DECA: The Doping-driven Evolutionary Control Algorithm

Pieter Spronck, Ida Sprinkhuizen-Kuyper and Eric Postma
Universiteit Maastricht, MICC-IKAT
P.O.Box 616, NL-6200 MD Maastricht, The Netherlands
{p.spronck,kuyper,postma}@micc.unimaas.nl

Abbreviated title: DECA

5th February 2007

Abstract

In task control, evolutionary optimisation tends to favour controllers that solve the easier task instances but that fail to solve the harder ones. We call this the problem of hard instances. The Doping-driven Evolutionary Control Algorithm (DECA) is introduced to deal with the problem. The effectiveness of DECA is assessed on two task-control problems: a box-pushing task and a food-gathering task. The experimental results show DECA to generate controllers that can solve both the easy and hard instances of both task-control problems. We discuss the results by offering a qualitative explanation for DECA's success and comparing it to related techniques. We conclude that the problem of hard instances is alleviated by the application of DECA.

1 Introduction

Evolutionary learning techniques (Bäck, 1996) are effective techniques for optimising the controllers of situated agents (Arkin, 1998). When applying evolutionary learning

to (neural) controller design, the mapping executed by the controller is generated automatically by setting the controller parameters. The quality of controllers is defined in terms of an appropriate measure of fitness as determined by the fitness function. In general, the fitness function is based on the evaluation of a controller on a series of typical task instances varying in difficulty from easy to hard. In earlier work we observed that good solutions to easy task instances are more abundant and are located in "flat" regions of the search space. In contrast, good solutions to hard task instances turned out to be rare and located at "peaks" surrounded by inferior solutions (Spronck et al., 2001a). In the evolutionary learning process new controllers are generated by recombining elements of previously-generated controllers, favouring those that have a relatively high fitness. Obviously, a controller that solves at least one of the task instances is assigned a higher fitness value than one that solves no instances at all. Since it is very likely that controllers that cope with easy task instances are discovered before those that cope with harder instances, the performance on the easy task instances determines the course of the evolutionary process to a great extent. Therefore, the evolutionary search is more or less constrained to the regions of search space where most of the solutions to easy instances reside. Unless a good solution that covers both easy and hard instances is found in the vicinity of these regions, the end result is a controller that handles easy instances well, but fails on the hard ones. We call this the problem of hard instances. If the problem of hard instances is not dealt with, evolutionary algorithms are bound to produce inferior solutions to task control problems. The research question we address reads: in what way can the problem of hard instances be dealt with? In answering the research question, we propose the Doping-driven Evolutionary Control Algorithm (DECA).

The outline of the remainder of this paper is as follows. Section 2 discusses the problem of hard instances in more detail. Section 3 introduces DECA and explains how it can be used in task learning problems. Section 4 describes the experimental procedure employed for evaluating DECA. Sections 5 and 6 are devoted to two experiments that show the success of DECA. Section 7 provides a general discussion of the experimental results, including an evaluation of possible alternatives for DECA. Section 8 presents our conclusions.

2 Easy vs. Hard Instances

In evolutionary computation it is quite common that populations converge on broader, but less fit peaks in the solution space, while avoiding fitter but narrower peaks (Soule, 2006). The premise of the problem of hard instances is that the presence of easy instances may interfere with the search for an optimum that represents a good solution for all instances. In this section, we demonstrate the interference by means of an example. In the example we define two partial fitness functions, that specify the fitness for easy and hard instances, respectively. Each partial fitness function is of the form $F_i : [0, 1000] \times [0, 1000] \rightarrow [0, 1]$, $i \in \{1, 2\}$. These functions can be interpreted as instance-specific fitness functions. A true fitness function for both instances is a combination of F_1 and F_2 and will be defined below. F_1 and F_2 are defined as:

$$F_1(x, y) = \frac{2 + cos(2\pi x) + cos(2\pi y)}{4}$$

$$F_2(x, y) = \frac{e^{-|x-500|} + e^{-|y-500|}}{2}$$

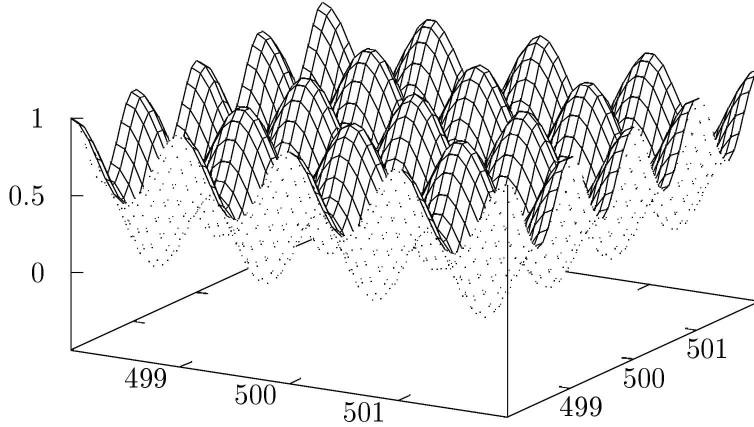


Figure 1: Part of easy-instance function F_1.

The function F_1, illustrated in Figure 1, reveals that instance 1 is a typical example of an easy instance. It achieves an optimum for each (x, y) when $x \in \mathbb{N}$ and $y \in \mathbb{N}$. F_2, illustrated in Figure 2, shows that instance 2 is a typical example of a hard instance. It achieves its sole optimum at $(x, y) = (500, 500)$. The largest part of the solution space of F_2 is quite flat, but it contains two fairly high "ridges" which pass through the optimum. Most hillclimbing algorithms will easily discover an optimum for each of the partial fitness functions. However, for stochastic optimisation techniques such as evolutionary optimisation, it is much easier to find a solution for F_1 than for F_2. Given random initial values for x and y, the optimal value is much more likely to be near for F_1 than for F_2. As a consequence, on average a solution to the easy instance will be found much faster than for the hard instance.

Of course, the separation of fitness into easy and hard instance components is quite unrealistic. In realistic problems, a single fitness function is defined that applies to all instances. In our case, a more realistic fitness function is defined as follows.

$$F = C \cdot F_1 + (1 - C) \cdot F_2$$

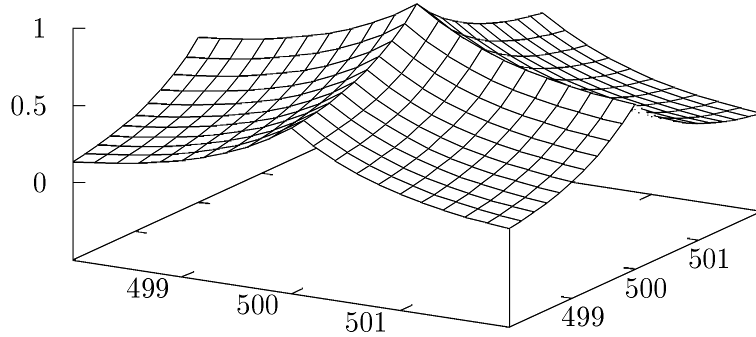


Figure 2: Part of hard-instance function F_2.

In this equation, C is a value in the range $\langle 0, 1\rangle$ that balances the contribution of both instance-specific fitness functions to the overall fitness. For the combined fitness measure, the global optimum is located at $(500, 500)$, independent of the value of C.

With this overall fitness function we can demonstrate the problem of hard instances by arguing that evolutionary optimisation tends to generate solutions that favour the optimisation of F_1 over F_2. In doing so, we consider two cases: (1) $C = 0.5$ (both instance-specific fitness functions carry an equal weight), and (2) $C = \varepsilon$, where ε is an arbitrary small positive number (the hard-instance fitness functions carries a much larger weight than the easy-instance fitness function).

For $C = 0.5$, any randomly initialised hillclimbing algorithm will converge to a value of F close to 0.5 in most cases and to the optimum $F = 1.0$ only rarely. The reason is, again, that a random initial choice of x and y is much more likely to be in the vicinity of the peaks of F_1 than the peak of F_2. Therefore, whenever a fitness function weights the fitness contributions of easy and hard instances equally, the problem of hard instances may appear. As a case in point, we performed a series of 30 tests using the simple Genetic Algorithm (sGA; Goldberg, 1989) with a population of 50 chromosomes to find

the optimum for F. None of the tests succeeded in finding the optimum, despite the sGA being able to quickly discover the optima for both F_1 and F_2 separately. Even when we added an extra genetic operator, which seeded the population with random new genetic material, the optimum was only found when by chance a chromosome was created that was already very close to the optimum.

The tendency of a hillclimbing algorithm to get stuck in a local optimum that represents a solution to the easy instance but not to the hard instance may be mitigated by assigning more weight to the hard-instance fitness function, i.e., setting C at a value near zero. This decreases the height of the local optima associated with F_1 (i.e., the peaks in Figure 1), while the global optimum (i.e., the peak in Figure 2) maintains its height. Decreasing the value of C expands the region around the global optimum for which hillclimbing will find the optimal solution. However, the region for which $F_1 > F_2$ is still much larger depending on the range of values for x and y. We repeated the previously mentioned series of 30 tests, now with $C = 0.01$, and found that only three of them succeeded in discovering the global optimum for F.

This illustrates that assigning more weight to hard instances does not always solve the problem of hard instances. In particular for the typically much larger search spaces in realistic problems, the problem of hard instances is likely to hamper the performance of stochastic hillclimbing algorithms such as evolutionary optimisation.

3 DECA

To deal with the problem of hard instances we propose the Doping-driven Evolutionary Control Algorithm (DECA). The algorithm is based on the notion of “doping”. Doping

is defined as the addition of some very good solutions to a population (usually the initial one) in order to facilitate the evolution process. These solutions may be generated by a different algorithm or may express the user's knowledge about the problem domain (Dumitrescu et al., 2000). Common terms used for similar techniques are "seeding", "case injection" (Louis, 2002) and "infusion" (Spronck et al., 2001a). If there are differences between the exact meanings of these terms, they are not well-defined. The term "seeding" is used in literature most often. It refers to the injection of any kind of genetic material into a population. We chose to use the term "doping" as it was used by Dumitrescu et al. (2000), to refer to the injection of complete solutions into a population, rather than the injection of any kind of genetic material. We discuss prior work on doping in Subsection 3.1 and describe DECA in detail in Subsection 3.2.

3.1 Doping

The application of doping (or seeding) is restricted to those cases where it is important to retain specific genetic material in the population (Dumitrescu et al., 2000). The best-known example is in the "messy Genetic Algorithm" (mGA), where in the primordial phase of the evolution the population is doped with all possible building blocks of a specific length (Goldberg et al., 1991). Sometimes doping takes the form of inserting manually designed solutions into the initial population. An example is Matthews et al.'s (2000) work on a problem in land-use planning where the initial population was doped with heuristic and expert-based solutions. In Case-Initialised Genetic Algorithms (Louis and Johnson, 1999) a solution to a problem similar to the target problem is inserted in the initial population to facilitate the evolution process in finding a good solution to the target problem. Grefenstette and Ramsey (1992) created an initial population that

consisted of 50% solutions that worked well in the past, 25% manually designed solutions for the problem in general, and only 25% solutions generated randomly.

While the examples mentioned above all report beneficial effects of doping, it should be considered whether doping can be detrimental to the evolution process. Doping genetic material that is unrelated to any known solution, as is done in the mGA, does little harm to the final solution. However, doping an initial population with known solutions may lead to inferior results. The reason is that within a population of random solutions, a fairly good solution is likely to have the highest fitness, which leads to convergence to a local optimum in the vicinity of the doped solution. The evolution process is used as a local optimisation process, rather than as a method to scan the solution space. Good solutions that are too remote from the doped solution are likely to be missed. In order for doping to yield good results in task-control problems, the evolutionary process needs to be biased to deal with hard instances. This is exactly what is done in DECA as will be detailed below.

3.2 Doping in DECA

The Doping-driven Evolutionary Control Algorithm (DECA) ensures that the evolutionary search is confined to those regions of the search space where the solutions to hard instances are likely to be found. In order to achieve the bias, DECA applies doping as described in the following six steps.

1. *Training set design*: Select a series of instances that encompass all relevant aspects of a task.

2. *Hard instance selection*: Identify a hard instance that encompasses most of the relevant aspects.

3. *Hard instance evolution*: Evolve a good solution to the hard instance selected in the previous step.

4. *Initialisation*: Generate a random population and "dope" this population with the solution evolved in the previous step.

5. *Evolution*: Evolve good solutions to the complete series of instances selected in step 1 using the doped population.

6. *Validation*: Evaluate the validity of the evolved solution on a new selection of task instances.

If no domain knowledge is available to select hard instances in step 2, a (time-consuming but generally applicable) way to identify hard instances is to attempt to evolve separate solutions to all the instances in the training set, and observe for which instances the evolution process takes the longest on average.

DECA is expected to yield good results because we assume that there is an asymmetry in the search space with respect to easy and hard solutions (i.e., local optima of the fitness function). Solutions to easy instances are readily found in the vicinity of solutions to hard instances, whereas the reverse is not true. The asymmetry is caused by the abundance of solutions to easy instances and the relative scarcity of solutions to hard instances. We discuss the validity of our assumption in more detail in Subsection 7.1.

4 Experimental Procedure

To evaluate the effectiveness of DECA, we performed experiments with two very different tasks. The first task is a box-pushing task wherein a robot has to push a box between two walls. The second task is a food-gathering task in which an agent has to collect food while avoiding to be damaged. For both tasks we used neural controllers, which are suitable adaptive structures for situated agents (Arkin, 1998). The weights and architectures of the controllers were generated using an evolutionary algorithm.

The freely available "Elegance" environment (Spronck and Kerckhoffs, 1997) was used to perform the experiments. Elegance, which is the acronym for Engineering Laboratory for Experiments with Genetic Algorithms for Neural Controller Evolution, is an environment designed to do experiments with evolutionary neural controllers. It is easily extendable and supports both feedforward and recurrent neural controllers, both weight determination and architecture design, a wide range of genetic operators and evolutionary algorithm parameters, and a variety of control tasks. It is easy to add new control tasks to the program.

The main set-up of the experiments is illustrated in Figure 3. The evolutionary algorithm maintains a population of potential solutions represented by chromosomes. Each chromosome is translated into a controller. The performance of the controller on the closed-loop task is expressed in terms of a fitness value, which is experimentally derived by observing the functioning of the controller on typical processes, i.e., its behaviour in simulated environments. For the box-pushing task, the task consists of pushing a box between two walls. For the food-gathering task, the task consists of collecting food while avoiding poison (a more detailed description of both tasks is given in Sections 5 and 6).

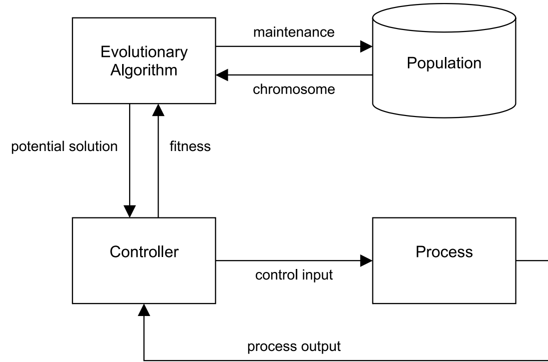


Figure 3: Evolving a controller for closed-loop control.

4.1 The Neural Controller

Preliminary experiments with the evolution of a neural box-pushing controller indicated that a layered recurrent neural controller outperforms various kinds of feedforward controllers on this particular task (Sprinkhuizen-Kuyper et al., 2000b). We therefore decided for both experiments to use a layered recurrent network with one hidden layer, a maximum of four hidden nodes, and the network output values constrained by applying a sigmoid function. As illustrated in Figure 4, a layered recurrent network has its nodes ordered in layers. It is less general than a completely recurrent network, because only recurrent connections within a layer are allowed. The recurrent connections feed the node values from the previous time step into the target nodes and may therefore be interpreted as a memory of previous node values. This architecture represents an Elman network (Elman, 1990).

4.2 The Evolutionary Algorithm

Inspired by the encoding of Maniezzo (1993), the evolutionary algorithm employed in Elegance allows evolving the network's weights in parallel with its architecture. The

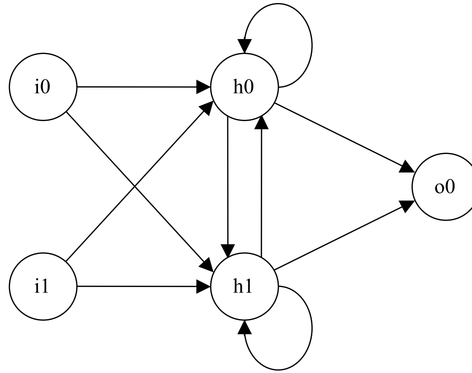


Figure 4: Illustration of a recurrent network of the type used in the experiments. The recurrent connections feed the node values from the previous time step into the target nodes. The network drawn here contains two input nodes (i0 and i1), two hidden nodes (h0 and h1), and a single output node (o0). All connections within the hidden layer are recurrent connections.

network is directly encoded into a chromosome consisting of an array of "connection genes." Each connection gene represents a single possible connection of the network and consists of a single bit and a real number. The bit represents the presence or absence of a connection and the real value specifies the weight of the connection. It should be noted that, in this encoding scheme, even absent connections have a weight associated with them. The weight values of inactivated connections function as a kind of latent memory that can be reactivated by a mutation of the connection bit.

In our experiments we employed the following six genetic operators, which we found to perform well in evolving solutions for other neural control problems.

1. *Uniform crossover.* Child chromosomes are created by copying each allele from one of two parents, each parent having a 50% chance of being selected for each allele.

2. *Biased weight mutation* (Montana and Davis, 1989). Child chromosomes are copies of parent chromosomes, with each weight having a 5% chance to be mutated by

adding a random value selected from the range $[-0.3, 0.3]$.

3. *Biased nodes mutation* (Montana and Davis, 1989). Child chromosomes are copies of parent chromosomes, whereby all the input weights of one randomly selected node are changed by adding a random value selected from the range $[-0.3, 0.3]$.

4. *Nodes crossover* (Montana and Davis, 1989). Child chromosomes are created by copying each of their nodes (including their input connections) from one of two parents, each parent having a 50% chance of being selected for each node.

5. *Node existence mutation* (Spronck and Kerckhoffs, 1997). Child chromosomes are copies of parent chromosomes, with a 95% chance of having all incoming and outgoing connections of one randomly-selected node being removed, and a 5% chance of having all absent connections of a randomly-selected node being activated.

6. *Connectivity mutation* (Spronck and Kerckhoffs, 1997). Child chromosomes are copies of parent chromosomes, whereby each connection has a probability of 5% to switch from being connected to being disconnected and vice versa.

During evolution, one of these six operators is selected at random. For the crossover operators, we arbitrarily decided to add only the fittest of the two children to the population, while we rejected the other child. To alleviate the problem of competing conventions (i.e., the occurrence of solutions in the population which are equal but have different representations; Hancock, 1992) the hidden nodes of the parents are rearranged to make their signs match (insofar as possible) before a crossover takes place (Thierens et al., 1993). Newly generated individuals replace existing individuals in the population, while taking into account elitism. For the selection process, size-k tournament selection

is used ($k = 2$ for the box-pushing experiment and $k = 3$ for the food-gathering experiment). Crowding (which is similar to tournament selection, but the worst of the selected chromosomes is replaced) with a factor of 3 is used as replacement policy.

In all experiments, the population size is equal to 100. In preliminary experiments we tested larger population sizes, with a maximum of 250, but these did not give significantly better results. We set a maximum to the number of generations based on the observed convergence rates (35 generations for the box-pushing experiment and 30 generations for the food-gathering experiment). Preliminary experiments showed that in rare cases we could achieve slightly better solutions if we let evolution continue for more generations, but in our view the severe increase in computation time required was not worth the small improvement in performance.

Having discussed the experimental procedure, we now turn to the description of the two experiments to evaluate the effectiveness of DECA.

5 The Box-pushing Experiment

The box-pushing task is the first task to evaluate DECA. The task involves the pushing of a box between two walls. A simpler version of the task was introduced by Lee, Hallam and Lund (1997). Pushing an object (in our case a circular box) between two walls is an elementary behaviour that is relevant in, for instance, the game of robot soccer in which a ball has to be pushed towards the opponent's goal (Asada and Kitano, 1999). The task is non-trivial, because it requires the agent to adapt continuously to the position of the ball as perceived through the noisy sensors. Elementary behaviours, of which the box-pushing task is only an example, are believed to underlie more complex behaviours such

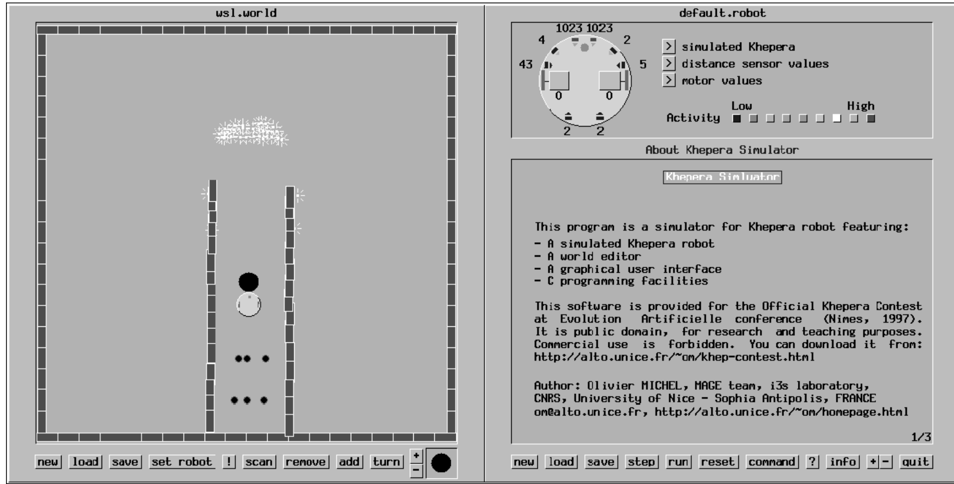


Figure 5: Simulation environment of the Khepera robot. The square area on the left side is the robot world and measures 1000×1000 units. The grey circle represents the robot, the black circle the box, and the six small black dots the starting positions of the robot and the box.

as target following, navigation and object manipulation. We describe the box-pushing task in Subsection 5.1, present the achieved results using DECA in Subsection 5.2 and provide a discussion of the results in Subsection 5.3.

5.1 The Box-pushing Task

In our studies of box-pushing behaviour we employed a publicly available mobile robot simulator based on the widely used mobile robot Khepera (Mondada et al., 1993) illustrated in Figure 5. The (simulated) Khepera displayed in Figure 6 is equipped with eight sensors and two motors, one for each of the wheels. The sensors provide the robot with proximity values. For the purpose of the experiment, we coupled the simulator to the Elegance environment.

The Khepera simulation is controlled by a neural network with fourteen inputs, provided by the eight proximity sensors and six additional virtual "edge-detector" sensors. The outputs of the virtual sensors are defined as the differences in proximity values

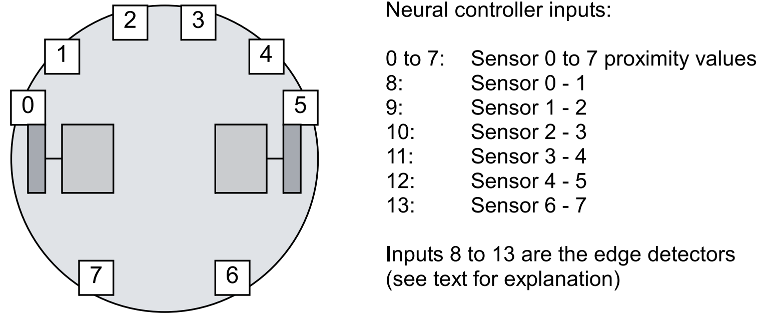


Figure 6: Schematic overview of the Khepera robot, with a mapping to the neural controller inputs.

between all pairs of neighbouring sensors. It is important to note that the Khepera simulation is stochastic because the sensors and controller outputs generate noisy signals. The motors driving the wheels are controlled by the outputs of two neural networks, one for the left and one for the right wheel. Exploiting the mirror symmetry of the perception-to-action mapping, the two neural networks are identical except for the mapping of sensors to network inputs and the definition of the signs of the edge-detecting inputs. Figure 7 illustrates the different mapping and signs for both networks. In the figure, the small rectangles at the left of the neural networks indicate the sensors. In these rectangles, $x - y$ indicates an edge detector in which the value of sensor y is subtracted from the value of sensor x.

The task set to the robot was to push the box as far away as possible from its starting position within a limited period of time. The partially-observable and stochastic nature of the environment makes this task quite complex (Wooldridge, 2000). Nine task instances (that differ in the initial configuration of robot and box) were defined for the box-pushing task. Figure 8 illustrates the nine instances numbered 0 to 8. The box-pushing task is difficult because the robot (i) must identify the box, (ii) must remain behind the box while pushing, (iii) must prevent the box from getting stuck, and (iv)

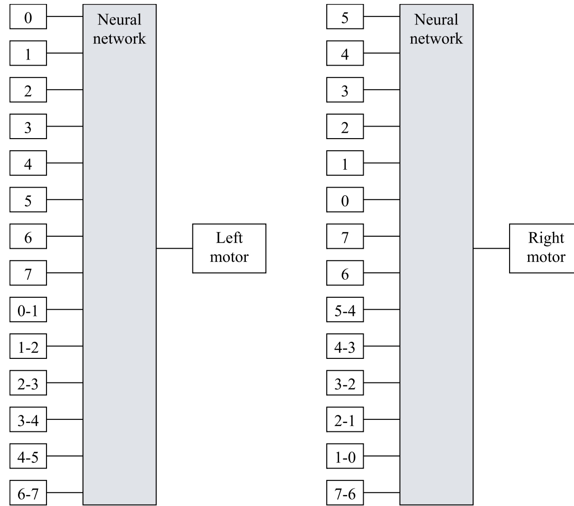


Figure 7: The two almost identical networks to drive the left and right network motors. The network inputs are proximity values derived from the Khepera robot shown in Figure 6.

must deal with noise generated by the sensors and the motor controls. Preliminary experiments revealed that the nine task instances exhibited these difficulties in various degrees. For instance, in task instances 0, 4, and 8, the box is positioned directly in front of the robot, which means the robot can perform its task by simply moving forward and correcting for small deviations. Task instances 2 and 6 are harder since the initial separation of the robot and box is larger than in task instances 0, 4, and 8. Task instances 3 and 5 can be considered the most difficult because in these tasks the robot suffers more from the roughness of the walls than in any of the other task instances (Spronck et al., 2001a).

At first glance it may seem that tasks 2 and 6 are equally difficult, if not more difficult than tasks 3 and 5. However, in preliminary experiments we discovered that this is not the case. In tasks 2 and 6, the robot travels a longer distance from its starting position to the box than in tasks 3 and 5. The longer distance allows the robot more time and more room to manoeuvre in a good position to slide the box along the wall. In tasks 2

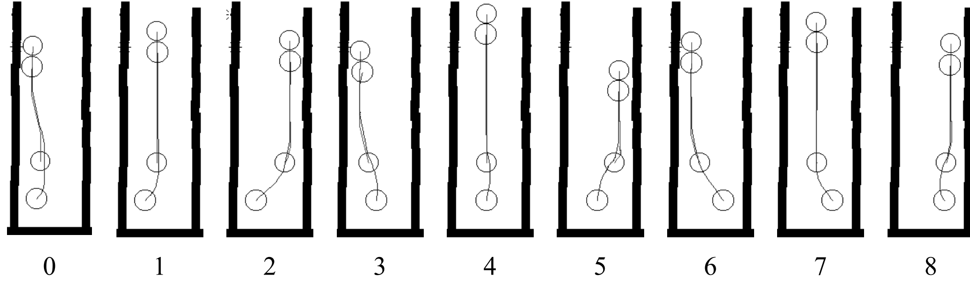


Figure 8: The nine task instances (0 to 8) and typical trajectories of the robot and the box. The circles shown are the robot (largest circles) and the circular box (slightly smaller circles) at their initial (bottom) and final (top) positions. The lines connecting the initial to the final positions represent the paths followed by the robot and the box. It should be noticed that the roughness of the walls hinders the robot in sliding the box along a wall.

and 6 the robot learns to position itself directly “below” the centre of the box. In tasks 3 and 5, the robot has less time and less room to manoeuvre into a good position, and so it tends to push the box “sideways”, thereby hitting the wall under an inconvenient angle. Figure 8 illustrates this robot behaviour.

If $robot_t$ is the position of the robot at time t, and box_t is the position of the box at time t, the fitness value assigned to a robot upon completion of a single instance i is defined as

$$F_i = d_i(box_T, box_0) - \frac{1}{2} d_i(box_T, robot_T)$$

In this equation $d_i(box_T, box_0)$ represents the Euclidian distance between the initial $(t = 0)$ and final positions $(t = T)$ of the box, and $d_i(box_T, robot_T)$ the Euclidian distance between the robot and the box at their final positions for task instance i (all distances are calculated between the centres of the objects). An experimental trial comprises $T = 100$ steps on each of the nine instances. We defined the average fitness F_{avg} on a trial as the average fitness over all instances, i.e., $F_{avg} = \frac{1}{9}\sum_{i=0}^{8} F_i$ (Sprinkhuizen-Kuyper et al.,

2000a).

To reduce the effect of the noise, we defined the overall fitness F as the average of the trial fitness values over a number of R repetitions of trials, i.e., $F = \frac{1}{R}\sum_{r=1}^{R} F_{avg}^{r}$, with F_{avg}^{r} representing the average fitness F_{avg} obtained at the r–th repetition. Computational resources constrained the number of repetitions. In our simulations we varied the number of repetitions between $R = 1$ and $R = 100$ depending on the following considerations. In preliminary experiments we already determined that controllers with a fitness value of 250 or less on a single trial are inferior (i.e., we determined empirically that, in general, they work well on the easier task instances 0, 1, 4, 7 and 8, but are unable to deal with the harder task instances 2, 3, 5 and 6), and remain inferior on replications of the trial. The contribution of inferior controllers to the evolution process is limited, and consequently their ranking need not be very precise, especially since we use tournament selection. Therefore, in case of such low fitness values, a single trial suffices ($R = 1$). For higher fitness values, the number of repetitions was set to $R = 10$. For a controller that has the potential to be the best of the population, the overall fitness was determined on the basis of $R = 100$ repetitions. Using this procedure the overall fitness of the fittest controller has a standard error of the mean of about 1.3, yielding an accuracy of about 2.5 fitness points (reliability of 95%; Cohen, 1995).

We confirmed the validity of the evolved controllers by testing them on a real Khepera robot. The controllers proved to be effective and efficient in letting a real Khepera robot push a circular box between walls. It is our opinion that this success is due to the high amount of noise inherent in the simulation, which requires an evolved controller to be robust (Jakobi, 1997).

5.2 Results of the Box-pushing Experiment

One experiment without doping and ten experiments with doping using various solutions were performed, and the overall fitness values were determined. For the doping experiments we applied DECA by executing the six steps described in Subsection 3.2. Figure 8 shows examples of the trajectories of successful robots on the nine task instances. In order to compare the effect of doping with a solution to a hard instance to doping with a solution to an easy instance, instead of selecting a hard instance (as prescribed in step 1), we performed separate doping experiments with solutions to each of the task instances (that vary from easy to hard). In addition, we performed a doping experiment using solutions to all instances. The average fitness values were obtained by averaging over the highest fitness values obtained after 35 generations in seven replications of each of the experiments.

We expected doping with controllers optimised on task instances 3 or 5, which are the hardest, to yield the best results. Indeed this was what we found. Figure 9 displays the results obtained with doping using controllers optimised on a single task instance (labelled '0' to '8'), without doping (labelled 'no') and with doping using controllers optimised on all nine task instances (labelled 'all'). Doping with controllers optimised for task instances 3 and 5 yield the best results (average fitness of 320.3 and 319.5, respectively), and the most consistent results (highest/lowest fitness values 322.9/318.1 and 322.1/316.8, respectively). Doping with controllers optimised on all tasks yields better results than doping with controllers optimised on task instances that are easy or moderate (i.e., task instances 0, 1, 2, 4, 6, 7, and 8). Presumably, the inclusion of solutions to the hardest instances contributes to the high fitness obtained in this case. It should be noted, however, that while doping with all instances gives the highest fitness,

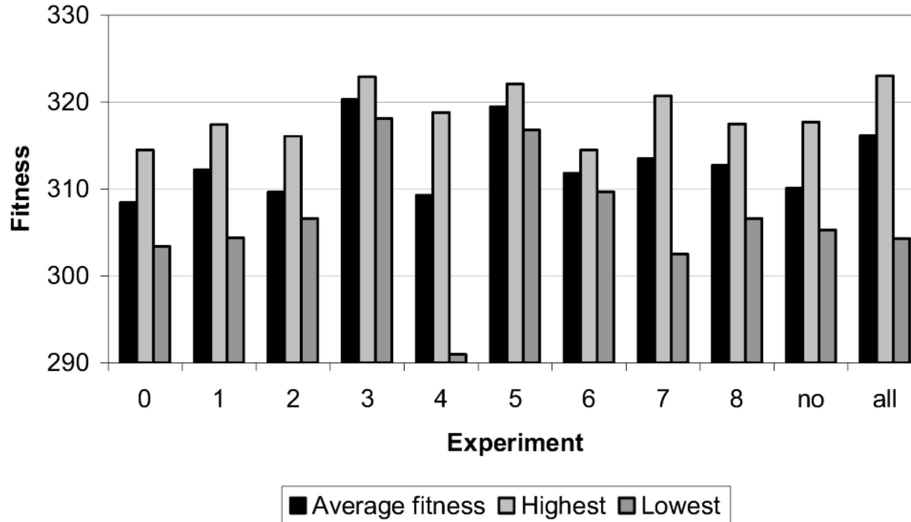


Figure 9: Fitness values of experiments with doping of a solution to a single task instance ('0' to '8'), without doping ('no') and with doping of all solutions ('all'). The left bar represents the average fitness, the middle bar the highest fitness, and the right bar the lowest fitness.

the results have a much higher variance than those obtained by doping with controllers optimised for task instances 3 and 5 (highest/lowest fitness values 323.0/304.3). Presumably, the reason for this is that the evolutionary algorithm occasionally converges to a local optimum near to the optimal solution for task instances other than 3 and 5.

Overall, these results show that on the box-pushing task DECA gives a significant improvement over non-doped evolutionary optimisation.

5.3 Discussion of the Box-pushing Experiment

Figure 9 shows that for tests performed with doping solutions to all instances, the highest fitness results are not significantly different from doping a solution to instances 3 or 5 only. However, in this comparison the lowest fitness results for the first are significantly lower than those for the latter. In contrast, when we performed tests with doping solutions to *both* instances 3 and 5, we found that they did not produce results different

from doping with either of these solutions. We surmise that the lower results with doping solutions for all instances are the effect of early convergence in the neighbourhood of a doped solution to an easy instance. Therefore we conclude that the doping of easy-instance solutions (even in combination with the doping of hard-instance solutions) should be avoided.

The box-pushing experiment was not specifically designed to test DECA. Yet, we were pleasantly surprised by the improved results obtained by applying DECA. Notwithstanding these results, it must be acknowledged that the box-pushing experiment task is of limited value for evaluating DECA, because it suffers from two main shortcomings. First, the task is based on a stochastic simulation requiring many repetitions to obtain reliable results. Second, the lack of variety in possible task instances precludes the assessment of the ability to generalise beyond the task instances given (even though the controller's ability to generalise was demonstrated by applying it to a real Khepera).

We expected that the success of DECA can be generalised to other evolutionary control tasks. To confirm our expectation, we decided to evaluate DECA on a second control task, designed to deal with the limitations of the box-pushing experiment.

6 The Food-gathering Experiment

The food-gathering experiment was designed to have the following two requirements. First, the task should be deterministic. Second, it should allow for generating instances with variable levels of difficulty. We describe the food-gathering task in Subsection 6.1, present the achieved results using DECA in Subsection 6.2 and provide a discussion on the results in Subsection 6.3.

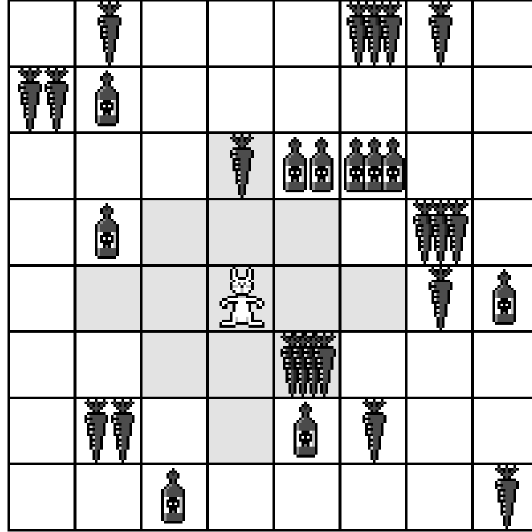


Figure 10: Part of the grid defined as the environment of the rabbit. The environment contains food (carrots) and danger (poison bottles). The rabbit's field of vision consists of all cells (squares) that can be reached in a maximum of two moves (the shaded squares in the image), i.e., the Manhattan distance = 2.

6.1 The Food-gathering Task

The food-gathering task is designed as follows. A rabbit is placed on a square two-dimensional grid of $N \times N$ cells. The rabbit can move by one step in each of the four main directions: north, east, south and west. The grid has periodic boundary conditions, i.e., it is defined as a torus. As illustrated in Figure 10, the rabbit's field of vision encompasses all cells that are within two moves from its current position. A cell may be empty, it may contain one or more carrots, or it may contain one or more poison bottles. If the rabbit enters a cell that contains c carrots, it removes (eats) all of them leaving an empty cell, and increases its score by c points. If the rabbit enters a cell with p poison bottles, it decreases its score with p points. In contrast to carrots, poison bottles are not removed from the grid when visited by the rabbit. In each experimental trial, a rabbit has to score as many points as possible within 100 moves. Initially, the rabbit is always positioned in an empty cell.

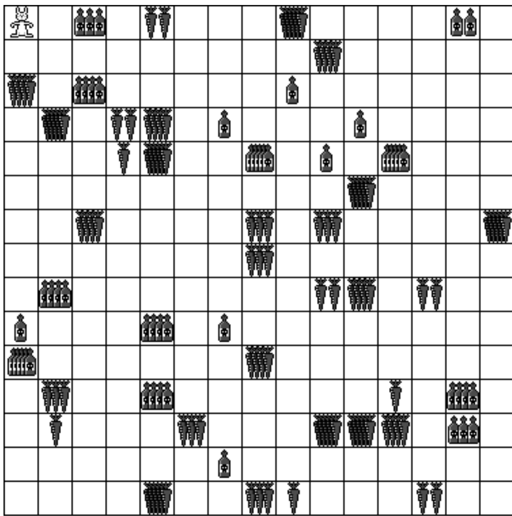
The rabbit is controlled by a neural network with thirteen inputs. Each input I is defined as the value of a cell visible to the rabbit (see Figure 10). The magnitude $|I|$ of the input value represents the number of elements within the patch occupying the cell. The sign of the input indicates whether the patch contains carrots ($I > 0$) or poison bottles ($I < 0$). An empty cell is represented by zero input ($I = 0$). The network has four outputs, representing the four directions of movement of the rabbit. The rabbit moves in the direction corresponding to the output with the highest value.

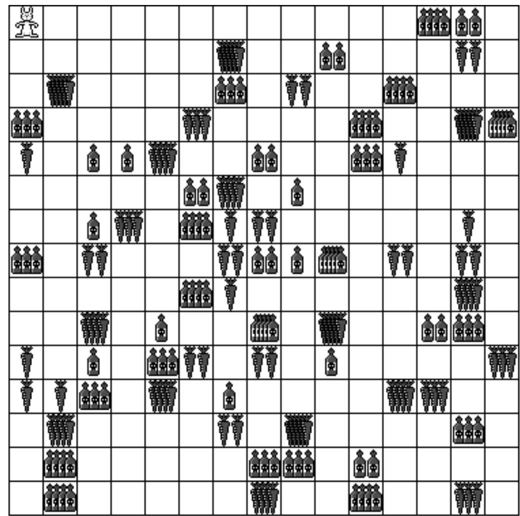
For the training set, we randomly generated grids with $N = 15$, a total number of carrots $C = 100$ and a total number of poison bottles P varying between 0 and 150. Carrots and poison bottles are clustered in small patches of one to five carrots or poison bottles per patch. The number of poison patches directly bordering a carrot patch also varies according to a density value d ($d \in \{0, 1, 2, 3, 4\}$). Arguably, the complexity of an instance is proportional to d and P, because an increased total number of poison bottles and an increased density of poison bottles adjacent to carrots make it harder for the rabbit to collect carrots without losing points.

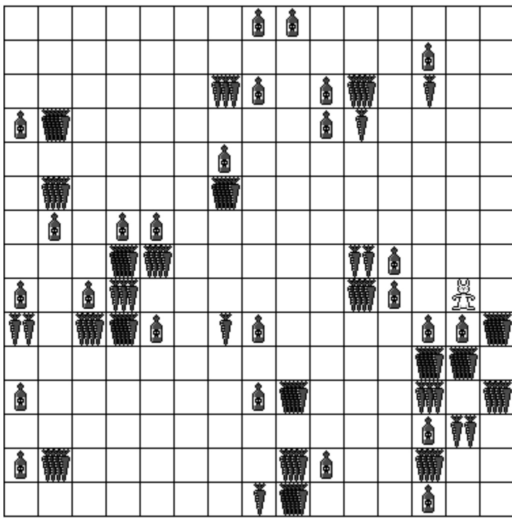
Table 1 displays the twenty instances (numbered 0 to 19) in the training set in relation to the parameters d and P, including a qualification of their difficulty. For instances 5 to 17, the parameter d is defined as a range. Figure 11 shows six of the twenty instances that serve as the training set.

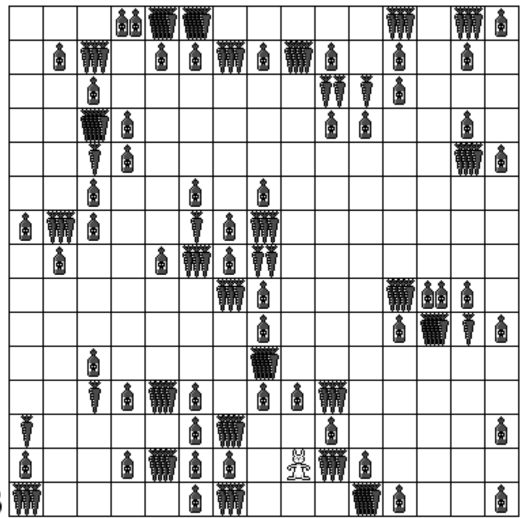
To assess the generalisation performance of evolutionary optimised rabbits, we generated an extensive test set of a hundred instances comprising five subsets of randomly-generated instances each. The instances within each subset were generated according to the same values of d and P as specified in Table 1.

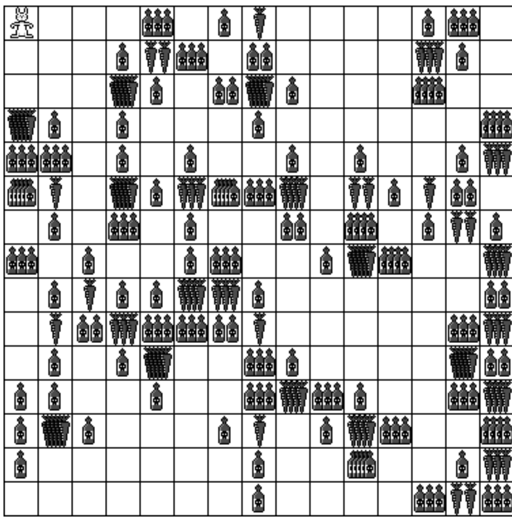
The fitness F of a controller (or rabbit) is defined as the average score on the twenty











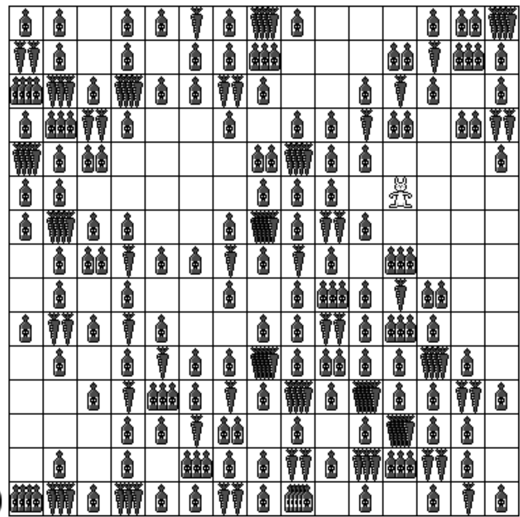


Figure 11: Six of the 20 task instances in the training set of the food-gathering experiment.

instance	d	P	difficulty	instance	d	P	difficulty
0	0	0	very easy	10	1–2	50	medium
1	0	25	very easy	11	1–2	100	medium
2	0	50	easy	12	1–2	150	hard
3	0	100	easy	13	2–3	50	medium
4	0	150	medium	14	2–3	100	hard
5	0–1	25	easy	15	2–3	150	hard
6	0–1	50	easy	16	3–4	100	hard
7	0–1	100	medium	17	3–4	150	very hard
8	0–1	150	medium	18	4	100	very hard
9	1–2	25	easy	19	4	150	very hard

Table 1: Specification of the twenty instances (numbered 0 to 19) used in the food-gathering experiments in relation to the density value d and the number of poison bottles P. In all instances $C = 100$.

task instances of the training set. Since each task instance contains 100 carrots, an upper bound to the fitness is 100. In most instances it is impossible to reach this upper bound, because even without poison patches, the shortest path connecting all carrot patches in the grid usually is longer than 100 steps.

6.2 Results of the Food-gathering Experiment

For the food-gathering experiment we compared two series of tests. In the first series, we used the evolutionary algorithm discussed in Section 4 to evolve, in 30 generations, a neural controller for the rabbit, with a fitness function defined as the average score of the controller on the twenty grids in the training set. In the second series we applied DECA, as follows. First we evolved a good controller for a single task instance. Then we evolved a neural controller for the rabbit with the overall fitness function in 27 generations, using an initial population doped with the solution found for the single task instance. The reason for using 27 (rather than 30) generations for the evolution with the overall fitness function was to ensure that the computational resources used for

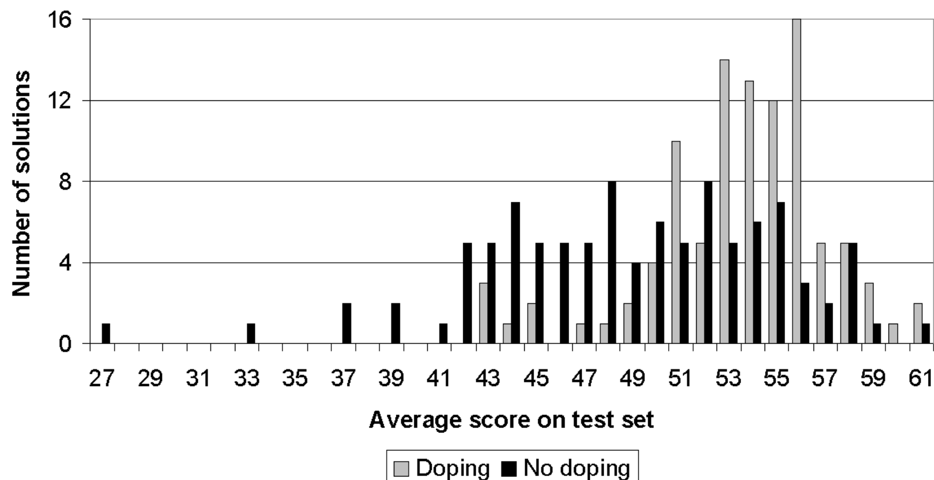


Figure 12: Comparison of the scores of 100 tests with doping and 100 tests without doping in the food-gathering experiment.

both series of experiments were approximately equal.

We decided to use task instance 17 as the hard task instance to develop a good controller for doping. In this task instance $P = 150$ and $d = 3$–4. We preferred task instance 17 over the seemingly harder task instance 19 (with $P = 150$ and $d = 4$), because in task instance 19 all carrot patches are completely surrounded by poison. We suspected this would reduce the complexity of the task, because it would be impossible for a controller to avoid damage to get to carrots. Therefore, damage avoidance is of less importance for instance 19 than for instance 17.

We ran $R = 100$ repetitions of each of the series of tests. Of each of the tests, the controller with the highest fitness on the training set containing twenty grids, was used as the solution found. We then evaluated this controller on the test set containing 100 grids. In our statistical analysis we defined the fitness of a controller as its score on the test set. In Figure 12 the histograms of the experiments with and without doping are displayed. The doped solution itself achieves a fitness of 32 on the test set.

As is evident from the histograms, the experiments with doping tend to give better

solutions than those without doping. The minimum score achieved without doping is 27, while the minimum score achieved with doping is 43. We also see that the highest score achieved is 61 both with doping (twice) and without doping (once). For doping, the bulk of the scores range from 50 to 60, whereas the bulk of the scores obtained without doping are more widely distributed, i.e., between 40 and 60.

Without doping, the score of evolutionary-optimised controllers averaged over 100 experiments is 48.9 with a standard error of the mean of 0.6. With doping, the score averaged over 100 experiments equals 53.6 with a standard error of the mean of 0.4. From these numbers we can conclude that the results achieved with doping are significantly better than those achieved without doping (reliability $> 99.9\%$; Cohen, 1995).

6.3 Discussion of the Food-gathering Experiment

The food-gathering task is deterministic and allows for the generation of novel task instances. Both features offer the advantage that the effect of doping can easily be assessed. Clearly, our results show that doping is useful for enhancing the quality and generalisation performance of evolutionary-optimised controllers.

To illustrate the type of solutions obtained, we discuss a striking example. Figure 13 shows a path followed by a successfully optimised rabbit (controller) on a hard task instance ($P = 150$ and $d = 3$–4). Despite the ability to move in four directions, the rabbit moves to the east and south only. In a post-hoc analysis of successful controllers, we noticed that two of their four outputs (namely one of the two longitude outputs and one of the two latitude outputs) were disconnected. Constraining the movement to two orthogonal directions diminishes the risk that the rabbit gets in a loop where it moves between a few neighbouring cells, and thus makes it easier to express behaviour that

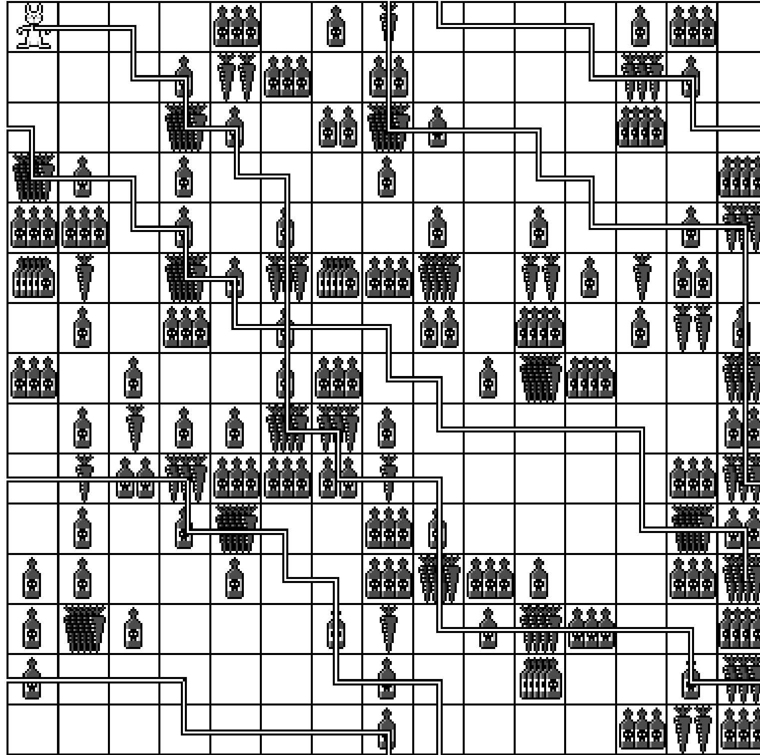


Figure 13: An example of the path taken by a successful rabbit in a hard environment.

allows the rabbit to cover as many different cells as possible within 100 moves.

7 Discussion

The application of DECA to two different tasks showed the feasibility of the approach. We discuss DECA by suggesting an explaining for the success of doping in Subsection 7.1, and by comparing doping to hillclimbing in Subsection 7.2. We discuss five search techniques as potential alternatives for DECA in dealing with the problem of hard instances, namely multitask learning in Subsection 7.3, multi-objective learning in Subsection 7.4, constraint-satisfaction reasoning in Subsection 7.5, and island-based evolutionary learning in Subsection 7.6.

Note that we do not claim that evolutionary learning of a neural controller with

DECA provides the *best* solutions for the problem domains discussed here. Other techniques that use a training set, such as reinforcement learning, may generate solutions of a quality comparable to, or even higher than the quality of the solutions discovered by evolutionary learning. However, we expect that these other techniques will also produce better results when the doping effect is taken into account. The development of such doping-like approaches is left for future study.

7.1 Explanation of the Doping Effect

Why is DECA a successful strategy? We attempt to provide a qualitative explanation for the success of doping.

The solution space of task control problems is spanned by the adaptable parameters defining the controllers, i.e., by the connection weights in the neural networks. Hence, the dimensionality of the solution space is defined by the number of adaptable parameters specifying the controllers (the dimensionalities of the box-pushing and food-gathering controllers are 81 and 92 respectively). As stated in Subsection 3.2 we assume that the high-dimensional solution space contains many large regions where solutions to easy task instances are found, but only a few small regions where solutions to harder task instances reside. Because the hard task instances encompass many, if not all of the difficulties posed by the environment, a solution that applies to instances of arbitrary complexity is likely to be found relatively near to a hard-instance region. Hence, doping the initial population with a solution specialised to hard task instances leads to good generalised solutions.

Our explanation is supported by the development of the fitness of evolution processes with and without doping. In Figure 14 the developments of fitness in the evolution of a

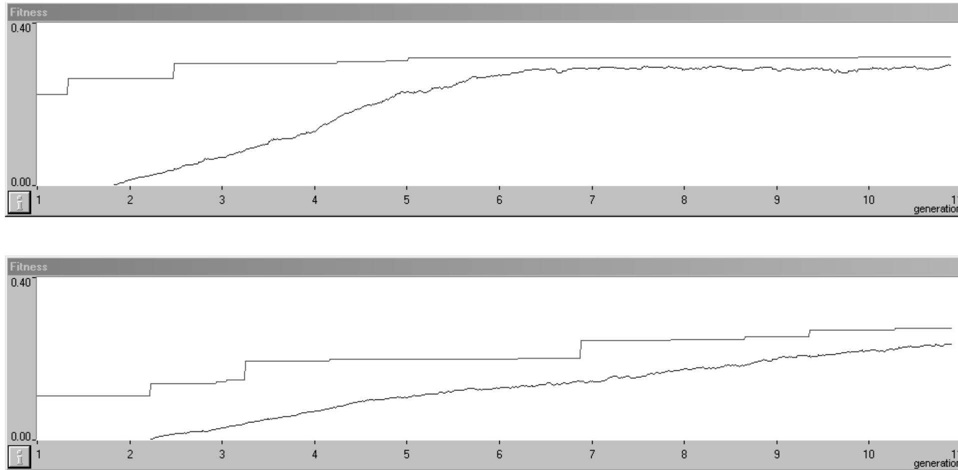


Figure 14: Two graphs showing typical developments of fitness for evolutionary learning with doping (top) and without doping (bottom). In both graphs the fitness (divided by 1000) is plotted against the generation. The top curve in each of the graphs shows the maximum fitness in the population, the curve below it the average fitness.

box-pushing task with doping (the upper graph) and without doping (the lower graph) are compared. While these are only two examples, we found that they are typical for all our experiments. With doping the fitness of the best controller in the population starts between 200 and 250. Within one or two generations, the fitness jumps to around 300. After that, the fitness slowly increases towards a value around 320. Without doping, the fitness starts anywhere between 0 and 200. Initially, the fitness increases quickly to a value between 200 and 250. After that, the fitness progresses slowly towards a value of about 310. These different patterns of development, in particular the quick rise in fitness at the start of the doped evolution process, suggest that DECA takes the best available solution (the doped one) and adapts it to handle the other task instances.

Further support for our explanation is found in experiments that showed that solutions to hard instances also performed reasonably well on the easier instances, whereas the same was not true the other way around. For the box-pushing task this is illustrated in Figure 15. It shows, for each of the doped controllers used in the box-pushing experi-

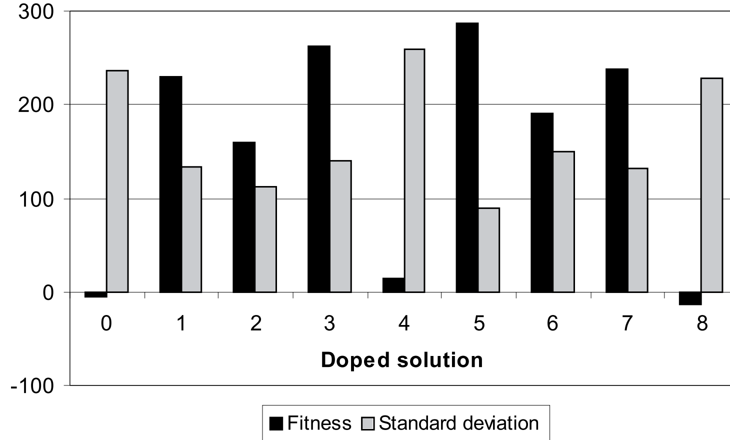


Figure 15: The fitness of doped solutions to a single task, tested on all task instances, averaged over 100 tests. For doping with solutions to each of the nine task instances (0 to 8), the graph shows the fitness (the left, black bar), and the standard deviation (the right, shaded bar).

ments, the fitness and standard deviation on all task instances, averaged over 100 tests. The hardest task instances 3 and 5 yield solutions that have the highest fitness on all other instances, combined with relatively the lowest standard deviation.

We acknowledge that control problems exist in which solutions to hard task instances are not in the neighbourhood of solutions to easy task instances. For such problems DECA will probably not provide the desired good results. We consider DECA a tool in the toolbox of the researcher of evolutionary algorithms, the effectiveness of which depends on the nature of the problem at hand.

7.2 DECA and Hillclimbing

Since our explanation for the doping effect states that the evolution process adapts the doped solution to become a general solution, the question may be posed whether DECA may be combined with hillclimbing. Given a doped solution, hillclimbing may represent a good alternative to standard evolutionary optimisation to obtain good results. We

believe, however, that hillclimbing is not a good alternative to evolutionary optimisation in DECA for the following reason. While the generalised solution may be in the vicinity of the solution to a hard task instance, it may be near the solution on some dimensions and far on other dimensions of the solution space. Sometimes, adapting the solution to the hard instance to generalise over all task instances requires large steps in one or a few directions in the solution space. Evolutionary algorithms may be more efficient in performing those large steps.

To support our argumentation against hillclimbing as an alternative to DECA, we performed an experiment with the foodgathering task, in which we used stochastic hill-climbing to improve the doped solution to task instance 17 in Section 6. This solution (which has a fitness of 32) contains 43 weights. We randomly changed the weights in 10,000 small steps, retaining those changes that resulted in better or equal fitness. After 30 repetitions of the experiment, we found that, on average, the hillclimbing process converged to a solution with a fitness of 40.0, with a standard deviation of 1.1. The maximum found was 42.8. Usually only a few thousand steps were needed for convergence. The average fitness of 40.0 is much less than the average fitnesses achieved with the evolutionary processes described in Section 6 (48.9 without doping and 53.6 with doping), and even the maximum fitness of 42.8 is not higher than the minimum fitness found by applying DECA (which is 43). Thus we have shown that, at least for the food-gathering task, hillclimbing is no suitable substitute for DECA.

Of course, the nature of the solution space depends on the type of problem. Hence, hillclimbing may yield good results in some cases. Montana and Davis (1989) support our line of reasoning, by stating that hillclimbing does not work well for neural network optimisation, because it tends to force convergence to a local optimum instead of a

global optimum. They recommend using hillclimbing only in those cases where the best solution achieved is close to the global optimum.

7.3 DECA and Multitask Learning

The principal goal of multitask learning is to improve generalisation performance of a controller on a task, by leveraging information obtained from controlling related tasks. It does this by training tasks in parallel using a shared representation. Caruana (1997) claims, and shows empirically, that it is more difficult to train a controller on an isolated, hard task, than it is to train a controller on a combination of related tasks that includes the hard one. At first glance, this seems to invalidate our claim that doping with a controller for a hard task instance leads to a better performance than doping with a controller for an easy task instance.

As Caruana (1997) explains, the "related tasks" used in multitask learning are simpler subtasks, rather than variations on the task. With DECA the task is the same for each task instance, only the complexity of the instances differs. The claims Caruana (1997) makes about multitask learning are, therefore, not in conflict with the claims we make about DECA. Moreover, we suspect that multitask learning actually suffers from the problem of hard instances, because it deliberately focusses on easier tasks before tackling a hard one. It does that for a good reason, namely that the hard task cannot be solved directly. Obviously, DECA is not intended to deal with these "unsolvable" tasks.

Louis and Li (1997) use an approach to multitask learning reminiscent of DECA. They evolve solutions to subtasks and use those to dope the initial population of an evolution run that solves the overall task. They discovered that doping with the best solution to each of the subtasks actually results in worse overall solutions than starting

with a randomly initialised population. However, doping with solutions to subtasks that also give good results on the overall task, leads to significantly better solutions than achieved with a randomly initialised population. This result supports our suggestion in Subsection 7.1, where we state that the doping effect results from solutions to hard task instances encompassing characteristics that are needed to solve the easier instances.

Obviously, when using appropriate genetic operators it should be possible to combine solutions to two different hard instances if the characteristics that comprise the two solutions reside at distinct parts of the chromosome. However, problems for which this is the case are likely to be rare. Furthermore, an instance that needs the combination of the two solutions would be considerably harder than the two hard instances for which the original solutions were derived. The harder instance is the one that should be used in the application of DECA. Nevertheless, it is possible that a combination of multitask learning and DECA, where controllers for hard instances of subtasks are doped, may improve the performance of either technique alone. This is an interesting notion that we feel warrants to be explored in future work.

7.4 DECA and Multi-objective Learning

Multi-objective learning aims to find a solution that performs well with regard to all individual objectives (Van Veldhuizen and Lamont, 2000). The main problem of multi-objective learning is that it tends to get stuck in a local optimum once a solution is found for one of the objectives. It is generally appreciated (Horn, 1997; Van Veldhuizen and Lamont, 2000) that a successful Multi-Objective Evolutionary Algorithm (MOEA) needs a secondary population to store Pareto-optimal solutions (e.g., solutions to single objectives), sometimes actually involving the secondary population in the evolution

process.

The task instances used in our DECA experiments bear some resemblance to the objectives in multi-objective optimisation. Interpreting the task instances as different objectives, multi-objective learning techniques can be applied to the problem of hard instances, since they seek a balance between several conflicting objectives (Van Veldhuizen and Lamont, 2000). However, the task instances in our DECA experiments do not represent different objectives, but different instances with varying levels of complexity, while the task to be performed is the single objective. Furthermore, the different instances do not require solutions that are in conflict with each other. Since, in general, multi-objective learning techniques are geared towards conflicting objectives, they do not exploit the similarity between the various instances. Therefore, we believe DECA to be better suited for handling the particular domain of task control problems.

7.5 DECA and Constraint Satisfaction Reasoning

Constraint satisfaction reasoning (CSR) deals with problems where the solution has to satisfy a given set of restrictions or constraints (Tsang, 1993). A solution is invalid unless it fulfils all the constraints. Hence, in CSR the problem is to find a solution that takes into account all constraints rather than one that addresses some of the constraints. Interpreting the task instances as constraints, CSR seems applicable to alleviate the problem of hard instances. However, CSR cannot be readily applied to the problem. The reason is that in CSR all constraints must be strictly satisfied, whereas in task learning it suffices if the task instances are handled reasonably well.

7.6 DECA and Island-based Evolutionary Learning

Evolutionary algorithms are inherently parallel. On multi-processor computers this is commonly exploited by dividing the population into smaller sub-populations, each of which is handled by a different processor. The sub-populations are often referred to as "islands" (Goldberg, 1989). On each island the population is evolutionary optimised to a particular task. The islands exchange genetic material on a regular basis. Apart from enabling parallel processing, the islands may converge to different solutions. The exchange of genetic material might result in an overall solution that combines the best of the island-based solutions (Skolicki and De Jong, 2004).

Island-based evolutionary learning (Spronck et al., 2001b) is an attempt to exploit the principles behind parallel evolutionary algorithms to solve the problem of hard instances. The basic idea of island-based evolutionary learning is to distribute the population evenly over a few islands, whereby each island is assigned a different task instance. After all island populations have converged to a solution to their assigned task, a new population of the best solutions of each of the islands and a number of random solutions is created. A conventional evolutionary algorithm is applied to this new population that is optimised to deal with all task instances. The idea is that the evolution combines genetic material developed using single task instances to solve the general task.

Clearly, island-based evolutionary learning may very well be applied to the problem of hard instances. However, empirical studies, using the box-pushing task, have revealed that island-based evolutionary learning tends to generate solutions that perform well on the harder task instances (even better than when a regular evolutionary algorithm is applied), but show an inferior performance on the easier task instances. As a consequence, a gain in overall fitness is not obtained (Spronck et al., 2001b). Furthermore,

since island-based evolutionary learning evolves a separate solution for all instances, the computational time required by the island-based evolution process is much larger than the computational time required by DECA.

8 Conclusion

In this paper we identified the problem of hard instances and presented DECA as an approach to deal with it. Specifically, we showed how doping an initial population with a solution to a single hard task instance improved the performance on two quite different tasks. Given the results on the box-pushing and food-gathering tasks we conclude that the problem of hard instances is alleviated by the application of DECA. Moreover, compared to traditional evolutionary algorithms, solutions discovered by DECA not only perform better on hard instances, but also perform better overall, i.e., achieve a significantly higher average fitness.

Our future work aims at determining for which tasks and under which conditions DECA performs better or worse than alternative techniques. Specifically, we will perform empirical studies in which DECA is compared with multitask learning (Subsection 7.3) and multi-objective learning (Subsection 7.4).

In addition to these empirical studies, an in-depth analysis of the doping effect is required to identify more problems to which DECA can be applied successfully. Future work aims at achieving such an analysis. To this purpose, the key assumption in our explanation for the doping effect, namely the supposed asymmetry of the search space with respect to easy and hard solutions (Subsection 3.2), needs verification. Furthermore, confirmation is needed for our belief that solutions to harder task instances

encompassing characteristics of solutions to easier task instances underlies DECA's success (Subsection 7.1). We intend to test DECA on a variety of benchmark problems, designed to exhibit specific characteristics with respect to the structure of the search space. Tracing the lineage of the best evolved solutions back to the doped solutions will be a key activity in understanding the factors responsible for DECA's success.

References

Arkin, R. C. (1998). *Behaviour-Based Robotics*. MIT Press.

Asada, M. and Kitano, H. (1999). The robocup challenge. *Robotics and Autonomous Systems*, 29(1):3–12.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28:41–75.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. MIT Press.

Dumitrescu, D., Lazzerini, B., Jain, L. C., and Dumitrescu, A. (2000). *Evolutionary Computation*. CRC Press, Boca Raton, FL.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley.

Goldberg, D. E., Deb, K., and Korb, B. (1991). Don't worry, be messy. In Belew, R. K. and Booker, L. B., editors, *Fourth International Conference on Genetic Algorithms*, pages 24–30. Morgan Kaufmann Publishers.

Grefenstette, J. and Ramsey, C. (1992). An approach to anytime learning. In Sleeman, D. H. and Edwards, P., editors, *Proceedinghs of the Ninth International Conference on Machine Learning*, pages 189–195, San Mateo, CA. Morgan Kaufmann.

Hancock, P. J. B. (1992). Genetic algorithms and permutation problems: a comparison of recombination operators for neural structure specification. In Whitley, L. D. and Schaffer, J. D., editors, *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 108–122, Los Alamitos, California. IEEE Computer Society Press.

Horn, J. (1997). Multicriterion decision making. In Bäck, T., Fogel, D., and Michalewicz, Z., editors, *Handbook of Evolutionary Computation*, pages F1.9:1–15. Oxford University Press, Oxford, England.

Jakobi, N. (1997). Evolutionary robotics and the radical envelope of noise hypothesis. *Adaptive Behavior*, 6(2).

Lee, W., Hallam, J., and Lund, H. H. (1997). Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots. In *Proceedings of IEEE 4th International Conference on Evolutionary Computation*, pages 495–499. IEEE Press.

Louis, S. J. (2002). Learning from experience: Case injected genetic algorithm design of combinational logic circuits. In Parmee, I. C., editor, *Adaptive Computing in Design and Manufacture V*, pages 295–306, Springer-Verlag.

Louis, S. J. and Johnson, J. (1999). Robustness of case-initialized genetic algorithms. In AAAI Press California Edts, editor, *Proceedings of the 12th International Florida Artificial Intelligence Research Symposium*, pages 129–133.

Louis, S. J. and Li, G. (1997). Combining robot control strategies using genetic algorithms with memory. In Angeline, P. J., Reynolds, R. G., McDonnell, J. R., and Eberhart, R.,

editors, *Evolutionary Programming VI*, pages 431–441, Berlin. Springer. Lecture Notes in Computer Science 1213.

Maniezzo, V. (1993). Searching among search spaces: Hastening the genetic evolution of feedforward neural networks. In Albrecht, R. F., Reeves, C. R., and Steel, N. C., editors, *Artificial Neural Nets and Genetic Algorithms*, pages 635–642. Springer-Verlag.

Matthews, K. B., Craw, S., Elder, S., Sibbald, A. S., and MacKenzie, I. (2000). Applying genetic algorithms to multi-objective land use planning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 613–620. Morgan Kaufmann.

Mondada, F., Franzi, E., and Jenne, P. (1993). Mobile robot miniaturisation: A tool for investigating in control algorithms. In Yoshikawa, T. and Miyazaki, F., editors, *Proceedings of the Third International Symposium on Experimental Robotics*, pages 501–513. Springer-Verlag.

Montana, D. and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 762–767. Morgan Kaufmann.

Skolicki, Z. and De Jong, K. (2004). Improving evolutionary algorithms with multi-representation island models. In *Proceedings of 8th International Conference on Parallel Problem Solving from Nature – PPSN VIII*, pages 420–429.

Soule, T. (2006). Resilient individuals improve evolutionary search. *Artificial Life*, 12(1):17–34.

Sprinkhuizen-Kuyper, I. G., Kortmann, R., and Postma, E. O. (2000a). Fitness functions for evolving box-pushing behaviour. In Van den Bosch, A. and Weigand, H., editors, *Proceedings of the Twelfth Belgium-Netherlands Artificial Intelligence Conference*, pages 275–282.

Sprinkhuizen-Kuyper, I. G., Postma, E. O., and Kortmann, R. (2000b). Evolutionary learning of a robot controller: Effect of neural network topology. In Feelders, A., editor, *Proceedings of the Tenth Belgian-Dutch Conference on Machine Learning*, pages 55–60. Tilburg University.

Spronck, P. H. M. and Kerckhoffs, E. J. H. (1997). Using genetic algorithms to design neural reinforcement controllers for simulated plants. In Kaylan, A. and Lehmann, A., editors, *Proceedings of the 11th European Simulation Conference*, pages 292–299. SCS Europe Bvba.

Spronck, P. H. M., Sprinkhuizen-Kuyper, I. G., and Postma, E. O. (2001a). Infused evolutionary learning. In Hoste, V. and De Pauw, G., editors, *Proceedings of the Eleventh Belgian-Dutch Conference on Machine Learning*, pages 61–68. University of Antwerp.

Spronck, P. H. M., Sprinkhuizen-Kuyper, I. G., and Postma, E. O. (2001b). Island-based evolutionary learning. In Kröse, B., De Rijke, M., Schreiber, G., and Van Someren, M., editors, *Proceedings of the 13th Dutch-Belgian Artificial Intelligence Conference*, pages 441–448. Universiteit van Amsterdam.

Thierens, D., Suykens, J., Vandewalle, J., and De Moor, B. (1993). Genetic weight optimization of a feedforward neural network controller. In Albrecht, R. F., Reeves, C. R., and Steel, N. C., editors, *Artificial Neural Nets and Genetic Algorithms*, pages 658–663. Springer-Verlag.

Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

Van Veldhuizen, D. A. and Lamont, G. B. (2000). Multiobjective evolutionary algorithms: Analyzing the State-of-the-Art. *Evolutionary Computation*, 8(2):125–147.

Wooldridge, M. (2000). On the sources of complexity in agent design. *Applied Artificial Intelligence*, 14(7):623–644.