // Call function $activation\_schedule$ to read file activation schedule
    $activation\_schedule(sched)$;
    // Write parameters in history file: call function $produce\_headers\_file\_history$
    $produce\_headers\_file\_history(hist);$
    // Write initial configuration in history file: call function $produce\_history\_file$
    $produce\_history\_file(0, hist);$
    // Call function $uniform\_transformation$ which represents the main implementation of the algorithm
    uniform transformation
    $uniform\_transformation()$;
  **end for**
**end for**
return 0;
// end main
```

Function $uniform\_transformation$

```
void $uniform\_transformation()$
// Compute the list of angles in which the robots are positioned
// Call function $computation\_list\_angles$
$computation\_list\_angles(list\_loc)$;
// Check if all robots are uniformly spread on the boundary of the circle
Call function $uniform\_spread()$
**if** ($uniform\_spread() == 1$)) **then**
```

```
    cout<<"ll the robots are uniformly spread on the circle";
**else**
    // Iteration of the different generations in an activation schedule
    **for** $(int g = 0; g < n\_gens; g++)$ **do**
        // Iteration of the active robots by generation
        **while** $(activation[g][rob \&\& rob <= n\_robs)$ **do**
            // Check if the angle previous and next to the robot are equal
            // Call function $previous\_next\_equal$
            **if** $(previous\_next\_equal(id\_active) == 1)$ **then**
                // Do no thing
            **else** {previous and next angles not equal}
                // Compute a new position
                //Compute mid point of previous and next angle and its mid point
                // Call function $half\_way\_mid\_point\_previous\_next$
                $next\_pos$ = $half\_way\_mid\_point\_previous\_next$($id\_active$, $str\_count$);
                // Update of the new position of the robot.
                $table\_loc$[$id\_active$-1] = $next\_pos$;
            **end if**
            // Write in file history
            // Call function $produce\_history\_file$
            $produce\_history\_file$((g, "history"));
            // Call function $computation\_list\_angles$ to compute the list of the new angles in which robots are located
            // $computation\_list\_angles(list\_loc$);
            // Check if all robots are uniformly spread on the circle
            // Call function $uniform\_spread()$
            **if** $(uniform\_spread())$ **then**
                cout<<" All the robots are uniformly spread on the circle ";
                return ;
            **end if**
            rob++;
        **end while**
    **end for**
**end if**
return ;
//End function $uniform\_transformation$
```

Chapter 4

Simulation of the Circle Formation Algorithm

This chapter presents an evaluation of both algorithms *circle formation* and *uniform transformation* by computer simulations using different metrics and different parameters of the system. The results show the convergence of these algorithms and how the convergence is affected by the parameters of the system. The objective of this analysis is to predict the performance of the algorithms when it is implemented on physical robots.

4.1 Experimental Setup

4.1.1 Terminology

In the experiments, we used the following terminologies (see Figure 4.1):

- Generation: It is the sub-population (or population) of robots activated at a time instant t according to an activation schedule (gen$_0$, gen$_1$, ...gen$_n$ in Figure 4.1).
- Activation step: It corresponds to the activation of a single robot r_i in a generation (a colored box in a column in Figure 4.1).
- Round Robin: The robots are activated in turn and only a single robot is activated in each generation (see black boxes in Figure 4.1).

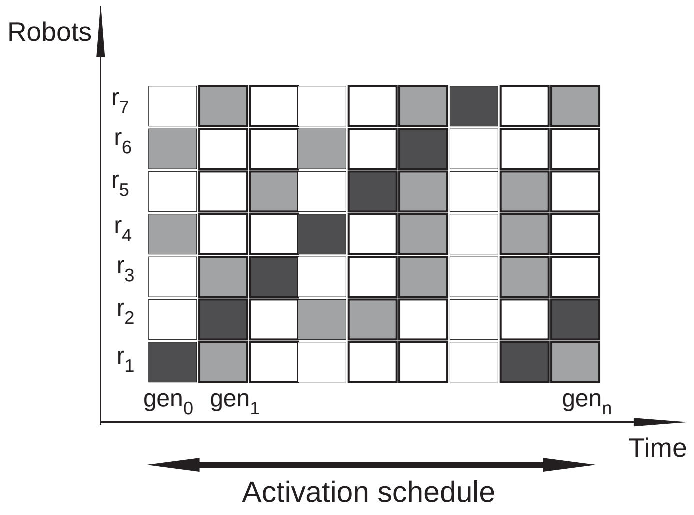


Figure 4.1. Terminology.

4.1.2 Parameters

To analyze the convergence of both algorithms, we have simulated them under various parameters. These parameters are summarized in the Table 4.1.

The system is composed of n robots. These robots are activated according to a probability of activation p in addition to the robots activated by the round-robin. n_gens stand for the number of generations of their activations and n_runs correspond to the number of runs performed for the same initial configuration by different activation schedules. There are two parameters related to the distance in the circle formation problem; $r_i(t)$ denotes the radius of a robot at time instant t,i.e. the distance from its current position to the center of the smallest enclosing circle at time t and R denotes the radius of the smallest enclosing circle. There are two parameters related to the angles when the robots are on the boundary of the circle; θ_{min} represents the minimum angular distance between any two consecutive robots and θ_{max} represents the maximum angular distance between any two consecutive robots at each time instant .

Table 4.1. Simulation parameters

Parameters	Description
n	the number of robots in the team
p	the probability of activation of other robots in addition to the one activated by the round robin
$r_i(t)$	the position of r_i according to the center of the SEC (radius of r_i)
R	the radius of the SEC
θ_{min}	the minimum angular distance between any two consecutive robots at each time instant
θ_{max}	the maximum angular distance between any two consecutive robots at each time instant
n_gens	the number of generations in an activation schedule
n_runs	the number of runs performed

4.2 Circle Formation: Experimental Performance

Termination criteria: The termination criteria of a team of robots P to form a circle is defined as:

$$\forall r_i \in P(t), \frac{r_i(t)}{R} = 1 \tag{4.1}$$

In order to examine the behavior of the *circle formation* algorithm, we carried out the following metrics:

4.2.1 Trace of the Relative Positions of the Farthest Robot From the Boundary

The purpose of this metric is to measure the number of generations required by the algorithm to reach the termination state; state where all the robots are on the boundary of the circle.

Figure 4.2 shows the relative positions of the farthest robot from the boundary of the circle. The team is composed of a population of 16 robots and the probability of their activation is equal to 0.25. As can be seen, the quotient $\frac{r(t)}{R}$ is increasing or stable after each generation until it is equals to 1. The algorithm terminates in finite time after 10 generations of activation of the robots. Each robot is activated almost 1 or 2 times ($\frac{24}{16}$).

4.2.2 Percentage of the Population on Boundary

The goal of this metric is to show the percentage of the robots located on the boundary after each generation.

Figure 4.3 presents the percentage of the population on boundary after each activation step. As can be seen, the percentage of the robots on the boundary is increasing or stable with the generations. The algorithm terminates when the population on the boundary is equal to 100%

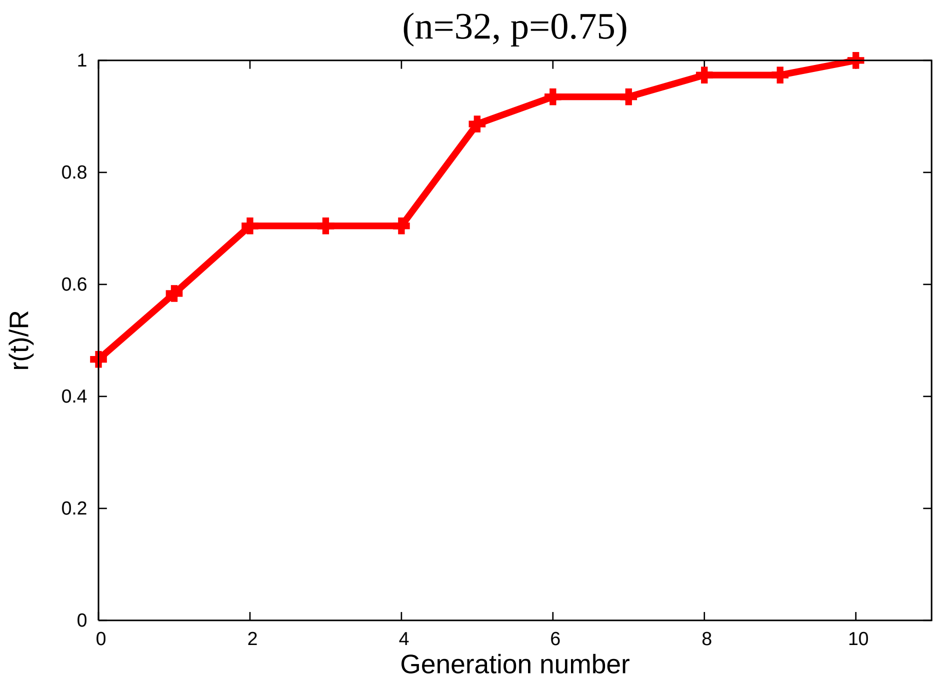


Figure 4.2. Trace of the relative positions of the farthest robot from the boundary.

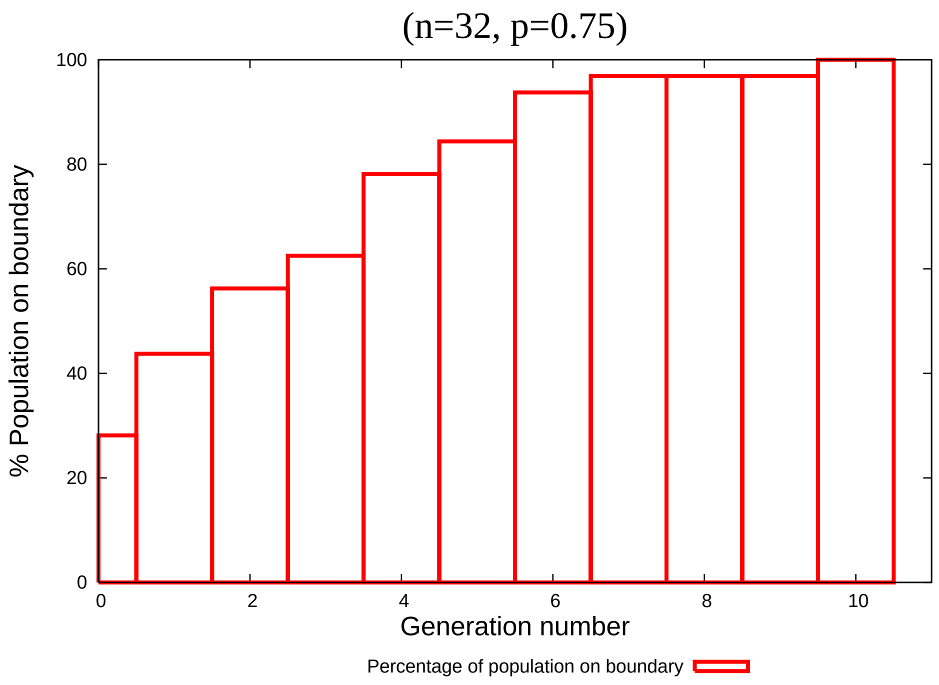


Figure 4.3. Percentage of the population on boundary of the circle.

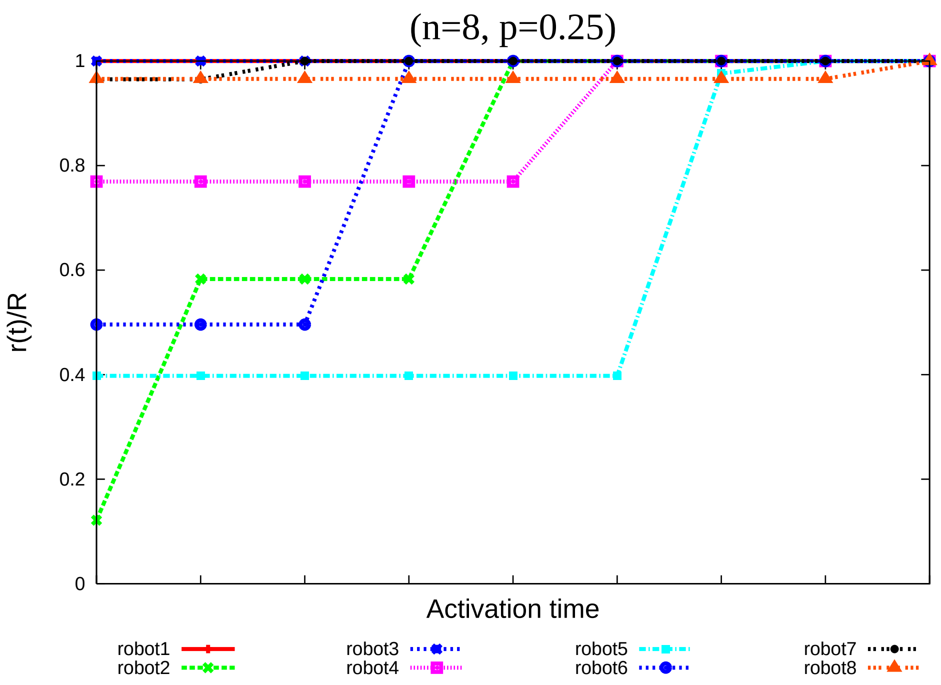


Figure 4.4. Distribution of robots according to the boundary.

according to the criteria mentioned before. The termination time is same as in Figure 4.2 since the parameters applied are same.

4.2.3 Distribution and Progress of Robots Towards the Boundary of the Circle

The purpose of this experiment is to show that no robots move to any position backward closer to the center of the circle.

Figure 4.4 reveal the progress of each robot in the team toward the boundary. The team is composed of 8 robots and their probability of activation is equal to 0.25. Initially, the two robots labelled 1 and 3 are already on the boundary and remains there. The 6 other curves represents the progress and termination time of the robots labelled $2, 4, 6, 7$ and 8.

We can notice that the quotient $\frac{r(t)}{R}$ is always increasing or stable for all robots after each activation step. This proves experimentally that robots move always to positions closer to the boundary and no robot moves backward to any position closer to the center of the circle.

4.2.4 Termination for Different Populations

In order to be able to compare the termination rate for different populations of robots, in Figure 4.5 we show the termination rate expressed in terms of number of generations for different

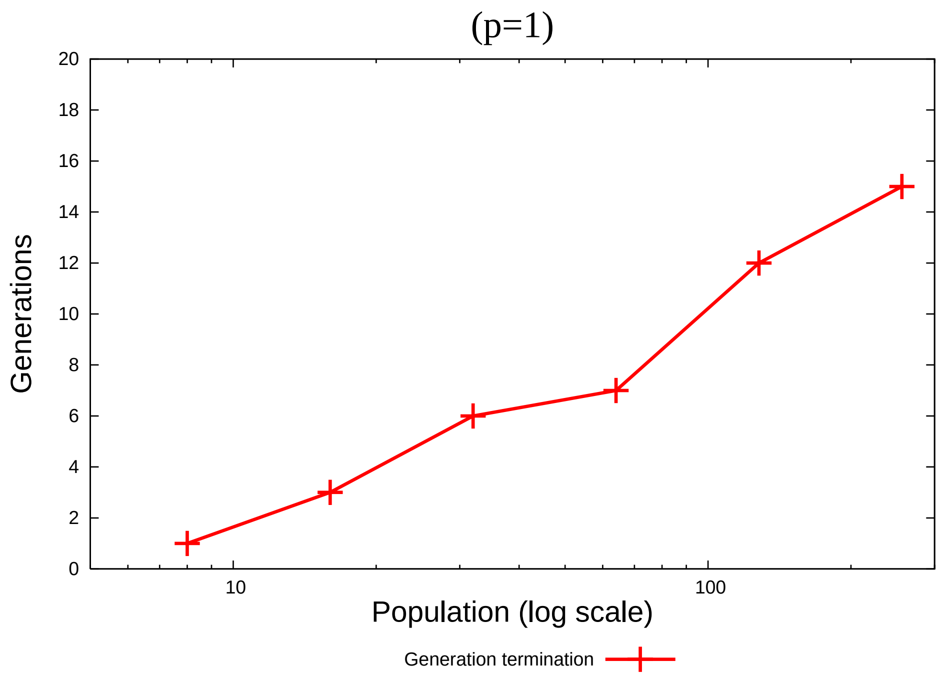


Figure 4.5. Termination for different populations $(p = 1)$.

populations of robots. In each team, we consider that all the robots are activated simultaneously $(p = 1)$. In this case, we can give the exact termination rate since the activation of the robots is not sensitive to the randomness of the activation schedule because all robots are activated together.

In Figure 4.5, the $x - axis$ represents the number of robots in each team expressed in log scale and the $y - axis$ represents the number of generations. As can be seen in the figure, the number of generations is proportional to the $\log$ of robots. Hence, the termination rate is in the order of $\log n$.

4.3 Uniform Transformation: Experimental Performance

To examine the behavior of the *uniform transformation* algorithm, we performed the three metrics as given below based on the following criteria :

Convergence criteria: The convergence of the algorithm *uniform transformation* is defined by:

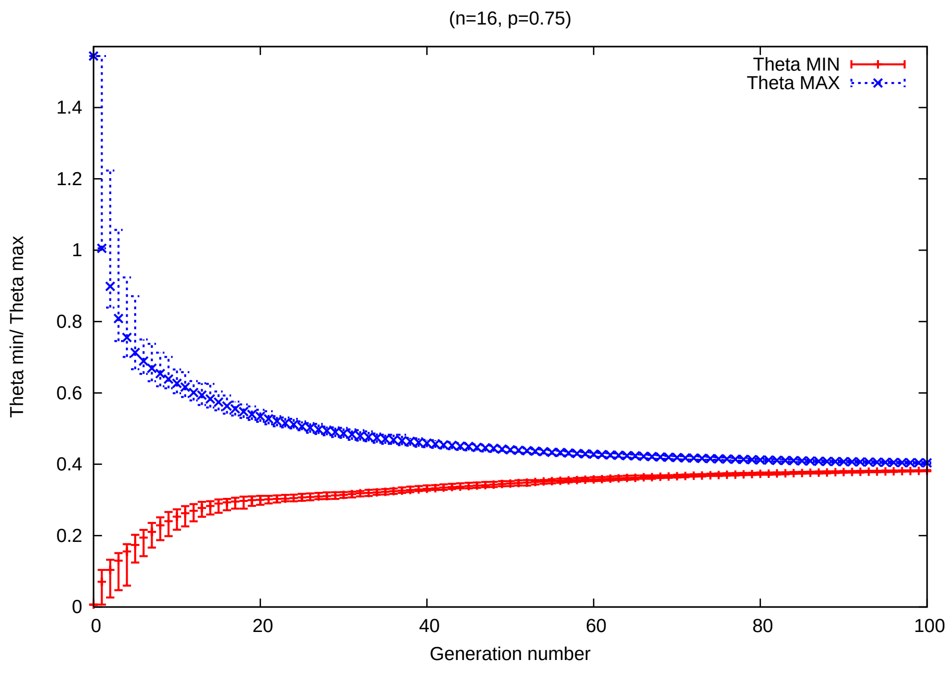


Figure 4.6. θ_{max} **is monotonically decreasing and** θ_{min} **is monotonically increasing** $(n = 16, p = 0.75)$**.**

$$\theta_{min} = \theta_{max} = \frac{2\pi}{n} \tag{4.2}$$

4.3.1 Algorithm Convergence

The purpose of this metric is to show the convergence of the algorithm. The algorithm converges when the difference between θ_{max} and θ_{min} is closer to zero. It has been proved in [7] that θ_{max} is monotonically decreasing and θ_{min} is monotonically increasing and as depicted in Figure 4.6, we can see similar results by simulation.

4.3.2 Convergence for Different Values of Probability

The goal of this metric is to observe the effect of the probability of activation of robots on the convergence of the algorithm. In Figure 4.7, we plotted the median of 50 runs for different probability of activation of robots. The 4 curves represents respectively the convergence for the probability equal to 0, 0.25, 0.5 and 1. The obtained results show that, when the probability of

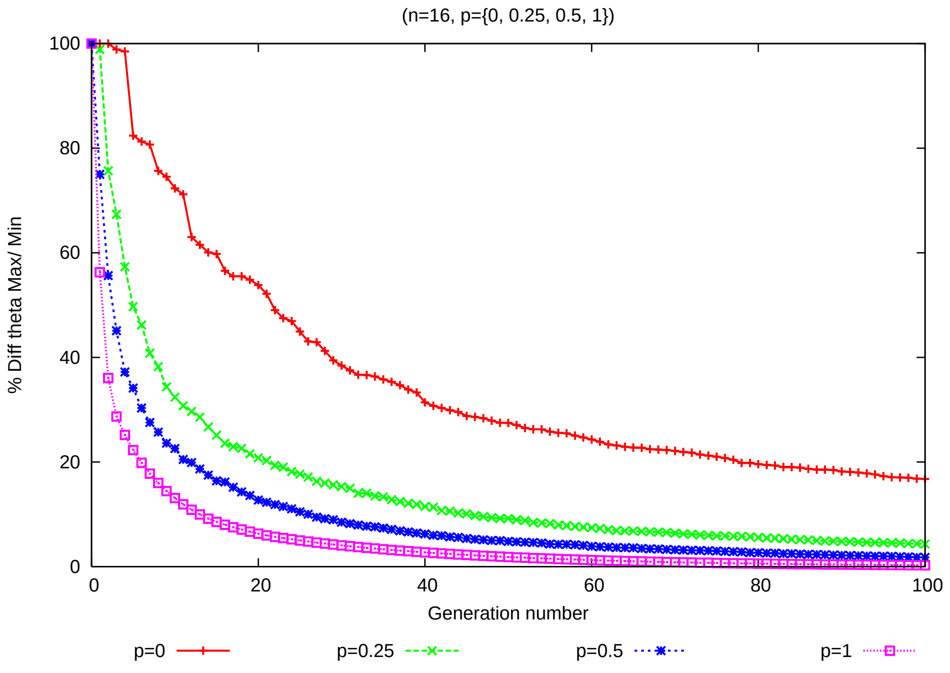


Figure 4.7. The rapidity of the convergence increases with the probability of activation.

activation of robots is *high*, the convergence is *faster* and when the probability of activation of robots is *small*, the convergence is *slower*. For instance, when $p = 0$, after 100 activation steps, the algorithm does not converge. Whereas, when $p = 1$ after almost 60 activation steps, the algorithm converge.

4.3.3 Convergence for Different Populations of Robots

The purpose of this metric is to evaluate the impact of the number of robots on the convergence of the algorithm.

In Figure 4.8, we plotted the convergence of different populations $(16, 32, 64, 128)$ using the same probability of activation (0.25). The results in Figure 4.8 show that if the population of robots is *big* for instance 128, the convergence toward the uniform configuration is *slower* and if the population is *small* for instance 16, the convergence is *faster*.

It is noteworthy that Figure 4.9 present an interesting phenomena; the curve which represents the 128 robots is below the curve which represents the 64 robots in the first 120 generations. Afterward the curve which characterizes the 64 robots is below and the curve which characterizes the 128 robots is above. The behavior in the first 120 generations is justified by the random generation of the initial positions of the robots. Initially, with the population of 128 robots the

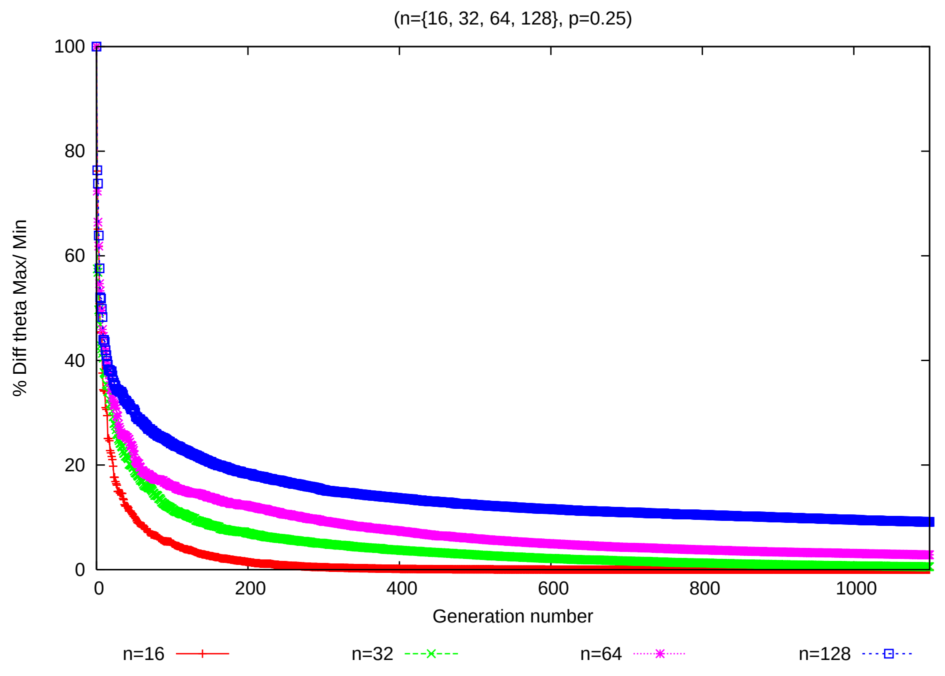


Figure 4.8. The rapidity of convergence decreases with the number of robots forming the team.

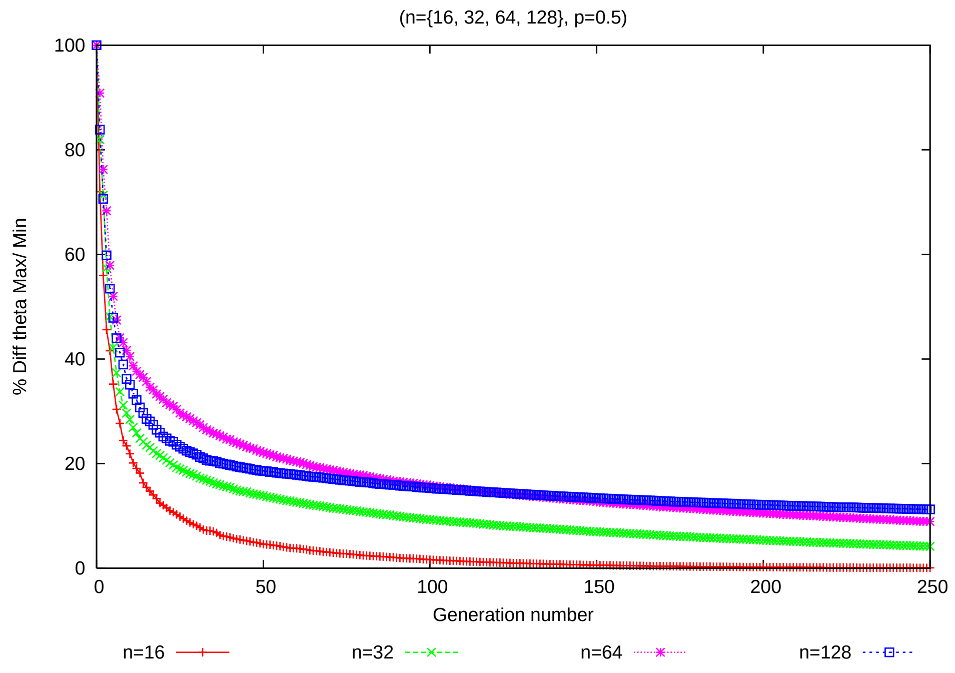


Figure 4.9. The unpredictability of the activation schedule has impact on the convergence.

difference between θ_{max} and θ_{min} is greater than the difference between θ_{max} and θ_{min} for the population of 64 robots. The normal behavior of the system is restored after 120 generations. Therefore, we can claim that the initial configuration (positions) of the robots has also an impact on the speed of convergence of the algorithm.

Chapter 5

New Circle Formation Algorithm

In this chapter, we propose a new algorithm for forming a uniform circle in the plane by a team of mobile robots. This algorithm actually builds upon the work of Défago and Konagaya [7] and function under the hypothesis that robots (1) are *oblivious* in the sense that they are unable to recall past actions and observations, (2) share *no common sense of direction*, and (3) have *no direct communication* only through observing each others position.

The algorithm is self-stabilizing since starting from any arbitrary state always converges towards the formation of a circle.

5.1 Algorithm Description

The proposed algorithm is based on the Suzuki and Yamashita model described in Chapter 2. It takes an arbitrary configuration in which all robots have distinct positions and regardless of their activation, eventually brings the system toward a configuration in which all robots are uniformly located on the boundary of the circle.

The algorithm consists of a deterministic function φ that is executed by every robot r_i each time it becomes active. The arguments to φ consist of the current position of the robot, and a multiset of points containing the observed position of all robots at the corresponding time instant. All positions are of course expressed in terms of the local coordinate system of r_i. The value returned by φ is the new position for r_i.

The algorithm relies on the fact that the environment observed by all robots is the same and makes sure that the Smallest Circle Enclosing all robots remains invariant and use it as a common reference. The invariance is ensured by imposing the following restrictions on the movements of robots:

Restriction 5. *No robot ever moves beyond the boundary of the smallest circle enclosing all robots.*

Restriction 6. *All robots located on the boundary of the smallest enclosing circle remain on that boundary.*

Restriction 7. *Robots located on the circumference of the smallest enclosing circle do not move unless there are at least three such robots with distinct positions.*

5.2 Algorithm: Formation of a Uniform Circle

Notations:

- $p_i(t)$: denotes the position of a robot r_i at time t according to some global $x-y$ coordinate system.
- $P(t) = \{p_i(t) | 1 \leq i \leq n\}$: denotes the multiset of the positions of all robots at time t.
- $L(P)$: represents a circular list of all robots, thus uniquely defining previous and next neighbors for each robot.
- $\prec$: denotes *previous* neighbor.

The principle of the algorithm is described as follows. Each time a robot becomes active, it executes the following steps of the algorithm:

- Computes the Smallest Enclosing Circle (SEC) of the observed positions of all robots in the team. The SEC is *unique* and *common* between robots. The robots must keep the SEC *invariant*. This is satisfied by the restrictions on the movements of robots described in Section 5.1.
- Computes a circular list, describing for each robot its *previous* most neighbor and its *next* most neighbor. The ordering of the robots is based on the angles they are located according to some global $x - y$ coordinate system in counterclockwise orientation of the circle. In the case, when some robots are collinear (their angles are equal), the ordering is done based on their positions according to the center of the Smallest Enclosing Circle. The robot with small radius is considered as *previous* to the one with bigger radius.

Algorithm 3 Formation of a uniform circle (code executed by robot r_i)

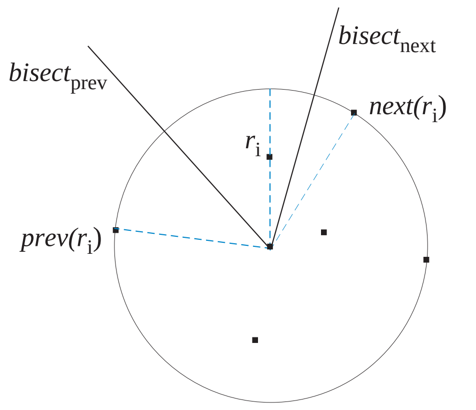
```
function $\varphi_{circle}(P, p_i)$
 1: Compute SEC(P)
 2: Orientation of SEC(P):= counterclockwise
 3: ORIGIN := $o$ of SEC
 4: $\theta_j$ := the angle $\angle ox, op_i$
 5: if $p_i$== $o$ then {robot in the center of SEC}
 6:     Move to any free position
 7: else
 8:     $p_j \prec p_i \Leftrightarrow (\theta_j \langle \theta_i) \vee ((\theta_j = \theta_i) \wedge \| \overrightarrow{op_j} \| \langle \| \overrightarrow{op_i} \|)$
 9:     Compute L(P) according to $\prec$
10:     Sort L(P) according to $\prec$
11:     if ($p_j \prec p_i \prec p_k$) and ($\theta_j = \theta_i = \theta_k$) then {collinear with prev and next neighbors}
12:         stays still
13:     else
14:         prev($p_i$):= previous direct most neighbor of $p_i$ in L(P)
15:         next($p_i$):= next direct most neighbor of $p_i$ in L(P)
16:         bisect $_{prev}$($p_i$):= bisector of the angle $\angle op_i, oprev(p_i)$
17:         bisect $_{next}$($p_i$):= bisector of the angle $\angle op_i, onext(p_i)$
18:         bisect ($p_i$):= bisector of the angle formed by bisect $_{prev}$($p_i$) and bisect $_{next}$($p_i$)
19:         target ($p_i$) := bisect($p_i$) $\cap$ SEC(P)
20:         Move toward target($p_i$)
21:     end if
22: end if
```

- Considers its two direct neighbors and computes first the *bisector* of the angle formed by the center of the circle, the robot itself, the center of the circle and its previous neighbor *bisect*$_{prev}$ and then the bisector of the angle formed by the center of the circle, the robot itself, the center of the circle and its next neighbor *bisect*$_{next}$ (See Figure 5.1(b)). The movement of a robot is delimited by *bisect*$_{prev}$ and *bisect*$_{next}$.

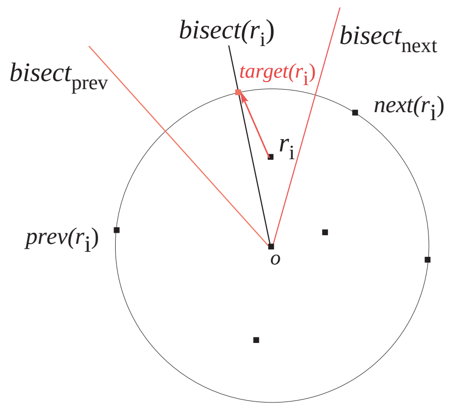
- Computes its *target* position on the boundary of the circle which corresponds to the intersection of the SEC with the *bisector* of the two bisectors *bisect*$_{prev}$ and *bisect*$_{next}$ (*target(*r_i*)* in Figure 5.1(b)).

- Moves gradually toward its target position (*target(*r_i*)*). The target position of the robot is always inside its reachable area (area delimited by *bisect*$_{prev}$ and *bisect*$_{next}$). Therefore, the target position of the robot is always reachable. The robot can progress to its target by moving along the path defined by its current position and the target on the boundary.

The algorithm presents two very simple and special cases. The first one is when there is a robot which is located in the center of the circle. In this case, the robot cannot identify its



(a) The movement of a robot r_i is delimited by its previous and next bisectors.



(b) Robot r_i computes its target position.

Figure 5.1. Robot r_i computes and moves to a new target position.

previous and next neighbor. One solution to this problem is to let this robot moves initially to any unoccupied position. Afterward, the problem is reduced to the general case.

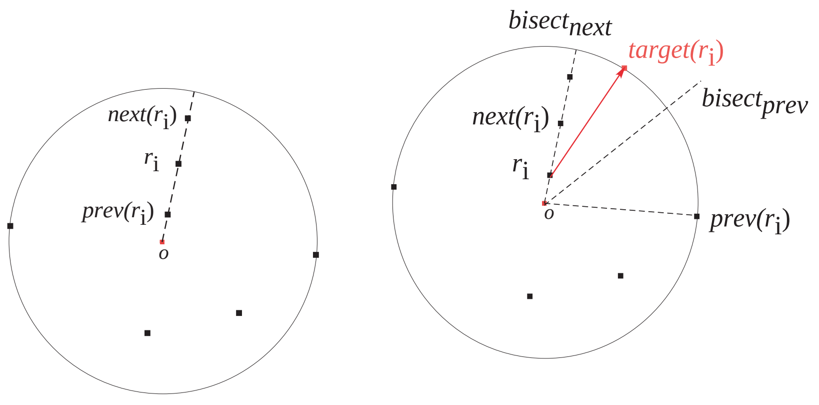
The second case is when the active robot is collinear with its direct neighbors *previous* and *next* as illustrated in Figure 5.2(a). In this case, it cannot compute *bisect*$_{prev}$ and *bisect*$_{next}$. Nevertheless, this problem is reduced to the general case from the time when one of the two robots in the extreme of the line moves. See Figure 5.2(c).

5.3 Correctness

Lemma 1. Under this algorithm, no two robots ever end up at the same position.

Proof. The current position of a robot r_i and its computed target target(r_i) is always a point in the area delimited by the two bisectors $bisect_{next}$ and $bisect_{prev}$ of the robot (points on the borders are not included). Let's denote this area by $area(r_i)$. Then, all possible movements of the robot will be as well in $area(r_i)$.

Assume by contradiction that robot r_i and r_j can move to the same position. Then, there must be an overlap of $area(r_i)$ and $area(r_j)$. However, if we take 3 tangent sectors with common origin and we draw the bisector in each angle. Then, the area delimited by the first two consecutive bisectors is tangent to the area delimited by the second two consecutive bisectors. Therefore, there is no overlap between the two areas. It follows that $area(r_i)$ and $area(r_j)$ cannot overlap



(a) Robot r_i is collinear with its previous and next neighbors.

(b) r_i moves toward a target position.

(c) The situation is reduced to the general case after r_i moves.

Figure 5.2. The case when robots are collinear is reduced to the general case when any of the extreme robots move.

which is in contradiction with the assumption. Thus, no two robots can ever move to the same position ■

Theorem 1. Under restrictions 2-4, the smallest enclosing circle of all robots is invariant.

Proof. The theorem has been proved in [7]■

Lemma 2. No robot moves backward to a position closer to the center of the circle.

Proof. Each time a robot r_i becomes active, it will find itself in one of the three cases:

- r_i is collinear with its previous and next neighbors: In this case, r_i will not move, it will stay still, then, obviously r_i does not move toward the center.
- r_i is in the center of the SEC, in this case, r_i will move away from the center of the circle. Then obviously, it does not move toward the center.
- r_i is not collinear with its previous and next neighbors. In this case, r_i will move to a position which is farther away from its current position and closer to the target position which is on boundary. Therefore, the radius of the robot is monotonically increasing. So, robot r_i actually progress away from the center.

The robot r_i is unable to move backward in any of the three cases, thus proving the lemma ■

Lemma 3. The target position (on boundary) of any robot r_i is always reachable

Proof. The target position of any robot r_i and its current position are in the same sector $sect(r_i)$ (the sector delimited by its previous and next bisectors). In addition, $sect(r_i)$ is private to r_i, then none of the other robots can block r_i to move toward its target position. Consequently, the target of r_i is always reachable ■

Lemma 4. All robots located in the interior of the $SEC(P)$ reach its boundary after a finite number of activation steps.

Proof. By Lemma 3, the target position of any robot r_i is reachable. By Lemma 2 no robot ever moves backward. Therefore, any robot r_i can reach $SEC(P)$ in finite number of steps. Since there is a finite number of robots in P, there exists some finite time after which all robots are located on the boundary of the smallest enclosing circle $SEC(P)$ ■

Lemma 5. Under this algorithm, for all $r_i, r_i \in SEC(P)$ is stable

Proof. Let us denote by C_{circle}, the set of all configurations with robots on the boundary. The proof is by induction, which consists on showing that if C_i is a configuration in C_{circle}, then by the algorithm, $C_{i+1} \in C_{circle}$ ■

Lemma 6. Any configuration in which robots are uniformly spread on the circle is stable under the algorithm.

Proof. The proof is by induction on the set of targets *T*. Assume a configuration C_i in which all robots are equally spaced on the circle, then C$_i \in$ *T*. When robots are in this configuration, each time a robot becomes active, it computes the same target position. Therefore, C$_{i+1} \in$ *T*. Hence, C$_{i+1}$ = C_i. Consequently, given a configuration in which robots are uniform spread on the circle, they stay still. This proving the lemma ■

Theorem 2. The algorithm deterministically solves the problem of circle formation.

Proof. By Lemma 4 and Lemma 3, there is a time after which all robots are located on the smallest enclosing circle, and from lemma 5, for each robot r_i which is on boundary it will stay on boundary. Thus, the algorithm solves the problem of circle formation in finite number of steps ■

Theorem 3. The algorithm converges towards a configuration wherein all robots are located evenly on the boundary of the circle.

Proof. By lemma 6, all the configurations wherein all robots are uniformly spread over the circle are stable. The proof consists in showing that the angular distance between any two consecutive robots is identical for all robots, that is $\frac{2\pi}{n}$ ■

Algorithm convergence The algorithm solves the circle formation problem in *finite time* and *converges* toward a uniform distribution of the robots over the circle. The advantages of the algorithm are its simplicity and its low cost computation.

Chapter 6

Decomposition of Basic Problems for Cooperative Mobile Robots

In this chapter, our objective is to outline a specification framework to define recurring basic problems for cooperative autonomous mobile robots. First, we propose four properties into which these problems can be decomposed. The advantage of this approach is that problems are expressed in a more consistent framework, thus making it easier to understand problems and study their relationships. Second, we give a list of eight problems and describe them according to a combination of the four properties. Note that, the definitions of the problems are model-independent.

6.1 Basic Properties

Among the four properties, the first two properties can be characterized as liveness properties, whereas the other two are safety properties. Roughly speaking, a *safety* property of a system is one which *specifies what the system should never do* (i.e. ”bad things never happen”). Conversely, a *liveness* property is one which *specifies that the system must eventually do something* (i.e. ”good things eventually happen”) [16].

6.1.1 Liveness properties

Property 1 (Forming shape). *Given some shape X, there is a time after which the shape defined by the location of all robots is homomorphic to X.*

Informally, the desired shape is formed eventually. In other words, after a finite number of moves, the final positions of the robots coincide with the points forming the input shape. Forming shape is a liveness property since while testing for violation of this property requires looking at infinite executions.

Property 2 (Reaching goal). *Given some point p, there is a time after which the center of gravity of all robots is co-located with p.*

Reaching goal property is the ability of a team of robots to reach a predetermined goal location. At the end of their computation process, the robots reach to the destination point at the same time or at different time. In the case when the robots are asked to attain a goal and preserving a shape at that destination, the property goal is satisfied when the center of gravity of the shape forming the robots superpose with the destination point.

6.1.2 Safety properties

Property 3 (Keeping shape). *Given some shape X, there is a time since which the shape defined by the location of all robots is always homomorphic to X.*

Roughly speaking, since the robots form the shape, they are required to keep the same shape in movement. Keeping shape is a safety property since the violation of this property can be observed in finite time (time in which the shape is modified).

Property 4 (Following path). *Given some oriented path $p(u)$, the following two predicates are satisfied.*

1. *At any time, the center of gravity of the location of all robots is a point on the path $p(u)$.*
2. *The center of gravity of the location of all robots progresses monotonically on the path $p(u)$.*

It is the ability of the robots to follow some given path while moving without getting out of that path. This property is satisfied when the center of gravity of the new positions of robots in each movement is a point in the specified path and the coordinates of the center of gravity increases according to the oriented path.

Table 6.1. Decomposition of common problems into the basic properties

	forming shape *(liveness)*	**keeping shape** *(safety)*	**following path** *(safety)*	**reaching goal** *(liveness)*
Shape formation	○			
Flocking		○		
Path-constrained flocking		○	○	
Goal reaching				○
Cooperative handling (“box-pushing”)		○		○
Path-constrained coop. handling		○	○	○
Shape formation w/goal	○			○
Path-constr. shape formation w/goal	○		○	○

6.2 Decomposition of Problems into Basic Properties

Table 6.1 summarizes the decomposition of problems into a combination of the four properties mentioned above. The problems surveyed in Chapter 1 are decomposed into the properties. In addition, several other combinations, that yield meaningful problems, are also presented in the table and discussed in the text.

Shape formation The shape formation problem encompasses the two problems of gathering and arbitrary pattern formation. This problem is simply defined by Property 1 (forming shape). In the literature, gathering and pattern formation are treated differently, probably because the former is easier to solve than the latter. Nevertheless, there is no reason to give a completely different definition, and so we combine these two problems as special cases of shape formation. In particular, the robots must eventually form the desired shape, which can be a point (i.e., gathering problem), a circle, a polygon, etc.

Flocking In the literature, the flocking problem has been defined according to a leader-followers approach [19, 13]. In that model, one of the robots is designated as the leader, and the others are followers. The problem requires that the followers follow the path of the leader in such a way that their relative positions maintains a given shape. However, we believe that this definition unnecessarily restricts the problem. In particular, the definition precludes fully symmetrical algorithms. For this reason, we try to provide an alternate definition which does not impose a leader-follower model, thus leaving this as an implementation issue.

Path-constrained flocking In the path-constrained variant of the flocking problem, the path that the robots must follow is given. This results in two safety properties; keeping the shape and following the path.

Goal reaching In the goal reaching problem, the robots are required to move towards a given destination point and reach in a finite number of moves. Hence, this problem is expressed by the liveness property reaching goal. The goal reaching problem is related to the problem of motion planning, except that there are no restrictions set on the path of the robots.

Cooperative handling (box-pushing) A problem known as ”box-pushing” has been studied in the literature on robotics. In that problem, a pair of small robots are required to collaborate in order to move a large box from one location to another. We can easily express a similar problem by combining flocking and goal reaching. Hence, the cooperative handling problem is defined in terms of one safety property (keeping shape) and one liveness property (reaching goal).

Other problems Several problems can be derived from those discussed above: for instance path-constrained cooperative handling is a combination of the problems path-constrained flocking and goal reaching. Therefore, the problem is defined by the two safety properties presented earlier and the the liveness property reaching goal. The problem of shape formation with goal (i.e., where the robots are asked to form a shape in a specific location known in advance) is also a combination of the shape forming problem and the goal reaching problem. When the property following path is added to this problem, it is called path-constrained shape formation with goal and in which the two liveness properties and the safety property following path must be fulfilled.

Chapter 7

Conclusion and Future Work

We first summarize our contributions and discuss the results of the thesis in Section 7.1. Then, we present our plans for future work.

7.1 Conclusion

First, we have surveyed some of the most important work in the field of distributed cooperative mobile robotics and more specifically, the basic recurring problems of coordination among robots . As a result of our survey, we found that in the literature there was a lack of specification of these problems and that they were expressed in a way that depends on the details of the system model in which they are studied. Hence, the motivation to propose a consistent framework of specification of recurring problems of cooperation between robots.

Second, we focused on a particular coordination problem; the problem of forming a circle by a team of mobile robots. In particular, we studied the convergence of a self-stabilizing circle formation algorithm proposed by Défago and Konagaya [7] using computer simulation. This algorithm solves the problem in two steps by different algorithms. The first algorithm solves the circle formation problem and the second one solves the uniform transformation problem. As a result of simulating the circle formation algorithm, we observed that the percentage of the population of robots on the boundary of the circle is increasing after each generation of robots' activation and the algorithm terminates in finite time. Moreover, we found that the number of generations required for termination (termination rate) is proportional to the log of robots.

In the uniform transformation algorithm, our results show how the convergence of the algorithm is affected by the parameters of the system. For instance, the speed of the convergence increases with the probability of activation of robots, and in contrast decreases with the size of the population. In addition, we noticed that the initial distribution of the robots over the circle has an impact on the speed of convergence of the algorithm.

Third, We proposed a new algorithm for forming a circle by a set of mobile robots. The algorithm solves the problem of uniform circle formation in one phase and by one algorithm. It solves the circle formation problem in *finite time* and it *converges* toward a uniform distribution of the robots over the circle.

The algorithm is simple in principle and it is based on the computation of bisectors of angles. It guarantees that no collision between robots would happen since each robot has its own exclusive zone in which it can move and there is no overlap between their zones. In addition, it guarantees that the target position of a robot is always reachable given that it is always inside the robot preserved zone.

The main advantages of the algorithm are its simplicity and its lower computation cost and we conjecture that it has fast convergence.

Finally, we have proposed a consistent specification framework of basic problems of cooperation expressed in terms of four properties (two safety and two liveness properties) along which these problems can be decomposed. Other combinations of the properties also yield meaningful problems, and variants thereof, that we did not see discussed in the literature yet. The main benefit of our approach is to make the study of inter-problems relationships easier. As we outlined a consistent framework for the definition of problems, it will be easier to generalize future results to classes of problems rather than individual ones.

7.2 Future Work

There is a number of issues that needs to be addressed in our future work:

- In our framework of specification of problems, the work we presented is still in progress, and many interesting research issues remain to be addressed. In particular, we are working on a formal definition of the properties, and hence the problems. We will also need to make sure that our specifications rule out trivial (and useless) solutions to the problems. At last, we will try to extend the framework with additional ”model-dependent” safety properties, such as preventing collisions with obstacles or other robots.

  In the longer run, we will use our framework to determine the minimal system requirements for solving each problem. We will hence investigate the relationship between the capabilities of robots and their ability to solve the different problems. From a practical standpoint, it is important to know what minimal set of functionalities is necessary to solve a given problem, as a way to reduce development and fabrication costs.

- In the new proposed algorithm, we will complete the proofs of correctness, study its convergence using simulations and then implement it on real robots.

- Defining more realistic system model. In our work, we considered the SYm model which is a well established model, however, it is simple because it models robots as mathematical points. We would like to propose a model which closer to reality. For instance, considering the volume of the robot, its direction, different speed for robots, etc.

- Finally, we want to investigate the issue of communication between robots. For instance, robots might need to exchange information on their states (positions, trajectories, orientation, etc.) to construct a complete configuration of the team in order to coordinate their actions and cooperate. The communication can be either *implicit* or *explicit*. We have studied a model with only implicit communication. Among our future work also, we want to study the role of explicit communication for solving similar problems. The question is what kind of communication is suitable for what problem and what kind of communication is effective for certain environment conditions? and how robots must maintain communication?

Bibliography

[1] H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Trans. on Robotics and Automation*, 15(5):818–828, October 1999.

[2] F. Aurenhammer. Voronoi diagrams–a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[3] M. G. Lewis B. R. Bellur and F. L. Templin. An ad-hoc network for teams of autonomous vehicles. *In Proc. of IEEE Symposium on Autonomous Intelligence Networks and Systems*, 2002.

[4] T. Blach and R. C. Arkin. Behavior-based formation control for multi-robot teams. *IEEE Trans. on Robotics and Automation*, 14(6):926–939, December 1998.

[5] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, (4):1–23, 1997.

[6] M. Cieliebak and G. Prencipe. Gathering autonomous mobile robots. In *Proc. 9th Colloquium on Structural Information and Communication Complexity (SIROCCO'02)*, Andros, Greece, June 2002.

[7] X Défago and A. Konagaya. Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In *Proc. 2nd ACM Intl. Workshop on Principles of Mobile Computing (POMC'02)*, pages 97–104, Toulouse, France, October 2002.

[8] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[9] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Hard tasks for weak robots: The role of common knowledge in pattern formation by autonomous mobile robots. In *Proc.*

10th Intl. Symp. on Algorithms and Computation (ISAAC'99), volume 1741 of *LNCS*, pages 93–102, Chennai, India, December 1999. Springer.

[10] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous oblivious robots with limited visibility. In *Proc. 18th Annual Symp. on Theoretical Aspects of Computer Science (STACS 2001)*, volume 2010 of *LNCS*, pages 247–258, Dresden, Germany, February 2001. Springer.

[11] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Pattern formation by autonomous robots without chirality. In *Proc. VIII Intl. Colloquium on Structural Information and Communication Complexity (SIROCCO 2001)*, pages 147–162, Vall de Núnia, Spain, June 2001.

[12] V. Gervasi and G. Prencipe. Flocking by a set of autonomous mobile robots. Technical Report TR-01-24, Dipartimento di Informatica, Università di Pisa, Italy, October 2001.

[13] V. Gervasi and G. Prencipe. Need a fleet? use the force! In *Fun With Algorithms 2 (FUN 2001)*, pages 149–164, Elba, Italy, May 2001.

[14] S. Hert and V. Lumelsky. Moving multiple tethered robots between arbitrary configurations. *In IEEE/RSJ IROS*, pages 280–285, 1995.

[15] K. Konolige, C. Ortiz, R. Vincent, A. Agno, M. Eriksen, B. Limketkai, M. Lewis, L. Briesemeister, E. Ruspini, D. Fox, J. Ko, B. Stewart, and L. Guibas. Centibots: Large-scale robot teams. *In Multi-Robot Systems: From Swarms to Intelligent Autonoma*, 2003.

[16] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, March 1977.

[17] M. J. Matarić. Designing emergent behaviors: From local interactions to collective intelligence. In *Proc. Intl. Conf. on Simulation of Adaptive Behavior*, pages 423–441, 1993.

[18] J. Ota, N. Miyata, T. Arai, E. Yoshida, D. Kurabayashi, and J. Sasaki. Transferring and regrasping a large object by cooperation of multiple mobile robots. *In IEEE/RSJ IROS*, pages 543–548, 1995.

[19] G. Prencipe. CORDA: Distributed coordination of a set of autonomous mobile robots. In *Proc. 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, pages 185–190, Bertinoro, Italy, May 2001.

[20] K. Schreiner. Nasa's jpl nanorover outposts project develops colony of solar-powered nanorovers. *In IEEE DS Online*, 2(3), 2001.

[21] S. Skyum. A simple algorithm for computing the smallest enclosing circle. *Information Processing Letters*, 37(3):121–125, 1991.

[22] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal of Computing*, 28(4):1347–1363, 1999.

[23] D. Yeung and G. Bekey. A decentralized approach to motion planning problem for multiple mobile robots. *In IEEE ICRA*, pages 1779–1784, 1987.

Publications

1. S. Souissi, X.Défago, T.Katayama. ”Convergence of a Uniform Circle Formation Algorithm for Distributed Autonomous Mobile Robots”. *In Proc. Joint Japan-Tunisia Workshop on Computer Systems and Information Technology (JT-CSIT'04), July 2004 Tokyo, Japan*.

2. S. Souissi, X.Défago, T.Katayama. ”Decomposition of Fundamental Problems for Cooperative Autonomous Mobile Systems”. *In Proc. of 2nd International Workshop on Mobile Distributed Computing*, pp. 554-560, Mar. 2004, *IEEE Computer Society Press*.

3. S. Souissi, X.Défago, T.Katayama. ”Specification of Recurrent Problems in Distributed Cooperative Mobile Robotics”. *In Proc. Scientific French-speaking Workshops (JSF'03), November 2003, Tokyo, Japan*.

Appendix A

Source Code

A.1 Program: Generation of Initial Configurations

```
#!/local/pkg/all/bin/ruby
##/*==============================================================*/
 ##This program allow to generate random initial positions
 ## of robots (Initial configuration).
 ##Last update: 16/04/2004....
##/*==============================================================*/

BEGIN {
  NUMBER_OF_RUNS          = 10
  NUMBER_OF_ROBOTS        = [8, 16, 32, 64]
  WIDTH  = 100
  HEIGHT = 100
  NAME_CONFIG  = "configuration"
  ROBOT_ID = 0
  srand
}

######################################################################
class Robot
######################################################################
  attr_accessor :id
  attr_accessor :x_coord
  attr_accessor :y_coord

  def initialize(id, x=rand(WIDTH)*rand , y=rand(HEIGHT)*rand)
    # set id
    @id = id
    # set coordinates
    @x_coord = x
    @y_coord = y
  end
 end  #end class

#*********************** End class Robot *******************/*
#######################################################################
class RobotTeam
#######################################################################
  def initialize(n_robs=0)
    @list = Array.new(n_robs)
    if (n_robs>0)
      @list.each_index {|i|
        @list[i] = Robot.new(i+1)
      }
      @list.sort! { |a,b| a.id <=> b.id }
   end
 end

  def << rob
    if ! @list.include?(rob)
      # do not accept duplicates and keep the list sorted by increasing id
      @list << rob
      @list.sort! {|a,b| a.id <=> b.id}
    else
      p rob
      fail "element already in team"
    end
    self
  end

  def size
    @list.size
  end
def to_array

long =  @list.size

lst_robot = [] long.times {lst_robot << [] }

    @list.each_index{|i|
      id   = @list[i].id
      x = @list[i].x_coord
      y = @list[i].y_coord

      lst_robot[i] = [id, x, y]
    }
    lst_robot
  end

end

#*********************** End class RobotTeam *****************/*
###############################################################
################ Create  configuration
###############################################################

def create_config(n_runs, n_robs)

   dir_path = ("run_"   + String(n_runs)   + \

              "_rob_"  + String(n_robs) )

  if File::exist?(dir_path)
    puts "#{dir_path} skipped (exists)"
    return
  end
  current_dir = Dir::pwd

begin
    Dir::mkdir(dir_path)
    Dir::chdir(dir_path)
```

```
      n_runs.times{|run|

        print "\n ----******A run*****------ \n"

      File::open(NAME_CONFIG+".r"+String(run), "w") {|f_config|

      init_team  = RobotTeam.new(n_robs)

     # print initial positions
          (init_team.to_array()).each() {|rob|
                x= rob[1]
                y= rob.last

            print "\n The result is: \n"
            print "#{x}\t#{y}"
            f_config.print "#{x}\t#{y}"
            f_config.puts
  }
}

} # end run

  rescue
    puts "Error when making #{dir_path}"
    raise
  ensure
    Dir::chdir(current_dir)
  end
end


def main
 print "\n Please wait, currently creating initial configuration !!\n \n"

 NUMBER_OF_ROBOTS.each {|n_robs|
      create_config(NUMBER_OF_RUNS, n_robs)
    }
```

```
end main

####Desired output: n_runs initial configuration ##Each file
configuration.rn  has the following format:

##  x_position \t y_position  (robot that has the id =1)
##x_position \t y_position  (robot that has the id =n) ##
```

A.2 Program: Generation of Activation Schedules

```
#!/local/pkg/all/bin/ruby

##/*================================================================*/
 ##This program permits to generate the activation schedules.
 ##Last update: 30/03/2004....
##/*================================================================*/
BEGIN {
  NUMBER_OF_RUNS         = 10
  NUMBER_OF_GENERATIONS = 50
  NUMBER_OF_ROBOTS      = [8, 16, 32, 64]
  ACTIVATION_PROB       = [0.00, 0.50, 0.75, 1.00]
  ACTIVATION_RANDOM     = true

  # do not change below this line
  NAME_SCHEDULE  = "schedule"
  ROBOT_ID = 0
  srand
}

#####################################################################
class Robot
#####################################################################
  attr_accessor :id
  def initialize(id)
    # set id
    @id = id
  end
 end  #end class

######################################################################
class RobotTeam
######################################################################
  def initialize(n_robs=0)
    @list = Array.new(n_robs)
    if (n_robs>0)
      @list.each_index {|i|
        @list[i] = Robot.new(i+1)
      }
   end
 end

  def << rob
    if ! @list.include?(rob)
      # do not accept duplicates
      @list << rob

    else
      p rob
      fail "element already in team"
    end
    self
  end

  def size
    @list.size
  end


  def next_generation (activated)
    new_gen = RobotTeam.new
    @list.each_index{|i|
      current = @list[i].clone

     new_gen << current
    }
    new_gen
  end
end

######################################################################
class ScheduleFactory
######################################################################
def initialize (size, prob=0, shuffle = true)
    @size = size
    @prob = prob
    @fixed = []
    @shuffle = shuffle
    self.fixed_initialize
  end # initialize

  def next
    fixed = self.fixed_pop
    sched = (1..(@size)).to_a
    sched.delete_if{|a| rand>@prob and a!=fixed}
    sched
  end # next

#private

  def fixed_pop
    self.fixed_initialize if @fixed.empty?
    @fixed.shift
  end # fixed_pop

  def fixed_initialize
    @fixed = (0..(@size-1)).to_a
    # random shuffle
    if @shuffle
      @size.times{
        pt = rand(@size)
        lft = @fixed[0..pt]
        rgt = @fixed[pt+1..(@size-1)]

        if ! lft.empty?
          lft.reverse! if rand < 0.5
          rand(lft.size).times{
            lft << (lft.shift)
          }
        end
        if ! rgt.empty?
          rgt.reverse! if rand < 0.5
          rand(rgt.size).times{
            rgt << (rgt.shift)
          }
        end
        @fixed = rgt + lft
      }
    end
  end # fixed_initialize
end # class ScheduleFactory

########################### end schedule ####################

###############################################################
################ Run simulation
###############################################################

def run_simulation(n_runs, n_gens, n_robs, act_prob, act_shuffle)
  init_team  = RobotTeam.new(n_robs)

  dir_path = ("run_"   + String(n_runs)    + \
              "_gen_"  + String(n_gens)    + \
              "_rob_"  + String(n_robs)    + \
              "_prob_" + String(act_prob) + \
              (act_shuffle ? "_shuf.dat" : "_noshuf.dat"))

  if File::exist?(dir_path)
    puts "#{dir_path} skipped (exists)"
    return
  end
  current_dir = Dir::pwd

begin
    Dir::mkdir(dir_path)
    Dir::chdir(dir_path)

    ### simulation
    n_runs.times{|run|

        File::open(NAME_SCHEDULE+".r"+String(run), "w") {|f_sched|

          cur_team = init_team
          sched    = ScheduleFactory.new(cur_team.size, \
                                         act_prob, \
                                         act_shuffle)
          n_gens.times{|gen|
            cur_act  = sched.next()
            print "\n current act \n"
            print cur_act

      cur_team = cur_team.next_generation(cur_act)

            # dump schedule
          #   f_sched.print String(gen+1)
            cur_act.each() {|elt| f_sched.print "#{elt}\t"}
            f_sched.puts
        }
      }
    }

  rescue
    puts "Error when making #{dir_path}"
    raise
  ensure
    Dir::chdir(current_dir)
  end
end

def main

 print "\n Please wait, currently processing activation schedules !!\n \n"
  NUMBER_OF_ROBOTS.each {|n_robs|
    ACTIVATION_PROB.each {|act_prob|
      run_simulation(NUMBER_OF_RUNS, NUMBER_OF_GENERATIONS, \
                     n_robs, act_prob, ACTIVATION_RANDOM)
    }
  }
end main

# Desired output: n_runs activation schedules

# Each activation schedule file  has the following format:
#id_activated robot \t id_activated robot....

# (a line correspond to one generation) # the id of Robots are
numbered from 1 to n..
```

A.3 Program: Implementation of the Circle Formation Algorithm

```
/*============================ Main program
============================*/ /*------------------------last
modified 26/06/2004 12:00 a.m------------*/
/*********************************************************************/
//**/**/*/*/*/*/*/*/*/*/*/*/* Remarks: */*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/
/*---The number of robots is a parameter
<=9999-------------------------*/ /*--The coordinates are double
with 15 numbers after the comma.--------*/ /*--This program
generates the history of all the positions of robots--*/
/*=====================================================================*/

/*============please do delete schedule factory*/
#include<stdio.h> #include<iostream.h> #include<stdlib.h>
#include<string.h> #include<math.h>

#include<LEDA/window.h> #include<LEDA/graphwin.h>
#include<LEDA/rat_kernel.h> #include<LEDA/rat_kernel_types.h>
#include<LEDA/geo_alg.h>

#include<LEDA/array.h> #include<LEDA/list.h>
#include<LEDA/graph.h> #include<LEDA/map.h>
```

```
#include<LEDA/rational.h> #include<LEDA/integer.h>

#include<LEDA/rat_circle.h> #include<LEDA/rat_line.h>
#include<LEDA/rat_segment.h> #include<LEDA/rat_ray.h>
#include<LEDA/rat_point.h>

#define n_robs 32 #define n_gens 50 #define n_runs 10 #define
samples 1 #define prob 0.75

/*================= Global variables ========================*/
struct coord{
    double x;
    double y;
    };

coord tab_pos[n_robs]; int activation[n_gens][n_robs]; rat_circle
SEC; rat_point table_points[n_robs]; double tab_angles[n_robs];

list<rat_point>list_points; list<double> list_angles;

GRAPH<rat_circle,rat_point> VD; GRAPH<rat_point,int> DT; const
float ratio = 0.99; const float pi = 3.14; const double range =
0.000000001; coord *positions; window W (600,600, "Simulation of
the algorithm uniform circle formation for mobile robots");

/*================= Declaration of Functions ==================*/

int current_directory (char *); int read_temp_file(const char *,
char *); coord *read_file_configuration(char *, int); double
read_x(coord *, int); double read_y(coord *, int); int
activation_schedule(char *); int schedule_factory(char *); void
display_matrix(int [n_gens][n_robs]); int all_on_boundary(
rat_point [n_robs]); list<node> node_neighbors(list<rat_point> ,
rat_point); list<rat_point> list_neighbors(list<rat_point>,
rat_point); void Draw_Voronoi(GRAPH<rat_circle,rat_point>&); int
segments_on_cell(rat_point); int rays_on_cell(rat_point);
rat_point Robot_next_position(list<rat_point>,rat_point);
rat_point Robot_inside_next_position(list<rat_point>, rat_point);
int main_processing(char *, char*); int produce_history_file(int,
char *); int produce_headers_file_history(char *); char*
concat_string(char*, char*);

/*==================== Program Main ========================*/

int main( int argc, char *argv[]) { char *str_run, *str_gen,
*str_rob, *str_prob, *str_s, *str_r, *current1, *current2,
*current3; char *path_schedule, *path_config, *file_schedule,
*file_config, *path_history, *file_history, *str_directory,
*create_dir, *str_history, *new_directory, *rm_directory,
*path_history2;

current1 = (char *)malloc(800); current2 = (char *)malloc(800);
current3 = (char *)malloc(800); str_history = (char *)malloc(800);

file_config = new char[2000]; file_schedule = new char[2000];
path_history = new char[2000];

path_history2 = new char[2000]; file_history = new char[2000];
path_schedule = new char[2000]; path_config =  new char[2000];

str_directory = new char [2000]; create_dir = new char [2000];
new_directory = new char [2000]; rm_directory = new char [2000];

str_run = new char[100]; str_gen = new char[100]; str_rob = new
char[100]; str_prob = new char[100]; str_s = new char[500]; str_r
= new char[500];

W.display(window::center, window::center);
sprintf(str_run,"%d",n_runs);
sprintf(str_gen,"%d",n_gens);
sprintf(str_rob,"%d",n_robs);
sprintf(str_prob,"%1.2f",prob); // to modify

current_directory(current2); path_schedule =
strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(current2,
"/run_"),str_run),"_gen_"),str_gen),"_rob_"), str_rob
),"_prob_"),str_prob ),"_shuf.dat"); path_schedule =
strcat(path_schedule, "/schedule.r");
```

```
current_directory(current3); path_config =
strcat(strcat(strcat(strcat(current3,"/run_"),str_run ),
"_rob_"),str_rob); path_config =
concat_string(path_config,"/configuration.r");

/*****************************************************************/
    //iteration of the different configurations(samples)
for(int k=0;k<samples;k++)
 {
  cout<<"\n @#@#@#@#@#@#@#@#@# Sample @#@#@#@#@#@#@#@#@#@ \n"<<k+1;
  printf("\n Initial configuration:\n");
  sprintf(str_s,"%d",k);
  file_config = concat_string(path_config, str_s);
  cout<<"\n ==== file config ==== \t"<<file_config<<endl;

  str_directory = strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(str
tory_"), "run_"),str_run),"_gen_"),str_gen),"_rob_"), str_rob),"_prob_"),str_prob), "_sam-
ple_"), str_s);
  cout<<"\n str_directory: "<<str_directory;
  sprintf(create_dir," mkdir %s",str_directory);
  sprintf(new_directory, "ls %s", str_directory);
  sprintf(rm_directory, "rm -rf %s", str_directory); // be carful in modify-
ing this function

  if (!system(new_directory)){
      cout<<"\n"<<str_directory<<"exists (Skipped)"<<endl;
      return 0;
  }
      //  system (rm_directory);
      //  cout<<"\n Suppression of old history directory done";}

   system (create_dir);
   current_directory(current1);
   path_history = strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(strcat(str
tory_"), "run_"),str_run),"_gen_"),str_gen),"_rob_"), str_rob),"_prob_"),str_prob ), "_sam
ple_"),str_s);

   path_history2 = concat_string(path_history,"/history.r");
  //iteration of the different schedules (runs)
  for(int run=0; run<n_runs; run++)
   {
     positions = read_file_configuration(file_config, n_robs);
     sprintf(str_r,"%d",run);
     file_schedule = concat_string(path_schedule, str_r);
     cout<<"\n ==== file schedule ==== \t"<<file_schedule<<endl;

     file_history = concat_string(path_history2, str_r);
     cout<<"\n ==== file history ==== \t"<<file_history<<endl;

     main_processing(file_schedule, file_history);

   }
 } // for 1




return 0;

}  //end main

/***************************************************************/

/*------------------- Robot_next_position
-------------------------*/

rat_point Robot_next_position(list<rat_point>list_p, rat_point
act_rob ) { rat_point p_bye, next_pos, rob;
list<point>list_intersect_tmp, list_intersect_seg,
list_intersect_ray, list_candidates; list<rat_point>list_robots;;
edge e; segment segm; int n, i, j, m, test; point q;
GRAPH<rat_circle,rat_point> VD_new; double min_dist; node v;

list_robots = list_p;

forall(p_bye, list_p){ if (p_bye == act_rob){
```

```
list_p.remove(act_rob);
//cout<<"\n list p"<<list_p;
break;}}

VORONOI(list_p, VD_new); n=0; m=0;

forall_edges(e,VD_new){

   node u = source(e);
   node v = target(e);
  list_intersect_tmp.clear();

  if (VD_new.outdeg(u) == 1 && VD_new.outdeg(v) == 1){
       continue;

   } else

     if (VD_new.outdeg(u) == 1 && VD_new.outdeg(v)>=1){
        rat_vector vec = VD_new[u].point3() - VD_new[u].point1();
        rat_point cv = VD_new[v].center() + vec.rotate90();
        rat_ray r (VD_new[v].center().to_point(), cv.to_point());

        list_intersect_tmp = SEC.to_float().intersection(r.to_float());
// cout<<"\n list tmp"<<list_intersect_tmp<<"size\t"<<list_intersect_tmp.size()<<endl;
        if (list_intersect_tmp.size() != 0) {   // one segment in voronoi can in-
tersect SEC only in 0 or 1 point
        double d1 =   p_bye.to_float().distance(list_intersect_tmp.front()) ;
            double d2 =   p_bye.to_float().distance(list_intersect_tmp.back()) ;
         if (d1<d2)
         list_intersect_ray.append(list_intersect_tmp.front());
          else
         list_intersect_ray.append(list_intersect_tmp.back());

         n++;
        }
   }else

    if (VD_new.outdeg(u) >=1 && VD_new.outdeg(v)==1){
     rat_vector vec = VD_new[v].point3() - VD_new[v].point1();
     rat_point cv = VD_new[u].center() + vec.rotate90();
     rat_ray r (VD_new[u].center().to_point(), cv.to_point());

     list_intersect_tmp = SEC.to_float().intersection(r.to_float());
       if (list_intersect_tmp.size() == 1) {
         double d1 =   p_bye.to_float().distance(list_intersect_tmp.front()) ;
         double d2 =   p_bye.to_float().distance(list_intersect_tmp.back()) ;
         if (d1<d2)
         list_intersect_ray.append(list_intersect_tmp.front());
          else
         list_intersect_ray.append(list_intersect_tmp.back());
        n++;
       }

    } else {
      segment segm (VD_new[u].center().to_point(), VD_new[v].center().to_point());
      list_intersect_tmp = SEC.to_float().intersection(segm);

    if (list_intersect_tmp.size() != 0) {   // one segment in voronoi can in-
tersect SEC only in 0 or 1 point

        double d1 =  p_bye.to_float().distance(list_intersect_tmp.front()) ;
        double d2 =   p_bye.to_float().distance(list_intersect_tmp.back()) ;
         if (d1<d2)
         list_intersect_seg.append(list_intersect_tmp.front());
          else
         list_intersect_seg.append(list_intersect_tmp.back());
      m++;
      }
 }
}


list_intersect_ray.merge(list_intersect_seg);
//cout<<"list_intersect_ray size"<<list_intersect_ray.size();

forall(q,list_intersect_ray) { test = 1; forall(rob, list_robots)
{
    if (q.distance(rob.to_float())<q.distance(p_bye.to_float()) && rob != p_bye){
```

```
test = 0; break;} }

if (test == 1) list_candidates.append(q);

}

list_candidates.sort();

//cout<<"\n list Candidates"<<list_candidates;

forall(q, list_candidates){ W.draw_circle(q,1,blue);
//leda_wait(2.2);
}

next_pos = rat_point(list_candidates.front()); //to modify
//cout <<"\n Next position"<<next_pos<<endl;
W.draw_point(next_pos.to_float(),red); return next_pos; }


/*--------------------------------------------------------*/

rat_point Robot_inside_next_position(list<rat_point>list_p,
rat_point act_rob) {

rat_point target_point; point q; list<point> list_verts,
list_candidates; edge e; int i, j; double  x1,x2,y1,y2, dis,x,y,
theta; node u,v;

rat_point o = SEC.center(); forall_edges(e,VD){

 if (VD[e]==act_rob){
   node u = source(e);
   node v = target(e);

segment segm (VD[u].center().to_point(),
VD[v].center().to_point());

point p1 = VD[u].center().to_point(); point p2 =
VD[v].center().to_point();
```

```
if ((o.to_float()).distance(p1)<= (SEC.to_float()).radius())
list_verts.append(p1);

if ((o.to_float()).distance(p2)<= (SEC.to_float()).radius())
list_verts.append(p2); }

}

list_verts.sort(); list_verts.unique(); /* forall(q,list_verts) {
    cout<<"\n list vertices after erasing duplicat: \t"<<q<<endl;
} */

target_point = rat_point(list_verts.front()); double max_dist =
(o.to_float()).distance(target_point.to_float());

cout<<"\n max distance, target
point:"<<max_dist<<target_point.to_float()<<endl;

forall(q,list_verts) {
    if(((o.to_float()).distance(q) >= max_dist) && (q!= list_verts.front())){
    if ((o.to_float()).distance(q) == max_dist)
        list_candidates.append(q);
        else{
           max_dist = (o.to_float()).distance(q);
         target_point = rat_point(q);}
    }
}

//cout<<"\n max distance, target point:"<<max_dist<<target_point.to_float()<<endl;

//determine the point in which the robot will move in the segment.
if (!list_candidates.empty()) { list_candidates.sort();
///-----cout<<"\n !!!!!!list_candidates!!!!\t"<<list_candidates<<endl;

target_point = rat_point(list_candidates.front()); }

//determine the point the robot will move with respect to the ratio.
/* x1= act_rob.xcoord(); x2= act_rob.ycoord(); y1=
target_point.xcoord(); y2= target_point.ycoord(); theta =
atan((y2-y1)/(x2-x1)); dis = sqrt(((y2-y1)*(y2-y1)) +
((x2-x1)*(x2-x1))) * ratio; x = dis * cos(theta) + x1;
//---cout<<"\n x:\t"<<x;
y = dis * sin(theta) + y1;
////---cout<<"\n y:\t"<<y;
point next_pos((dis * cos(theta) + x1), (dis * sin(theta) + y1));


cout <<"\n Next position robot inside"<<next_pos<<endl;

*/ cout <<"\n Max distance  robot inside:"<<max_dist; cout<<"\n
next position  robot inside:"<<target_point.to_float(); point
p_draw = target_point.to_float(); W.draw_point(p_draw,red);
//leda_wait(1.1);
return target_point;
//return next_pos;
}

/*----------------------main_processing--------------------------------*/
int main_processing(char *sched, char*hist) {

int counter_g, id_active, test_boundary, res_r, res_s, res; double
dis_to_circle, rad_reduce, dis_reduce; rat_point new_pos,it,
pos_active; rat_point table_positions[n_robs];

list<rat_point>list_positions; FILE *fp_hist; list<rat_point>
list_neighb; list<node> node_neighb; node v; edge e;
list<edge>list_adj_edges; int i; char str[10]; GRAPH<rat_circle,
rat_point>VD_temp; GRAPH<rat_point,int> DT_temp; char
*path_history, *str_rad, *str_dis; W.display(window::center,
window::center); cout<<"\n   shed"<<sched;
  // read the x and y coordinates

i = 1; while (i <= n_robs) {
  double x_coord =  read_x(positions, i);
  double y_coord =  read_y(positions, i);
  point p (x_coord, y_coord);
  table_points[i-1] = rat_point(p);
  W.draw_point(p, red);
  sprintf(str,"%d",i);
  W.draw_text(p,str);
  list_points.append(rat_point(p));
  i++;
}

SEC = SMALLEST_ENCLOSING_CIRCLE(list_points); rat_point o =
SEC.center(); cout<<"\n Center: \t"<<o.to_float(); double rad =
(SEC.to_float()).radius();
printf("\n  Radius: \t %4.15lf \n", rad);
W.draw_circle(o.to_float(),rad,blue);

if (produce_headers_file_history(hist)==0)  // write parameters in
history file return 0; if (produce_history_file(0, hist)==0)   //
write init configuration in history return 0;

VORONOI(list_points,VD); DELAUNAY_TRIANG(list_points,DT);
Draw_Voronoi(VD);
//leda_wait(1.20);

////schedule_factory(sched);

activation_schedule(sched); display_matrix(activation);

 //check if all on the boundary
if(all_on_boundary(table_points)){
    cout<<"\n All on the boundary \n";}
else {  //if not on the boundary

for(counter_g=0; counter_g <n_gens;counter_g++) { cout<<"
/*/*/*/*/*/*/*/ Counter_gen */*/*/*/*\n "<<counter_g;
//W.draw_text(500, 10, "new generation");
    int counter_rob = 1;

while(activation[counter_g][counter_rob] != 0 &&
counter_rob<=n_robs)
      {
```

```
     // check if on the boundary

    int id_active = activation[counter_g][counter_rob];
        pos_active = table_points[id_active-1];
        W.draw_circle(pos_active.to_float(), 2,red);
leda_wait(1.1);

        printf("\n @@@@@ Active @@@@@:\t %d", id_active);
        cout<<"\t"<<pos_active.to_float();
             // test boundary
        dis_to_circle =   o.to_float().distance(pos_active.to_float());
        double value = fabs(1-(dis_to_circle/rad));
        test_boundary = (value<= range);

        cout<<"\n test boundary!!" <<test_boundary;

if (test_boundary ==1) // if the robot on the boundary do no
thing:Just write in history

   produce_history_file(counter_g, hist);

else{  //not on the boundary
          node_neighb = node_neighbors(list_points, pos_active);
          list_neighb = list_neighbors(list_points, pos_active);
       //   cout<<"\n list neighbors: "<<list_neighb<<"\n";

forall_nodes(v, DT) { if (pos_active == DT[v]){
 list_adj_edges = DT.adj_edges(v);
 break;}
}
    // check if cell inside circle
             res_r = rays_on_cell(pos_active);
          if (res_r != 0) {
            cout <<"\n"<< pos_active.to_float()<<"\t  case 1: intersect with SEC!!!!\n"<<en
            new_pos = Robot_next_position(list_points, pos_active);
          }

          else {
              res_s = segments_on_cell(pos_active);
          if (res_s == 1){
            cout <<"\n"<< pos_active.to_float()<<"\t case 2 :intersect with SEC!!!!\n"<<enc
            new_pos = Robot_next_position(list_points, pos_active);}
          else{
             cout <<"\n"<< pos_active.to_float()<<"\t case 3 :does not inter-
sect with SEC: inside circle\n"<<endl;
            new_pos = Robot_inside_next_position(list_points, pos_active);}
          }

//leda_wait(1.2);
W.clear(); table_points[id_active-1] = new_pos; cout<<"\n
new_pos...."<<new_pos; list_points.clear();

// MAJ Matrix result
int i = 0; while (i < n_robs) {
    rat_point p = table_points[i];
   //  cout<<"\n" <<p.to_float();
    W.draw_point(p.to_float(), red);
    sprintf(str,"%d",i+1);
    W.draw_text(p.to_float(),str);
    list_points.append(p);
    i++;
}

produce_history_file(counter_g, hist);

rat_point o = SEC.center(); double rad =
(SEC.to_float()).radius(); W.draw_circle(o.to_float(),rad,blue);
W.draw_point(o.to_float(), blue);
W.draw_text(o.to_float(),"o",blue);

VORONOI(list_points,VD); Draw_Voronoi(VD);
DELAUNAY_TRIANG(list_points,DT);
//leda_wait(1.5);
}
             //check if all on the boundary
if(all_on_boundary(table_points)) {
 cout<<"\n All on the boundary \n";
 return 1;
```

```
}

   counter_rob++;
} //end while  iteration of robot....

} //end for // iteration generations... }

return 1;

//W.close();

} //end funct.


/*------------------ read initial configuration --------*/

coord *read_file_configuration( char *filename, int robs)
 {
FILE *fc; double position; int valtest, j,k; char *str_file,
*str_current; /*
sprintf(str_file, "cd %s", path_config);
system(str_file); cout<<"\n str_file \t"<<str_file; */

 fc = fopen (filename, "r");
if (fc == NULL) { printf("\n File configuration is not available
\n"); exit(1); }
           // Save the positions in tab_pos
j=0; k=0;
while ((valtest = fscanf(fc, "%lf", &position)) == 1)
{
    j++;
if (j % 2 != 0)  // even number(pair)
{
  printf(" The position x is: %4.15f", position);
  tab_pos[k].x = position;
} else {
  printf("\t y: %4.15f \n", position);
   tab_pos[k].y = position;
   k++;
}

}

fclose(fc); /*
sprintf(str_current, "cd %s", current);
system(str_current); cout<<"\n str_current \t"<<str_current; */
return tab_pos; }

/*-------------------segments_on_cell------------------------*/
int segments_on_cell(rat_point act_rob) { edge e;
map<int,rat_segment>tab_segm; rat_ray r; int n, i, res;
list<point>list_intersect_points;

forall_edges(e,VD){

 if (VD[e] == act_rob){
   node u = source(e);
   node v = target(e);

   if (VD.outdeg(u) == 1 && VD.outdeg(v) == 1){
       continue;

   } else

     if (VD.outdeg(u) == 1 && VD.outdeg(v)>=1){
       rat_vector vec = VD[u].point3() - VD[u].point1();
        rat_point cv = VD[v].center() + vec.rotate90();
        rat_ray r (VD[v].center().to_point(), cv.to_point());

   }else

    if (VD.outdeg(u) >=1 && VD.outdeg(v)==1){
     rat_vector vec = VD[v].point3() - VD[v].point1();
     rat_point cv = VD[u].center() + vec.rotate90();
     rat_ray r (VD[u].center().to_point(), cv.to_point());

   } else
```

```
      {
       segment segm (VD[u].center().to_point(), VD[v].center().to_point());
       tab_segm[n] = segm;
       n++;}
   //     cout<<"\n Segment: "<<segm<<"\n";
      }
}

res = 0; if (n!=0) forall_defined(i, tab_segm) {
    list_intersect_points = (SEC.to_float()).intersection((tab_segm[i]).to_float());
if (list_intersect_points.size()!= 0) {
    // cout<<"\n Segment \t"<<(tab_segm[i]).to_float()<<"\t intersect with the cir-
cle !!!!";
    res = 1;
    break;
    }
}

return res;   // if res == 0 then no intersection with the circle
then cell is inside circle. }

/*------------------segments_on_cell------------------------*/
int rays_on_cell(rat_point act_rob) {

map<int,rat_ray>tab_ray; rat_ray r; edge e; int n, i; n=0;


forall_edges(e,VD){
 if (VD[e] == act_rob){
   node u = source(e);
   node v = target(e);

   if (VD.outdeg(u) == 1 && VD.outdeg(v) == 1){
       continue;

   } else

     if (VD.outdeg(u) == 1 && VD.outdeg(v)>=1){
        rat_vector vec = VD[u].point3() - VD[u].point1();
        rat_point cv = VD[v].center() + vec.rotate90();
        rat_ray r (VD[v].center().to_point(), cv.to_point());
        tab_ray[n] = r;
        n++;
       // cout<<"\n ray: "<<r<<"\t"<<n<<"\n";
   }else

    if (VD.outdeg(u) >=1 && VD.outdeg(v)==1){
     rat_vector vec = VD[v].point3() - VD[v].point1();
     rat_point cv = VD[u].center() + vec.rotate90();
     rat_ray r (VD[u].center().to_point(), cv.to_point());
     tab_ray[n] = r;
     n++;
     // cout<<"\n ray**: "<<r<<"\t"<<n<<"\n";
   } else
     segment segm (VD[u].center().to_point(), VD[v].center().to_point());

 }
} /* cout<<"\n Rays:"<<endl; forall_defined(i, tab_ray) {
    cout<<tab_ray[i]<<"\t"<<endl;
} */ return n; }

/*----------------------Function read x-coordinate------*/

double read_x(coord *pos, int robot_id) { int i; double x_var;

i = 0; do {
  if ((robot_id - 1) == i)
   {
    // y = *(&(pos+(i))->x);
     x_var = (*(pos+i)).x;
   }
      i++;
   }
while(i != robot_id);
  return (x_var);
}

/*----------------------Function read y-coordinate------*/
```

```
double read_y(coord *posit, int id) { int j; double y_var;

j = 0; do {
 if ((id - 1) == j)
  {
    // y = *(&(pos+(i))->x);
      y_var = (*(posit+j)).y;
  }
  j++;

} while(j != id);
  return (y_var);
}

/*------------ Display Matrix --------------------------*/

void display_matrix(int matrice[n_gens][n_robs]) {

int i,j; printf("\n");

for(i=0;i<n_gens;i++)
   {
     for(j=0;j<n_robs;j++)
     {
     printf("%d\t", matrice[i][j]);
     }
         printf("\n");
   }
}

/*------------ all_on_boundary --------------------------*/

int all_on_boundary( rat_point table_p[n_robs])

{ int i=0, result=1;

double rad = (SEC.to_float()).radius(); rat_point o =
SEC.center(); int test_boundary=1;
```

```
while (i < n_robs) {
   rat_point p = table_p[i];

double dis_to_circle =   o.to_float().distance(p.to_float());
double value = fabs(1-(dis_to_circle/rad));

if (value > range)
      {
       result = 0;
       cout<<"\n still not yet on the boundary of the circle!!\n";
       printf("(i=%d,value=%4.15lf,range=%4.15lf)\n", i, value, range);
       break;
      }
i++; } return result; }

/*----------------- node_neighbors ------------------------*/
list<node> node_neighbors(const list<rat_point> list_p, rat_point
active_rob) {

list<node>list_nodes; node v; forall_nodes(v, DT)
    {
    if (active_rob == DT[v])
        break;
    }
      list_nodes = DT.adj_nodes(v);
  return list_nodes;
}

/*---------------- list_neighbors ---------------------------*/
list<rat_point> list_neighbors(const list<rat_point> list_p,
rat_point active_rob) { list<rat_point>list_neighb;
list<node>list_nodes; node v, w;

forall_nodes(v, DT){
  if (active_rob == DT[v])
    break;
}
```

```
list_nodes = DT.adj_nodes(v);

forall(w, list_nodes) {
 rat_point p = DT[w];
 list_neighb.push_back(p);
}
  return list_neighb;

}

/*----------------------
Draw_Voronoi----------------------------*/

void Draw_Voronoi(GRAPH<rat_circle,rat_point>& VD)

{ node v;edge e; map<edge, bool> drawn(false);

W.display(window::center, window::center);

forall_nodes(v,VD){

  if(VD.outdeg(v) < 2) continue;
   rat_point p = VD[v].center();
   W.draw_circle(p.to_point(), 1 ,green);

  }

forall_edges(e,VD){
   if(drawn[e]) continue;
   drawn[e] = drawn[VD.reverse(e)] = true;
   node u = source(e);
   node v = target(e);

   if (VD.outdeg(u) == 1 && VD.outdeg(v) == 1){
    rat_line l = p_bisector(VD[u].point1(), VD[u].point3());
      W.draw_line(l.to_line(), green);

   } else
```

```
     if (VD.outdeg(u) == 1 && VD.outdeg(v)>=1){
        rat_vector vec = VD[u].point3() - VD[u].point1();
        rat_point cv = VD[v].center() + vec.rotate90();
        W.draw_ray(VD[v].center().to_point(), cv.to_point(), blue);

   }else

    if (VD.outdeg(u) >=1 && VD.outdeg(v)==1){
     rat_vector vec = VD[v].point3() - VD[v].point1();
     rat_point cv = VD[u].center() + vec.rotate90();
     W.draw_ray(VD[u].center().to_point(), cv.to_point(), blue);

   } else
     W.draw_segment(VD[u].center().to_point(), VD[v].center().to_point(), blue);
}

}

/*-----------------------read_temp_file----------------------*/
int read_temp_file(const char *nom, char *s) {
  FILE *fp;
  int k=0;
  char ch;

  if ((fp = fopen(nom,"r")) == NULL)
    {
      printf("\n Problem in read_file: (open) \n");
      exit;
    }

  while ((ch =fgetc(fp))!=EOF)
    {

      if(ch!='\n')
    {
      *(s + k)=ch;
      k++;
    }
      else
    {
      if(feof(fp))
        break;
    }

    }//while

  *(s + k)='\0';

  return 1;
}

/*---------------------------------------------------------------*/
int current_directory (char *str) {
  system("pwd>tmp.dat");
  read_temp_file("tmp.dat", str);
  system("rm tmp.dat");
  return 1;
}

/*--------------------------------------------------------------------*/
int schedule_factory(char *filename ) {
  FILE *fp;
  char *str, ch;
  int i, j, k;
  int mx_lines, mx_cols;

for(j=1; j<=n_gens; j++) {
  for(i=1; i<= n_robs; i++)
    {
      activation[i][j]= 0;
    }
}

cout<<"\n file name, argument in Schedule factory:"<<filename; if
((fp = fopen(filename,"r")) == NULL) {
 printf("\n Problem in open\n");
 exit(1);
}
```

```
i=1; j=1; k=0; mx_lines =1; mx_cols =1;

str = (char *)malloc(200);

while ((ch =fgetc(fp))!=EOF) {

  if((ch!='\t' ) && (ch!='\n'))
    {
      *(str + k)=ch;
      k++;
    }
  else
    {
      if(ch=='\t')
    {
      *(str + k)='\0';
      activation[i][j]= atoi(str);
      printf("%d\t", activation[i][j]);
      if(i> mx_lines)
        mx_lines =i;
      if(j> mx_cols)
        mx_cols =j;

      k=0;
      j++;
    }
      if(ch=='\n')
    {
      if(feof(fp))
        break;

      *(str + k)='\0';
      activation[i][j]= atoi(str);
      printf("%d\n", activation[i][j]);
      if(i> mx_lines)
        mx_lines =i;
      if(j> mx_cols)
        mx_cols =j;
      k=0;
      i++;
      j=1;
    }
    }

}//while

return 1; }

/*===============================================================================*/


/*--------------------------------------------------------*/

int produce_headers_file_history(char *filename) {

FILE *fp_hist;

if((fp_hist = fopen(filename,"a")) == NULL){
 cout<<"\n !!!! Can't write headers in file history !!!! \n";
 return 0;}
else {
  fprintf(fp_hist,"======================== Headers ==================\n");
  fprintf(fp_hist,"n_runs=%d\tn_gens=%d\tn_robs=%d\tprob=%1.2f", n_runs, n_gens, n_robs, pr
  fprintf(fp_hist,"\nCenter_SEC:x= %4.15lf\ty=%4.15lf\tRadius=%5.15lf", (SEC.center().to_fl
  fprintf(fp_hist,"\n====================== Fin Headers ==============");
  fprintf(fp_hist,"\nBegin Data\n");
}

fclose(fp_hist); return 1; } //end func

/*--------------------------------------------------------*/

int produce_history_file(int gen_num, char *filename)

{ int i; FILE *fp_hist;
```

```
if((fp_hist = fopen(filename,"a")) == NULL){
 cout<<"\n !!!! Can't write in file history !!!! \n";
return 0;} else {
 fprintf(fp_hist,"%d\t",gen_num);

 i = 0;
 while (i < n_robs)
 {
  rat_point  p = table_points[i];

  double x = (p.to_float()).xcoord();
  double y = (p.to_float()).ycoord();

if (i != n_robs-1)
   fprintf(fp_hist,"%4.15lf\t%4.15lf\t",x,y );
else
   fprintf(fp_hist,"%4.15lf\t%4.15lf",x,y );

  i++;
 }
fprintf(fp_hist,"\n"); }

fclose(fp_hist);

return 1; } //end func

/*-------------------------------------------------------------*/
char* concat_string(char*str1, char*str2)

{ char *str3;

str3 = (char *)malloc(strlen(str1)+ strlen(str2)+10);

for (int i=0;i<strlen(str1);i++)
    str3[i] = str1[i];

for (int i=0;i<strlen(str2);i++)
    str3[i + strlen(str1)] = str2[i];
```

```
str3[strlen(str1) + strlen(str2)] = '\0'; return str3;

}

/*====================== Activation Schedule ===============*/
int activation_schedule(char *filename ) {

FILE *fp; char str[10], ch; int value, i, j, k, n;

fp = fopen (filename, "r");
   // check file existance
 if (fp == NULL)
 {
 printf("\n
 File schedule not available \n");
 exit(1);
 }

   // Matrix initialization
i=0; j=0; while (i <= n_gens)
  {
    while (j <= n_robs)
      {
    activation[i][j]=0;
    j++;
      }
    i++;
    j=0;
  }

    //Browse the file
 i=0;
 j=0;
 k=0;
for (n=0;n<=8;n++)
          str[n]= ' ';
   str[9]='\0';

while ((ch=fgetc(fp))!=EOF) { if ( (ch =='\n') || (ch == '\t'))
```

```
   {
      if (ch == '\t')
      {
         value = atoi(str);
          activation[i][j] = value;
          for (n=0;n<=8;n++)
            str[n]= ' ';
           str[9]='\0';
          k=0;

          j++;
       }
       else
       {
            value = atoi(str);
            activation[i][j] = value;
           for (n=0;n<=8;n++)
               str[n]= ' ';
            str[9]='\0';
            k=0;
           i++;
           j=0;

          if(feof(fp))
              break;
       }
    }
else {
  str[k] = ch;
  k++;

}

}

fclose(fp);

return 1;
```

```
} //end


/*------------------inside_convex_hull-------------------------*/
int inside_convex_hull(const list<rat_point> list_p, rat_point
active_rob) {

list<rat_point> list_convex; rat_point it; int res;

list_convex =  CONVEX_HULL(list_p); cout<<list_convex; res =1;
forall(it, list_convex)
    {
    if ((it.xcoord() == active_rob.xcoord()) && (it.ycoord() == active_rob.ycoord()))
       {
        res = 0;
        cout<<"\n On the convex hull!!! \n";
        break;
       }
   }
return res; }

/*-------------------------Remarks-----------------------------------*/
/* The number of robots in file configuration must equal in file
schedule*/ /*the number of generations in schedule must equal the
number of generations in program main*/

/*===================================================================*/

/*------------Function read activation schedule -----*/
/* int
*active_by_generation(int gen) {
  int j, k;
  int *table;

  //  table = (int *) malloc(n_robs * sizeof(int));

for(j=0;j<n_robs;j++)
    *(table +j)= activation[gen][j+1];
```

```
return table; } */
```

A.4 Program: Implementation of the Uniform Transformation Algorithm

```
/*============================ Main program ====================*/
/*----------------------last modified 26/06/2004 6:00 p.m------*/
/*************************************************************/
//**/**/*/*/*/*/*/*/*/*/*/*/* Remarks: */*/*/*/*/*/*/*/*/*/*/*/*/*/*/
/*---The number of robots is a parameter <=9999----------------*/
/*--The coordinates are double with 15 numbers after the comma.*/
/*============================================================*/

#include<stdio.h> #include<iostream.h> #include<stdlib.h>
#include<string.h> #include<math.h> #include<LEDA/window.h>
#include<LEDA/graphwin.h> #include<LEDA/rat_kernel.h>
#include<LEDA/rat_kernel_types.h> #include<LEDA/geo_alg.h>
#include<LEDA/array.h> #include<LEDA/list.h>
#include<LEDA/graph.h> #include<LEDA/map.h>
#include<LEDA/rational.h> #include<LEDA/integer.h>
#include<LEDA/rat_circle.h> #include<LEDA/rat_line.h>
#include<LEDA/rat_segment.h> #include<LEDA/rat_ray.h>
#include<LEDA/rat_point.h> #define n_robs 16 #define n_gens 100
#define n_runs 10 #define prob 0.5

/*==========================================================*/
struct coord{
    double x;
    double y;
    };

coord table_positions[n_robs]; coord *positions_in_circle; int
activation[n_gens][n_robs]; double tab_angles[n_robs];
list<double> list_angles;

list<point>list_loc; point table_loc[n_robs]; //table of positions
rat_circle SEC;
```

```
const double pi = 3.14159; const double range = 0.1;


coord *read_file_configuration_all_on_boundary( char *filename);
double read_x(coord *, int); double read_y(coord *, int); int
activation_schedule(char *); int schedule_factory(char *); void
computation_list_angles(list<point>); point
half_way_mid_point_previous_next(int); int uniform_spread(); void
uniform_transformation(); double compute_angle(double, double,
double, double); int previous_next_equal(int); char*
concat_string(char*, char*); int produce_history_file(int, char
*); void draw_sector(double, double, double); void new_display();
void display_matrix(int [n_gens][n_robs]);

window W (600,600, "Simulation of the algorithm uniform circle
formation for mobile robots");
/*=============================================================*/

int main() { char str[10]; W.display(window::center,
window::center); point p; coord *positions_in_circle;

positions_in_circle =
read_file_configuration_all_on_boundary("history_demo");

int i = 1; while (i <= n_robs) {
  double x_coord =  read_x(table_positions, i);
  double y_coord =  read_y(table_positions, i);
  point p (x_coord, y_coord);

  table_loc[i-1] = p;
  W.draw_point(p, red);
  sprintf(str,"%d",i);
  W.draw_text(p,str);
  list_loc.append(p);
  i++;
}

SEC = SMALLEST_ENCLOSING_CIRCLE(list_loc);
```

```
W.screenshot("/home/i2009/ssouissi/circle_sim/circle_sim_first_part/demo_uniform/demo_init1

point o = SEC.to_float().center(); cout<<"\n SEC center:"<<o;
W.draw_point(o,blue);

new_display();

uniform_transformation(); cout<<"\n Treatment finished
succesfully.\n";

//leda_wait(1.1);
W.close();
  return 1;
}

/*------------------------computation_list_angles
-------------------*/

void computation_list_angles(list<point> list_posit) {

double theta; point q; point o = SEC.to_float().center();

int i =0;

list_angles.clear(); forall(q,list_posit) {

  theta = compute_angle(o.xcoord(), o.ycoord(), q.xcoord(), q.ycoord()); //func-
tion to compute angle
  cout<<"\n Theta "<<i<<"\t:"<<theta<<endl;
   list_angles.append(theta);

   tab_angles[i]= theta;

   i++;
}

}
```

```
/*--------------------half_way_mid_point_previous_next------*/

point half_way_mid_point_previous_next(int id_active) {

double ang_active, next_pos_ang, mid_ang; list_item it,
previous_ang, next_ang;

list_angles.sort(); list_angles.unique();

double r = SEC.to_float().radius(); point o =
SEC.to_float().center();

ang_active = tab_angles[id_active-1]; cout<<"\n angle
active\t"<<ang_active;

forall_items(it,list_angles) {
    if (list_angles[it] == ang_active){
       previous_ang = list_angles.cyclic_pred(it);
       next_ang =  list_angles.cyclic_succ(it);
      break;
    }
}
  cout<<"\n previous angle:"<<list_angles.contents(previous_ang);
  cout<<"\n next angle:"<<list_angles.contents(next_ang);
  double previous = list_angles.contents(previous_ang);
  double next = list_angles.contents(next_ang);

 // draw_sectors
 draw_sector(ang_active,previous, next);

if (previous>ang_active)
      previous = previous -2*pi;

if (next<ang_active)
      next = next +2*pi;

  mid_ang = (previous + next)/2;
  cout<<"\n mid angle:" <<mid_ang;
  next_pos_ang =  (ang_active+mid_ang)/2;
  cout<<"\n next position angle:" <<next_pos_ang;
  point p = table_loc[id_active-1];
  double r_pos = sqrt((p.ycoord()-o.ycoord())*(p.ycoord()-o.ycoord()) + (p.xcoord()-
o.xcoord())*(p.xcoord()-o.xcoord()));
  cout <<"\n cos angle is :"<<cos(next_pos_ang);
  cout <<"\n sin angle is :"<<sin(next_pos_ang);

  double x = r_pos*cos(next_pos_ang);
  double y = r_pos*sin(next_pos_ang);
  point p_tar (x+o.xcoord(),y+o.ycoord());
  cout <<"\n The coordinates x and y:"<<x+o.xcoord()<<"\t"<<y+o.ycoord();

  ray rr (o,p_tar);
  list<point> l = SEC.to_float().intersection(rr); // to get the point exactly on the circl

 point q (l.back().xcoord(), l.back().ycoord());

  W.draw_point(q,red);
  W.draw_circle(q,2,red);

return q;

}

/*------------------Uniform spread-------------------------*/ int
uniform_spread() {

double ang_active; list_item it, previous_ang, next_ang;
computation_list_angles(list_loc); list_angles.sort();
list_angles.unique();

 int i=0, result=1;
 int nbr_uni = 1;

while (i < n_robs) { ang_active = tab_angles[i];

forall_items(it,list_angles) {
    if (list_angles[it] == ang_active){
        previous_ang = list_angles.cyclic_pred(it);
        next_ang =  list_angles.cyclic_succ(it);
       break;
     }
}
   double previous = list_angles.contents(previous_ang);
   double next = list_angles.contents(next_ang);

if (previous>ang_active)
     previous = previous -2*pi;
if (next<ang_active)
     next = next +2*pi;

        cout<<"%%%%%%%%%%% convergence range %%%%%%%%%% "<<fabs((n_robs*(next-
ang_active)/(2*pi))-1);

    if (fabs((n_robs*(next-ang_active)/(2*pi))-1) >range){
     result =0;
     cout<<"\n Not uniformly spread on the boundary of the circle!!\n";
     break;
    }
    else{
     nbr_uni++;
     cout<<"\n nbr uniform\t"<<nbr_uni;
    }
i++; } //end while

if (nbr_uni == n_robs) result = 1;

return result; }

/*---------------------- Uniform transformation -------------*/

void uniform_transformation() {

point pos_active, next_pos; int id_active; char *str_g, *str_demo,
*str_r;

str_demo = new char[500]; str_g = new char[10]; str_r = new
```

```
char[10];

computation_list_angles(list_loc);

activation_schedule("schedule_demo");

display_matrix (activation);
 //check if all uniformly spread
W.screenshot("/home/i2009/ssouissi/circle_sim/circle_sim_first_part/demo_uniform/ademo_init

if(uniform_spread() ==1)
    cout<<"\n !!!!! All the robots are uniformly spread on the circle !!!\n";
else {

   for(int g=0; g<n_gens;g++)
    {
     cout<<"\n Generation:\t "<<g;
     int rob = 0;
     while(activation[g][rob] != 0 && rob<n_robs)
       {

    id_active = activation[g][rob];
        pos_active = table_loc[id_active-1];
        cout<<"\n pos active:\t"<<pos_active;
        W.draw_circle(pos_active, 2,green);
        // Take a screen shot...
         sprintf(str_g,"%d",g);
         sprintf(str_r,"%d",rob);
         str_demo = concat_string(concat_string(concat_string("demo_first",str_g),str_r ),
         str_demo = concat_string("/home/i2009/ssouissi/circle_sim/circle_sim_first_part/de
         W.screenshot(str_demo);

//--    leda_wait(1.1);
        printf("\n &&&&&&&&&&&&&& Active &&&&&&&&&&&&&& id:%d",id_active);
        cout<<"\n position active:"<<pos_active;

     if (previous_next_equal(id_active) ==0){
        // compute next position
    next_pos = half_way_mid_point_previous_next(id_active);
```

```
        //MAJ table positions;
         table_loc[id_active-1] = next_pos;

         produce_history_file(g, "history_uniform_demo");

         list_loc.assign(list_loc.get_item(id_active-1), next_pos);
         computation_list_angles(list_loc);

//    leda_wait(1.1);
         new_display();

       } //end if

        // check if uniformly spread
        if(uniform_spread()) {
          cout<<"\n !!!!! All the robots are uniformly spread on the circle !!!\n";
          return;
          }

     rob++;
       }  //while

    }//for

}//else

W.close(); return; }//end funct.


/*-------------------------------------------------------------------*/

 int activation_schedule(char *filename ) {

FILE *fp; char str[10], ch; int value, i, j, k, n;

fp = fopen (filename, "r");
   // check file existance
 if (fp == NULL)
 {
 printf("\n
 File schedule not available \n");
 exit(1);
 }

   // Matrix initialization
i=0; j=0; while (i <= n_gens)
  {
    while (j <= n_robs)
      {
    activation[i][j]=0;
    j++;
      }
    i++;
    j=0;
  }

    //Browse the file
 i=0;
 j=0;
 k=0;
for (n=0;n<=8;n++)
          str[n]= ' ';
   str[9]='\0';

while ((ch=fgetc(fp))!=EOF) { if ( (ch =='\n') || (ch == '\t'))
   {
      if (ch == '\t')
      {
         value = atoi(str);
          activation[i][j] = value;
          for (n=0;n<=8;n++)
            str[n]= ' ';
           str[9]='\0';
          k=0;

          j++;
       }
       else
       {
          if(feof(fp))
             break;
            value = atoi(str);
            activation[i][j] = value;
            for (n=0;n<=8;n++)
               str[n]= ' ';
            str[9]='\0';
            k=0;
           i++;
           j=0;
       }
    }
else {
  str[k] = ch;
  k++;

}

}

fclose(fp);

return 1;

} //end

/*---------------------------------------------------*/

void new_display() {

char str[10]; point p; leda_wait(1.1); W.clear();

point o = SEC.to_float().center(); double rad =
SEC.to_float().radius();

W.draw_circle(o,rad, blue); W.draw_point(o,blue);
```

```
int i = 0; while (i < n_robs)
 {
    point p = table_loc[i];
    W.draw_point(p, red);
    sprintf(str,"%d",i+1);
    W.draw_text(p,str);
    W.draw_segment(o,p, blue);
    i++;
 }
}

/*-------------------------------------------------------*/

double compute_angle(double x1, double y1, double x2, double y2) {
double  theta, deltax, deltay;

deltay = y2-y1; deltax = x2-x1;

  theta = atan(deltay/deltax);
if (deltax<0)
     theta = theta + pi;
if (deltax>0)
    if(deltay<0)
      theta = theta + 2*pi;
if (deltax==0)
   {
     if(deltay >0)
        theta = pi/2;
     if(deltay<0)
        theta = -pi/2;
     if(deltay==0)
       theta = 0;
   }


return theta; }

/*------------------ read initial configuration --------*/
```

```
coord *read_file_configuration_all_on_boundary( char *filename)
 {
FILE *fc; double position; int valtest, j,k; char *str_file,
*str_current;


fc = fopen (filename, "r"); if (fc == NULL) { printf("\n File
configuration all on boundary is not available \n"); exit(1); }
           // Save the positions in tab_pos
j=0; k=0;
while ((valtest = fscanf(fc, "%lf", &position)) == 1)
{
    j++;
if (j % 2 != 0)  // even number(pair)
{
  printf(" The position x is: %4.15f", position);
  table_positions[k].x = position;
} else {
  printf("\t y: %4.15f \n", position);
   table_positions[k].y = position;

   k++;
}

}

fclose(fc);

return table_positions; }

/*----------------------Function read x-coordinate------*/

double read_x(coord *pos, int robot_id)
{ int i; double x_var;

i = 0; do {
  if ((robot_id - 1) == i)
   {
     x_var = (*(pos+i)).x;
   }
      i++;
   }
while(i != robot_id);
  return (x_var);
}

/*----------------------Function read y-coordinate------*/

double read_y(coord *posit, int id)

{ int j; double y_var;

j = 0; do {
 if ((id - 1) == j)
  {

      y_var = (*(posit+j)).y;
  }
  j++;

} while(j != id);
  return (y_var);
}


/*---------------------- previous_next_equal----------*/

int previous_next_equal(int id_active) {

double ang_active; list_item it, previous_ang, next_ang;

list_angles.sort(); list_angles.unique(); ang_active =
tab_angles[id_active-1];

forall_items(it,list_angles) {
    if (list_angles[it] == ang_active){
       previous_ang = list_angles.cyclic_pred(it);
       next_ang =  list_angles.cyclic_succ(it);
      break;
    }
}

  double previous = list_angles.contents(previous_ang);
  double next = list_angles.contents(next_ang);

if (previous>ang_active)
     previous = previous -2*pi;

if (next<ang_active)
     next = next +2*pi;

    double diff = fabs(((ang_active-previous)/(next-ang_active))-1);
    cout<<"\n ======Range difference: "<<diff;

if (fabs(((ang_active-previous)/(next-ang_active))-1) >range)
   return 0;
else
  return 1;

}//end

/*--------------------------------------------------------*/

void draw_sector(double ang_active, double ang_previous, double
ang_next) {

point p_active, p_previous, p_next, o; int i,j;

for (i=0;i<n_robs;i++) {
    if (tab_angles[i] == ang_active){
       for (j=0;j<n_robs;j++) {
     if (j==i) {
          p_active = table_loc[j];
         break;}
       }
    }
```

```
} cout <<"\n Point active:"<<p_active;

for (i=0;i<n_robs;i++) {
    if (tab_angles[i] == ang_previous){
       for (j=0;j<n_robs;j++) {
     if (j==i) {
          p_previous = table_loc[j];
         break;}
       }
    }
} cout <<"\n previous:"<<p_previous;

for (i=0;i<n_robs;i++) {
    if (tab_angles[i] == ang_next){
       for (j=0;j<n_robs;j++) {
     if (j==i) {
          p_next = table_loc[j];
         break;}
       }
    }
} cout <<"\n next:"<<p_next;

   o = SEC.center().to_float();
   W.draw_segment(o,p_active, green);
   W.draw_segment(o,p_next, red);
   W.draw_text(p_next, "     next",red);
   W.draw_segment(o,p_previous, red);
   W.draw_text(p_previous, "     prev",red);


leda_wait(1.1);

return; }


/*-------------------------------------------------------*/

int produce_history_file(int gen_num, char *filename)
```

```
{ int i; FILE *fp_hist;

if((fp_hist = fopen(filename,"a")) == NULL){
 cout<<"\n !!!! Can't write in file history !!!! \n";
return 0;} else {
 fprintf(fp_hist,"%d\t",gen_num);

 i = 0;
 while (i < n_robs)
 {
  point  p = table_loc[i];

  double x = p.xcoord();
  double y = p.ycoord();

  fprintf(fp_hist,"%4.15lf\t%4.15lf\t",x,y );
  i++;
 }
fprintf(fp_hist,"\n"); }

fclose(fp_hist);

return 1; } //end func

/*-----------------------------------------------------------------*/
void display_matrix(int matrice[n_gens][n_robs]) {

int i,j; printf("\n");

for(i=0;i<n_gens;i++)
   {
     for(j=0;j<n_robs;j++)
     {
     printf("%d\t", matrice[i][j]);
     }
         printf("\n");
   }
}
```

```
/*-------------------------------------------------------*/ char*
concat_string(char*str1, char*str2)

{ char *str3;

str3 = (char *)malloc(strlen(str1)+ strlen(str2)+10);

for (int i=0;i<strlen(str1);i++)
    str3[i] = str1[i];

for (int i=0;i<strlen(str2);i++)
    str3[i + strlen(str1)] = str2[i];

str3[strlen(str1) + strlen(str2)] = '\0'; return str3;

}

/*-----------------------------------------------------------*/
```