EVOLVING AGGREGATION BEHAVIORS FOR SWARM ROBOTIC SYSTEMS: A SYSTEMATIC CASE STUDY

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERKİN BAHÇECİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2005

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Erol Şahin
Supervisor

Examining Committee Members

Assist. Prof. Dr. Erol Şahin	(METU, CENG)	
Assoc. Prof. Dr. Göktürk Üçoluk	(METU, CENG)	
Prof. Dr. Faruk Polat	(METU, CENG)	
Dr. Onur Tolga Şehitoğlu	(METU, CENG)	
Assist. Prof. Dr. A. Buğra Koku	(METU, ME)	

ABSTRACT

EVOLVING AGGREGATION BEHAVIORS FOR SWARM ROBOTIC SYSTEMS: A SYSTEMATIC CASE STUDY

Bahçeci, Erkin

M.S., Department of Computer Engineering

Supervisor : Assist. Prof. Dr. Erol Şahin

August 2005, 57 pages

Evolutionary methods are shown to be useful in developing behaviors in robotics. Interest in the use of evolution in swarm robotics is also on the rise. However, when one attempts to use artificial evolution to develop behaviors for a swarm robotic system, he is faced with decisions to be made regarding some parameters of fitness evaluations and of the genetic algorithm. In this thesis, aggregation behavior is chosen as a case, where performance and scalability of aggregation behaviors of perceptron controllers that are evolved for a simulated swarm robotic system are systematically studied with different parameter settings. Using a cluster of computers to run simulations in parallel, four experiments are conducted varying some of the parameters. Rules of thumb are derived, which can be of guidance to the use of evolutionary methods to generate other swarm robotic behaviors as well.

Keywords: Swarm Robotics, Evolutionary Methods, Genetic Algorithms, Grid Computing, Neural Networks

ÖZ

OĞUL ROBOT SİSTEMLERİ İÇİN TOPLANMA DAVRANIŞI EVRİMLEŞTİRİLMESİ: SİSTEMATİK BİR ÖRNEK PROBLEM İNCELEMESİ

Bahçeci, Erkin

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Assist. Prof. Dr. Erol Şahin

Ağustos 2005, 57 sayfa

Evrimsel yöntemlerin robot sistemleri için davranış geliştirilmesinde kullanılabileceği bilinmektedir. Bu yöntemler oğul robot sistemleri için de giderek daha önemli bir çözüm yöntemi olarak ortaya çıkmaktadır. Ancak yapay evrim yöntemleriyle bir oğul robot sistemi için davranış evrimleştirilmeye çalışıldığında evrimin parametreleri ile ilgili bazı kararlar verilmesi gerekmektedir. Bu çalışmada örnek problem olarak toplanma davranışı seçilmiş ve robotların kontrol programı olarak benzetim ortamındaki bir oğul robot sisteminde evrimleştirilen yapay sinir ağları kullanılmıştır. Farklı evrim parametreleriyle oluşturulan yapay sinir ağlarının performansı ve ölçeklenebilirliği araştırılmıştır. Benzetim programları bir bilgisayar kümesinde paralel olarak çalıştırılacak şekilde bazı parametrelere verilen değerler değiştirilerek dört ayrı deney yapılmıştır. Oğul robot sistemlerinde başka davranışların da evrimsel yöntemlerle oluşturulmasına yardımcı olabilecek pratik kurallar çıkarılmıştır.

Anahtar Kelimeler: Oğul Robot Bilimi, Evrimsel Yöntemler, Genetik Algoritmalar, Dağıtık Hesaplama, Yapay Sinir Ağları

To my family

ACKNOWLEDGMENTS

The author wishes to thank his supervisor Assist. Prof. Dr. Erol Şahin for his guidance and assistance throughout the research.

The author would like to thank Onur Soysal for developing most of the Parallelized Evolution System, Emre Uğur for his LaTeX support, both of them together with Mehmet R. Doğar for their team work in simulator development, and all members of Kovan Research Laboratory for their support.

This work was partially funded by the SWARM-BOTS project, a Future and Emerging Technologies project (IST-FET) of the European Community, under grant IST-2000-31010, and by the "Kontrol Edilebilir Robot Oğulları" Career Project awarded to Erol Şahin (104E066) by TÜBİTAK (Turkish Scientific and Technical Council).

The computational resources used in this study were provided by the TÜBİTAK ULAKBİM High Performance Computing Center. This study would not be possible without their support.

Finally, the author would like to thank his family for their patience and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Apart from their essential contributions to science fiction, robots have been utilized in industry, especially in automotive sector since 1956 [1]. Different forms of these *industrial robots* or *industrial manipulators* include *robot arms* and *Cartesian coordinate robots*. They significantly speed up and improve accuracy of tasks such as welding and painting, and reduce cost and time requirements for manufacturing processes. Their main task is to replay a programmed sequence of motor actions to move its parts to exactly the same positions over and over again. This accuracy in movement is made possible in an industrial robot with its high-precision (therefore expensive) actuators and with the help of motion sensors. Environment sensing is optional, but usually the outside world is assumed to stay the same, and external disturbances are not considered.

With a need for remote operation of robots in non-industrial settings, *teleoperated* robots were built. They were driven with remote controls, and as well as actuators, some included sensors such as a camera to allow easier and more distant control. An example is the study by Farry *et al.* [2]. They have implemented teleoperation of a robot hand by transforming myoelectric signals produced by the operator's hand muscles into motor actions on the robot hand. Using this conversion they were able to replicate the movements of the operator's hand on the robot hand including grasping and several types of thumb motion.

Another type of control is *semi-autonomous control*, which has two forms: *shared control* and *control trading* [1]. The former implies that a person continuously watches the robot while it performs a task autonomously, but takes the control over when a delicate job arises. Control trading differs from shared control in level of autonomy. In control trading, a person tells the robot the task or sub-task to be done, which is carried out by the robot autonomously. Communication with the robot is only needed to assign a new job or to interrupt a currently running one. Sojourner [3], used in Mars missions, is an example to the use of control trading.

This type of controller is necessary in most space missions because of the extra long signal transmission times. Real-time teleoperation is simply not possible when it takes 5 minutes to send a command and 10 minutes to get the feedback.

The idea of *autonomous robotics* takes the challenge one step further: humans should not interfere in the robot's operation at all, i.e., robots are on their own from beginning of a task to its completion. An autonomous robot should be able to cope in an unknown environment that may have different external factors. The robot must decide on its own how to act according to its changing environment, while simultaneously trying to accomplish its objective. The robot cannot be programmed beforehand with rules on what to do for each possible combination of sensor data. Not just because it would require a huge list of sensor-actuator pairs, but also because nobody actually knows what the outputs of the robots should be for all combinations of sensor readings [4].

The performance of a robot is determined not only by its controller, but also by its interaction with the environment. Even if most environment factors such as surface roughness and amount of light and wind are stabilized, factors such as the friction between robot wheels and ground, and therefore slippage of wheels are difficult to be determined. Thus, manual design of an *ultimate* controller (that will act properly in all cases) is a rather difficult and challenging task.

1.1 Swarm Robotics

Robotics research focused on single-robot systems until late 80's. This was when research also began on multiple robots [5] because robots got cheaper to build or buy. However, multi-robot studies are not substitutes for single-robot studies since domains of *single-robot* and *multi-robot* tasks are mostly discrete. A single robot is more useful in some errands such as picking and carrying large objects, assisting people, and mimicking human actions, whereas multiple robots are more useful in tasks, which require spatial distribution, robustness, or scalability.

With the advancements in manufacturing techniques, mass production of macro, micro, and nano scale robots is expected to become possible in the near future, which will enable applications with tens to thousands of small robots. There are task domains where it is more appropriate to use such large number of simple robots instead of one big complex robot. These include [6]:

- tasks that need large area coverage (such as search and rescue missions),
- tasks that favor redundancy for robustness (such as communication networks and future nano swarms of medical robots in blood vessels) or for security (such as landmine detection and elimination),
- tasks that require scalability when the problem at hand might grow larger or smaller (such as monitoring or securing borders of enemy troops).

When we look at nature, we see that certain animals have properties that make them suitable for such tasks: the animals living in colonies, such as ants or bees. One might ask why not to get inspiration from these social animals. At this point *swarm robotics* comes into scene. It is a new approach for the coordination of large numbers of relatively simple robots [6, 7]. The approach takes its inspiration from the system-level functioning of social insects which demonstrate three desired characteristics for multi-robot systems: *robustness*[1], *flexibility*[2], and *scalability*[3]. Most of the studies [7, 8] on swarm robotics focus on developing behaviors with these desired characteristics.

By definition, in swarm robotics, robots are simplistic with respect to the task they tackle. They have simple control architectures, such as neural networks, and only small amounts of memory, if any. This fact makes them highly dependent on the current sensor inputs to decide the outputs of its actuators. They also have local communication, i.e., they can only communicate with nearby robots and do not send out broadcast messages. However, this fact does not prevent swarm systems to show interesting global behaviors.

1.2 Emergence and Self-Organization

In a swarm of social animals, local interactions of individuals result in highly organized colony behavior although there is no leader orchestrating the colony *from above*. The global behavior seems to be a consequence of local actions and communications alone. Such behaviors are called *emergent*.

Ronald C. Arkin gives the following definitions for *emergence* [9]:

[1] *Robustness* is the ability to finish the aimed task in the presence of a partial failure of the system, therefore exists in a group of robots more than in a single robot, where failure of one robot can be compensated by others.

[2] *Flexibility* is the capacity to coordinate in different ways so that different tasks can be accomplished with the same robots, for example to be able to carry objects together as well as moving in a connected group to move on rough terrain and over gaps.

[3] *Scalability* is the ability to function in groups of different sizes, so that different sized tasks can be handled.

- “Emergent behavior implies a holistic ability where the sum is considerably greater than its parts.
- Emergence is the appearance of novel properties in whole systems.
- Global functionality emerges from the parallel interaction of local behaviors.
- Intelligence emerges from the interaction of the components of the system (where the system’s functionality, i.e., planning, perception, mobility, etc., results from the behavior-generating components).
- Emergent functionality arises by virtue of interaction between components not themselves designed with the particular function in mind.”

Emergence of group behaviors or global patterns from the seemingly unrelated interactions and relations among the lower level constituents, without the need for a leader or a centralized control of any kind, is the phenomenon called *self-organization*. Simple local rules in physical or biological systems can lead to interesting global patterns. For example, suppose we add a small pie slice next to another one, with the following rule

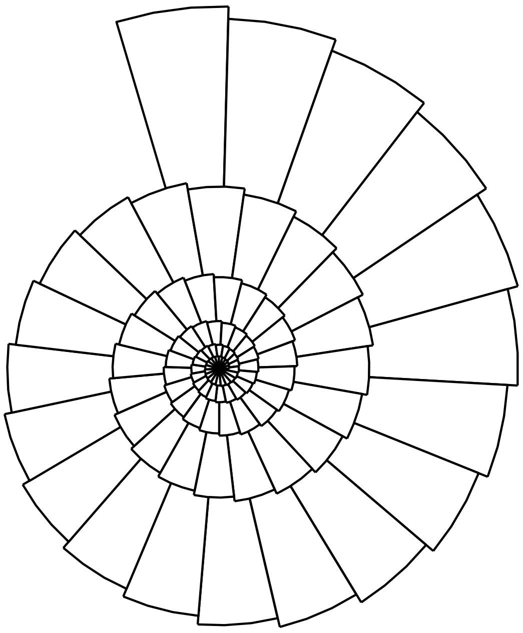
$$r(t+1) = 1.035 \times r(t) \tag{1.1}$$

where $r(t)$ is the radius of the t^{th} slice and angle of each slice is the same. After about a hundred iterations, the resulting global pattern, which can be seen in Figure 1.1(a), resembles a seashell (Figure 1.1(b)) or a snail’s shell.

Examples of such patterns can be seen in nature quite often both in the form of visual and behavioral patterns [11]. An interesting example is the series of patterns, shown in Figure 1.2, formed during aggregation of *dictyostelium* amoeba, which is a unicellular organism. They group in spirals which then turns into a multicellular formation, and finally into a stream-like pattern toward the center of cells. This is caused by local changes in concentration of a chemoatractant which is secreted by amoeba that are starving. The observed aggregation performance, which is in the order of some hundred thousands in cell count, is incredible compared to the difficulty of obtaining aggregation for even 10 robots [6].

Another self-organizing aggregation behavior happens in cockroach larvae. Jeanson *et al.* [12] investigated their behaviors at both individual and collective levels at different densities in a homogeneous medium. Jeanson *et al.* modeled the larvae’s observed behaviors which depend on the presence of nearby larvae. Their model showed that using only local information can lead the whole group into clusters.

Self-organization is observed in robotics studies as well. In [13], Owen Holland *et al.* report that in their experiments, they obtained spatial sorting behaviors, i.e., sorting two kinds


(a)

(b)

Figure 1.1: **(a)** Global pattern emerging from a simple local rule. **(b)** Similar pattern in a seashell. The image is taken from a collection [10] of interesting natural patterns by Paul Doherty.

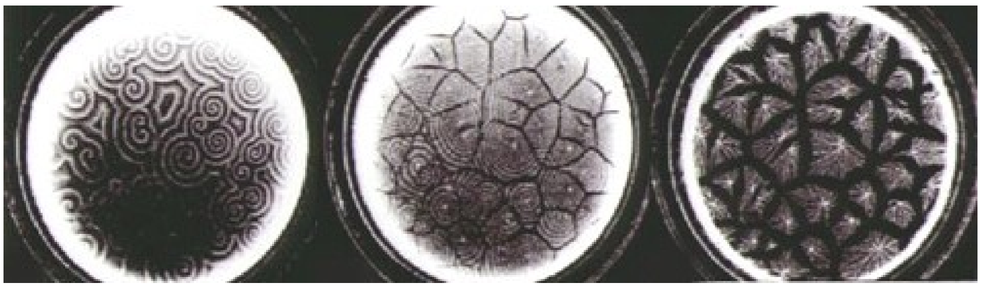


Figure 1.2: Aggregation patterns of dictyostelium amoeba into Dictyostelium discoideum slime mold. Each step of the transitions from left to right takes about 30 minutes. Images courtesy of P.C. Newell.

of pucks into distinct homogeneous clusters. They used a group of mobile robots that only sense the presence and color of the pucks. Here, an important point is that coordination and regulation of the task is done via the items themselves instead of via robot communication. The robots detect pucks and obstacles (other robots are considered obstacles as well) and act according to simple rules depending on this detection. The robots do not communicate at all. This type of interaction that is done through the environment is called *stigmergy* and is utilized among social insects in their everyday tasks such as nest building and brood sorting. Holland's study proves possibility of self-organization in robots as well, with the puck sorting behavior happening through local interactions alone, without any global communication or centralized control.

1.3 Evolving Controllers for Robots

The behavior of a robot can be described in two perspectives: proximal and distal. These are defined by Nolfi [14] as follows:

> "The *distal* description is from the observer's point of view and is based on words or sentences in our own language which are used to describe a sequence of sensorimotor loops. The *proximal* description is from the robot's point of view and is based on a description of how the robot reacts to each sensory state."

We, as humans, want the robots to perform certain behaviors, which we can accurately describe only in distal terms. However, a robot can only observe its environment from *its own* sensory view and react to it within *its own* actuation capabilities. We cannot accurately know and tell what it should do depending on its every possible combination of inputs, hence lack the proximal description of the desired behavior. Therefore we cannot easily determine the desired controller for the robots.

What we need is a method for the robots to implement their own control mechanism that depends on their *proximal* view of the environment, where we only need to (and actually only can) define a means to evaluate their *observed* behavior in our *distal* terms. A very suitable method fulfilling these requirements is artificial evolution.

Starting with a set of (possibly random) controllers, evolutionary computation methods generate increasingly better performing controllers using performance evaluations (by means of distal descriptions) of the current set. Performance increase is obtained through selection of more fit controllers and generating *possibly* higher performing ones using genetic operators, just as in the natural evolution.

A successful application of artificial evolution to robot controller design is reported by Floreano *et al.* [4], who evolved a controller for a single Khepera robot to travel in a looping maze, while avoiding the walls as much as possible. They used a suitable fitness function to maximize forward movement and obstacle avoidance together. This study shows that it is possible to utilize artificial evolution to develop robot controllers, at least for a simple task such as this.

1.4 Aggregation

As the case for our study on artificial evolution, we chose the aggregation behavior[4]. Aggregation is basically the approaching behavior of the members of a group so that they are close together for various purposes. It is observed in many types of animals, such as slime molds, beetles, flies, fish, and birds [11]. It can be considered as a self-organizing process in these animals, because only local interactions are used in global aggregation of the group[5]. Animals either use aggregation to increase their chances of survival, or they use it as a precursor of other behaviors in which they need to act together. These behaviors include carrying large prey, exchanging supplies, building nests, reproducing, preserving their body temperatures, and coordinated and more efficient movement [11]. All of these require prior aggregation at the site of interest.

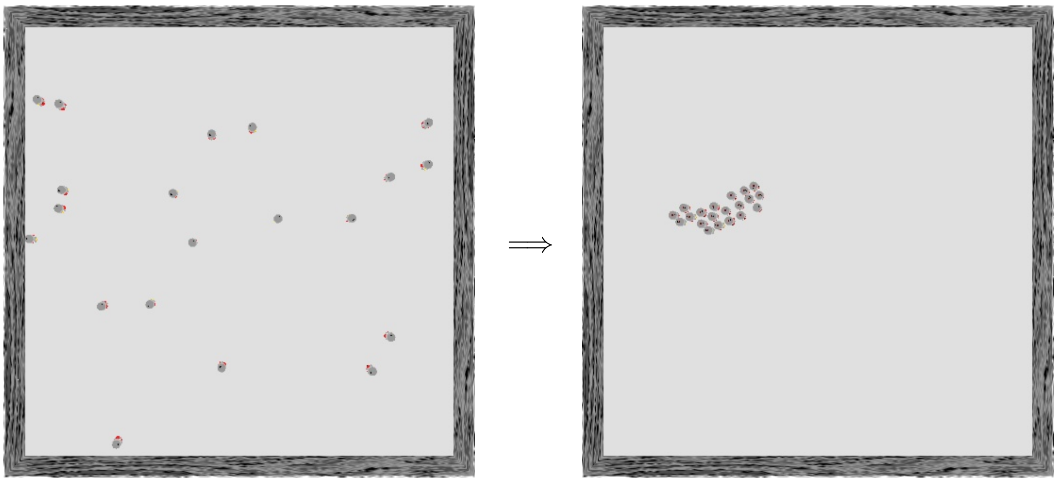


Figure 1.3: Aggregation task involving 20 robots.

The case we chose in this systematic study can be described as "*aggregation of robots with limited ranged perceptual abilities in a closed arena*". Aggregation, or clustering, of robots (Figure 1.3) is useful when the task to be accomplished requires or is done easier with a group of robots. Aggregation is especially valuable in swarm robotics, where by definition of swarm robotics [6], robots interact and communicate locally. However, aggregation in a

[4] There are also studies in literature [15] that study behaviors called as aggregation, which is actually gathering passive items (such as pucks) by the robots. This behavior should not be confused with the aggregation that is the task case in this study.

[5] See the definition of self-organization in Section 1.2

group of robots that can only communicate locally is a challenging problem, because as in the animals mentioned above, a self-organizing behavior should be developed. Moreover, this controller should be scalable as much as possible like in animals, i.e., it should work as well as it can with different numbers of individuals. Furthermore, the robots are not capable of identifying other robots or distinguishing a single close robot from a group of distant ones (as seen in Figure 3.5(c)). Therefore, one may have difficulty in determining the right kind of interactions among the robots with their limited sensing capabilities to obtain scalable global aggregation. A straightforward strategy such as "go toward the loudest signal, if the sound is more than some threshold then stop" results in small clusters, which are far from desired large clusters containing as many robots as possible. Hence, discovering better strategies is required.

To study performance in accomplishing a task, one needs a way to measure the quality of the desired behavior. The assessment of the quality of aggregation can be done in more than one way. For simulations, the assessment methods may be computing average distances between every pair of robots, calculating average distance to the center of mass of robots (as done in [16]), counting robots in clusters (in closely packed groups), computing minimum spanning tree among robots (over distances to each other), etc. The method used in this study will be described in detail in Section 5.2.4.

1.5 Systematic Analysis of Aggregation in Swarm Robotic Systems

Due to their appropriateness as a controller generation tool, evolutionary methods are becoming promising candidates to develop behaviors in robotics studies. This is especially true for swarm robotic systems, on which it is necessary to discover the required interactions among a group of very simple robots that should act in a coordinated fashion using only local communication. This causes a larger gap between distal and proximal descriptions of the desired behavior, and makes swarm robotic systems more suitable for evolutionary algorithms. On the other hand, when one uses evolutionary algorithms in swarm robotics, he should determine some evolution and fitness evaluation parameters.

Using aggregation of robots, we studied the performance and the scalability of evolved behaviors for a simulated swarm robotic system. We conducted four systematic experiments varying some parameters and analyzed the effect of different parameter choices on performance and scalability of aggregation behaviors.

The rest of the thesis is organized as follows: Chapter 2 will describe evolutionary methods, and in particular *Genetic Algorithms*. It will also review some evolutionary robotics studies in literature. Physical simulation environment and robot architecture will be discussed in Chapter 3. Chapter 4 will elaborate on a system (with installation and usage guide given in Appendix A) to run genetic algorithms in a distributed manner, which is needed due to high computation times required by physical simulations. Chapter 5 will describe the details of the infrastructure used for evolution experiments. The details and results of these experiments will be stated in Chapter 6. Finally, conclusions and future work will be presented in Chapter 7.

CHAPTER 2

Evolutionary Robotics

2.1 Artificial Evolution

Evolutionary computational methods are inspired by the natural evolution. In nature, a population of animals struggle to survive and reproduce to produce the next generation. The principle of “survival of the fittest” applies: individuals that are fitter within their environments are more likely to survive and also more likely to produce offspring, transferring their genetic material onto the next generation. In this way, nature eliminates weak individuals and the population gets more adapted to the environmental conditions generation by generation.

The idea of evolution, i.e., animals getting selected over their *survival performance* to produce better adapted populations, started to be used as an optimization method, in 60’s. Evolutionary computation grew in four areas:

- Genetic algorithms (Holland)
- Evolutionary programming (Fogel *et al.*)
- Evolution strategies (Rechenberg and Schwefel)
- Genetic programming (Koza)

John H. Holland began publishing on adaptive systems theory in 1962 [17] and wrote his book on *Genetic Algorithms* in 1975 [18], which mimics natural evolution and is basically adaptation of a *population* of candidate solutions for a problem with the use of genetic operators such as *selection*, *mutation* and *crossover*.

Meanwhile, Fogel *et al.* came up with a method they named *Evolutionary Programming* [19, 20] which is similar to genetic algorithms, but does not make use of crossover. This was one of the earliest attempts to evolve behavior. Evolutionary Programming was applied to evolution of finite state machines and function optimization [21].

Another evolutionary computation method is Rechenberg's *Evolution Strategies* [22, 23], which makes use of different crossover (or *recombination*) techniques where one new population member is formed using genes of either two or all of the population members, and choosing an intermediate value of among parent parameter values in the chromosome instead of directly inheriting one of them.

The fourth method, *Genetic Programming*, is attributed to Koza [24] and is usually considered a subset of genetic algorithms. It deals with evolving computer programs in the form of trees instead of strings, where the individuals in the population are evaluated for fitness by being executed.

According to [21], which describes and discusses these four evolutionary algorithms comprehensively, evolutionary computation methods, are different from other search and optimization methods in several aspects:

- *A population of candidate solutions rather than a single one is maintained.* This increases variety in solution space. It also enables applications where a group of solutions is needed rather than only one.

- *The search in solution space is done more randomly compared to deterministic progression of other methods.* Doing so makes it possible to discover different and better solutions at consequent evolutions.

- *Fitness of solutions are used directly instead of being utilized as derivatives and second derivatives.* This enables applications in optimizing functions that do not have continuous derivatives.

2.2 Genetic Algorithms

Holland's Genetic Algorithms, works roughly as depicted in Figure 2.1. Suppose we have a given problem that we want to find a solution to. For example, let this problem be minimizing a function $f(x)$. For the genetic algorithm to operate on a problem, we need an encoding for a candidate solution of the problem. In our example, a candidate solution is an x value, which can be encoded as a floating point number or an integer.

Genetic algorithms work with a set of candidate solutions rather than a single one as in other optimization methods. The encoded candidate solution is called a *chromosome* and the set of solutions is called a *population* analogous to a set of animals in a population, each of

```
Initialize population
repeat
   Evaluate fitness of individuals
   Select individuals to mate depending on fitness
   Pair individuals to be mated
   Apply crossover
   Apply mutation
until a termination condition is met
```

Figure 2.1: The genetic algorithm.

which are *encoded* in a single DNA molecule.

The genetic algorithm needs a way to evaluate the *goodness* (or *fitness*, as widely used) of a candidate solution. For the function minimization example, this would simply be evaluation of the function with the value of the variable that is the candidate solution. The smaller the result, the better the solution.

The whole population of candidate solutions is evaluated in this manner. Depending on their fitness values, the population is sorted and a subset of the population is *selected* among the better ranking individuals. This selected set is then used to produce the new population, i.e., the population of the next *generation*. It is this part of the Genetic Algorithm that implements the *survival of the fittest* principle. The new population, created through some genetic operators such as *crossover* and *mutation*, is expected to have higher fitness values.

Crossover or *recombination* is applied to the selected set of individuals, with a certain probability. Crossover swaps parts of two chromosomes, i.e., pairs from the selected set, where it can be applied at one point on the chromosomes or on multiple points. Its main purpose is to join two *useful* segments of two different chromosomes in one chromosome, where the resulting chromosome has more *useful* parts than the two initial chromosomes. This does not always happen, but when it happens often enough, it will result in a better performing population through improved individuals.

After crossover operations, individuals in the population are subjected to the *mutation* operator with a small probability. Mutation is used to alter a small portion of the chromosome at a random position. It helps in creating individuals that are randomly and slightly perturbed versions of the previous populations. In short, crossover combines solutions whereas mutation generates new ones.

Use of genetic operators such as crossover and mutation improves chances of introducing more fit individuals, however these operators may also destroy some highly fit ones. To overcome this disadvantage, a fraction of the top ranking individuals in the population may be transferred to the next generation without applying crossover or mutation. This helps preserving the best chromosomes, and usually accelerates evolution. This modification is called elitism and is commonly used in studies using genetic algorithms.

The encoding of a candidate solution and the fitness function is specific to the problem at hand. Furthermore, the crossover and mutation operations can be defined to suit the chromosome encoding. For example, random bits in the chromosome encoding can be altered or if the encoding consists of a set of numbers, the value of a randomly chosen one can be increased or decreased by a random amount. Genetic algorithms are executed in the same way (as shown in Figure 2.1) once the following are supplied:

- an encoding,
- a fitness function,
- a mutation operator,
- a crossover operator,
- a termination condition.

The Genetic Algorithm continues to produce new populations, or generations, in this manner until a termination condition is met, which is usually reaching a maximum number of generations. Then, the best candidate solution of the final generation is considered to be *the* solution produced by the evolution. Furthermore, if the problem requires so, not one but a set of the candidate solutions in the final population can be used, which is not possible with more traditional optimization algorithms.

2.3 Use of Artificial Evolution in Swarm Robotic Systems

Early studies on evolving behaviors for swarm robotic systems reported limited success and expressed pessimistic conclusions. In one of the earliest studies, Zaera *et al.* [25] used evolution to develop behaviors for dispersal, aggregation, and schooling in fish. Although they had evolved successful controllers for dispersal and aggregation; the performance of the evolved behaviors for schooling was considered disappointing, and they concluded that for complex actions like schooling, manual design of a controller would require less time and effort than evolving one, mainly due to the difficulty of determining a useful evaluation function for the specific task.

Mataric *et al.* [26] have made a comprehensive review of the studies until 1996 on evolving controllers to be used in physical robots and they have discussed the key challenges. They addressed approaches and problems such as evolving morphology and/or controller, evolving in simulation or with real robots, fitness function design, co-evolution, and genetic encodings. They emphasized that for an evolved controller to be beneficial, the effort to produce it in evolution should be less than the effort needed to manually design a controller for the same robotic task. They stated that it has not been the case, yet; but when the challenges and problems are handled, they may become a practical alternative to controllers designed by hand.

In [27], Lund *et al.* used evolution to develop minimal controllers for exploration and homing task. They evolved controllers for the Khepera robot (K-Team, Switzerland) for the task considered where the robot was desired to leave a light source, i.e., home, explore the surrounding for some time, and then return back home where it is virtually recharged. To obtain this periodic behavior, they used sampled sensory input and a minimal network architecture without recurrent connections, which can be used to obtain the notion of *return period*. Instead their evolution exploited the geometrical shape as perceived by robot and produced a suitable controller.

In contrast to some of these pessimistic conclusions, during the recent years optimistic results are being reported on the evolution of swarm behaviors. In the Swarm-bots Project [28], Baldassarre *et al.* [29] successfully evolved controllers for a swarm of robots to aggregate and move toward a light source in a clustered formation. Moreover, for this specific task, several distinct movement types emerged: constant formation, amoeba (extending and sliding), and rose (circling around each other). In [28], Trianni *et al.* also evolved successful controllers for a swarm of robots that can grip each other, called a swarm-bot, to fulfill tasks such as aggregation, coordinated motion in a common direction, cooperative transport of heavy loads (as in ants), and all-terrain navigation to avoid holes (connected in swarm-bot formation). Their evolved controllers made use of sound sensors, traction sensors, and flexible links. Trianni *et al.* [30] has also identified two types aggregation behaviors emerged from evolution: a dynamic and a static clustering behavior. In static clustering, robots move in circles until they are attracted to a sound source. Then they *bounce* against each other until an aggregate is formed. The clusters are tight and static with the robots involved turning on the spot, whereas in dynamic aggregation, the clusters are loose and they flock around. This study is a good example of evolution of different strategies, or behaviors, for a specific task. Furthermore, in [16] Dorigo *et al.* evolved aggregation behaviors for a swarm robotic system. They analyzed two of the evolved behaviors and showed that evolution was able to discover rather scalable behaviors.

Ward *et al.* [31] have evolved neural network controllers for such a survival scenario where there are two populations of animals, predators and preys that co-evolve to produce a schooling behavior. They have also studied on the connection of physiology with behavior and they claim that prey need a wide-range low-resolution visual sensors whereas predators are better off with visual input concentrated in the front.

Despite these studies, the use of artificial evolution to generate swarm robotic behaviors for a desired task is a rather unexplored field of study. The effort in using evolutionary methods can be reduced by suggestions on choosing parameters required in applying artificial evolution to swarm robotics. To the best of our knowledge, no systematic study has been made to investigate effects of parameters to help such choices. In this study, we addressed this lack of systematic analysis studies to deduce some rules of thumb on the choice of some parameters used in evolution of swarm robotic behaviors.

CHAPTER 3

Simulator

The previous chapter discussed the benefits of using artificial evolution to develop robot controllers. However, once a controller is evolved using simulations, it is difficult to be sure that this controller will actually work on a physical robot as it does in simulation. Therefore, to increase confidence in transferring an evolved controller onto physical robots, robot simulations are done using physical simulators. These simulators realistically model movements and interactions of bodies under gravitational and other external forces. Bodies have mass and shape properties (collision geometries) and can be connected to each other with several types of joints. Collisions between bodies are also simulated. Upon collision, an instantaneous joint (contact joint) is formed between the colliding bodies that simulates the desired amount and type of frictional forces.

3.1 Simulation Environment

In this study, a port[1] of MISS, a cut-down version of the Swarmbot3D simulator [32] is used. Swarmbot3D is a physics-based simulator developed within the Swarm-bots project that modeled the s-bots (mobile robots with the ability to connect to each other). Swarmbot3D simulator includes simulation models of the s-bot at different levels, all obtained from and verified against the actual s-bot. As Mataric *et al.* mentioned [26], evolving controllers for physical robots in simulation requires modeling of noise and error models to maximize transferability of controllers onto physical systems. This is ensured in this simulator with the sensor models implemented with sensory data coming from the physical s-bot. The minimal s-bot model of Swarmbot3D simulator is used here, with which evolution of aggregation behavior was first

[1] In Kovan Research Lab, we ported MISS and Swarmbot3D simulators from Vortex, a commercial physics development platform, to the Open Dynamics Engine (ODE), a free physics-based simulation library. Extensions to ODE were done to add XML file loading capabilities, to improve rendering and camera handling, which were packaged under the name Kovan ODE eXtensions (KODEX). Using KODEX, converting Swarmbot3D from Vortex to ODE was possible with little effort.

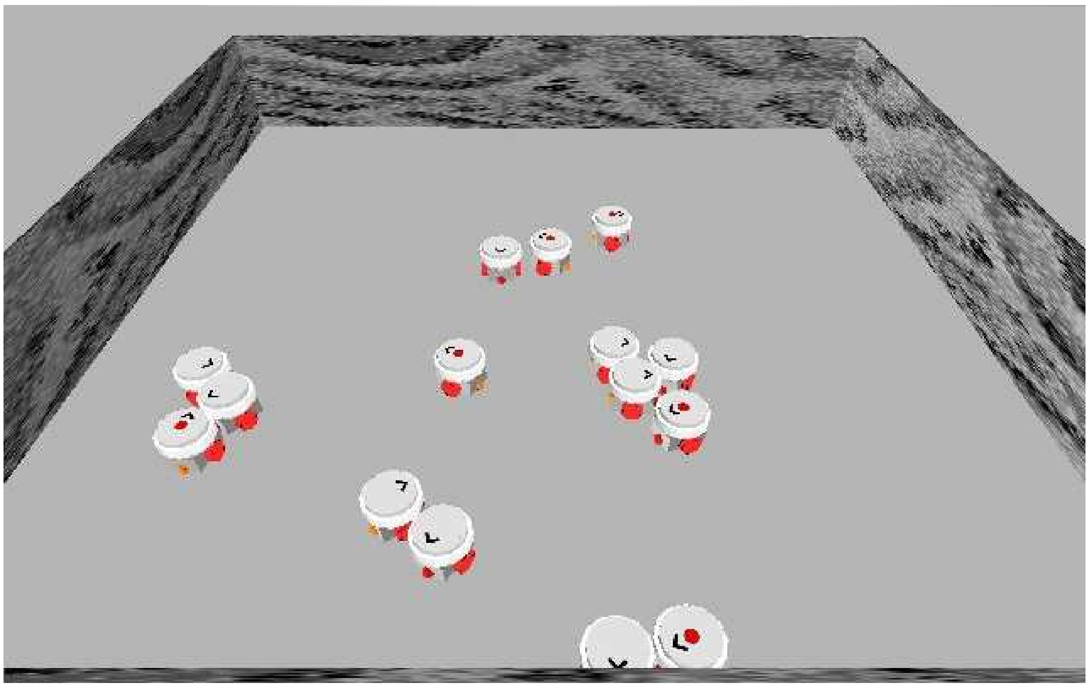


Figure 3.1: A screenshot of the simulator.

studied by Dorigo *et al.* in [16]. A snapshot of the simulator is shown in Figure 3.1.

3.2 Robot Architecture

A schematic view of the robot indicating the sensor and signal source configuration used in our experiments is shown in Figure 3.2. The robot is modeled as a differential drive robot with two wheels. The model has 8 infrared range sensors around the robot, and one omni-directional speaker and 4 directional microphones placed at the center of the robot.

3.2.1 Robot Controller

The part of a robot that computes actuator outputs as a function of the robot inputs is called the *controller*, i.e., brain, of the robot. In the experiments performed, robots act reactively depending only on their inputs. They don't have any memory or long deliberative processes to decide on the outputs. The controller is chosen to be a single-layer perceptron which has 12 input neurons (4 connected to microphones and 8 connected to infrareds), 3 output neurons (1 to control the omni-directional speaker and 2 to control the wheels) as seen in Figure 3.3. This controller is a reactive architecture because outputs are determined directly by the inputs: no planning is done and no memory is used.

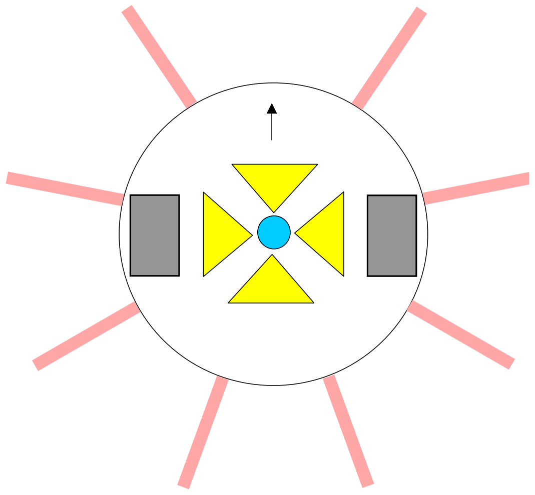


Figure 3.2: A schematic view of the robot model. The robot has a diameter of 5.8 cm. The 8 bars emanating from the body of the robot indicate the IR sensor direction and range. The 4 triangles are placed at the center represent microphones, 2 rectangles at the sides represent wheels, and the circle at the center represents an omni-directional speaker.

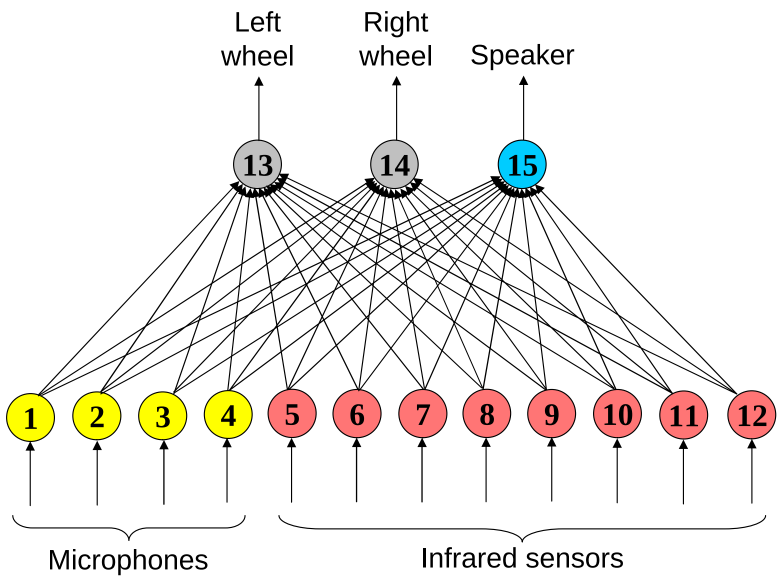


Figure 3.3: Neural network controller used as the controller for robots. Neurons match corresponding parts in Figure 3.2 as follows: 1-4: microphones, 5-12: IR sensors, 13-14: wheel actuators, and 15: speaker.

3.2.2 Sensor Specifications

The details of the sensor models are described in detail in [32]. The infrared sensors are modeled using sampling data obtained from the real robot with the addition of white noise as described in [29] and [32]. The characteristics of the sampled IR sensors can be seen in Figure 3.4. These were recorded with an obstacle near the robot at certain distances and angles.

Figure 3.5(c) shows the characteristics of the sound sensor model which drives the long-range interactions among the robots. As it is, the sound sensor model can be regarded as unrealistic due to its simplicity. However, using a proper placement of microphones robust sound source localization can be done as in [33], where Valin *et al.* has localized sound sources with a precision of 3 degrees in 3 meters range using an array of 8 sound sensors placed at the corners of a rectangular prism.

However, it should be noted that our simulator was neither verified against the original Swarmbot3D simulator, nor against the physical robots. Therefore, we make no claims about the portability of the evolved controllers onto the physical robots. Yet, for the purpose of this study, we believe that the sensor and signaling models which were taken from the Swarmbot3D simulator are sufficient since our study aims to deduce general rules of thumb for evolving behaviors in swarm robotic systems.

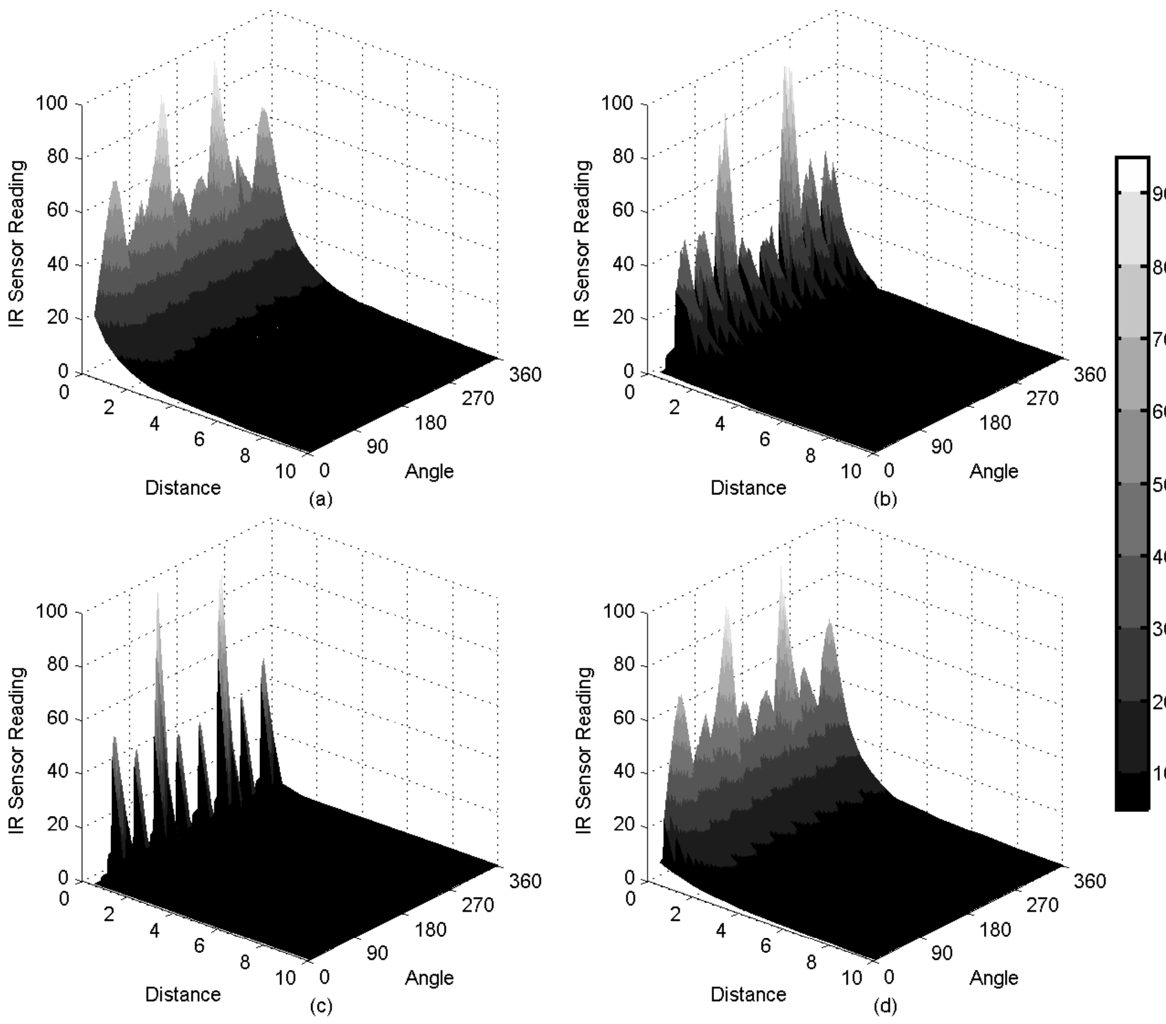
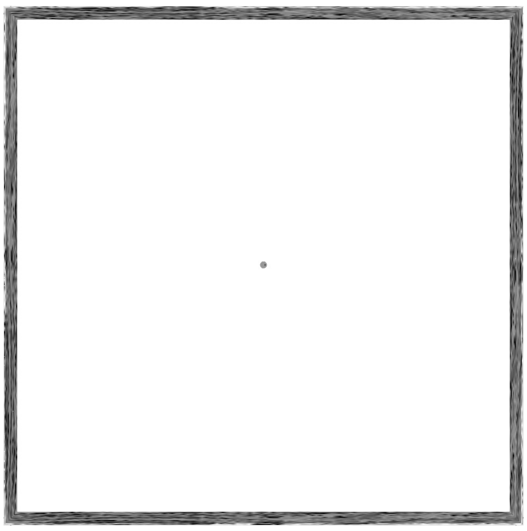
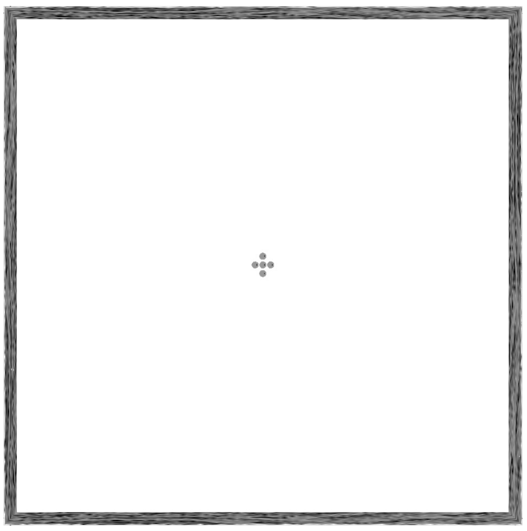


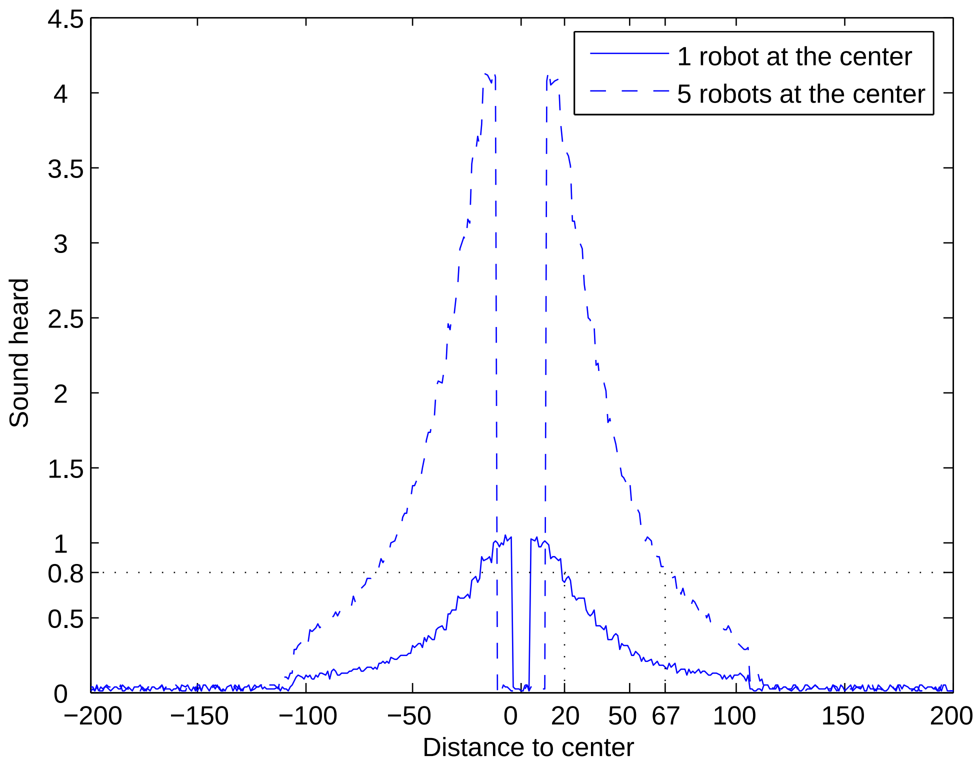
Figure 3.4: Readings of sampled IR sensors obtained in [29]. The four graphs belong to samples obtained when near **(a)** a straight wall, **(b)** another robot, **(c)** a small obstacle, and **(d)** a big obstacle. The vertical axis shows the observed proximity value as the maximum among the readings of the eight actual IR sensors around the physical robot, hence the eight peaks in each graph. The other two axes show the angle and distance of the obstacle. The colors show the range of sensor values, which tells us that in all samples the observed proximity is below 10% at distances greater than about 3 cm. (The robot has a diameter of 5.8 cm).



(a)



(b)



(c)

Figure 3.5: **(a)** A 400×400 cm arena with one robot at the center. **(b)** A 400×400 cm arena with five robots at the center. **(c)** Sound heard from one and five sound emitting robots at the center, shown with a solid and a dashed line, respectively. The audibility values shown are the maximum of sensory input values recorded by the four microphones of a virtual robot which is placed at different distances from one wall to the opposite on a horizontal line intersecting the center of the two arenas shown in (a) and (b). Higher values indicate higher audibility. The pits at the center indicate the regions occupied by the sound emitting robots. It is important to note that one robot at distance 20 and 5 robots at distance 67 are heard at the same audibility level. These two very different situations cannot be distinguished by the robots used in this study.

CHAPTER 4

Parallelized Evolution System (PES)

Evolutionary robotics studies require large numbers of simulation runs. For an evolution of 50 generations that uses a population of 50 individuals and 5 different runs for evaluating the fitness of single individual, a total of 12500 simulation runs are needed. When physical simulations are used to achieve realistic behaviors, a single simulation takes time on orders of 10's of seconds to complete. For an average simulation run time of 50 seconds, a single evolution requires 625000 seconds, or 10416.67 minutes, or 173.61 hours, or 7.23 days on a single computer. Due to these heavy costs of processing requirements, evolutionary robotics studies would be definitely limited by the total amount of CPU-time available.

Fortunately, fitness evaluations done in one generation of an evolution are completely independent of each other. Hence a possibility of parallelization arises for these fitness evaluations of the same generation. And this is exactly what the Parallelized Evolution System (PES) does.

PES [34] is a platform to parallelize evolutionary methods on a group of computers connected via a network. It separates the fitness evaluation of genotypes from other tasks (such as selection and reproduction) and distributes these evaluations onto a group of computers to be processed in parallel. PES consists of two components: (1) a server component, named PES-Server, that executes the evolutionary method, the management of the communication with the client computers, and (2) a client component, named PES-Client, that executes programs to evaluate a single individual and return the fitness back to the server. Figure 4.1 shows the structure of a PES system.

An easy interface is provided to the user by PES, which relieves him from dealing with the communication between server and client processes. PES-Client is developed for both Windows and Linux, enabling the PES system to harvest computation power from computers running either of these operating systems. An easy-to-use framework for implementing

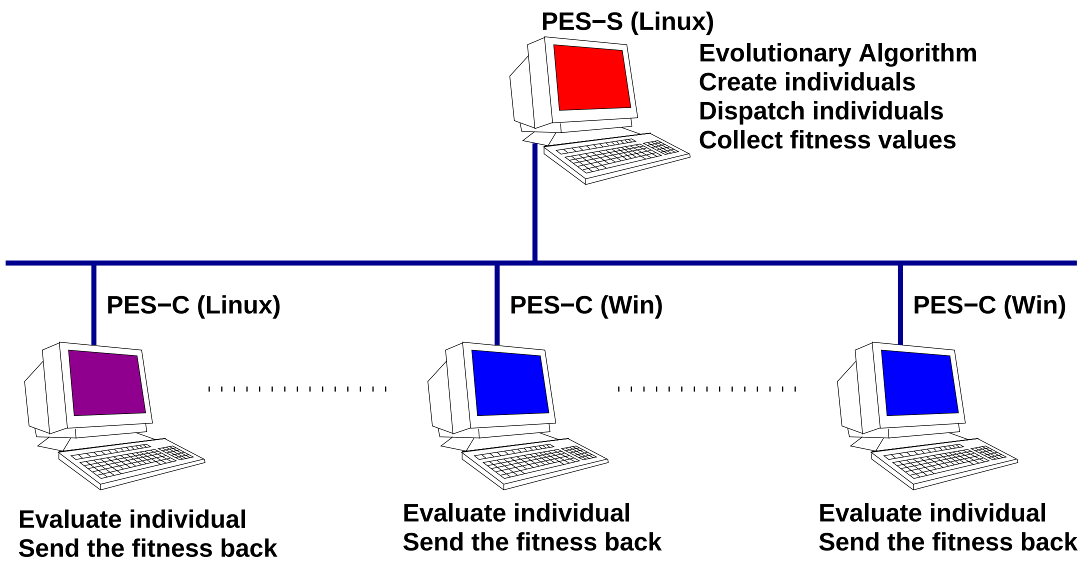


Figure 4.1: Structure of the PES system. The PES-Server runs on a Linux machine and handles the management of the evolutionary method. It executes the selection and reproduction of the individuals (genotypes) which are then dispatched to a group of PES-Clients (running both Windows and Linux systems). The individuals are then evaluated by the clients and their fitness values are sent back to the server.

evolutionary methods, and the inter-operability of the system distinguishes PES from other systems available and makes it a valuable tool for evolutionary methods with large computational requirements.

PES uses PVM (Parallel Virtual Machine)[35][1], a widely utilized message passing library in distributed and parallel programming studies, for communication between the server and the clients. We have also considered MPI [36] as an alternative to PVM. MPI is a newer standard that is being developed by multiprocessor machine manufacturers and is more efficient. However PVM is more suitable for our purposes since (1) it is available in source code as free software and is ported on many computer systems ranging from laptops to CRAY supercomputers, (2) it is inter-operable, i.e. different architectures running PVM can be mixed in a single application, (3) it does not assume a static architecture of processors and is robust against failures of individual processors.

PES wraps and improves PVM functionality. It incorporates a time-out mechanism to detect processes that have crashed or have entered an infinite loop. PES provides *ping*, *data* and *result* message facilities. Ping messages are used to check the state of client processes. Data messages are used to send task information to client processes and result packages are

[1] Available at http://www.csm.ornl.gov/pvm/pvm_home.html.

used to carry fitness information from clients.

The following sections describe the PES-Server and PES-Clients.

4.1 PES-Server

PES-Server provides a generic structure to implement evolutionary methods. This structure is based on Goldberg's basic Genetic Algorithm [37] and is designed to be easily modified and used by programmers. The structure assumes that fitness values are calculated externally. In its minimal form, it supports tournament selection, multi-point cross-over and multi-point mutation operators.

PES-Server maintains a list of potential clients (computers with PES-Client installed), as specified by their IP numbers. Using this list, the server executes an evolutionary method and dispatches the fitness evaluations of the individuals to the available clients. The assignment passes the location of the executable to be run on the client as well as the parameters that represent that particular individual and the initial conditions for the evaluation. Then it waits for the clients to complete the fitness evaluation and get the computed fitness values back.

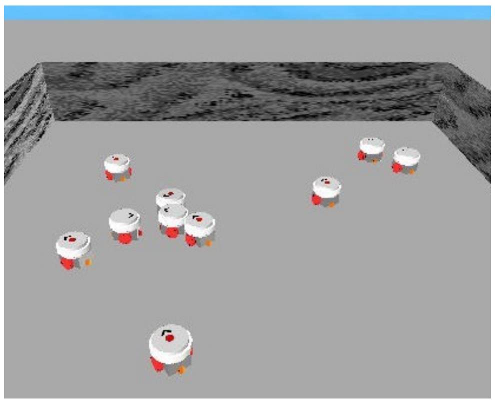
PES-Server contains fault detection and recovery facilities. Using the *ping* facility the server can detect clients that have crashed and assign the uncompleted tasks to other clients. In its current implementation, the server waits for the evaluation of fitness evaluations from all the individuals in a generation before dispatching the individuals from the next generation.

4.2 PES-Client

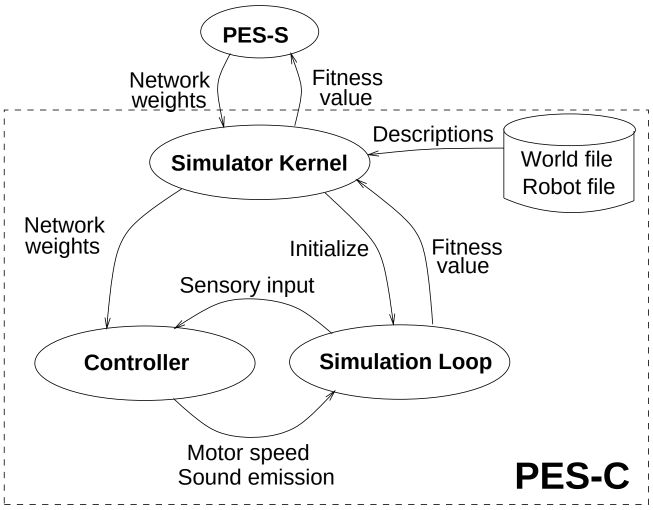
PES-Client acts as a wrapper to handle the communication of the clients with the server. It fetches and runs a given executable (to evaluate a given individual) with a given set of parameters. It returns the fitness evaluations, and other data back to the server.

Client processes contain a loop that accepts, executes and sends result of tasks. Client processes reply to *ping* signals sent by the PES-Server to check their status. Crashed processes are detected through this mechanism.

PES-Clients are developed for single processor PC platforms running Windows and Linux operating systems. Note that, to use clients with both operating systems, the fitness evaluating program should be compilable on both systems. In its current implementation, PES-Client has the fitness evaluation component embedded within itself (as seen in Figure 4.2(b)) to simplify communication with PES-Server. For another problem with a different fitness evaluation,



(a)



(b)

Figure 4.2: **(a)** A snapshot of the environment being simulated: Mobile robots distributed in an arena are enclosed by walls. **(b)** Architecture of PES-Client being used. In its current implementation, fitness evaluation and communication parts are not separated.

the PES-Client should be altered to suit that problem. Ideally, the communication component of PES-Client should be separated from the fitness evaluation component, but this is not implemented yet.

4.3 Experimental Results of PES

We developed the PES platform as part of our work within the Swarm-bots project[2] [38] to develop swarm robotic behaviors. We conducted experiments to evolve behaviors for clustering of a swarm of mobile robots and analyzed the speed-up and efficiency of the PES system in this task.

The swarm of robots and their environment are modeled using ODE (Open Dynamics Engine), a physics-based simulation library, Figure 4.2(a). The parameters of this simulation and parameters of the controller (network weights) are passed to the PES-Client from the PES-Server. The simulator constructs the world and runs the simulation, by solving physical dynamics equations. Movements of robots are determined by their controller as specified by the genotype. This controller uses the sensors of the robots and moves the robots for 2000 time steps. Then a fitness value is computed, based on a measure of clustering achieved. This fitness value is then returned to the PES-Server, as shown in Figure 4.2(b).

[2] More information can be found at http://www.swarm-bots.org.

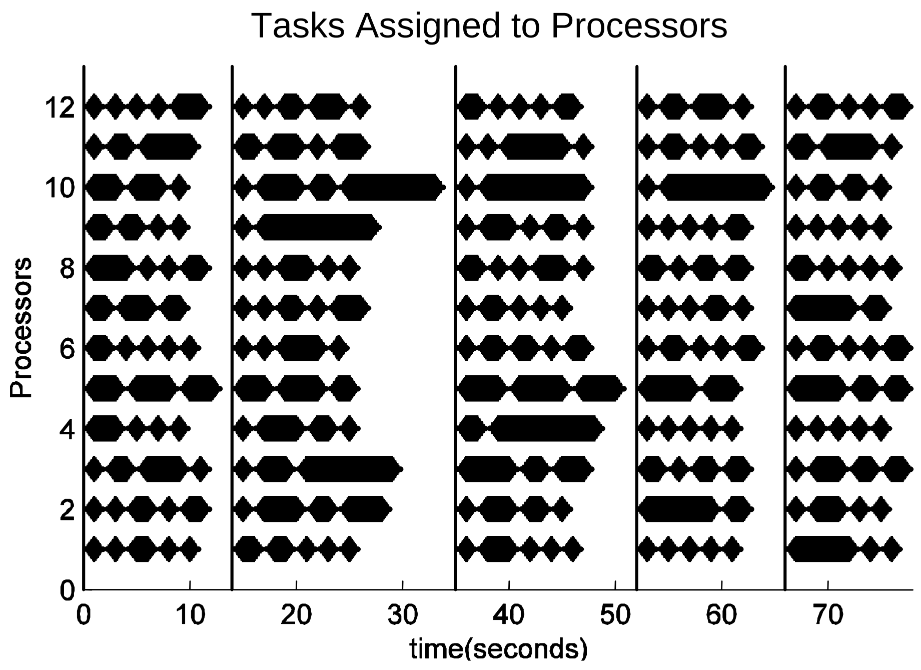


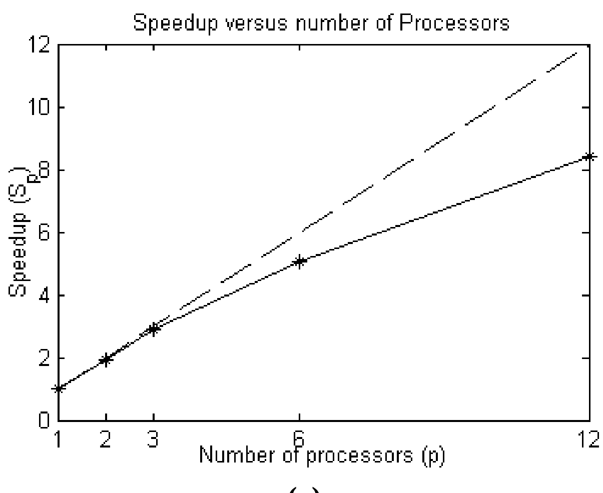
Figure 4.3: Load of 12 processors during 5 generations of evolution.

4.3.1 Experimental Set-up for Testing PES

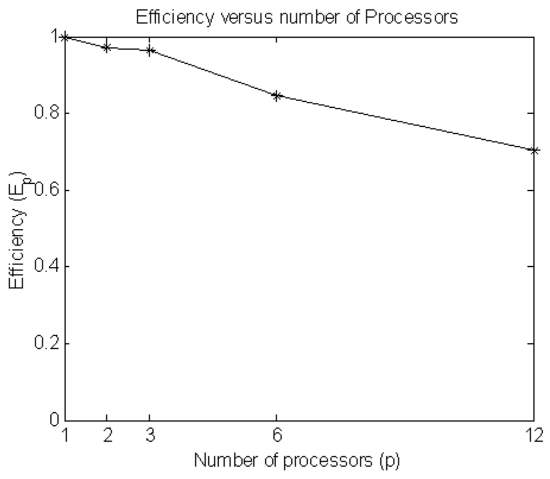
To test the performance of PES, we installed PES-Clients on 12 PC's of a student laboratory at the Computer Engineering Department of Middle East Technical University, Ankara, Turkey. During the experiment, these computers were being used by other users and each of them had different workloads that varied in time. The population size was set to 48, requiring 48 fitness evaluations to take place during each generation. The evolution was run for 30 generations.

Figure 4.3 plots the load of the 12 processors in time during the evaluation of five generations. The PES-Server waits for fitness evaluations of all the individuals in a generation before selection and reproduction of the individuals of the next generation. In the plot, the vertical lines separate the generations. Within each generation, 48 fitness evaluations are calculated, which are visible as dark diamonds or dark horizontally stretched hexagons. It can be seen that the fitness evaluation time varies between different cases. There are two major causes of this. First, each processor has a different and varying workload depending on the other users of that computer. Second, the physics-based simulation of the swarm of robots slows down dramatically as robots collide with each other and the walls in the environment.

In order to analyze the speed-ups achieved through the use of the PES system and its efficiency, we have repeated the evolution experiment using 1, 2, 3, 6 and 12 processors. The



(a)



(b)

Figure 4.4: **(a)** Speed-up is plotted against number of processors. **(b)** Efficiency is plotted against number of processors.

data obtained is used to compute the following two measures:

$$S_p = \frac{\text{Time required for single machine}}{\text{Time required for p machines}}$$
$$E_p = \frac{\text{Speed-up with p processors}}{\text{p}}$$

The results are plotted in Figure 4.4(a,b). Ideally S_p should be linear to number of processors and E_p should be 1. The deviance is a result of the requirement that all individuals need to be evaluated before moving on to the next generation. As a consequence of this, after the dispatching of the last individual in a generation, all but one of the processors have to wait for the last processor to finish its evaluation. This causes a decrease in the speed-up and efficiency plots. Note that, apart from the number of processors, these measures also depend on two other factors: (1) the ratio of total number of fitness evaluations in a generation to the number of processors, (2) the average duration of fitness evaluations and their variance.

Earlier in this chapter, we had described a ping mechanism that was implemented to check whether a processor has crashed or not. This mechanism was crucial since we envision PES to harvest idle processing powers of existing computer systems and cannot make assumptions about the reliability of the clients. Figure 4.5 shows the ping mechanism at work during an evolution where we had 20 fitness evaluations in each generation run on 4 processors. Similar to the plot in Figure 4.3, this plot shows the loads of the processors. The numbers in the hexagons are labels that show the number of the individuals being evaluated. The continuous vertical bars separate the generations. the dotted vertical lines that are drawn at

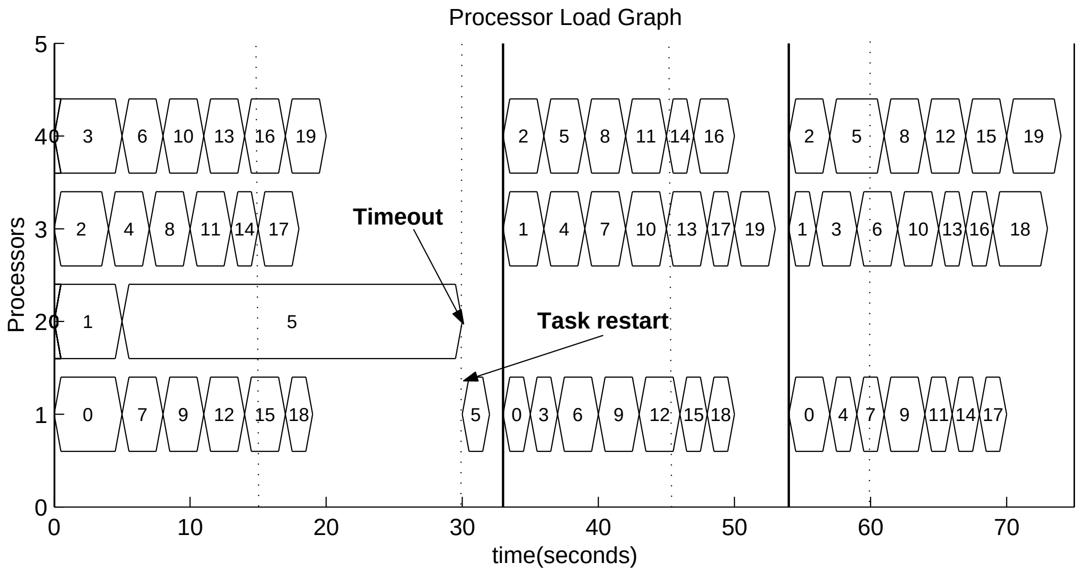


Figure 4.5: Load of processors during a run in which a processor fails.

15, 30, 45, and 60 seconds mark the pings that check the status of the processors. In this experiment, processor 2 crashed on while it is evaluating individual 5 after the first ping. PES-Server detected this at the second ping (at time 30), assigned the evaluation of individual 5 to processor 1, and removed processor 2 from its list.

For the actual experiments of this thesis that aim to systematically analyze aggregation performance of evolved controllers PES was installed on a cluster of 128 computers provided by TÜBİTAK ULAKBİM. Using this cluster with PES, we could reduce evolution time from one week to 2.5 hours with 100 computers.

Furthermore, to run simulations for scalability evaluation in parallel on this cluster of computers, we derived a program (called PES-Dist) from PES. This program is a tool that utilizes PVM to distribute execution of a series of programs if the complete command line for each program to be run is supplied.

CHAPTER 5

Experimental Framework

5.1 Introduction

Regarding evolutionary methods in developing controllers for swarm robotic studies, there are some parameters that should be considered, such as the number of generations, the number of simulation steps used for fitness evaluations, number of robots, and size of arena. We performed four experiments that altered some parameters and compared the results for different choices of parameter values.

Tasks constituting each of the four experiments are shown in Figure 5.1. An evolution suite (dashed box) consists of an evolution and a scalability evaluation for each choice of parameters. An evolution, shown as the box shaded in light gray, is run to produce a controller for each specified choice of parameter values. This box is enlarged in Figure 5.2 and explained in detail in Section 5.2.

The evolved controller is then analyzed for its scalability performance, i.e., performance in different sized set-ups to measure its *scalability*. This step that is shown as the box shaded in dark gray in Figure 5.1 is enlarged in Figure 5.6 and described in Section 5.3. The result of the dashed box is a set of scalability performances, one for each parameter choice.

The genetic algorithm, in a way, does an adaptive random search over the solution (in this case controller) space and it is not guaranteed to find the optimal solution. In this study, the best performing controller at the final generation of evolution is taken as the solution produced by an evolution. However, there may be better performing solutions that the genetic algorithm has not discovered. Also variance in performance of controllers in simulation affects the observed fitness of a controller. A controller does not get the same score in two different simulation runs with different initial random robot distributions. Therefore, *in a single evolution*, if a chosen value i of a specific parameter produces a better performing controller

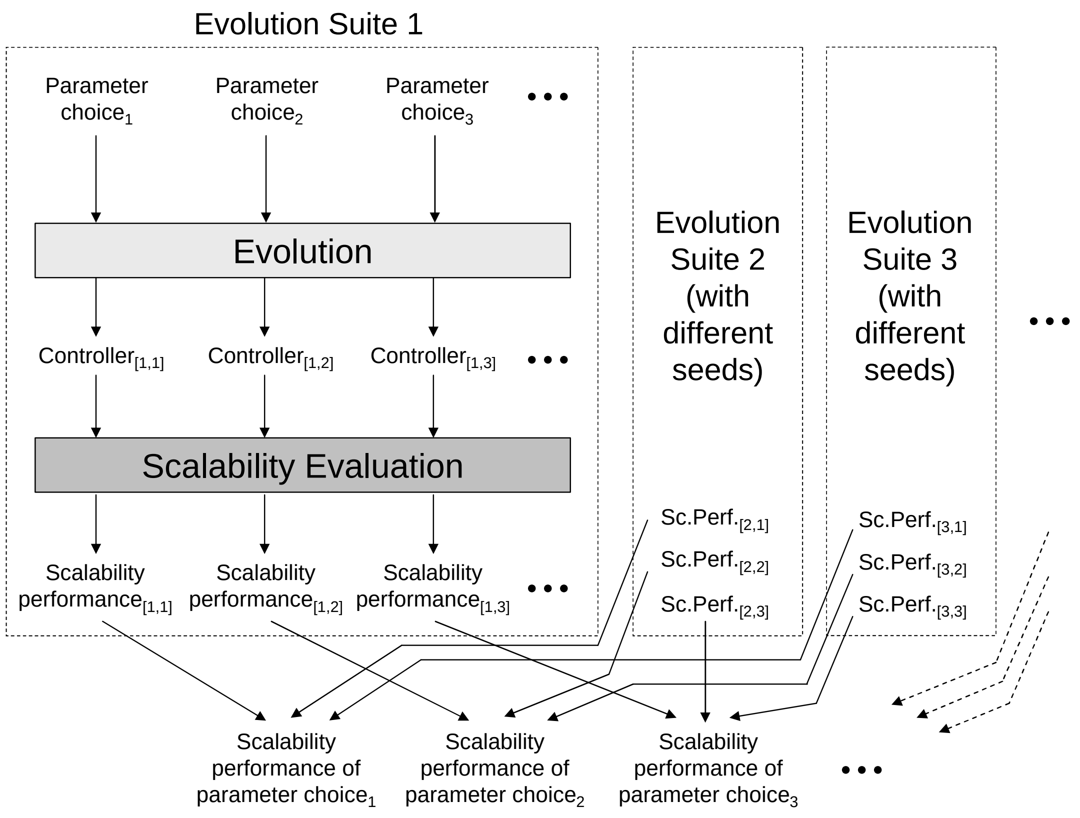

Evolution Suite 1
Parameter choice$_1$
Parameter choice$_2$
Parameter choice$_3$
Evolution
Controller$_{[1,1]}$
Controller$_{[1,2]}$
Controller$_{[1,3]}$
Scalability Evaluation
Scalability performance$_{[1,1]}$
Scalability performance$_{[1,2]}$
Scalability performance$_{[1,3]}$
Evolution Suite 2 (with different seeds)
Sc.Perf.$_{[2,1]}$
Sc.Perf.$_{[2,2]}$
Sc.Perf.$_{[2,3]}$
Evolution Suite 3 (with different seeds)
Sc.Perf.$_{[3,1]}$
Sc.Perf.$_{[3,2]}$
Sc.Perf.$_{[3,3]}$
Scalability performance of parameter choice$_1$
Scalability performance of parameter choice$_2$
Scalability performance of parameter choice$_3$

Figure 5.1: Flow of operations in the experiments. Evolution uses a specified set of parameters to produce an evolved controller.

than a controller produced by a chosen value j, this does not necessarily mean that i is a better value choice for this parameter than j. This may lead to results that are not so strong.

Since this study aims to derive rules of thumbs for evolutionary robotics, the credibility of results regarding the choice of parameter values should be as high as possible. To accomplish this, more than one evolution with different random seeds (shown as multiple dashed boxes in Figure 5.1) is carried out for each parameter value choice. The scalability performances for each parameter choice are then combined by averaging, shown at the bottom of Figure 5.1. This thesis extends [39] in this sense by conducting four evolutions for each parameter value choice instead of only one.

5.2 Genetic Algorithm and Evolution Scheme

5.2.1 Genetic Algorithm Details and Parameters

The genetic algorithm used in this study is shown and described in Figure 5.2. This genetic algorithm is run with a population of $p = 50$ chromosomes. Fitness evaluation of a single controller that is shown in the box shaded with a gradient is enlarged in Figure 5.5 and is described in detail in Section 5.2.4.

5.2.2 Encoding of the Robot Controller

The connection weights of the perceptron seen in Figure 3.3 (plus bias weights for output neurons) are encoded as $3 \times 12 + 3 = 39$ floating point numbers on a chromosome, or a population member.

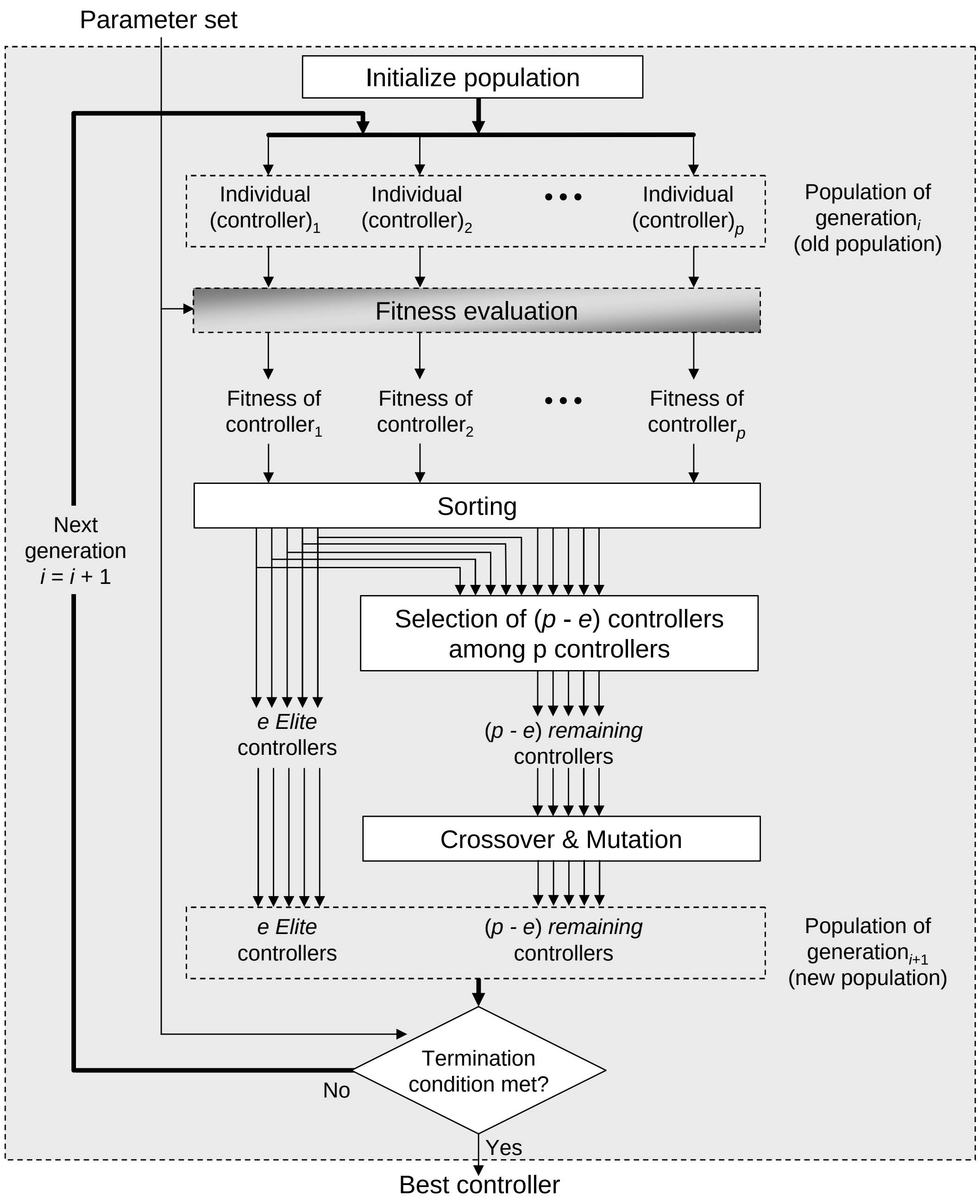
5.2.3 Genetic Operators

Tournament selection was chosen as the selection method because of its simplicity. After selection, crossover is applied to the members of the population with a probability of 0.8. The mutation method used is defined as choosing one weight out of all 39 weights on the chromosome and adding a random value uniformly in $[-1.0, +1.0]$ range. Each chromosome is subjected to this type of mutation with a probability of 0.5. This means that each network weight in the population has a mutation probability of $\frac{0.5}{39}$. At each generation, depending on their fitness, the best 10% ($e = 5$) of the population is copied unchanged to the next generation, i.e., elitism, together with the rest, which is the result of selection, crossover, and mutation operations.

5.2.4 Fitness Evaluation

For the genetic algorithm to function correctly, the chromosomes should be evaluated for their fitness, i.e., in our case, how good the encoded controller performs aggregation. Aggregation quality can be assessed in several ways. One way is to compute sum of the distances of each pair of robots. This measure gives smaller values as the robots get closer to each other. However, we chose another aggregation measure, which counts the robots in the formed clusters and computes the fitness as the size of the largest cluster with respect to the whole group, because this measure is a direct method of calculating what percent of the robots have clustered together.

In order to do the evaluation, the perceptron defined by that particular chromosome is replicated as the controller for all the robots in the swarm, and the swarm robotic system is simulated for a certain number of steps. At the end of a simulation run, sizes of clusters are computed. This is done as follows.



Parameter set
Initialize population
Individual (controller)$_1$
Individual (controller)$_2$
Individual (controller)$_p$
Population of generation$_i$ (old population)
Fitness evaluation
Fitness of controller$_1$
Fitness of controller$_2$
Fitness of controller$_p$
Sorting
Next generation $i = i + 1$
Selection of (p - e) controllers among p controllers
e Elite controllers
(p - e) remaining controllers
Crossover & Mutation
e Elite controllers
(p - e) remaining controllers
Population of generation$_{i+1}$ (new population)
Termination condition met?
No
Yes
Best controller

Figure 5.2: The genetic algorithm in detail. It starts with a population of size p initialized randomly. Each individual, which encodes a controller, is evaluated for its fitness. Using the computed fitness values, the controllers are sorted in descending order. The top e controllers form the *elite* group and go into the new population untouched. Among the whole population, a set of $(p - e)$ controllers are selected with tournament selection and are subjected to crossover, then mutation to give $(p - e)$ new controllers. The resulting $(p - e)$ controllers, together with the elite controllers form the new population of size p. If the termination condition is not met yet, the new population goes through the same process again. Otherwise, the best performing controller in the last fitness evaluation is accepted as the solution. The given parameter set is used in fitness evaluation and the termination condition.

Robots i and j are referred to as neighbors if the $Neighbor(i, j)$ relation, defined in Equation 5.1, is true. Also, the two robots are in the same cluster, or aggregate or group, if the $Connected(i, j)$ relation, defined in Equation 5.2, is true. $Connected(i, j)$ is actually the ***transitive closure*** of the $Neighbor(i, j)$ relation. Transitive closure is computed using Warshall's algorithm, which has $O(n^3)$ complexity over the number of robots [40].

$$Neighbor(i,j) = \begin{cases} true & \text{if distance between} \\ & i \text{ and } j \leq 4 \text{ cm} \\ false & \text{otherwise} \end{cases} \tag{5.1}$$

$$Connected(i,j) = \begin{cases} true & \text{if there is a path from} \\ & i \text{ to } j \text{ over the} \\ & \text{relationship } Neighbor \\ false & \text{otherwise} \end{cases} \tag{5.2}$$

Using the transitive closure, each robot is assigned to a cluster, while calculating the size of clusters. This is done with the algorithm, shown in Figure 5.3, of $O(n^2)$ complexity over the number of robots. The primary purpose of this algorithm is to determine the largest cluster l. The aggregation performance, or $fitness$, of a single evaluation run is defined as $\frac{size(l)}{n_{robots}}$, i.e., ratio of size of the largest cluster to the number of all robots, where $size(c)$ is the number of robots in cluster c.

Different initial positions of robots in the arena lead to a significant bias for the resulting aggregation performance, as seen in Figure 5.4. Therefore, a fair evaluation of different controllers requires multiple performance evaluation simulations per controller, each starting with a different random initial placement. The number of simulations per controller will be called n_{runs} from now on.

The fitness of a chromosome is defined as in Equation 5.3.

$$Fitness = F_{combine}(fitness_1, ..., fitness_{n_{runs}}) \tag{5.3}$$

where $F_{combine}$, *fitness combining function*, is used to join the fitness values of n_{runs} simulation runs done for a single chromosome. These runs differ in their randomization seed, which determines the initial placement of robots. $fitness_i$ in this equation refers to the fitness value of a simulation run with the i^{th} random seed. In this study, the $F_{combine}$ function is one of the parameters altered in experiments and is chosen among the following functions: *average*, *median*, *minimum*, and *maximum*.

```
for all robot $r$ do
    for all cluster $c$ do
        if $Connected(r,$ first robot of $c)$ then
            Assign robot $r$ to cluster $c$
            Increment $size(c)$
            if cluster $l$ not initialized OR $size(c) > size(l)$ then
                $l \Leftarrow c$
            end if
            Continue with next robot $r$ in outer loop
        end if
    end for
    Create new cluster $c'$
    Put robot $r$ into cluster $c'$
    $size(c') \Leftarrow 1$
    if cluster $l$ not initialized then
        $l \Leftarrow c'$
    end if
end for
```

Figure 5.3: Algorithm to determine the largest cluster.

5.2.5 Arena Set-ups for Evolution

Simulations involve robots that are initially randomly distributed in a closed square arena. Arena sizes that are used in the evolutions are 110×110 cm, 140×140 cm, 200×200 cm, and 282×282 cm. Initial positions and orientations of robots are random and are determined using the random seed coming from the genetic algorithm.

5.3 Scalability Evaluation

During scalability analysis, each evolved controller is tested with 50 different seeds on 5 different set-ups, which are all set-ups used for evolution plus a 400×400 cm arena, shown in Table 5.1. In all these set-ups the robot density over the arena is kept the same. The number of simulation steps is increased in larger arenas to allow more time for aggregation.

The results, i.e., final largest cluster ratio of robots, obtained from the 50 runs are averaged and yield the result for a single controller and a single evaluation set-up. We had mentioned 4 different evolutions for each parameter value choice. Each one of the 4 controllers produced by these evolutions are evaluated for their scalability in the same manner, on the 5 different evaluation set-ups. The results are plotted for each evaluation set-up and each different parameter value choice averaged over the 4 evolution suites. The *scalability metric* we use

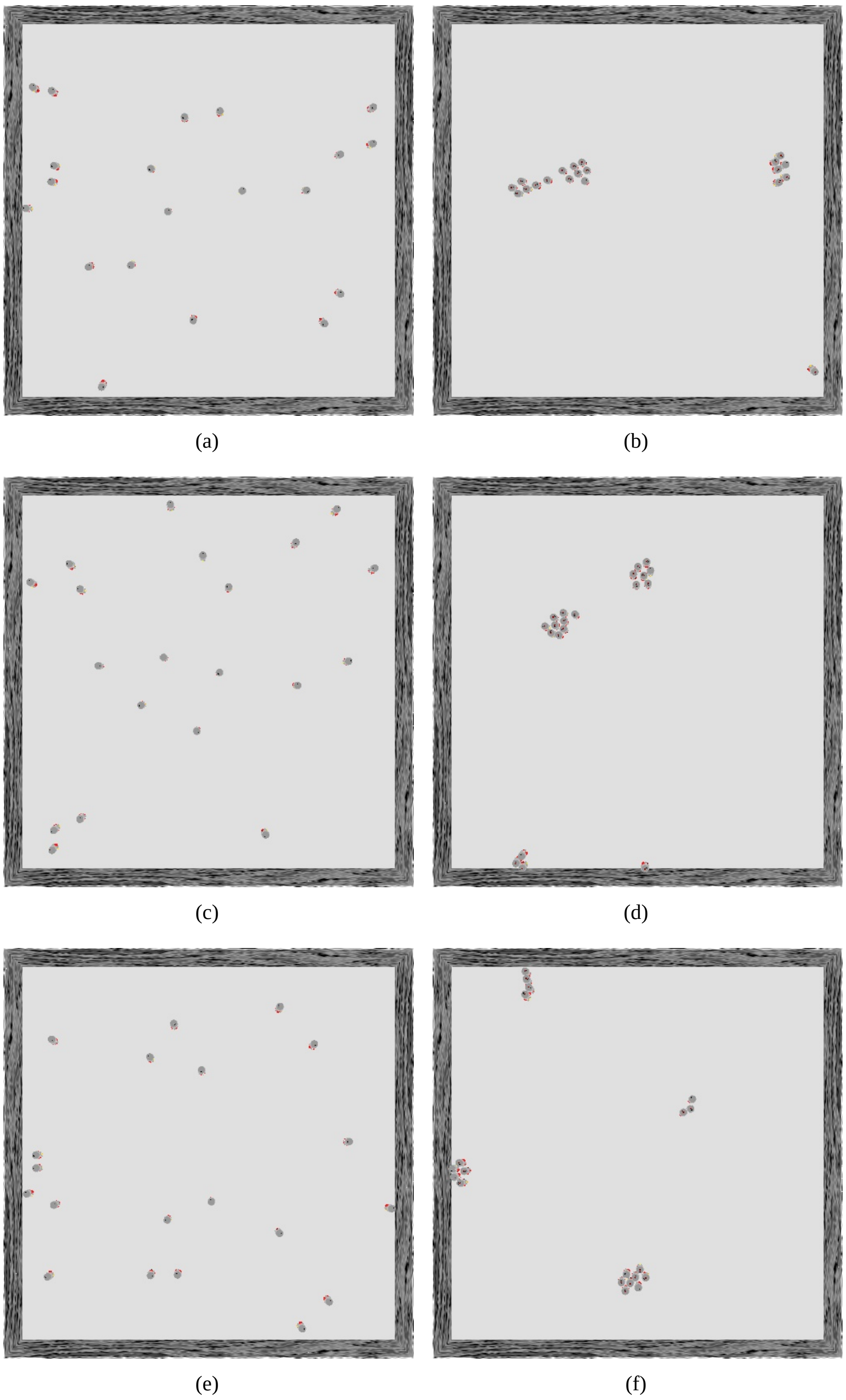


Figure 5.4: Different initial positions and their consequences. The pairs of images (**(a)-(b)**, **(c)-(d)**, and **(e)-(f)**) show the state of robots at steps 1 and 6000 for three different seeds using the same controller.

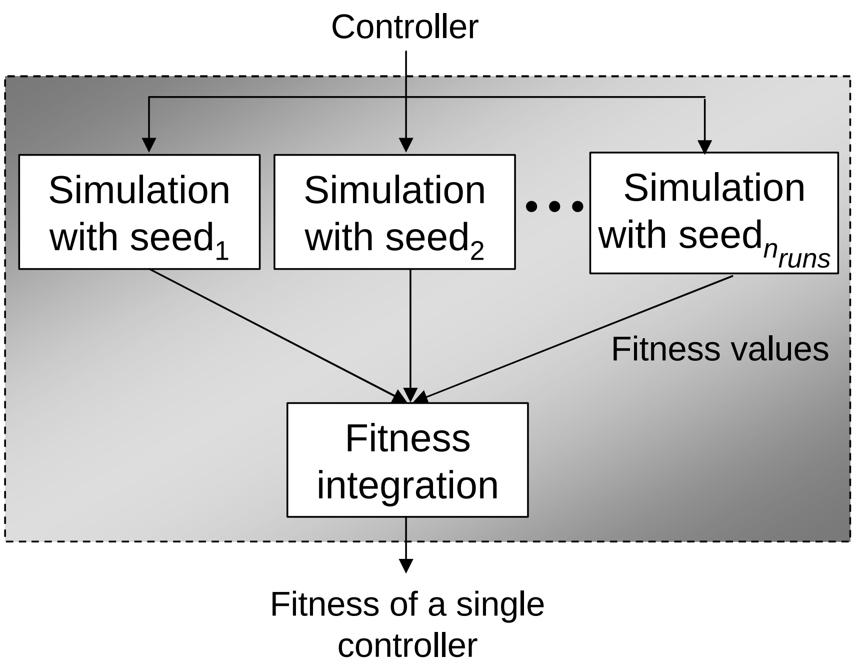


Figure 5.5: Fitness evaluation of a single controller in the genetic algorithm. The same controller is used in n_{runs} different simulation runs with different initial random robot placements and the results are combined into one single value. This value is used as the fitness of the controller.

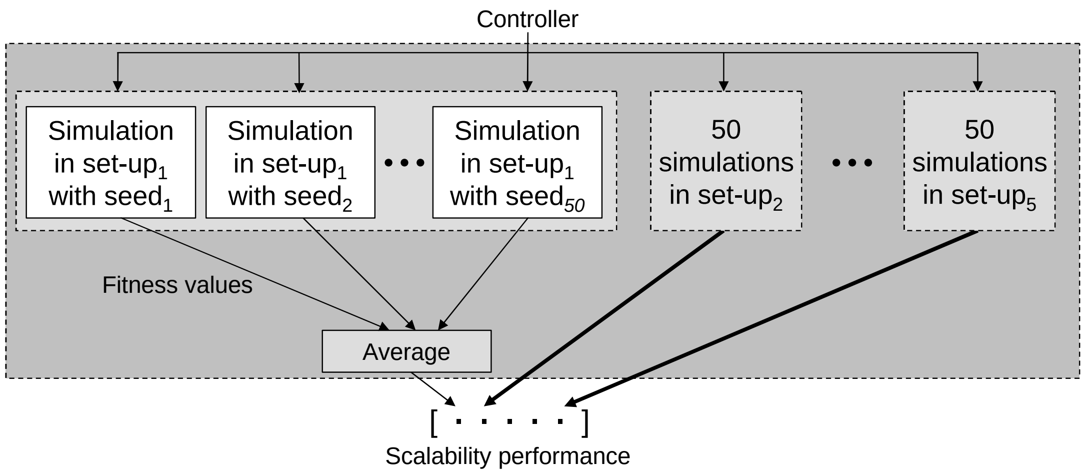

Controller
Simulation in set-up$_1$ with seed$_1$
Simulation in set-up$_1$ with seed$_2$
Simulation in set-up$_1$ with seed$_{50}$
50 simulations in set-up$_2$
50 simulations in set-up$_5$
Fitness values
Average
[· · · · ·]
Scalability performance

Figure 5.6: Scalability evaluation of the given controller is done by running each controller on 50 times on each of the 5 different sized set-ups and averaging the results.

is simply a vector of 5 numbers that are the average fitness values for 5 different scalability set-ups.

The numbers given above mean 50 *runs* $\times$ 5 *set-ups* $\times$ 4 *evolutions* $=$ 1000 *evaluation simulation runs* for each parameter value choice and 1000×4 *parameter alternatives* $= 4000$

Table 5.1: Set-ups used for scalability evaluation.

set-up	**# Robots**	**Arena Size**	**# Simulation Steps**
1	3	110×110	3000
2	5	140×140	6000
3	10	200×200	9000
4	20	282×282	12000
5	40	400×400	15000

total scalability evaluation runs in a single experiment. As one can realize, this is huge amount of computation for a single computer to overcome. Hence, just like we parallelized fitness evaluations of the genetic algorithm, we also distributed these simulation runs with PES-Dist mentioned in Chapter 4. These simulation runs were also executed on the cluster of ULAKBİM.

CHAPTER 6

Experiments and Results

Our main purpose in the experiments is to derive hints, in evolutionary swarm robotics, to how the available limited processing time should be utilized for evolving a desired robotic behavior. Should the available time be given to more generations, to more runs per controller, or to longer simulations for fitness evaluation? Choices of some parameters that do not affect total evolution time were also investigated.

With the experimental framework described in the previous chapter, we conducted four experiments to investigate the effect of different parameters on performance and scalability of evolved aggregation behaviors. Parameters altered in the experiments can be seen in Table 6.1. The specifically investigated parameters in each experiment are shown in Table 6.2.

Due to long computation times for simulations, for each parameter, a limited number of values could be investigated, which can be seen in Table 6.2. Different ranges and more values for parameters could also be considered, which would extend the results of this study.

Table 6.1: Parameters altered in evolution experiments.

Parameter Name	**Parameter Description**
$F_{combine}(\cdot)$	Fitness combining method
n_{runs}	Number of simulation runs per controller
n_{steps}	Number of simulation steps
n_{gens}	Number of generations in evolution
Set-up Size	Set-up size in terms of number of robots, arena size, and number of simulation steps

In the first three experiments total number of simulation steps used in evolution was kept constant. This roughly corresponds to keeping the total amount of processor time constant[1].

The first experiment considered effects of changing a single parameter, while the second

[1] Roughly, because two simulations that run for S simulation steps do not necessarily have equal execution durations due to longer execution when there are more collisions in simulation.

Table 6.2: Investigated parameters in evolution experiments.

Experiment	**Investigated parameters**
1	$F_{combine}(\cdot)$
2	n_{steps} vs. n_{runs}
3	n_{gens} vs. n_{runs}
4	Set-up Size

and third experiments specifically investigated trade-off's between the parameters shown in Table 6.1 and the fourth experiment altered set-up size used in evolution. In the trade-off experiments, the aim was to answer the set of questions "If I have parameters that cause longer evolutions when increased, should I use the limited amount of CPU time for increasing parameter A and decreasing parameter B, or vice versa?" In other words, the question is "which parameter is worth increasing, A or B?"

Table 6.3: Constant and variable parameters in the experiments are shown. Variable parameters are given as different values that the variable is assigned. Experiments 2, 3, and 4 used median as the $F_{combine}$ function, because initial trials had shown that median caused slightly better performance. However, this was observed to be incorrect when all results of experiment 1 were obtained as seen in Section 6.1.

Exp.	n_{robots}	**Arena size**	n_{steps}	n_{runs}	n_{gens}	$F_{combine}$
1	10	200×200	6000	5	50	avg., median, min., max.
2	10	200×200	18000, 6000, 3600, 1800	1, 3, 5, 10	50	median
3	10	200×200	6000	1, 3, 5, 10	150, 50, 30, 15	median
4	5, 10, 20	140×140, 200×200, 282×282	6000, 9000, 12000	3	50	median

6.1 Experiment 1: Fitness Integration Method

In Section 5.2.4 we had mentioned a *fitness combining function*, $F_{combine}$, that determines how the n_{runs} different fitness evaluations (each obtained from different initial robot positions) of a controller are combined. Due to a standard deviation that can be as high as 24% (Figure 6.1(a)) among fitness values obtained with the same controller, the function may influence the course of evolution and performance of resulting controller significantly. The first experiment is motivated by the question of how should the $F_{combine}$ function be chosen to obtain the best results. We examined the effect of fitness combining method, where the four functions, *average*, *median*, *minimum*, and *maximum* were used as $F_{combine}$.

The results, which can be seen in Figure 6.1(a), indicate that the four functions are sorted according to their performance in the order *minimum*, *maximum*, *median*, and *average*. Each one dominates, i.e., is better in all evaluation set-ups than, the next one in this order. The *minimum* function, which corresponds to pessimistic evaluation, is clearly the best of the four.

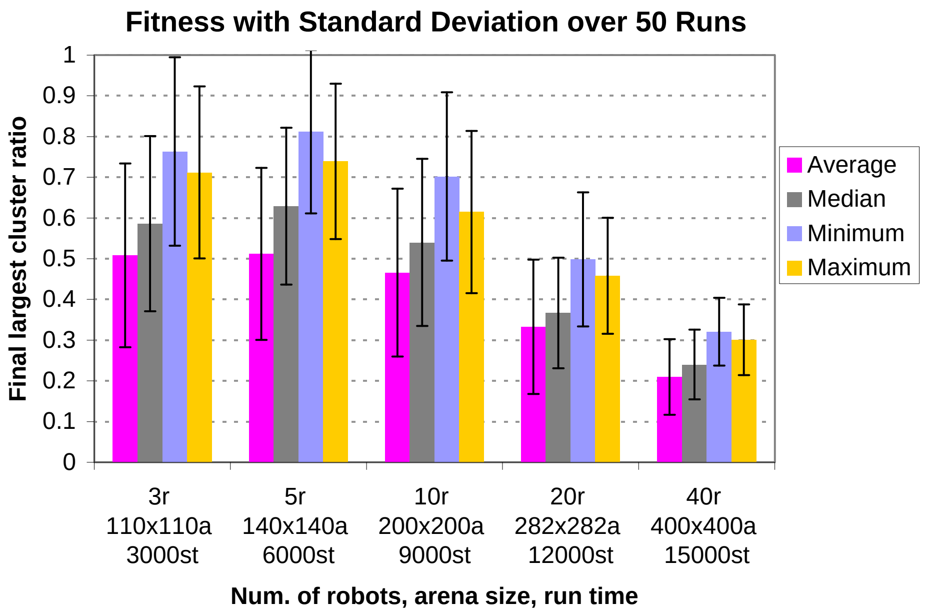
It is also worth mentioning that, considering the standard deviations among the scalability performances of 4 distinct evolutions (Figure 6.1(b)), the *minimum* function performed close in each of the 4 evolutions, i.e., showed small variance, as well performing the best. On the other hand, the *maximum* function, which is the second best in performance, showed significant variance among evolutions compared to the other functions. This means that although both seem to perform well, the *maximum* function is riskier to use in evolution than the *minimum* function.

One interesting observation would be that the functions that consider only extrema in fitness, i.e., *minimum* and *maximum* functions, performed better than the functions emphasizing all fitness values, i.e., *median* and *average* functions.

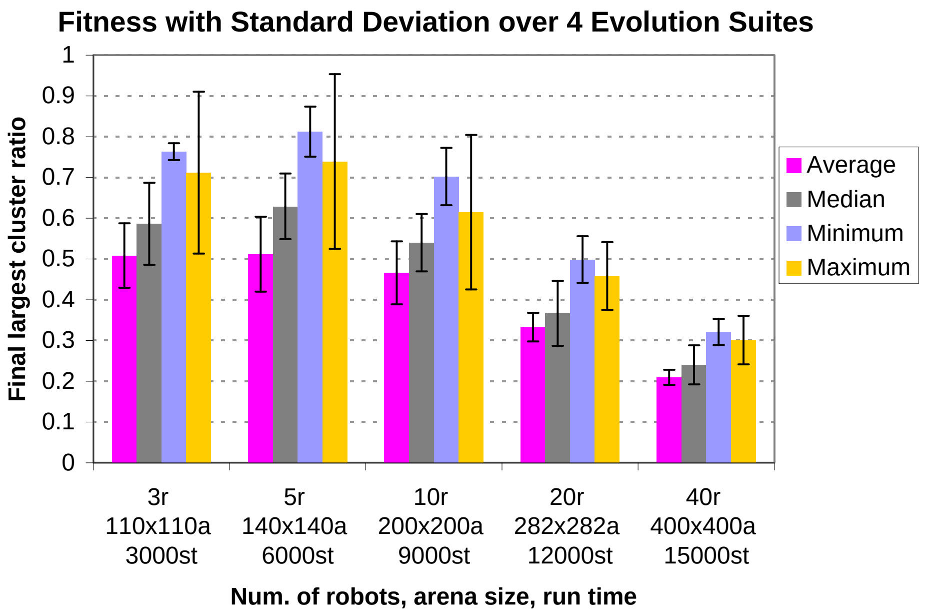
The behavior of one of the best controllers evolved in this experiment can be seen in Figure 6.2(a) and Figure 6.2(b). The evolved strategy can be described as "if no sound is heard then go straight; if there is wall then avoid it; if a sound is heard then approach the loudest sound source; but if the sound is very loud then turn on the spot". The emergent behavior of formed groups is to move slowly toward the loudest sound source. This can be seen in paths of groups in Figure 6.2(b), which are slowly going toward each other.

6.2 Experiment 2: Simulation Duration vs. Number of Runs per Controller

Choice of parameters that affect evolution time are hard to choose, since to alter one of them and to keep the total evolution duration constant, one needs to sacrifice another parameter. To shed light onto this situation, we considered a trade-off in the second experiment, between the number of runs per controller (n_{runs}) and the number of simulation steps in fitness evaluation (n_{steps}) while keeping the number of total simulation steps executed for a specific controller constant. Figure 6.3 shows a significant monotonous increase in performance as n_{runs} increases although while duration of simulation decreases. However, as n_{runs} is increasing,

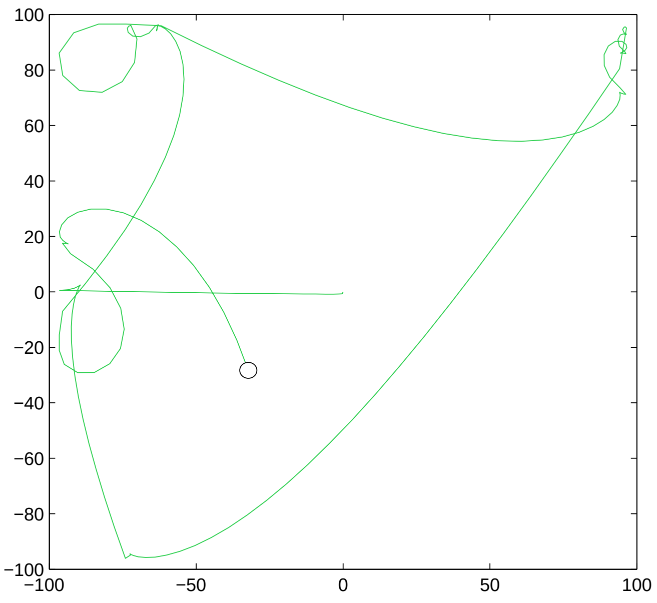


(a)

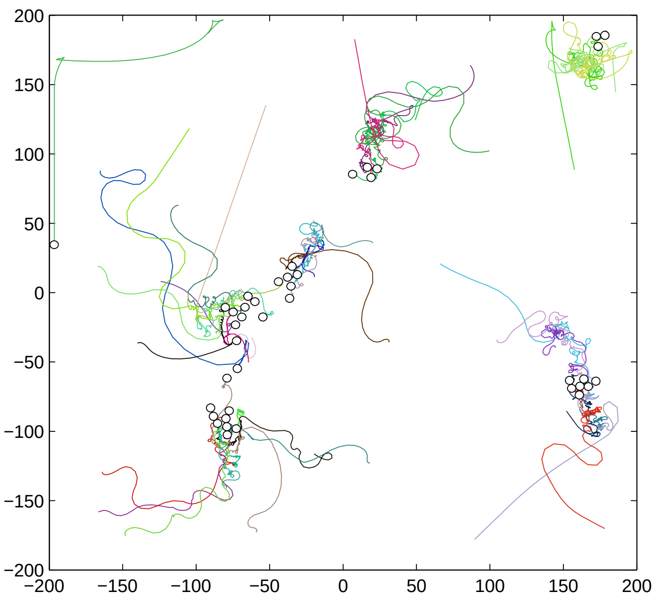


(b)

Figure 6.1: Results of experiment 1. Different integrations of fitness values of the same controller are compared. The y-axis shows final largest cluster ratios, i.e., aggregation performance, whereas the x-axis designates 5 different set-ups used to evaluate scalability performance of produced controllers by the evolutions depicted on the legend. The evaluation set-ups increase in number of robots, size of arena, and number of simulation steps from left to right. For each of the 5 setups on x-axis, **(a)** shows $mean(mean$ over 50 runs$)$ over 4 evolution suites with the error bars indicating $mean(std.dev.$ over 50 runs$)$ over 4 evolution suites. **(b)** differs only in error bars, which indicate $std.dev.(mean$ over 50 runs$)$ over 4 evolution suites.



(a)



(b)

Figure 6.2: The behavior of an evolved controller with **(a)** a single robot in an arena of size 200×200 after 10000 time steps, and **(b)** 40 robots in an arena of size 400×400 after 15000 time steps. Final positions of robots are shown as circles together with the paths they followed during the whole simulation run.

performance increase is slowing down, which can be seen as decreasing gaps between the lines in the plot. This is probably due to decreased n_{steps} in high n_{runs} evolutions,

This implies that the number of runs for a controller is more important than the number of simulation steps up to a certain level, where the gain coming from high n_{runs} is surpassed by the loss from low n_{steps}.

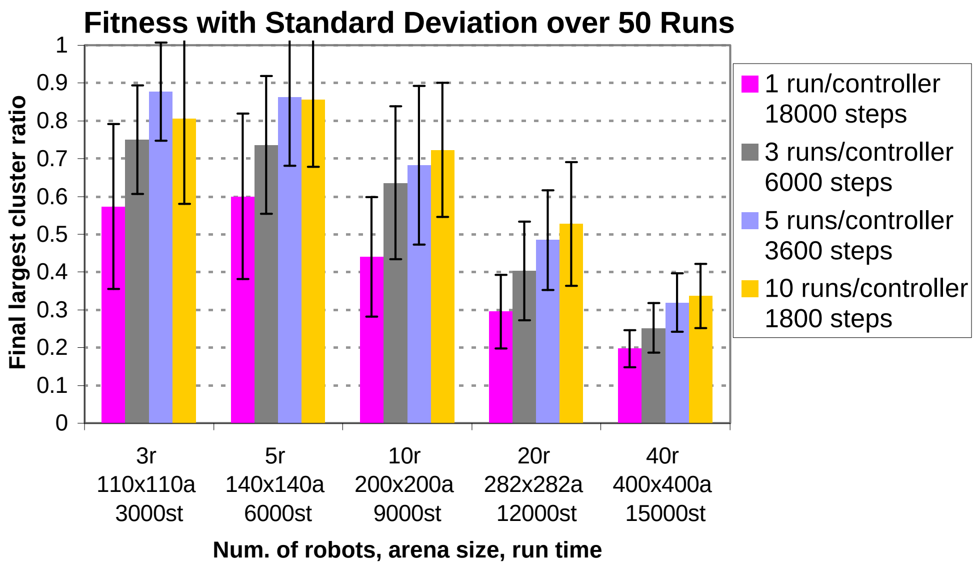


Figure 6.3: Results of experiment 2. The number of runs for the same controller and the number of simulation steps are varied while keeping total number of steps for a controller constant. Plot axes are the same as in Figure 6.1. For each of the 5 setups on x-axis, **(a)** shows $mean(mean$ over 50 runs$)$ over 4 evolution suites with the error bars indicating $mean(std.dev.$ over 50 runs$)$ over 4 evolution suites.

The degree of general drop in performance toward larger scalability evaluation arena setups in Figure 6.3 and the result plots of other experiments show the amount of deviation from perfect scalability, which would be shown as completely horizontal lines, i.e., no performance loss with bigger scales.

6.3 Experiment 3: Number of Generations vs. Number of Runs per Controller

Another trade-off we considered in total evolution time is between n_{runs} and number of generations (n_{gens}) while keeping the number of total simulation steps in the whole evolution constant. Our third experiment investigates the choice of parameters in this trade-off. In the experimented values for the parameters, as n_{runs} is increased n_{gens} is decreased, so that the total number of simulation runs done in the evolution is constant.

The resulting performance and variance plots are shown in (Figure 6.4). The resulting scalability performances are rather close, which is pretty surprising because it shows that the decrease in performance by decreasing number of generations (as suggested by the fact that if elitism is used in a genetic algorithm, performance increases or stays the same at consequent generations) is compensated by the increase in performance caused by the increase in n_{runs} (as shown in Experiment 2). Moreover, this balance of the two parameters seems to exist at the same parameter value ranges (that is $n_{gens} = [15, 150]$ and $n_{runs} = [1, 10]$) for all 5 set-ups of different size.

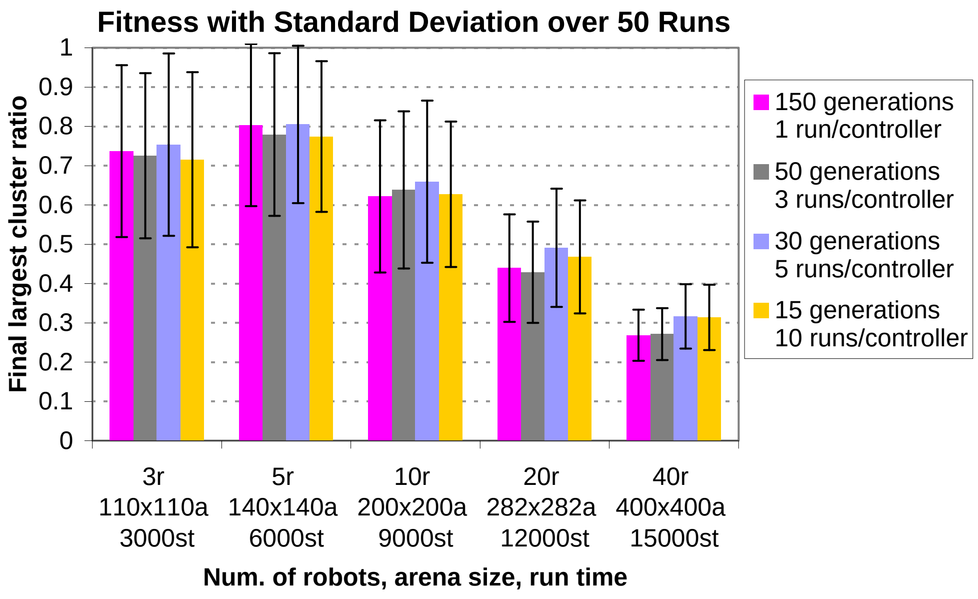


Figure 6.4: Results of experiment 3. The number of generations and the number of runs for the same controller are varied while keeping total number of steps constant. Plot axes are the same as in Figure 6.1. For each of the 5 setups on x-axis, **(a)** shows $mean(mean$ over 50 runs$)$ over 4 evolution suites with the error bars indicating $mean(std.dev.$ over 50 runs$)$ over 4 evolution suites.

6.4 Experiment 4: Set-up Size

Scalability is an important issue in swarm robotic systems, since it significantly affects the usefulness of a swarm robotic controller. Therefore, one might ask the question "how well does a controller, that is evolved using a set-up of certain size, perform on different sized set-ups?", or more generally "how does chosen set-up size in evolution affect the scalability performance of evolved controllers?". One would expect that each evolution produces a controller that runs best on its evolution set-up among controllers that are evolved on different sized set-ups. Is that really so?

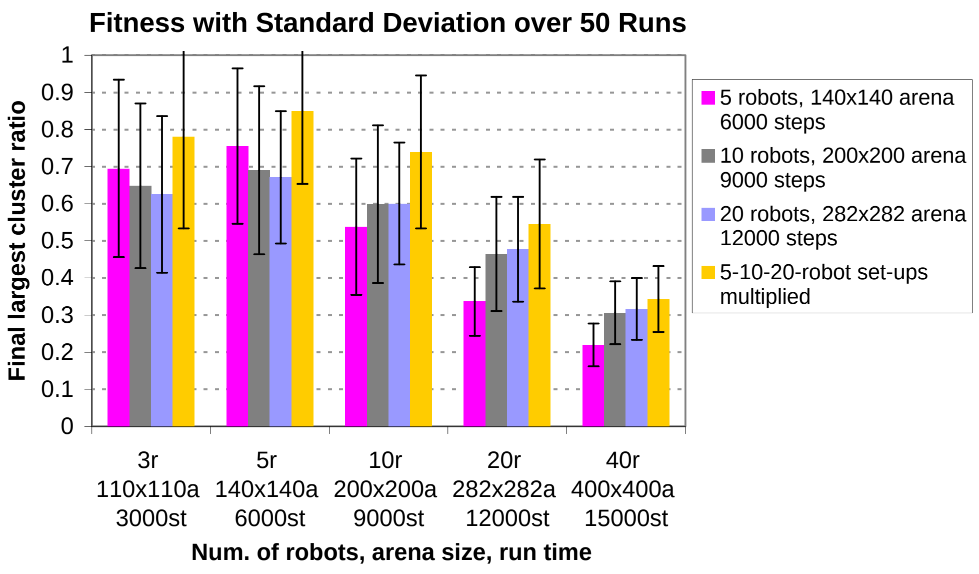
As explained in earlier chapters, in each evolution, controllers are evaluated for their fitness by running a simulation on a set-up of certain size (arena size, number of robots, and number of simulation steps combined). For evaluating the fitness of a controller, if we use multiple set-ups instead of a single one, would we be able to evolve a more *scalable* controller, i.e., a controller that performs higher in all different sized set-ups?

These questions were tackled in our fourth experiment. However, the type of scalability considered here does not involve solutions to *sparser* arenas, i.e., having lower *robot densities*, or shorter times to complete aggregation. If arena size and simulation duration were kept constant while number of robots was altered, the problem would change considerably. A sparser arena would mean that finding other robots would be a much more challenging task because of limited sensor ranges. The tactic required to aggregate in such an arena may be quite different from the one required in a less sparse one. It would also complicate the aggregation task if simulation duration was not increased together with number of robots. Aggregation in shorter times may require different strategies.

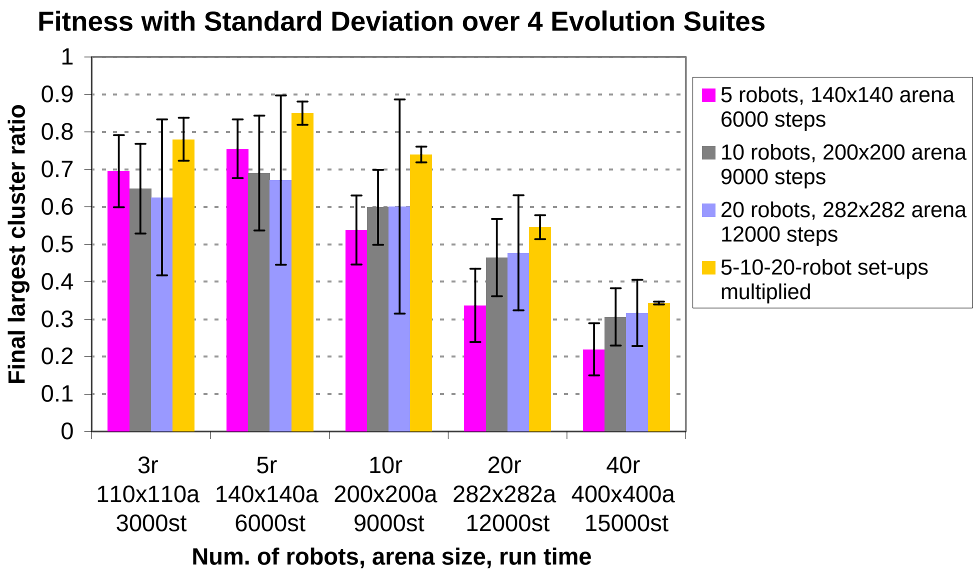
Therefore, the scalability we try to achieve is concerned with attempting the same task with different number of robots, while also altering the arena size to keep the robot density constant, and also simulation duration to allow enough time for the robots to find each other and aggregate. Thus, this experiment investigates the effect of set-up size, i.e., number of robots, arena size, and number of simulation steps together, on performance and scalability. Unlike the first three experiments, which were conducted to find out how to use total processing time most effectively, this experiment does not keep the total number of simulation steps in the whole evolution constant. Instead it analyzes how good controllers evolved with different set-ups perform on smaller/larger set-ups. It investigates which set-up size leads to better scalability and also whether using all set-ups in one evolution (where calculating fitness a controller by multiplying simulation results on different set-ups) improves overall scalability.

The results in Figure 6.5(a) show that, as expected, the multiple set-up evolution performed the best among the five evolutions and displayed the highest scalability. Also, the evolution with the smaller set-up (5-robot set-up evolution) showed higher performance in smaller evaluation set-ups (3-robot and 5-robot set-ups), and lower performance in larger evaluation set-ups (10-robot, 20-robot, and 40-robot set-ups) than the evolutions with bigger set-ups (10-robot and 20-robot set-up evolutions).

When we look at Figure 6.5(b), we see that multiple set-up evolution is not only the best performing evolution, but also the one with the least “variance among 4 evolutions”, hence



(a)



(b)

Figure 6.5: Results of experiment 4, where evolution set-up size, i.e. number of robots, arena size, and number of simulation steps, is varied. Plot axes are the same as in Figure 6.1. For each of the 5 setups on x-axis, **(a)** shows $mean(mean$ over 50 runs$)$ over 4 evolution suites with the error bars indicating $mean(std.dev.$ over 50 runs$)$ over 4 evolution suites. **(b)** differs only in error bars, which indicate $std.dev.(mean$ over 50 runs$)$ over 4 evolution suites.

the highest reliability among different evolutions. Furthermore, it is observed that 20-robot set-up evolution has the highest variance. This shows that if the 20-robot set-up is chosen for evolving controller, the resulting performance will be less predictable compared to evolutions on other set-ups. This may be caused by the size of the set-up, where even with a suitable controller for the task, 20 robots may end up at placements that have much more varying fitness the end of 12000 steps in a big arena with respect to smaller set-ups.

Among the three evolutions using only one set-up, each evolution produced a controller that performed best at its evolved set-up. The difference between the performance of a controller evaluated on the set-up that it is evolved on and performances of other two controllers becomes significant in the evaluation with the 20-robot set-up, where 20-robot set-up evolution scores much higher than 5-robot set-up evolution. However, the controller evolved in multiplied set-up evolution outperforms all others, even on their very own evolution set-ups. It is very interesting and can further be analyzed because the three evolutions get to have three evaluation simulation runs on their own set-up, while the multiplied set-up evolution gets to have only one on each of the three set-ups.

CHAPTER 7

Conclusions and Discussions

We studied how several parameters involved in using evolutionary methods in swarm robotic systems affect the performance and the scalability of behaviors. We chose the aggregation behavior as our case and made four systematic experiments. These experiments investigated trade-offs among number of runs per controller, number of generations in the genetic algorithm, and number of simulation steps to find out the most beneficial resource to dedicate processing time to. Furthermore, this study examined how to best merge fitness results obtained from simulation runs of the same controller with different seeds. Finally, one more experiment was done to better understand how evolution set-up size affects the scalability and performance on set-ups of different size.

In this study, we have made some assumptions regarding the problem domain in several aspects: the aggregation task, robot architecture, robot controller, and the genetic algorithm. For the aggregation task, we considered clustering of robots in a closed arena where a robot can perceive a portion of the arena. Therefore, the robots had no means of broadcast communication. However, they were able to hear a group of robots better than a single one at the same distance as seen in Figure 3.5. The controller of the robots was a single-layer perceptron, which means that the robots had reactive control, without any memory and acted without doing any planning. Furthermore, the robots were not capable of identifying each other. Also, the group of robots was homogeneous without a leader of any kind. This provides robustness for the system, which is one of the characteristics of swarm robotics.

There were also some assumptions on the genetic algorithm. The perceptron controller was encoded in the chromosome as a vector of floating point numbers with a mutation that adds or subtracts a random number with a probability of 1%. The size of the population used was 50, where the top 5 chromosomes were passed unaltered to the next generation. Tournament selection was used, where the selected chromosomes were subjected to crossover with 80% probability.

Based on the results obtained from the experiments conducted, we conclude the following rules of thumb, which can be accepted as true in the assumed domain we described above:

1. To combine results from different fitness evaluations of the same controller (to overcome bias from random initial distributions), the use of *minimum* function should be preferred over the functions *average*, *median*, and *maximum*.

2. When faced with the trade-off between the number of simulation steps for each run and the number of different runs per controller, one should choose the minimum sufficient number of simulation steps while maximizing the number of runs per controller. This will considerably eliminate negative effects of the high variance observed in robotics applications when initial positions are random.

3. The optimum value of the number of runs per controller and the number of generations (which is as important as number of runs) is not easy to obtain. Number of generations in evolution needed for emergence of a controller with acceptable performance, depends on architectural complexity of the controller and difficulty of the task. It is best to let the evolution run once initially for many generations to see about when the performance reaches a reasonable level.

4. In fitness evaluation, running simulations in multiple set-ups of different scale and multiplying the results improves scalability, since the controller is evaluated in multiple set-ups of different size, while being evolved. Also, evolving robot controllers on set-ups close in size to the actual set-ups that the robot is to be used will improve performance on that set-up. Therefore, using controllers evolved in set-ups too different than the application set-up is not recommended. Also evolving on a large set-up will increase variance of the evolved controller in performance among multiple evolutions.

We believe that these results, obtained through the systematic experiments, have a high chance of being relevant both for evolving other swarm robotic behaviors in simulation, and for evolving behaviors for physical robotic systems. Some of these rules of thumb are already employed earlier in the studies that evolve self-organizing behaviors by Dorigo *et al.* [16]. They use many runs per controller to obtain more reliable fitness evaluations (as suggested by item 2 above) and for each of these runs they use a random number of robots to obtain a more scalable controller (which supports item 4).

7.1 Future Work

This study can be extended by considering tasks other than aggregation and verifying the results on them. Also, more experiments can be carried out to investigate effects of other parameters which do not influence total run-time such as mutation-crossover rates, and different fitness measures. Moreover, experiments can be conducted to further explore optimal regions in trade-offs among parameters that affect total run-time, such as population size in the genetic algorithm, simulation run-time, and number of simulations for each chromosome. Finally, the results can be strengthened more by applying the evolved controllers onto physical robots and evaluating the performance and scalability with different number of robots and with arenas of different size or shape.

REFERENCES

[1] R. R. Murphy, *Introduction to AI Robotics*. Cambridge, MA, USA: MIT Press, 2000.

[2] K. Farry, I. Walker, and R. Baraniuk, “Myoelectric Teleoperation of a Complex Robotic Hand,” *IEEE Transactions on Robotics and Automation*, vol. 12, August 1996. Submitted.

[3] http://mpfwww.jpl.nasa.gov/MPF/rover/sojourner.html.

[4] D. Floreano and F. Mondada, “Evolutionary neurocontrollers for autonomous mobile robots,” *Neural Netw.*, vol. 11, no. 7-8, pp. 1461–1478, 1998.

[5] M. Mataric, “Issues and approaches in the design of collective autonomous agents,” *Robotics and Autonomous Systems*, vol. 16, pp. 321–331, December 1995.

[6] E. Şahin, “Swarm robotics: From sources of inspiration to domains of application,” in *Swarm Robotics Workshop: State-of-the-art Survey* (E. Şahin and W. Spears, eds.), no. 3342 in Lecture Notes in Computer Science, (Berlin Heidelberg), pp. 10–20, Springer-Verlag, 2005.

[7] M. Dorigo and E. Şahin, “Swarm robotics - special issue editorial,” *Autonomous Robots*, vol. 17, no. 2, 2004.

[8] E. Şahin and W. Spears, eds., *Swarm Robotics Workshop: State-of-the-art Survey*, vol. 3342 of *Lecture Notes in Computer Science*. Berlin Heidelberg: Springer-Verlag, 2005.

[9] R. C. Arkin, *Behavior-based Robotics*. Cambridge, MA, USA: MIT Press, 1998.

[10] http://www.exo.net/∼pauld/lectures/patternscostarica/patternsnature2004.htm.

[11] S. Camazine, N. R. Franks, J. Sneyd, E. Bonabeau, J.-L. Deneubourg, and G. Theraula, *Self-Organization in Biological Systems*. Princeton, NJ, USA: Princeton University Press, 2001.

[12] R. Jeanson, C. Rivault, J. Deneubourg, S. Blanco, R. Fournier, C. Jost, and G. Theraulaz, “Self-organised aggregation in cockroaches,” *Animal Behaviour*, vol. 69, pp. 169–180, 2005.

[13] O. Holland and C. Melhuish, “Stigmergy, self-organization, and sorting in collective robotics,” *Artificial Life*, vol. 5, no. 2, pp. 173–202, 1999.

[14] S. Nolfi, “Evolutionary robotics: Exploiting the full power of self-organization,” *Connect. Sci.*, vol. 10, no. 3-4, pp. 167–184, 1998.

[15] A. Martinoli, A. J. Ijspeert, and L. M. Gambardella, “A probabilistic model for understanding and comparing collective aggregation mechansims,” in *ECAL ’99: Proceedings of the 5th European Conference on Advances in Artificial Life*, (London, UK), pp. 575–584, Springer-Verlag, 1999.

[16] M. Dorigo, V. Trianni, E. Şahin, R. Groß, T. H. Labella, G. Baldassarre, S. Nolfi, J.-L. Deneubourg, F. Mondada, D. Floreano, and L. M. Gambardella, "Evolving self-organizing behaviors for a *swarm-bot*.," *Autonomous Robots*, vol. 17, no. 2-3, pp. 223–245, 2004.

[17] J. H. Holland, "Outline for a logical theory of adaptive systems," *J. ACM*, vol. 9, no. 3, pp. 297–314, 1962.

[18] J. H. Holland, *Adaptation in natural and artificial systems*. Ann Arbor, MI, USA: University of Michigan Press, 1992.

[19] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence through Simulated Evolution*. New York: John Wiley & Sons, 1966.

[20] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. NY: IEEE Press, 1995.

[21] J. Kennedy and R. C. Eberhart, *Swarm intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[22] I. Rechenberg, "Evolution strategies," in *Computational Intelligence: Imitating Life*, pp. 147–159, Piscataway, NJ: IEEE Press, 1994.

[23] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies - a comprehensive introduction," *Natural Computing: an international journal*, vol. 1, no. 1, pp. 3–52, 2002.

[24] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[25] N. Zaera, C. Cliff, and J. Bruten, "(Not) evolving collective behaviours in synthetic fish," in *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour*, (Cambridge, MA), pp. 635–6, MIT Press, 1996.

[26] M. J. Mataric and D. Cliff, "Challenges in evolving controllers for physical robots," *Journal of Robotics and Autonomous Systems*, vol. 19, pp. 67–83, Oct. 1996.

[27] H. H. Lund and J. Hallam, "Evolving sufficient robot controllers," in *Proceedings of the Fourth IEEE International Conference on Evolutionary Computation*, (Piscataway, NJ), pp. 495–499, IEEE Press, 1997.

[28] M. Dorigo, E. Tuci, R. G. V. Trianni, T. H. Labella, S. Nouyan, and C. Ampatzis, "The SWARM-BOTS project," in *Swarm Robotics Workshop: State-of-the-art Survey* (E. Şahin and W. Spears, eds.), no. 3342 in Lecture Notes in Computer Science, (Berlin Heidelberg), pp. 31–44, Springer-Verlag, 2005.

[29] G. Baldassarre, S. Nolfi, and D. Parisi, "Evolving mobile robots able to display collective behaviors.," *Artificial Life*, vol. 9, pp. 255–268, Aug. 2003.

[30] V. Trianni, R. Groß, T. Labella, E. Şahin, and M. Dorigo, "Evolving Aggregation Behaviors in a Swarm of Robots," in *Advances in Artificial Life - Proceedings of the 7th European Conference on Artificial Life (ECAL)* (W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, eds.), vol. 2801 of *Lecture Notes in Artificial Intelligence*, pp. 865–874, Springer Verlag, Heidelberg, Germany, 2003.

[31] C. R. Ward, F. Gobet, and G. Kendall, “Evolving collective behavior in an artificial ecology.,” *Artificial Life - Special issue on Evolution of Sensors in Nature, Hardware and Simulation*, vol. 7, no. 2, pp. 191–209, 2001.

[32] F. Mondada, G. C. Pettinaro, A. Guignard, I. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. Gambardella, and M. Dorigo, “Swarm-bot: A new distributed robotic concept,” *Autonomous Robots*, vol. 17, no. 2–3, pp. 193–221, 2004.

[33] J.-M. Valin, F. Michaud, J. Rouat, and D. Letourneau, “Robust sound source localization using a microphone array on a mobile robot,” in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 2, pp. 1228–1233, 2003.

[34] O. Soysal, E. Bahçeci, and E. Şahin, “PES: A system for parallelized fitness evaluation of evolutionary methods,” in *Proceedings of the Eighteenth International Symposium on Computer and Information Sciences (ISCIS)* (A. Yazici and C. Sener, eds.), vol. 2869 of *Lecture Notes in Computer Science*, pp. 889–896, Springer Verlag, Heidelberg, Germany, 2003.

[35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users’ Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA, USA: MIT Press, 1994.

[36] R. Hempel, “The MPI standard for message passing,” in *HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II*, (London, UK), pp. 247–252, Springer-Verlag, 1994.

[37] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

[38] E. Şahin, T. Labella, V. Trianni, J.-L. Deneubourg, P. Rasse, D. Floreano, L. Gambardella, F. Mondada, S. Nolfi, and M. Dorigo, “SWARM-BOTS: Pattern formation in a swarm of self-assembling mobile robots,” in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics* (A. El Kamel, K. Mellouli, and P. Borne, eds.), (Hammamet, Tunisia), pp. 145–150, Piscataway, NJ: IEEE Press, Oct. 6-9, 2002.

[39] E. Bahçeci and E. Şahin, “Evolving aggregation behaviors for swarm robotic systems: A systematic case study,” in *IEEE Swarm Intelligence Symposium*, (Pasadena, CA, USA), 2005.

[40] R. Sedgewick, *Algorithms in C*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

APPENDIX A

Parallelized Evolution System Manual

A.1 Installation

Parallelized Evolution System (PES) can be installed easily on both Linux and Windows (using Cygwin), as long as PVM is also installed, which is necessary for communications between server and client components. PES also requires remote services (namely rexec or rsh) to function. Remote services can be replaced by ssh as described in the documentation of PVM and ssh. Details of installation in both platforms are given in following subsections.

A.1.1 Linux Installation

Most recent Linux distributions come with the PVM library. Otherwise, the easiest way to install PVM is using its rpm distributions.

Remote services are also present in most of Linux distributions. Details of configuration are given in Section A.2.

The first step to install PES is to install PVM. For Linux machines, following the described process in packages is enough. Especially using rpm distribution facilitates the process.

A.1.2 Windows Installation

Installation on Windows is slightly more complicated than installation on Linux. On Windows, Cygwin is mandatory for PES to operate. Cygwin can be downloaded and installed from the official Cygwin web site[1]. Full distribution of Cygwin contains many useful utilities, but also uses great amount of drive space (around 1 GB).

In many distributed applications, the same executable program is run on each computer to be used for distributed processing. Therefore, compilers and development tools are not needed

[1] http://www.cygwin.com

on all computers, but only one, which will be used to compile the distributed application. Then, this executable can be transferred to the others, which require a minimal distribution of Cygwin. This distribution must contain a shell and *inetd* to work correctly. The *inetd* package contains remote call services. Packages that are dependencies of these must also be included.

After Cygwin installation, PVM must be compiled under Cygwin. Some recent distributions of PVM have some minor problems that cause compilation under Cygwin to fail. A working build can be found at the homepage of Kovan Research Laboratory[2].

A.2 Configuration

In all platforms, PVM requires two environment variables to be set: PVM_ARCH and PVM_ROOT. PVM_ARCH defines the architecture of the machine. The value that should be used can be learned by issuing the *pvmgetarch* command that comes with the PVM package. PVM_ROOT should contain the root path of the PVM installation.

Configuration of PES is independent of platform. However, PVM requires platform-specific issues to be addressed.

PES uses a single configuration file "Pesdef.txt". This file contains parameters used in the genetic algorithm. This configuration file is searched on the running directory of the server program. The *TaskManager* instance for the program reads this file and configures the program using the file. When the file is not present, default values are used. The parameters and their default values are listed on Figure A.1.

A.2.1 Linux Configuration

PVM requires remote services. Unless otherwise stated, remote services require password authentication. Password authentication is repetitive and clumsy to use with many machines. An alternative is rhosts authentication, which is simpler to use. The user should create a file named ".rhosts" in his/her home directory, which should contain a list of login names (together with the name of their servers) that are allowed to login to this account without password authentication. This file consists of lines composed of a server and login name pair. For further details on this file, see the manual page of *rlogin* command.

The last step of PVM configuration is to run the pvm daemon. This can be accomplished by executing the *pvm* command. This command runs the pvm console, which enables control

[2] http://www.kovan.ceng.metu.edu.tr

of PVM and the tasks that are running on it. The PVM console can be used to add/remove hosts at run time.

A.2.2 Windows Configuration

The *.rhosts* authentication mechanism is not supported in Cygwin. Instead, the “hosts.equiv” file can be used. This mechanism is more dangerous than using .rhosts since it allow the user to login to the system as any user. So this file should be prepared with care.

PVM daemon and console under Cygwin are similar to the Linux version. Cygwin provides a bash shell for this purpose. Cygwin must be configured as described in the PVM documentation. Cygwin environment is not available for remote calls, so environment variables should be added to the Windows environment. The simplest way to accomplish this is using the Windows registry scripts. A script for default directories can be also found at the Kovan web site.

A.3 Example

A simple example comes with the PES distribution. It features all basic functions of PES and demonstrates the implementation of PES-S and PES-C for a model XOR problem.

A.4 Frequently Asked Questions

1. **Where can I use PES?** PES is designed for problems where fitness evaluations are the dominating factor in complexity. It performs best on such problems, like problems attacked with evolutionary methods that involve long simulations as fitness evaluation.

2. **How many PC’s can PES work with?** PES currently assumes a maximum of 300 computers. In theory, it can work with arbitrary number of computers but it is tested with at most 128 PC’s.

3. **Is there any more information available on PES?** There is a PES technical report at the Kovan web site. It gives details on the performance of PES.

4. **Where is the documentation of the library?** It is included in the distribution under “doc” directory.

5. **Is PES free?** Yes. PES is provided as freeware.

6. **PES finishes execution with error -7, why?** This error signals that the executable file in clients can't be found. Check the "Pesdef.txt". See Figure A.1 for details on the contents of "Pesdef.txt".

7. **PES works too slowly, why?** The cause of this problem can be configuration. Check the hosts in the PVM environment. Type *pvm* to reach console. In this console, type *conf* to see the hosts available. Also, the PES server component displays available hosts on initialization. If the required hosts are not available, add them to the PVM host list as defined in the PVM documentation. If the problem persists, check "Pesdef.txt" for the location of the executable program in clients. See also Question 1.

Parameter	Type	Description
nproc	int	Number of PES-C's to use.
MAX_GENERATIONS	int	Max number of generations
POP_SIZE	int	Population size of genetic algorithm.
TASK_STEPS	int	Number of simulation steps. This can be used for simulated fitness functions.
pingtimeout	int	Timeout for PES-C's in seconds.
TASK_PER_EVAL	int	Number of cases to be tested for each fitness function evaluation.
SLAVENAME	string	Path of the PES-C executable in slave machines.

Figure A.1: Configuration file (Pesdef.txt) contents.