Computing and Informatics, Vol. 23, 2004, 575–596

DISTRIBUTED GENETIC ALGORITHM: A CASE-STUDY OF EVOLUTION BY DIRECT EXCHANGE OF CHROMOSOMES*

Aleš Kubík

Institute of Computer Science
Silesian University
Bezručovo nám. 13, 746 01 Opava
Czech Republic
e-mail: ales.kubik@fpf.slu.cz

Manuscript received 19 March 2003; revised 24 February 2004
Communicated by Patrick Brézillon

Abstract. It is considered a difficult task to design a controller from scratch for a robot in a dynamic environment. Evolutionary and genetic algorithms are frequently used to find a solution with desired properties. The evolution is supposed to run on one robot or agent processor. In this article we explore the possibility of dividing the genome among several robots. Robots exchange among each other successful controllers in the form of chromosomes that code for the weight vector in a neural controller. Thus; the evolution can be faster because it runs in a parallel manner over the whole robotic society, not in one robot. Second interesting point is the exchange of experience among robots. Robots can send each other parts of the controllers that they evolved on their own. They can learn behavioral strategies that other robots developed. We describe experiments done with small Khepera robots in the domain of a benchmark test for evolutionary robotics. We compare the behavior generated by one robot based on its individual evolution with the robot that profits from sharing the full set of chromosomes with the other robot.

Keywords: Distributed genetic algorithm, evolution, multi-agent system, experience exchange, computational cognitivism

* Author's research on the subject is partially supported by the Czech Science Foundation Grant No. 201/04/0528.

1 INTRODUCTION

It is not easy to find an optimal code for robot controllers from scratch. One of the most successful methods of programming robots is to design a mechanism which will evolve a controller that ensures a desired behavior. Evolutionary and genetic algorithms are suitable tools to achieve this [5, 4].

There are mainly two ways of evolutionary design for robots. Either the robot itself runs a genetic algorithm to produce a controller with desired behavioral properties (e.g. [3])[1], or the evolution is constrained by the robot-to-robot couplings in a multiagent scenario. This way it is possible to co-evolve different behavioral strategies concurrently by direct interactions among robots. The experiments with predator-prey co-evolution are well known(e.g. [11, 10]), and more attention is gained by cooperative scenarios in multiagent systems (see [12] for last achievements).

Another possibility is offered by the multi-robot solution where the evolution of controllers runs on the whole society of robots so that the genome is not constrained to one agent body. Here we have two basic design possibilities. Either the population of chromosomes is scattered over the multi-robot system and there is a central process that evaluates their performance. In this case each of the robots represents an autonomous process that tests only the part of the whole genome. In this case there is a need for coordination of evolutionary processes (ordering of chromosomes, crossover, mutation, etc.) and the central process has to allocate offspring chromosomes to robots. On the other hand there is no need of inter-robot communication. The process that runs all the evolutionary computing plays a central part and if it fails the whole system stops functioning immediately. Also, there is no possibility of sharing experience among robots.

The other solution is to create random sets of chromosomes on every robot and exchange successful population members among them. In this case we can go along without a central process but the robots have to communicate with each other and coordinate the exchange of chromosomes. This approach was taken by the authors in [13] or [14].

In our work we focus on the latter solution. We gain time savings due to a parallelism in controller evolution and robustness due to a lack of central process controlling robots. As a tradeoff there is a programming code and communication overhead present in this solution. As a testbed of our ideas, we have chosen collision avoidance representing almost benchmark behavior in experimental robotics. The experiments described in this work should be used as a starting point in the design of distributed genetic algorithm. Our aim was to design a simple solution that could be extended to other platforms. We focus on general applicability as well as depict more application-specific details. We also propose other improvements to our solution and extension to our work.

[1] A good overview of evolutionary robotics focusing on single robot scenarios represents the work [9].

The rest of the paper is organized as follows. In the following section we recall basic properties of genetic algorithms and distributed genetic algorithms and give information on relevant related work to our experiments. Section 3 contains basic information on robots and experimental settings we used in our work. In the next section (4) we present the experiment results. In Section 5 we discuss the result and problems that we encountered and propose extensions to our work. Section 6 summarizes and concludes.

2 GENETIC ALGORITHMS

Genetic algorithm is one of the most used search and optimization algorithm and in the last decade it is heavily used as an evolutionary learning method in applications of multiagent systems, artificial life, and robotics. On the theory of genetic algorithms and why they work see [5], or [4].

2.1 Basic Idea

The basic idea of a genetic algorithm is as follows. Suppose that we can represent potential solutions to a specified problem by strings of symbols organized in a population. Motivated by Darwinian evolution of species that try to survive under the pressure of environmental conditions, populations of strings evolve to form fitter generations under the constraints of a problem task. Strings of solutions can be thought of as genes (or genes carried by chromosomes)[2] that code some behavior representing a phenotype. Under evolutionary pressure only the fittest solutions survive and by means of reproduction and mutation give rise to solutions in subsequent generations. The process can repeat until good enough or even optimal solution is reached.

More precisely *population* P is a set of chromosomes denoted by

$$P = \{g_1, g_2, \ldots, g_p\} \in \{a, b, \ldots\}^k.$$

Chromosomes $\{g_1, g_2, \ldots, g_p\}$ are strings of length k composed of symbols from the set $\{a, b, \ldots\}$.[3] *Selection pressure* is represented by a *fitness function* that maps members of populations to positive real numbers:

$$f : P \to R_+.$$

The *population evolution* is a change of populations in a discrete fashion dependent on time t as a consequence of a reproduction operator R:

[2] In the sequel we will denote by chromosome a set of genes represented by strings.

[3] Traditionally chromosomes holding one gene are represented as binary strings, i.e. $P \in \{0, 1\}^k$. In our work chromosomes c_i are pairs of genes $c_i = \{g_{i_1}, g_{i_2}\}$, $i \in \{1, \ldots, 40\}$, where each gene is a string of natural numbers $g_{1,2} = \{n\}^8$, $n \in N, n \in (-30, 30)$.

$$P_{t+1} = R(P_t).$$

In evolution of generations, the key processes are *selection* of a subset of a population into an *intermediary generation*, *crossover* and *mutation* of its members leading to a new generation.

2.2 The Algorithm

The genetic algorithm can be described by following pseudocode:

```
function P_fin ← geneticAlgorithm(t_max, stopCriterion)
  begin
    t := 0;
    P_t ← initializePToRandomValues();
      evaluate(P_t);
      while ((t < t_max) and (not stopCriterion)) do
        begin
          t := t + 1;  P_interm := ∅;
          P_interm ← select(P_{t-1});
          P_t ← reproduce(P_interm);
          P_t ← mutate(P_t);
        end
      P_fin := P_t;
  end
```

2.3 Selection

Fitness function serves as a means to evaluate the performance of individual solutions. Evaluated chromosomes are selected to intermediary generation on a stochastic basis. There are several algorithms of selection used in the genetic algorithm.

Roulette wheel is a method of selection where the probability of survival for each solution is proportional to its fitness.

Elitism prefers only the portion of the best solutions ranked by their fitness.

Tournament selection is a method in which pairs of randomly selected chromosomes meet and the one with higher fitness will pass to an intermediary generation.

2.4 Recombination Operators

Members of an intermediary population are subject to recombination operators. This is due to three main reasons:

- in order to keep size of the next generation stable,
- to propagate features of successful individuals, and

- to keep diversity of solutions by introducing new features into the population.

Reproduction and *mutation* are standard recombination operators.

Reproduction of chromosomes proceeds in the following way. As chromosomes are selected they can directly pass to the next generation based on the given probability, or they can be *crossover*ed with another selected chromosome.

Crossover with *ncp* number of crossover points is an operator O_{cross}^{ncp} that maps a pair of chromosomes into a new pair of (offspring) chromosomes that hold part of the features (symbols) from their parents:

$$(g_1^{offsp}, g_2^{offsp}) = O_{cross}^{ncp}(g_i, g_j), i, j \in \{1, \ldots, p\}.$$

The point where crossover takes place is determined randomly and there can even be more than one crossover points. The application and the result of one-point-crossover can be seen in the following schema:

$$\begin{array}{lcl} a_1 \ldots a_i \mid a_{i+1} \ldots a_n & & a_1 \ldots a_i \; b_{i+1} \ldots b_n \\ & \overset{O_{cross}^1}{\longrightarrow} & \\ b_1 \ldots b_i \mid b_{i+1} \ldots b_n & & b_1 \ldots b_i \; a_{i+1} \ldots a_n. \end{array}$$

Selected or reproduced chromosomes can have their symbols mutated with the given probability. *Mutation* with *nmp* denoting number of mutated symbols[4] is a recombination operator O_{mut}^{nms} which changes one or more symbols in the chromosome string randomly:

$$(g^{mut}) = O_{mut}^{nms}(g^{offsp}).$$

An example of a 1-bit mutation of a chromosome represented by a 8-bit binary string can look like this:

$$10101010 \overset{O_{mut}^1}{\longrightarrow} 10111010.$$

2.5 Parallel Genetic Algorithm

Parallel genetic algorithm is an extension to traditional sequential genetic algorithm in that the population of chromosomes is partitioned to independent disjoint subpopulations. These evolve independently and exchange parts of their genome on a stochastic basis. What follows is a pseudocode of a decentralized version of a parallel genetic algorithm with no central process controlling the selection, reproduction and mutation of chromosomes in subpopulations.

function
$g_{opt} \leftarrow$ `parallelGeneticAlgorithm(`t_{max}`,` $stopCriterion$`,` $p_{exchange}$`)`
begin

[4] In our work we mutate natural numbers $n \in (-30, 30)$ in the string representing weight vector of a neural network connection.

```
    t := 0;
    for i := 1 to N do
        P_{i_t} ← initializePToRandomValues();
    while ((t < t_max) and (not stopCriterion)) do
        begin
            t := t + 1;
            for i := 1 to N do
                P_{i_t} ← oneGenerationGeneticAlgorithm();
            if (rand > p_exchange) then
                begin
                    P_{x_t} ← chooseRandomly(P_t);
                    P_{y_t} ← chooseRandomly(P_t − {P_{x_t}});
                    exchange(P_{x_t}, P_{y_t});
                end
        end
    g_opt ← bestChromosome(P_t);
end
```

In the pseudocode of a decentralized parallel genetic algorithm the set of chromosomes is divided to N independent subpopulations. In each step (until the stop criterion is fulfilled or time expired) classical sequential algorithm with only one population transition is then run on each of these subpopulations. Then two subpopulations are selected stochastically and their chromosomes are exchanged. This can be done in different ways. In our work one of the subpopulations sends 1/4 of its best chromosomes to the other one.

Parallel genetic algorithm can be modified in several ways. It can be controlled by central process that evaluates the subpopulations and distributes successful chromosomes, it can define a neighborhood so that only neighboring subpopulations can exchange chromosomes, the return value can be only one chromosome with the highest fitness or the whole subpopulation of best-valued chromosomes, etc. An overview of parallel genetic algorithms can be found in [2].

3 ROBOTS

For the experiments we used two Khepera robots ([8]) that are well suited for educational as well as experimental purposes (see [6]). Basic module consists of two interconnected boards with the diameter of $5.5\,cm$. It has two servomotors driving independently two wheels and eight infrared sensors (six in the front and two at the rear side of the robot – see Figure 1) that serve to avoid obstacles. Robots can move at the speed of $1\,ms^{-1}$.

Robot is an autonomous unit in that the whole program written in C language runs on the on-board processor. It is possible that the program runs on the host computer and communicates with the robot via the serial communication line. We

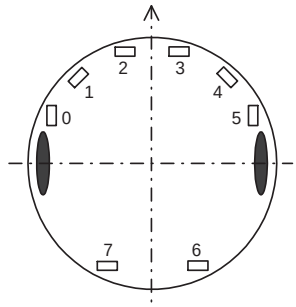

Fig. 1. Schematic view of a Khepera robot with two wheels and eight IR proximity sensors. The arrow indicates the front movement of the robot

maintained such type of a communication only for the purposes of collecting data from the experiments. Communication routines as well as program execution are managed by Motorola 68331 controller with 256 KB RAM and 512 KB ROM.

We used an additional component – a radio extension turret that is mounted on top of a robot in a plug and play fashion. It enables the communication of a robot with the host computer via a radio base, but in our experiments we use it for direct peer-to-peer communication between robots. Robots communicate via sending and reading data buffers whose size is 16 bytes. Messages are sent directly to a robot with a specified identification number. There is a built-in mechanism of checking the correctness of messages. The sender considers a message to be sent correctly only if it receives the acknowledgement from the receiver. It repeats the sending process after the timeout when it receives no acknowledgement. The message is considered lost after 10 unsuccessful trials. The robot with the radio extension used in the experiments can be seen in the Figure 2.

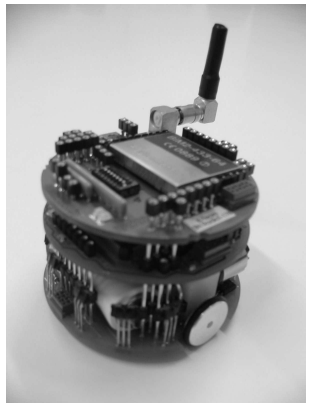

Fig. 2. A front-side view of a Khepera robot with the radio turret ontop

4 THE TASK

We tried to program our robots to learn to avoid dynamic or static obstacles. The behavior is coded by a very simple neural network. It controls the movements of a robot in a straightforward fashion. If the sensors of either side (left or right) of a robot perceive any kind of obstacle, the motor on this side will speed up the wheel it drives. The algorithm is inspired by abstract vehicles proposed in [1]. Genetic algorithm is used to evolve weight vector of input-output neuron connections. These are coded as chromosomes (so they are non-binary integer-valued codons). Each robot evolves its own controllers. One of them sends (we will call it a sender) the successful candidates to the second robot (denoted as receiver). The receiver combines its own evolved population of chromosomes with those received from the sender. We expect that it will learn better and faster than the sender. We try to compare its expected performance with the observed behavior.

4.1 Roles

Programming code of both agents differ in several ways. In this section we describe the algorithms of both robots in more detail.

4.1.1 The Sender

The body of program of the sender is composed of two main processes that are running interchangeably. One is responsible for running evolution of weight vectors and testing the resulting neural controllers. The second one is responsible for sending 1/4 of the most successful population chromosomes to the receiver. Concise pseudocodes of sender's processes are as follows:

```
process 1:
  begin
    t := 0;
    w_t[] ← createWeightVectorsWithRandomValues();
    while (t < 20) do
      begin
        for i := 1 to N do
          begin
            runNeuralController(w_{i_t});
            f_{i_t} ← fitness(w_{i_t});
            changeSpeedRandomly(s_1, s_2);
          end
        sort(w_t);
        buffer[] ← transformBestChromosomesToMessages(w_t);
        suspend(this);
        while (not signalReceived) do
```

```
                waitForSignal(process2);
            if (signalReceived) then
                begin
                    t := t + 1;  w_interm := ∅;
                    w_interm ← select(w_{t-1});
                    w_t ← reproduce(w_interm);
                    w_t ← mutate(w_t);
                end
        end
    end

process 2:
    begin
        while (true) do
            begin
                while (not signalReceived) do
                    waitForSignal(process1);
                if (signalReceived) then
                    sendMessages(buffer, receiver);
                sendSignal(process1);
            end
    end
```

The array of message buffers represents a global structure shared by both processes. After each generation the first process transforms the best valued weight vectors into an array of buffers 16 bytes each and gives control to the second process waiting for response in the background. After the process 2 sends all of the array buffers to an awaiting robot it sends a signal to the first process from which it awaits another signal. Processes of a sender don't run in a parallel manner because of problems with interference between the radio transmission and communication with the host computer that these processes maintain.

4.1.2 The Receiver

The receiver consists of two main processes that can be described by the following pseudocodes:

```
process 1:
    begin
        t := 0;
        w_t[] ← createWeightVectorsWithRandomValues();
        while (t < 20) do
            begin
                for i := 1 to N do
```

```
                begin
                   runNeuralController(w_{i_t});
                   f_{i_t} ← fitness(w_{i_t});
                   changeSpeedRandomly(s_1, s_2);
                end
             sort(w_t);
             replace(w_{t_{1:N/4}}, messages);
             t := t + 1; w_interm := ∅;
             w_interm ← select(w_{t-1});
             w_t ← reproduce(w_interm);
             w_t ← mutate(w_t);
          end
   end

process 2:
  begin
    oldMessages ← createNullBuffer();
    while (true) do
      begin
        receive(buffer, sender);
        if (not (equals(buffer, oldMessages)) then
          begin
            messages ← transformMessagesToWeightVectors(buffer);
            oldMessages := messages;
          end
      end
  end
```

The receiver must continually check for new messages from the sender. The buffers are not flushed which means that if no new message arrives the received buffer still holds old values. That is why the robot must remember the previously arrived message which it always compares with newly received information. If new array of weight vectors arrives, the receiver rewrites the received weight vectors and uses them in its own genetic algorithm as a second quarter of a population that will be crossed over with the best members of its own genome. Processes of the receiver run concurrently and there is no need of synchronization.

4.2 Neural Controller

The behavior of collision avoidance is performed by a neural controller with primitive architecture.[5] There are 8 input neurons fully connected with two output neurons

[5] It was a purposes to design an artificial neural network that can solve the task as simple as possible.

that compute the speed of two motors. These connections are labelled by integer-valued weights. Each of the input neurons has the information about the read value of one of the proximity sensors. Schematic view of a neural network architecture used in our experiments is in Figure 3.

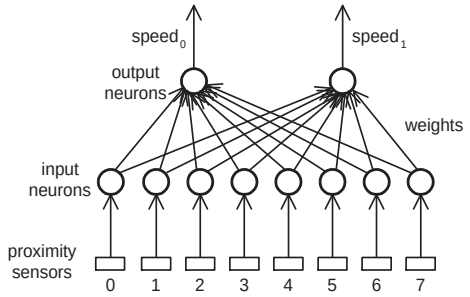

Fig. 3. Architecture of a neural network used in the experiment

The speed of each motor is computed according to eq. (1),

$$speed_i = potential_i/400 + 10;\ i = \{0, 1\}, \tag{1}$$

where

$$potential_i = \sum_{j=0}^{7}(weight_{ij} * irvalue_j), \tag{2}$$

where $weight_{ij},\ i = \{0, 1\},\ j = \{0, 1, \ldots 7\}$ is a vector of weights represented as a two-dimensional array of integer values from the open interval $(-30, 30)$, and $irvalue_j,\ j = \{0, 1, \ldots 7\}$ is a vector of values read by proximity sensors. Speed of motors must be normalized and set to some non-negative value if the value of $potential_i$ is close to 0.

4.3 Evolutionary Settings

The genetic algorithm depicted in the robot processes above was used to evolve the weight vector $weight_{ij},\ i = \{0, 1\},\ j = \{0, 1, \ldots 7\}$. In each run we evaluated the population of neural controllers with the fitness function consisting of three components from [3] (see eq. (3)).

$$\Phi = V(1 - \sqrt{|speed_0 - speed_1|})(1 - i), \tag{3}$$

where V is a normalized average speed of two wheels, $|speed_0 - speed_1|$ is an absolute difference of speed of two wheels, and i is a sensory value recorded by a proximity sensor with highest activation normalized so that it doesn't surpass the value 1. These three components of a fitness function measure average speed of a robot (it

should be maximized), speed difference of rotating wheels (it should be minimized), and highest activation of proximity sensors (it should be minimized).

From each population we picked 1/2 of chromosomes with highest fitness values and placed it in the next generation. Then we sequentially took pairs of parental chromosomes and performed the crossover at the random spot to produce offspring chromosomes for a new generation. Consequently we randomly changed a chromosome coding with 5% probability.

In case of the receiver robot we combined 1/4 of its own chromosomes with the best 1/4 of the sender's best population members to produce the offspring generation as explained in Section 3.1 (see also schema in Figure 4).

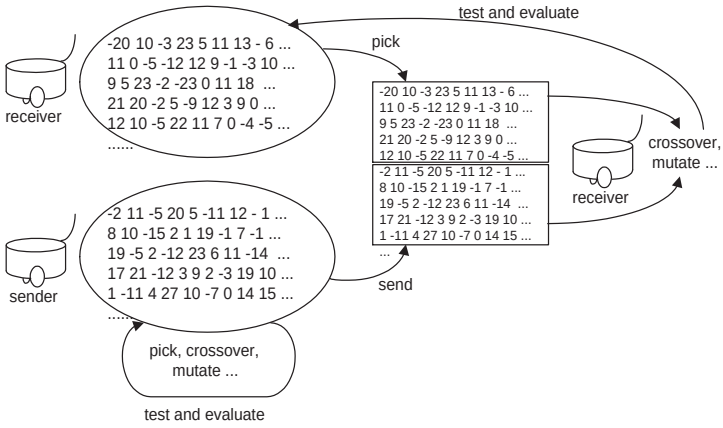

Fig. 4. Schema of a genetic algorithm for both robots

We experimented with various size of population – 12, 20, and 40 chromosomes. We set the smallest size of population so that it almost always finds "acceptable" solution for at least one of the robots – a controller that performs well judged not by the fitness value but by looking at the moving robot performance. We were mainly interested in the best controller, not the average performance that can be determined only computationally. It is very interesting how small population of neural controllers can evolve to a successful solution.

5 RESULTS

In this section we present results of our experiments. We describe experiments with the populations of size 12, 20, resp. 40 chromosomes. Each experiment consists of 5 independent runs that were evaluated for the highest and average values as well as for variance of both robots performance. Each run consisted of 20 generations of evolving neural controllers. The goal of our experiments was to compare the

performance of a sender and a receiver evaluating the controller fitness as well as smoothness of their movements just by observing robots in the arena. Both robots should learn to avoid obstacles satisfactorily and as fast as possible. The bigger the population, the better the performance with more time spent on learning. Our expectations are that the receiver will learn faster than the sender. Comparison to results reported in [3] is difficult because fitness values are normalized and the input of the neural network can have different values. So the only comparable result is time spent by robots to learn the task "well enough", which can be evaluated only by observing their behavior.

Both robots move separately in a small closed arena with randomly dispersed obstacles whose positions were changed during the experiments.

5.1 Experiment 1

In the first experiment we used 12 chromosomes in one population. The performance of the best controller of the receiver is much better and it also learns faster in comparison with the sender robot (see the Figure 5)[6]. Its performance is even better on average, though not so markedly than the performance of best controllers. Due to a small population size the learning peaks in the 14^{th} generation. To measure how

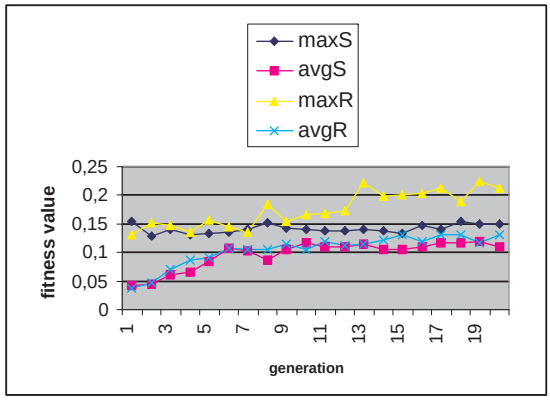

Fig. 5. Experiment with the population of 12 chromosomes. Max. and avg. fitness values for both robots

sharing the chromosome pool improved the performance of a receiver we compared in each generation the best evolved controllers of the sender and the receiver.

$$growth_{max} = \frac{max(fitness_{max}(gen^i_{sender}) - fitness_{max}(gen^i_{receiver})) * 100}{fitness_{max}(gen^i_{receiver})} \quad (4)$$

[6] Capital S and R in subsequent figures denote the sender and receiver, respectively.

where

- $real \leftarrow fitness_{max}(gen^{i}_{sender/receiver})$ is a function that returns the highest fitness value from its argument – the set of fitness values of a given robot from experimental run $i \in \{1, 2, 3, 4, 5\}$, and
- $real \leftarrow max(arg)$ is a function that returns the maximum difference of compared highest fitness values of both robots throughout the all experimental runs.

$$growth_{min} = \frac{min(fitness_{max}(gen^{i}_{sender}) - fitness_{max}(gen^{i}_{receiver})) * 100}{fitness_{max}(gen^{i}_{receiver})} \quad (5)$$

where $real \leftarrow min(arg)$ is a function that returns the minimum difference of compared highest fitness values of both robots throughout the all experimental runs.

$$growth_{real} = \frac{(fitness_{max}(gen^{1-5}_{sender}) - fitness_{max}(gen^{1-5}_{receiver})) * 100}{fitness_{max}(gen^{1-5}_{receiver})} \quad (6)$$

where $real \leftarrow fitness_{max}(gen^{1-5}_{sender/receiver})$ is a function that returns the highest fitness value from its argument - the set of fitness values of a given robot throughout the experimental runs 1–5.

The difference (that could be called maximum expected fitness growth) measured in percentage shows how big the improvement could be in case of best-performing receiver's controllers (eq. (4)). Figure 6 shows also the lowest expected fitness growth[7] (eq. (5)) along with the real improvement (eq. (6)). Real growth in fitness of the receiver's chromosomes is higher than we would expect in this experiment.

5.2 Experiment 2

In this experiment the population of controllers consisted of 20 members. Again the performance of the receiver was better in highest fitness values but approximately the same on average (see Figure 7). In this experiment the receiver learned even faster than in previous case (the highest fitness value in the 5^{th} generation) but from this point the learning started slowly to degrade.

The expected versus real growth in fitness of receiver's best controllers can be seen in Figure 8.

The learning of a receiver is much more unstable with big differences between the highest and the smallest fitness values in each generation illustrated in Figure 9 and denoted by eq. (7).

$$var_{min} = min(fitness_{max}(gen^{i}_{sender/receiver}) - fitness_{min}(gen^{i}_{sender/receiver})) \quad (7)$$

[7] This denotes to what degree the received controllers could negatively influence the performance of a receiver.

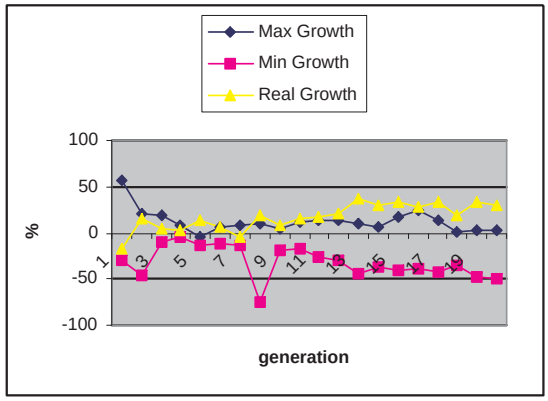

Fig. 6. Experiment with the population of 12 chromosomes. Expected vs. real fitness growth in receiver's chromosomes

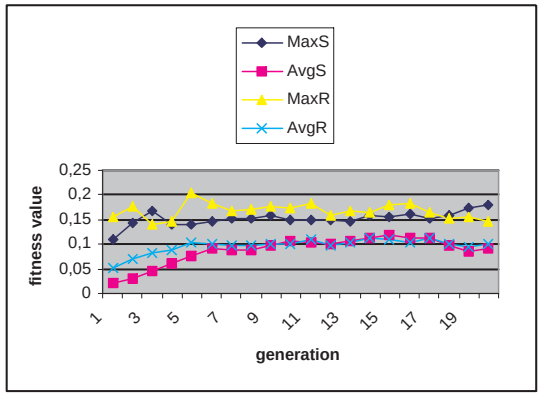

Fig. 7. Experiment with the population of 20 chromosomes. Max. and avg. fitness values for both robots

where

- $real \leftarrow fitness_{max}(gen^{i}_{sender/receiver})$ returns maximum fitness value of either a sender or a receiver in one of the experimental run $i \in \{1, 2, 3, 4, 5\}$,
- $real \leftarrow fitness_{min}(gen^{i}_{sender/receiver})$ returns minimum fitness value of either a sender or a receiver in one of the experimental run $i \in \{1, 2, 3, 4, 5\}$, and
- $real \leftarrow min(arg)$ is a function that returns the minimum difference of the highest and the lowest fitness values of a sender or a receiver robot throughout the all experimental runs.

There are mainly three reasons to this fact:

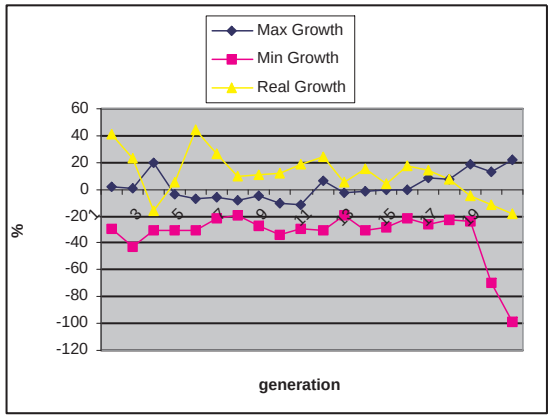

Fig. 8. Experiment with the population of 20 chromosomes. Expected vs. real fitness growth in receiver's chromosomes

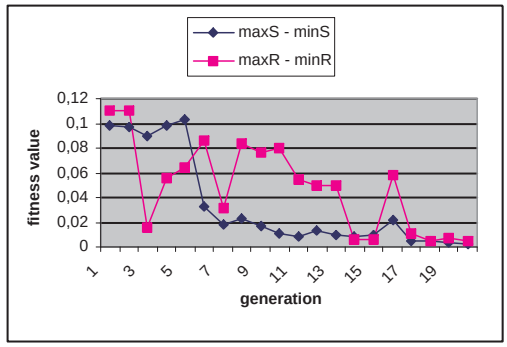

Fig. 9. Experiment with the population of 20 chromosomes. Max. – min. fitness values taken from the run with the smallest difference

- Controllers acquired from the sender with worse performance in comparison with the receiver's own population pool can perturb learning and destabilize it. Even if it should not influence the best chromosomes it has an influence on creating suboptimal solutions.
- Communication errors. The communication among Khepera robots is not very reliable. There appears loss of data during the transmission which we roughly estimate to be 5 to 15 %. If the robot fails to correctly determine when new array of message buffers arrives it uses old values instead which deteriorates the performance of a given generation. The bigger the population of chromosomes, the more fatal the consequences appear to be. Firstly, the robots must exchange a number of messages that equals to half of the number of chromosomes in the

population.[8] The more messages they send and receive, the more probable is the occurrence of transmission errors.

- Time delay in sending and testing chromosomes. Because of non-concurrent nature of sender's processes it takes more time for the sender to run all the generations when compared to the receiver because the sender stops the genetic algorithm while sending data to the other robot. That is why transmission lags can occur in that the sender can send the receiver chromosomes from generations that are older than the currently executed generations of the receiver. The learning of the receiver can suffer from this.

The consequences of the facts explained above will be more visible in the last experiment.

5.3 Experiment 3

Even if both robots achieved highest fitness values in this experiment (with 40 chromosomes in population), the receiver robot didn't perform better than the sender. Its best controllers were better when compared to the best-performing sender's controllers only in a couple of generations and it completely failed when we take average performance into account (see the Figure 10).

Figure 11 shows instability of the receiver's evolving populations that is on average higher than in the previous experiment.

In the last experiment the improvement in receiver's fitness got hardly positive values as depicted in Figure 12.

6 DISCUSSION

In this section we discuss our approach to controller evolution in the multi-robot societies. We summarize the results of our experiments and compare them to other reported works, we consider advantages and disadvantages of our solution, its potential for future work and how some technical difficulties could be addressed.

6.1 Summary of Experiments

Our experiments have shown that evolution of neurocontrollers in a multirobot society can be very fast even in a very small group of robots. In a couple of generations

[8] Radio turrets of used Khepera robots use protocol that enables to transfer only 16 bytes messages (more precisely unsigned bytes), i.e. only small positive integers. One weight vector that contains 16 weights must therefore be split into 2 messages. Each message holds 1/2 of the weight vector values plus the indication of a sign of particular weight value. E.g. message containing data {... 1 7 1 3 0 19 0 21 0 19 0 2 1 16 1 15} is a half of a chromosome with values {7 3 -19 -21 -19 -2 16 15}.

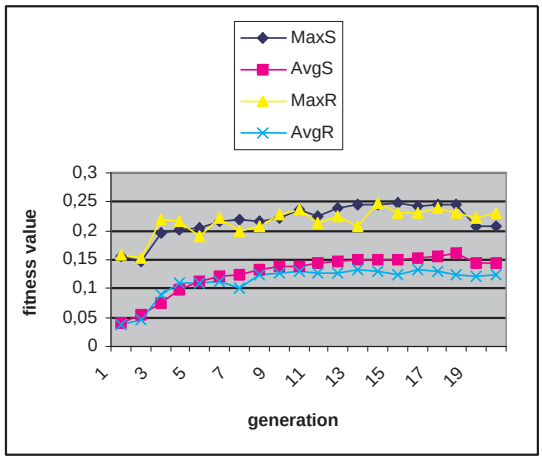

Fig. 10. Experiment with the population of 40 chromosomes. Max. and avg. fitness values for both robots

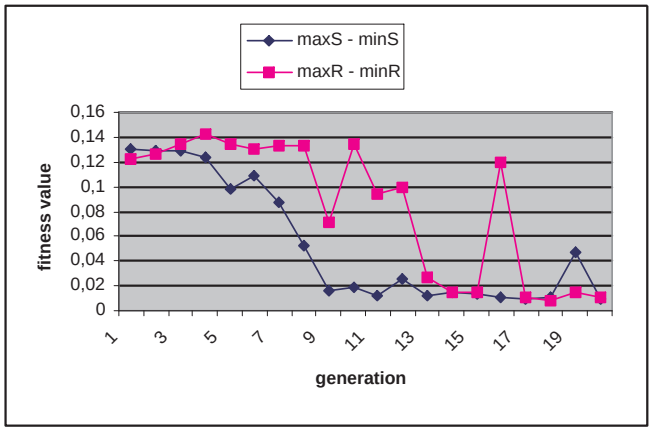

Fig. 11. Experiment with the population of 40 chromosomes. Max. – min. fitness values taken from the run with the smallest difference

(5–20) the good enough controller can evolve even in a population of several controllers (10–20). The learning can take 2 to many hours based on the evolutionary settings. Authors in [3] report on near optimal performance with 50 generations that was achieved in more than 30 hours. This is, nevertheless, influenced by the population size. Very promising results were confirmed by real growth in fitness of a robot that uses distributed chromosome pool that was even higher in some cases than we expected.

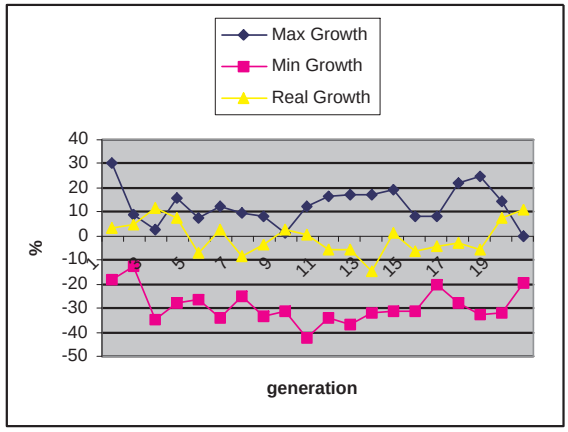

Fig. 12. Experiment with the population of 40 chromosomes. Expected vs. real fitness growth in receiver's chromosomes

Good results of a distributed genetic algorithm in comparison with serial genetic algorithm were obtained only in small populations of chromosomes. Problems with real environment and communication cause degradation of best as well as average performance of controllers. This can be seen also on bigger variance in fitness values of receiver's chromosomes. The possibility of population distribution to small independent groups in our solution nevertheless eliminates consequences of this fact.

6.2 Generality of Our Approach

Our aim was to extend the evolution of a single agent neural controller to a group of robots. We have shown that the learning by evolving robot controllers speeds up if robots share the experience with their own evolution. An advantage of our approach also consists in simplicity of implementation. We depicted the possibility to create a pool of solutions that could be shared by a group of robots. The idea rests in the fact that only controllers that were most successful are used in next generations. The exchange of experience is thus possible in a very direct fashion. This is applicable more generally in other tasks and controller architectures. The only condition is an existence of a part of controller identified as an evolvable and communicable unit in a genetic algorithm.

We implemented a parallel distributed algorithm with decentralized control and direct communication described in Section 2.5 that is to our knowledge unique on hardware platform. There are, nevertheless, various extensions and modifications that can be done in the future and that we describe in more detail in the next section.

6.3 Extensions and Future Work

In case of a bigger group of robots they should communicate using some protocol. The reason is possible loss of data due to repeated and failed transmission. Communicating robots must synchronize the process of sending and/or receiving. This is necessary when more than one robot want to send messages to one robot at the same time. The synchronization brings complexity and programming code overhead into the implementation of multi-robot evolution of controllers in decentralized environment. The advantage of using synchronization protocols is generality – more robots can evolve simultaneously which also (as we expect) brings the profit of massive parallelism of evolution. This can be done effectively exchanging very small groups of chromosomes (possibly containing only one chromosome) among robots. The learning could be very fast and optimally performing controllers could be achieved in several generations. In this case very high instability in the fitness values of chromosomes has to be expected, because we have no guarantee of receiving only better performing controllers. The possible solution could be to filter out received chromosomes that have lower fitness value than the chromosomes evolved by the receiving robot. This would require exchange of fitness values along with the weight vectors.[9]

To eliminate data transmission errors it would be useful to design a mechanism for continuous repetition of sending data if they have been lost from the part of the sender. More difficult for the receiver would be to "guess" where is the beginning of a new message array in case some of the messages get lost and the sender repeats the transmission of a whole array again (because in this case the receiver uses the contents of an old buffer, so it doesn't notice that something went wrong).

Decentralized form of a distributed genetic algorithm is useful if the communication load can be reduced by limiting the number of communication partners of each robot in each evolutionary run. Using direct communication is not very effective from the point of view of exchanged data if more than 2 robots evolve simultaneously and if after each population every robot exchanges chromosomes with each other in the group because each robot has to send the same set of data more than once. A better way to do this would be to use stigmergic communication – communication through the environment. The robots would drop the chromosomes to a destined file from which they could be read by other robots. This file could be used as a shared environment of robots. As a tradeoff we lose a power of decentralization which can have bad impact on the behavior of a robot society in case the file will go corrupt.

As for usefulness of our approach for cognitive science and robotics, it can be applied in the research of cognitivistic approach to behavior of artificial creatures and personalized agents. Robots share the environment but the experience with the given task differs from agent to agent. Their behavior is a direct consequence of what

[9] This cannot be done with Khepera robots because as mentioned in the previous section they can directly exchange only bytes of unsigned integers.

they learned online and the encoded artificial neural network or its parameters can be considered a symbolic representation of the experience as the robot interacts with the environment and other robots. This experience is grounded, i.e. it is not hand-coded by a constructor from the beginning. By direct exchange of chromosomes, robots share experience in a very elegant and straightforward fashion. Evolutionary algorithms (especially used along with connectionism) thus can be used in the study of personalized and individual approach to solving tasks by animats and the research of "computational consciousness" of artificial minds. Features of the environment as well as parameters of the solving task are extracted by the robot itself creating its personal image (a virtual reality) of experience. Being only a basic idea reported in this preliminary work we nevertheless hope it to serve as an influential source for scientists coping with the problem of cognition as well as with learning by sharing experience in artificial systems.

7 SUMMARY AND CONCLUSIONS

In this paper we reported the results of our work on distributed genetic algorithm applied to a population of neural controllers used and exchanged by two robots without a central control process. We discussed how technical problems that arise could be solved and how the work could be used as a starting point for further research. We proposed several ways of extending our design and implementation and what must be done in order to use our ideas in societies with more than two robots. Finally we proposed our solution to be applicable in broader context in the area of multiagent learning and computational cognitivism.

REFERENCES

[1] Braitenberg, V.: Vehicles. Experiments in Synthetic Psychology. MIT Press, Cambridge, MA, 1984.

[2] Cantu-Paz, E.: A Survey of Parallel Genetic Algorithms. IlliGAL report No. 97003. University of Illinois, 1997.

[3] Floreano, D.—Mondada, F.: Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural Network Driven Robot. In: Cliff, D.—Husbands, P.—Meyer, J.—Wilson, S. (eds.): From Animals to Animats III: Proceedings of the Third International Conference on Simulation of Adaptive Behavior. MIT Press-Bradford Books, Cambridge, MA, 1994, pp. 402–410.

[4] Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Redwood City, CA, 1989.

[5] Holland, J. H.: Adaptation in Natural and Artificial Systems. The University of Michigan Press, Ann Arbor, 1975.

[6] Kelemen, J.—Kubík, A.: RADIUS: Looking for Robot's Help in Computer Science Research and Education. ERCIM News, 2002, No. 51, pp. 48–49.

[7] Kubík, A.: Distributed Genetic Algorithm: Learning by Direct Exchange of Chromosomes. In: Banzhaf, W. et al. (eds.): Advances in Artificial Life. Proc. 7th European Conference on Artificial Life. Springer Verlag, Berlin, 2003, pp. 346–356.

[8] Mondada, F.—Franzi, E.—Ienne, P.: Mobile Robot Miniaturization: A Tool for Investigation in Control Algorithms. Proceedings of the Third International Symposium on Experimental Robotics. Kyoto, Japan, 1993.

[9] Nolfi, S.—Floreano, D.: Evolutionary Robotics: The Biology, Intelligence and Technology of Self-Organizing Machines. MIT Press, Cambridge, MA, 2000.

[10] Nolfi, S.—Floreano, D.: Co-evolving Predator and Prey Robots: Do 'Arm Races' Arise in Artificial Evolution? Artificial Life, Vol. 4, 1998, No. 4, pp. 311–335.

[11] Quinn, M.: Evolving Communication Without Dedicated Communication Channels. In: Kelemen, J.—Sosík, P. (eds.): Advances in Artificial Life: Proceedings of the 6th European Conference on Artificial Life. Springer Verlag, Berlin, 2001, pp. 357–366.

[12] Quinn, M.—Smith, L.—Mayley, G.—Husbands, P.: Evolving Teamwork and Role-Allocation with Real Robots. In: Standish, R. K.—Bedau, M. A.—Abbass, H. A. (eds.): Proceedings of the Eighth International Conference on Artificial Life. MIT Press, Cambridge, MA, 2002, pp. 302–311.

[13] Watson, R. A.—Ficici, S. G.—Pollack, J. B.: Embodied Evolution: Embodying an Evolutionary Algorithm in a Population of Robots. In: Angeline—Michalewicz—Schoenhauer—Yao—Zalzala (eds.): 1999 Congress on Evolutionary Computation. IEEE Press, 1999, pp. 335–342.

[14] Watson, R. A.—Ficici, S. G.—Pollack, J. B.: Embodied Evolution: Distributing an Evolutionary Algorithm in a Population of Robots. Robotics and Autonomous Systems, Vol. 39, 2002, No. 1, pp. 1–18.

Aleš Kubík studied economics and information technology at the University of Economics in Bratislava. He holds a PhD. degree from this institution (2003). Currently he is working as an asisstant professor at the Institute of Computer Science, Silesian University in Opava (Czech Republic). His research interests include experimental robotics, emergent computing in multiagent systems, artificial life, evolutionary and distributed computing. He has written two books on multiagent systems (in Czech).