Developing Box-Pushing Behaviours Using Evolutionary Robotics

Boris VAN LIERDE

2011
Abstract

The context of this report and the IRIDIA laboratory are described in the preface. Evolutionary Robotics and the box-pushing task are presented in the introduction.

The building of a test system supporting Evolutionary Robotics experiments is then detailed. This system is made of a robot simulator and a Genetic Algorithm. It is used to explore the possibility of evolving box-pushing behaviours. The bootstrapping problem is explained, and a novel approach for dealing with it is proposed, with results presented.

Finally, ideas for extending this approach are presented in the conclusion.
ACKNOWLEDGEMENTS

I would like to thank Elio Tuci for his help, interesting conversations, and enlightening guidance throughout the completion of this research project.

Hasan Fleyeh also has my extreme gratitude for his critical reading of this report, as well as for his enormous patience while waiting for the completion of this work.

Next, Dalarna University and the IRIDIA laboratory provided me with the opportunity for carrying out this project and for that I thank them too.

Finally I would like to thank all robotics and intelligent systems researchers, among which Rodney Brooks, C. Ronald Kube and Eric Bonabeau, whose work was an inspiration and a pleasure to read.
LIST OF FIGURES

Figure 1 The crossover operator ........................................................................................................................................ 11
Figure 2: visualising the simulated robots in 3D ............................................................................................................. 15
Figure 3: The whole system .................................................................................................................................................. 16
Figure 4: Schematic representation of the robot from above and from the front ............................................................. 19
Figure 5: Controller parameters that have evolved in the incubator are then tested in the simulator .................................................. 25
Figure 6: Evolution of the fitness for the Light Finding behaviour ................................................. 42
Figure 7: Screen shots of the progression of a robot implementing the Light Finding behaviour .......................................................... 43
Figure 8: First subcomponent ............................................................................................................................................ 44
Figure 9: Second subcomponent ..................................................................................................................................... 44
Figure 10: Third subcomponent ....................................................................................................................................... 44
Figure 11: Initial positions of the robots at each of the solution's nine evaluations ................................. 46
Figure 12: Box Pushing, fitness evolution ...................................................................................................................... 47
Figure 13: Simple Directed Box Pushing .................................................................................................................... 50
Figure 14: Directed box pushing, designed fitness function ............................................................................................ 52
Figure 15: Five successive configurations of the environment, of increasing complexity ................. 54
Figure 16: Incremental Box Pushing, version 3 ............................................................................................................. 56
Figure 17: Incremental Box Pushing, version 5 ............................................................................................................. 57
Figure 18: Incremental Box Pushing, version 6 ............................................................................................................. 58
Figure 19: Incremental Box Pushing, version 6 ............................................................................................................. 59
Figure 20: Incremental Box Pushing, version 6 ............................................................................................................. 60
Figure 21: Incremental Box Pushing, version 6 ............................................................................................................. 61
Figure 22: Incremental Box Pushing, version 6 ............................................................................................................. 62
Figure 23: Incremental Box Pushing, version 6 ............................................................................................................. 62
Figure 24: Configurations with a cost ............................................................................................................................. 65
LIST OF TABLES
Table 1: The genetic algorithm’s parameters................................. 26
Table 2: Experimental setup parameters........................................ 33
Table 3: Parameter values for the light finding experiment................. 41
Table 4: Parameter values for the collaborative undirected rectilinear box pushing experiment................................. 46
Table 5: Parameter values for the directed box pushing with minimal fitness function experiment.................................................. 49
Table 6: Parameter values for the directed box pushing with designed fitness function experiment.................................................. 52
1 Preface

1.1 Internship and research project

This report presents a research project completed during an internship finalizing a Master of Sciences in Intelligent Systems undertaken at the Dalarna University, Sweden.

The final project of the MSc's program is intended to be an application and a deepening of the subjects studied, as well as an investigation into related subjects.

In that way, robotics is an extremely interesting application of Intelligent Systems. They also present several noteworthy aspects, some of which are mentioned below.

Technically, robotics or more precisely (but not exclusively) robot controllers are increasingly bound to the various fields of AI.

From the commercial point of view robotics are said to be on the verge of becoming a major industry, with applications ranging from space exploration where it already occupies a prominent position, to nano-robotics through domestic robotics.

Philosophically, several concepts (stemming mainly from Complexity theory, Artificial Life, behaviour-based or bio-inspired robotics) such as emergence or self-organisation are thought-provoking on the subject of the nature of life.

Finally, robotics is a cultural subject of interest that is fascinated to many people, including myself. For those reasons, robotics was chosen as the subject of this project. The research was carried out at the IRIDIA laboratory, which is presented in the following section.
1.2 The IRIDIA laboratory

The Institut de Recherche Interdisciplinaire en Intelligence Artificielle (IRIDIA) is a laboratory of the Engineering faculty of the Université Libre de Bruxelles (ULB). The IRIDIA’s activities are grouped in four main domains, as described in the following introductory text from their website:

IRIDIA is the Artificial Intelligence research laboratory of the Université Libre de Bruxelles. It is deeply involved in theoretical and applied research in computational intelligence. The major domains of competence are: (i) swarm intelligence, (ii) metaheuristics for the solution of combinatorial and continuous space optimization problems, (iii) the foundational study of biological networks, and (iv) business intelligence applications.

Our research program in swarm intelligence is centered around the design of algorithms or distributed problem-solving mechanisms using the collective behavior of social insect colonies as main source of inspiration. In particular, we have proposed innovative algorithms for the solution of different types of optimization problems and for the control of swarms of robots.

A metaheuristic can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem. In this research area, we have proposed the ant colony optimization metaheuristic and we are world leaders in various stochastic local search methodologies such as iterated local search and evolutionary computation.

Concerning biological networks, our main interest is in the study of neural networks, immune networks, and chemical reaction systems and in the identification of what are their common features and mechanisms. We are also interested in exploiting the results of these studies for the conception of adaptive distributed engineering artefacts.

Finally on the very practical side, IRIDIA develops business intelligence applications such as data mining and object oriented solutions for companies and administrations.
2 Introduction

The subject of this research project was the development of controllers that generate behaviours allowing a robot to accomplish a specific task. After choosing a suitable task for the robot, the challenge was to generate such controllers by using a method named Evolutionary Robotics (ER). The term “suitable task” is to be understood as a task that is not trivial and -more importantly- that deals with various aspects of the behaviour of a robot.

This report presents the system that was built to carry out the experiments, the methods that were used, the main obstacle that was met, the way it was dealt with, and the results that were obtained.

2.1 Choosing a task: Box pushing

For this work, a non-trivial and novel (in the ER perspective) problem was chosen. A great deal of work in ER has been done on relatively simple tasks, such as obstacle avoidance [i][ii], with fewer studies of complex tasks, such as retrieving a sequence of locations [iii] or objects [iv] in a complex environment, or executing a series of movements [v][ix].

Box pushing has often been used by roboticists as a test-bed for new techniques in robotics [vi][vii][viii]. Although the task is simple to describe – finding a box and pushing it from an initial location to another given location – it involves treating several aspects that are important when designing robot controllers.

Together these aspects cover many of the difficulties raised by the interaction of any robot with the real world, and roboticists are most likely to meet them when designing systems. Therefore their study is of great importance.

Those aspects can be grouped into four categories:

- Perceptual aspects: the location, recognition and identification of other robots, of the obstacles, and of the box, all of which may be changing over time;
- Effectual aspects: the intentional modification of the environment by the robot, by application of a force to an object, directly with an arm or indirectly through the torque of its wheels;
• Cognitive aspects: the planning of actions based on the perceived world in order to accomplish a task;

• Collaborative aspects, when several robots are involved: direct communication [ix] or indirect “stigmergic” communication.

These categories are consistent with the logical task organisation used in traditional robotics: Perception – Cognition – Reaction (with the collaborative aspect being a superset of the others). The task organisation in traditional robotics design paradigm being: (1) acquire information about the outside world; (2) build an internal representation of the world; (3) elaborate a plan to solve the problem in the representation; (4) carry out the plan in the real world with a series of actions; and repeat.

An interesting facet of the box pushing task is that it involves all four categories, and intertwines them in a complex way. For example, the effectual aspect is not just about the robot moving its own body, but also about moving other objects including other robots, which increases the difficulty of the perceptual aspect, requires to perform more forward planning and to take into account the other robots.

All these aspects that are found in box pushing are also found in many other tasks. They can be considered building blocks for other behaviours. “Solving” box pushing allows insight into those aspects, which can then be used to solve different and potentially more complex tasks.

2.1.1 Previous work

The article that bought up the idea of working on this task is “Cooperative transport by ants and robots”, by Kube and Bonabeau [x] which is a presentation of their efforts in applying methods stemming from a theory called Behaviour-based robotics [xi] to the box pushing problem.

---

4 Stigmergic communication is an indirect process in which information transmission between two entities is not made directly or purposely, but rather indirectly and incidentally through the environment. For example, a box moving in a certain direction might be perceived by a robot and may or may not be interpreted by the perceiving robot as the result of the actions of other robots on that box, the point being that the movement of the box will have an effect on the perceiving robot's behaviour. And whether it was intentional or not, information transmitted from the robots pushing the box to the perceiving robot.

5 This does not necessarily mean there will be any “planning ahead” per se in the robot controller, but it is a convenient way of describing the phenomenon of a robot taking an action that has an effect several time steps ahead, as if it was entirely planned.
Behaviour-based robotics is a design paradigm in the designing of robot controllers where the concepts of embodiment, “local planning”, subsumption and emergence are crucial. Although they have been widely recognised as a significant and promising departure from traditional robotics, there still is a great deal of human work involved in the design of such systems.

The position of this research project in relatively to Kube and Bonabeau's work was to investigate whether a similar task could be solved with Evolutionary robotics, and whether it could be solved in an entirely autonomous manner.

Kube and Bonabeau present a successful implementation of a controller that allows a group of homogeneous robots to perform directed uncoordinated collaborative box pushing. Simply put, several identical robots with no means of direct communication manage to push a box from a random location to a predefined goal. The box, the robots and the light that defines the goal are all placed in a circular wall-delimited arena.

2.1.2 Restrictions relative to the previous work

In relation to Kube and Bonabeau's experiment, the scope of this work was reduced as to the number of robots used. Their work was essentially focused on collaborative transportation, while this work studied only individual robots. The reason for this was the necessity of reducing the complexity of the problem because of the time constraints on this research project, which was done by removing the communication aspects of the task. (Nevertheless, one experiment set is presented where two collaborating robots are evolved, but those multi-robot experiments were rather meant to test the ER system itself.)

There is another restriction regarding the physical deployment of the experiment. Typically, a successful ER-developed controller will be tested on the real-life robot upon which the simulated robot used in the evolutionary process was based. However, again due to time constraints and to keep the focus on the development itself, the physical implementation was not carried out.

2.2 Evolutionary Robotics

Evolutionary robotics (ER) is about using “soft” techniques to evolve robots.

Indeed, traditionally robots and robot controllers are engineered according to “hard” established design methods. In contrast ER systems, using AI techniques, create the robots or the controllers
autonomously. The particular AI techniques that are used include Genetic Algorithms and Artificial Neural Networks.

The main motivation in using ER is the possibility of producing robots or robot controllers that would have been difficult to engineer using traditional methods. The reason why such engineering might be difficult is a lack or total absence of knowledge on the task's specific requirements, available resources, or evaluation methods. This can occur for example when the robots have to operate in inaccessible and less well known environments such as deep water, space, extraterrestrial planetary surfaces, inside a living body or simply in familiar environments but at a nanoscopic scale.

Another motivation for using ER, only mentioned here for completeness, is the study of biological phenomena with the help of ER based simulations. Fields concerned by this application include biology and more precisely the theory of evolution, psychology, or neurology.

2.2.1 Historical background

Evolutionary robotics was born at the confluence of three similar paradigm shifts that occurred in three independent study fields: artificial intelligence, linguistics, and psychology.

The common element in the three shifts is the departure from a conception of mind and body, or intelligence and body, as two different and distinct entities that can be studied independently. In the new representations, the two are intertwined and cannot be isolated from each other.

Traditional Artificial Intelligence (AI) has attempted to model the human mind using concepts such as an inner representation of the world and reasoning based upon this representation. More precisely, the artificial entity (robot or computer) is designed to gather information about the world, build a representation of it based on that information, reason with elements extracted from this representation, and make conclusions or decide on actions to take.

In 1986, Rodney Brooks, frustrated by the lack of success of traditional AI-based robotics, developed a new approach, the subsumption architecture [xii], which eventually led to behaviour-based robotics.

Similarly, linguistics attempted to model human language with a set of rules applied to a set of symbols. Language was seen as a purely logical manipulation of these symbols. There was a belief that language could be reproduced by an artificial entity provided that the correct rules were extracted and implemented in the entity.
Confronted to the relatively poor performance of those attempts, McClelland and Elman [xiii] introduced a non-symbolic representation of language, and attempted to model it using Artificial Neural Networks.

Finally, psychology saw the arrival of concepts such as embodiment and “situationness” in the theory of Embodied Cognitivism (also known as Embodied Philosophy). According to this theory, the perceptual structures of the intelligent entity shape its conceptual structures. Those conceptual structures are not based on abstract symbols and representations, but on metaphorical representations constructed from passed experiences.

From these different concepts emerged new ideas. Artificial Neural Networks had traditionally been used to get the best possible output for a given input. Intelligence was seen as giving the correct answer to a question. But once confronted to the idea of embodiment, the Neural Network is used by an embodied entity such as a robot for giving the output that in return gives the best next input, effectively bringing the body to a satisfying situation, where satisfaction is measured by the entity's survival.

Intelligence is thus defined as the ability to stay alive, and mind and body are but one.

2.2.2 The bootstrapping problem

A resourceful nobleman of the German folklore once found himself trapped in a swamp (or sinking in a deep lake according to some versions), but resolutely pulled himself out of danger by grabbing the straps of his boots and lifting his own weight to safety. At least so goes the legend related by Rudolf Erich Raspe in “The Surprising Adventures of Baron Münchausen”. And that legend gave its name to the bootstrapping problem.

The bootstrapping problem in the context of ER refers to the initial phase of an artificial evolution. The entire population of solutions, having been initialized with random values, will most likely be given a small score by the fitness function. The more complex the task, the less likely it is that any individual succeeds in accomplishing even a small part of it. All individuals being equally mediocre in completing the task, the evolutionary algorithm will have no indication on which individuals have the most potential for generating the next generation, thus undermining the chances of that next generation of building up better behaviours. Because of that, the evolution never gets a chance to “take off” and might stagnate indefinitely. It must be understood that this problem only occurs when the problem to solve or the task to accomplish requires several intermediate steps.
If for example the robot is required to push a box to a goal, it must accomplish the following steps: find the box, go to the box, position itself behind the box in relation to the goal, push the box while maintaining the correct direction and the correct orientation in relation to the box, correct any errors in the trajectory, and finally stop pushing as soon as the goal is reached.

Trying to reach a behaviour which would accomplish all these tasks, and accomplish them in the correct order starting from a random initial behaviour is almost equivalent to performing a random search.

There are several ways to overcome this problem, perhaps the most obvious of which is to add components to the fitness function which would reward the accomplishment of sub-tasks defined by the experimenter. Continuing with the example, robots could be rewarded when they are touching the box, rewarded further when their body is facing the goal and even more when their body, the box and the goal are aligned, and so on.

It will be shown that this method presents a fundamental flaw and moreover gives unsatisfying results. An alternative method will be presented later on in the experiments section.

2.2.3 Using AI techniques

Evolutionary robotics is part of the wider family of Artificial Intelligence and more specifically of Artificial Life. They are built upon techniques that come from those fields.

Typically, an ER system will evolve a robot controller by tuning its parameters with a Genetic Algorithm (GA). Evolution of physical or morphological characteristics has been done as well, even though this is rarer. Generally that controller will be based on a Neural Network (NN), but the use of other controller cores has been studied, such as Fuzzy Logic systems [xiv].

More details on the specifics of an ER system are given below, while an introduction to the two particular AI techniques used in this work (GA's and NN's) is given in the following sections.

2.3 Genetic algorithms

2.3.1 Biological inspiration

When observing life in nature, one can see every living being, even within a same species, is different from another. Different species are all adapted to their respective environment: they are able to exploit
the existing resources no matter how scarce they are, and at the same time they are able to avoid the hazards of that same environment, letting them survive as a species.

Physical and physiological characteristics are individual and apparently at least partially inherited. In genetics, the set of all characteristics constitutes the phenotype. The phenotype is an expression of the genotype. The mapping from genotype to phenotype is a very complex process, where randomness plays at least some role.

The genotype is the set of all genes of one particular individual. A gene in turn is a functionally and geographically well defined set of information, which will express a certain characteristic, or contribute to the expression of a characteristic.

When sexual reproduction occurs, the genes of both parents are combined. Mutations may occur as well during the process, but in a more limited way.

The newly created individual's genotype is then translated during growth to form the individual's phenotype.

Among its set of physical and behavioural characteristics, some will differ to some extent from that of its parents. Most of these differences will have little influence on the individual’s fitness, fitness being the measure of an individual's adaptation to the environment it lives in. Some characteristics on the other hand might have a non null influence on the fitness. According to the evolutionary theory, those individuals with greater fitness have greater chances of survival and have more opportunities to reproduce, thus being more likely to disseminate their genetic material.

In that way, a phenomenon of constant, open-ended adaptation to the environment emerges.

The open-endedness of the process is a two-faced coin. The downside of it is the inefficiency of the search: many times an individual or a whole branch of the tree of solutions will find themselves stuck in a dead-end and disappear, potentially leading the particular species to extinction. The benefits on the other hand are the adaptability and the creativity of the process in finding genuine solutions.

### 2.3.2 Adapting the principles of genetics to artificial systems

Optimisation and search of solutions in high dimensionality spaces are domains in which classical software engineering still struggles, especially when confronted to problems which require algorithms that have high complexity bounds.
Some of those complex problems do happen to present similarities with the kind of problem with which biological evolution deals. The question thus arises of whether the genetic process presented above could be used to tackle them. That is where the idea of Genetic Algorithms came from.

2.3.3 Genetic algorithms principles

Genetic algorithms are a simulation of the mechanisms of natural evolution.

An individual in the artificial population represents one possible solution to a given problem, or one particular method for solving a given task. A solution is defined as a set of values for the parameters of the problem or task.

For instance, let there be a problem \( P_{a,b}(x, y)=z \), where the task is to obtain certain values for the output \( z \) given certain values for the input \( x \) and \( y \). The value to obtain for \( z \), given the values of \( (x, y) \) is defined by the experimenter. What the GA must do is find values for the parameters \( (a, b) \) of the problem so as to develop the “right behaviour” as defined by the experimenter.

At the first generation \( g_1 \), \((a, b)\) for each solution \( i \) in the pool of solutions are initialized with values \((s_{1,i}, t_{1,i})\). These values can either be random, predefined and constant across a set of experiments, or determined by heuristics.

At each generation \( g_n \), each one of the solutions \((s_{n,i}, t_{n,i})\) is evaluated by what is called the fitness function, and given a score. The fitness function gives a measure of the distance between the values given by the solution and the desired values.

The way the solutions \((s_{n+1,i}, t_{n+1,i})\) at generation \( g_{n+1} \) are created is by the use of one or a combination of three genetic operators applied to the solutions of the previous generation \( g_n \). The three operators are crossover, mutation, and selection.

In the crossover operation, two parents are chosen (most likely well performing individuals, corresponding to solutions that obtain a high fitness score), then extract values from both sides and recombine them (see Figure 1). The way the values are chosen is up to the experimenter, but is often random. For example the solution \( k \) at generation \( g_{n+1} \), \((s_{n+1,k}, t_{n+1,k})\) might have been constructed from solutions \( i: (s_{n,i}, t_{n,i})\) and \( j: (s_{n,j}, t_{n,j})\), where the first value of the new solution is the first value of solution \( i: s_{n+1,i} = s_{n,j}\) and the second value is the second value of solution \( j: t_{n+1,i} = t_{n,j}\). In that case, the genotype (of length two) is split in the middle.
Note that in the example, there are only two parameters to the problem to be searched, and the genotype consists of two values which are the values of the parameters, whereas in the figure there are fourteen elements in the genotypes. But this does not necessarily mean there are exactly fourteen parameters to be searched. The way the parameters are encoded is completely up to the experimenter, and there does not need to be a direct correspondence between the two.

In the mutation operation, one of the values of the child's genome is simply altered by a random value $\varepsilon$ (usually small). $s_{n+1,i} = s_{n,i} + \varepsilon$.

The selection operation is the process of allowing an unaltered solution to be reused in the following generation. Typically the solutions which yield the highest fitness value have a higher chance of being selected. In many cases, bad solutions are not systematically eliminated because they allow the exploration of the problem space without restricting the search to local maxima.

### 2.4 Continuous time recurrent neural networks

Artificial Neural Networks (NN) are a calculation method inspired by the functioning of biological neural networks. They are good at taking classification and generalisation decisions based upon learned perception mechanisms rather than formal logic. Another thing at which they are good is function approximation, be it of a function that is known but difficult to compute for a given input in a
short time, or of an unknown function. For this reason, they are suited to controlling robots that have to take real-time decisions based upon complex input.

Amongst the different types of NN that are used in ER, continuous time recurrent neural networks (CTRNN) are quite powerful \[xv\]. They use a mechanism that effectively creates a memory effect in the network. Roughly speaking, each neuron uses its state, or activation level, from previous iterations as one of its inputs, with the weight of the previous states changing as time goes by at an individual rate that can be subject to evolution just as the network’s other parameters. The connection map between the neurons is unconstrained, even allowing for a neuron to connect to itself. The result of such a configuration is a dynamical system the output of which may vary even when given an unvarying input. An information-retaining mechanism is thus implemented, allowing the system to “remember” past states.

A CTRNN is made of a network of connected neurons that each have a state. The new state of any given neuron \(i\) in a CTRNN at every step is given by

\[
\hat{y}_i = \frac{-y_i + \sum_{j=1}^{N} w_{ji} \sigma(y_j + \theta_j) + I_i}{\tau_i} \quad i = 1, 2, \ldots, N
\]

where \(y_i\) is the current state of the neuron \(i\), \(w_{ji}\) is the weight of the connection from neuron \(j\) to neuron \(i\), \(y_j\) is the current state of neuron \(j\), \(\theta_j\) is the bias of neuron \(j\), and \(\tau_i\) is the time-constant, and \(\sigma(x)\) is the standard logistic activation function

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
3 Building of a test system

In this section the steps taken in building a system to support this project's Evolutionary Robotics experiments are described, as well as the system itself and its characteristics.

3.1 Preliminaries

Simply put, for carrying out an Evolutionary Robotics experiment, two things are needed: an evolutionary process and robots. More precisely, an ER system must be able to generate and analyse an evolutionary process and must allow implementing either in a physical way or in a simulated way the solution that has been evolved. (There is no fundamental difference between a system that uses physical robots and one that uses simulated robots\(^6\), but for practical reasons simulations are very often used in ER, as explained further below).

So the system is made of two components: the incubator, for the evolutionary process, and the simulator, for the robotics aspect.

3.1.1 System requirements

The first step in designing any system is defining some general requirements. Most important to the study of behaviours developed with Evolutionary Robotics is the ability to conduct experiments and then to measure the results. Those abilities are covered by the following requirements:

- experimental flexibility;
- reproducibility;
- interpretability;
- traceability.

Below those terms are explained in detail and the components that were implemented to support them are presented.

---

\(^6\) Although R. Brooks would disagree: according to him the inherent simplifications present in any simulation are so great that they prevent a usable solution to be developed. This was one of the motivations for inventing the subsumption architecture.
First, “experimental flexibility” is the possibility for the experimenter to carry out many different types of experiments with varying parameters. For that a system of user-defined run-time parameters was implemented where many aspects of an experiment (amount and position of robots, of obstacles, shape of the arena, etc...) which all together define an experiment's setup can be defined by those parameters.

Thanks to this, the experimenter is not limited to a predefined setup but has the possibility of running different types of experiments and sets of experiments with varying values without having to recompile the whole system. Additionally, the values of the parameters of an experiment may be saved along with its output for later use.

Secondly, “reproducibility” is the ability to reproduce similar experiments and to retrieve results that are fairly consistent across the successive run of those experiments. This is made possible through the same mechanism of user-definable parameters mentioned above.

Thirdly, “interpretability” is the ability to qualify abstract numerical results, i.e. to see and to evaluate the evolution of the fitness of a robot, and to visualise the behaviour of the robot. The evolution of fitness can be visualised thanks to graph plotting tools which are fed the log of an experimental run, while the movements and behaviours of the robot and the arena can be visualised thanks to a 3D tool. Figure 2 is a screen-shot of the 3D visualisation tool, displaying an experiment involving two robots.

7 Insofar as evolutionary phenomenon can be reproducible. Being stochastic by nature, variations are always to be expected, and are actually beneficial to the evolution.

8 This tool is based on the 3D visualisation component of the ODE physics simulator. That component is a wrapper around the OpenGL API. The ODE library is the one used by the GA's fitness function.
The robots' direction is indicated by the short green lines. They are equipped with infra-red sensors, the beam of which is represented by a red line. (Here only one beam is visible because the other ones are being “absorbed” by the box). The on-board lights (green for the box, yellow for the robots), are represented by small spheres. Finally, four landmark pyramids delimit a one square meter area.

And finally, “traceability” is the ability to retrace the history, or the genesis of a species as well as to retrieve the initial setup of an experiment. This ability is of course key to the other two. It is provided by a logging and data processing mechanism.

3.1.2 Existing system

The simulator built by Elio Tuci in the scope of an experiment in collaborative behaviour [6] was used as a starting point.

The simulated robots are based on the SBot\(^9\), a modular and versatile platform that has been extensively used at the IRIDIA lab. All the sensors used in the experiments presented here are models of existing sensors on the Sbot. (The actual physical robot carries much more sensors than were used in this particular work).

\(^9\) http://www.swarm-bots.org/

http://en.wikipedia.org/wiki/Sbot
The physics simulation is built upon the Open Dynamics Engine (ODE) library\(^{10}\).

### 3.2 The whole system and the evolutionary process

In the figure below the whole system is presented schematically. The *incubator* contains the GA that evolves the robot controllers, and interacts with the *simulator* to evaluate the populations' solutions. (The simulator can also display and run any logged solution in a 3D graphics window). The *experiment* part initializes the other two components, launches the evolutionary process, logs the output, and generally orchestrates the system.

![Figure 3: The whole system.](image)

### 3.3 The simulator: an arena, robots and objects

The real-world simulator is a key component of any ER system. Its role is very straightforward: it serves as a sandbox for the virtual robots, which are given scores depending on how well they behave in the virtual world. Technically the simulator is used by the Genetic Algorithm during the evolutionary process as a tool for its fitness function.

There are two reasons why simulators, rather than real-world robots, are used in ER. Firstly for time considerations: given that several thousand generations may easily be necessary for the emergence of good quality behaviour, it is simply not practical to have the evaluation of solutions done in real time.

\(^{10}\) [http://www.ode.org](http://www.ode.org)
Secondly there is the risk of the robots physically damaging themselves should they be left in the control of a wild randomly initialized neural network.

Below is a description of the various objects that are simulated, presented from a physical and from a software point of view.

3.3.1 The arena

The arena is a flat rectangular area enclosed in four walls. It is the stage for every other object of the experiment. It may contain lights, obstacles and boxes (obstacles, such as the arena walls, are immoveable objects while boxes are movable).

In the simulation, the ground has a variable and randomly initialized friction factor across its area. Meaning the simulated boxes and robots will meet variable resistance while travelling across it. This has the advantage of offering to the controller the opportunity to learn how to deal with partially inconsistent reactions from a box that is being pushed around, thus preparing it to situations encountered in the real world.

Technically the area's total surface is divided in a number of sub-areas. This number is fixed as of now, independently from the area's size. At the start of an experiment or of an evaluation, depending on how the particular experiment was set up, each area is assigned a random value. During the simulation, at every time step, the collision points – which are the points at which the ground and the robot or the ground and the box touch each other – are assigned a friction factor proportional to said value.

On the software side of things, the arena is represented by the class Arena, which has responsibility over many aspects of the simulation. It is the Arena that creates the other simulated objects, and at every time step it instructs each of those objects to take one step. These objects do the same to all of their sub-objects, so that ultimately all the objects in the arena will have advanced one step in time.

\[11\] An evaluation is one of several runs, or “chances” that a solution is given during its fitness evaluation. It consists in a certain amount of time that the corresponding robot is allowed to spend in the arena to perform the given task. The solution's fitness is the sum of the results for all its evaluations.
3.3.2 The boxes and the obstacles

A box is a parallelepiped of a certain mass. It may be displaced if sufficient force is applied. It might be equipped with a lamp detectable by the robots' cameras. The obstacles on the other hand are not moveable.

The implementation of the simulated boxes and obstacles in software is straightforward, the concepts of those being close to the ODE concept of a Box. The only addition to the concept is the possibility that a box possesses to define “sectors” around itself. Boxes' sectors serve a particular use in a certain type of experiment's fitness evaluation and will be explained later on.

3.3.3 The ambient light

The ambient light is a light detectable by a certain type of sensor, and is used throughout the box-pushing experiments to indicate the goal towards which the robots must push the box.

3.3.4 The robots

The robots typically carry a number of the following sensors: active infra-red (IR) sensors that are used as low-precision high-noise proximity sensors, cameras that are able to return the distance to the nearest object of a given colour in each of a set of sectors, and ambient light sensors that return the global light intensity. Cameras, being more accurate than ambient light sensors, will rather be used to detect LEDs which are carried by robots, while ambient light sensors will detect the general direction of the lights placed in the arena.
Robots are also equipped with actuators. Two motor-driven wheels that are able to perform on-the-spot rotation are controlled by two continuous-valued outputs from the robot controller. The robots might also be equipped with an on-board light which can be triggered by their controller as well. In that case the continuous value of the controller output is mapped to a discreet (boolean) value using a simple threshold function.

Finally, the robot's controller is not simulated as a physical object. Its mechanisms are merely reproduced in a “regular” software component. Indeed, since the physical device – most likely an embedded controller – can be emulated by the computer, simulating its physics does not make much sense. The controller is fed the robot's sensory output, computes its own output, which is then fed to the robot's actuators.

All sensors and actuators are presented in more detail in the following section.
3.3.5 The sensors

Sensors, as well as actuators and controllers, will typically be placed on a robot. (Even though there is no restriction on placing them anywhere in the arena, be it in the real world or in the simulator).

3.3.5.1 Infra-red sensors

The infra-red (IR) sensors used here are in reality an infra-red light emitter coupled with a receiver, in which configuration they are called *active* infra-red sensors. They can be used to detect obstacles by measuring the amount of reflected light that bounces back from a nearby obstacle. Their precision is relatively low, being sensitive to noise and unable to distinguish their own reflected light from that of another sensor.\(^\text{12}\)

The implementation is based on a set of measurements made with real IR sensors such as those found on the S-Bot. The reading of a real sensor was recorded for different distances and angles from a flat obstacle. The simulator then picks the reading corresponding to the current situation in the simulated world, and this reading becomes the output of the simulated sensor, after a random amount of noise (between -15% and +15% of the initial value) has been added. In other words, the simulator measures the distance and relative angle of the IR sensor from the closest obstacle, and reads in a file the value corresponding to the same distance and angle in the real world.

In the 3D visualisation of the simulation, the IR sensors' light beams are represented by short red lines of slightly varying length. The variation is the expression of the random noise that is introduced into the reading.

3.3.5.2 Camera

On the S-Bot, the physical characteristics of the camera (resolution, range...) are hidden behind an interface that also acts as an image pre-processing unit. Indeed, the colour 360-degree image is converted to the following information: for every sector of the image, the distance from the camera to the brightest light in that sector, or 0 if no light is detected. The camera does not react to the diffuse illumination of a lamp (such as the arena lights), but only to the small sharp light of a LED (such as the robots’ and the boxes’ on-board lights). A sector is a pre-defined portion of the 360-degree

\(^{12}\) That characteristic is not necessarily disadvantageous. Even though it is not use it here, it could be used for robot communication.
horizontal angle of vision such that the whole range is divided in sectors of equal angular amplitude. There is a constraint with the real camera that restricts the number of sectors to be a power of 2.

### 3.3.5.3 Monochrome and proximity cameras

The monochrome and proximity cameras are similar to the basic camera, but the image pre-processing is slightly different. It is worth noting that although there are three different types of cameras, and that a simulated robot can have any number of cameras on board, there is always only one physical camera on the robot that makes a series of measurements which are separated during pre-processing into a set of inputs.

In the case of the monochrome camera, only one colour is recognised, that has to be defined beforehand (at the same time that the number of sectors is defined). In the case of the proximity camera, only the one closest light source across the whole angle of vision (across all sectors) is taken into account, and its location is returned in terms of sector and distance.

The utility of the monochrome camera is focusing the user's (the robot's) attention to a certain object. For example, there could be a green light on the box and a yellow light on every robot, and two monochrome cameras on each robot, set to recognise green and yellow respectively. That way the robots' controllers would have two distinct sets of inputs, making their evolution and their task easier.

The utility of a proximity camera is data dimension reduction, which is commonly performed in the pre-processing stage of an artificial neural network.

Both cameras are introduced to facilitate the controller's evolution, by data discrimination and by data dimension reduction. Theoretically, the GA could learn to discriminate the data by itself, and should well do so in order to stay true to the principle of autonomous evolution.

### 3.3.5.4 The ambient light sensors

Unlike the camera, the ambient light (AL) sensors react to diffuse, non-directional light. In the proximity of a light placed above the arena a robot with two ambient light sensors, one on each side, will relatively easily learn to turn towards the light simply by rotating until its two AL sensors receive the same amount of light and give the same readings.

The simulator uses a mechanism similar to that of the simulated IR sensor for the AL sensors.
3.3.6 The actuators

The SBot has a number of actuators including prehensile devices such as grippers, but only two types of actuators will be used here: the motor-driven wheels and the lights.

3.3.6.1 Motor-driven wheels

The robot's locomotion is achieved through a pair of motor-driven bi-directional wheels. Rotation is performed by varying the differential speed of the wheels. It also has a pair of threads located closer to the centre of the robot, but in a flat terrain such as the one used in this experiment they are functionally equivalent to the wheels, therefore they are not represented in the simulation.

In the simulator, the motors' inputs are a continuous signal in the range [0,1] that map to the wheel's range of speeds from full-speed backwards to full-speed ahead. Thanks to the realistic physics simulation provided by ODE, the wheels come up to speed progressively under the pressure of the applied forces, as opposed to instantly spinning at the target speed.

3.3.6.2 On-board lights

An on-board light is a light that is physically bound to an object of the simulation, typically a box or a robot. On-board lights are recognised by the various cameras, while ambient lights are recognised by the AL sensors as mentioned earlier. In the real world on-board lights are LEDs while ambient lights are high-intensity, wide-diffusion lights.

3.3.7 The controllers

Functionally the controller of a robot does nothing more than map an input received from sensors to an output which is then fed to actuators. In some cases, such as with the controller that was used in some of the experiments, previous states are taken into account as well.

The controllers that were used throughout this work are presented below. Those that were evolved were all neural networks of some kind, and one is a utility remote-controller to be used by the experimenter.

3.3.7.1 Neural-network based controllers

These controllers use a neural network for performing the transformation from input to output.
The way the transformation is done and thus the behaviour of the robots was dictated by the type and the topology of the network (perceptron, CTRNN, multi-layered or not, with recurrence or not,...), by the values of the parameters of each neuron in the network, and by the connection weights from one neuron to the other.

The type and the topology of the networks are fixed before the beginning of an experiment by the experimenter, and are not subject to evolution.

The values of the parameters and the connection weights on the other hand are evolved, and then are used in the controller of the robots present in the simulator. They are not modified any further during the duration of a simulation.

3.3.7.2 Remote controller

The remote controller is a purely software tool built into the simulator for testing purposes. It makes it possible to control a robot with the keyboard during a visualisation session. Such a controller is of no use during the evolutionary process, but can be useful while visualising the behaviour of the evolved robots. It has no equivalent in the physical world, although it certainly would not be difficult for an electronics engineer to build one using simple RF technology.

For example, one may wish to evolve a light-following behaviour. For that, a light would be placed at different random places inside the arena during the evolution, and the robot encouraged finding the light. Then, during the live visualisation of the behaviour, the arena light would be suppressed in favour of an on-board light fitted on a remote-controlled robot. The experimenter could then see how well the evolved robot would chase the light-carrying remote-controlled robot despite its movements.

3.4 The incubator: the Genetic Algorithm

There are two sets of values to be considered regarding a GA: those values that the GA is responsible for evolving and the values given to the parameters of the evolutionary algorithm itself. In this section, both the values being evolved during the experiments and the GA's parameters are presented.

3.4.1 What is being evolved

The object of the evolution is the values specifying the neural network that controls the robot. In these experiments there is no evolution of the topology of the network, of the morphology of the robot, or of anything related to the sensors or controllers.
The semantics and quantities of those specifying values depend respectively on the type and on the size of the particular neural network being used. (See a previous section in the introduction for details on the parameters of a CTRNN.)

The experimenter can define which of the parameters to evolve, how they are represented in the genome, and how the mapping from genome to phenome occurs, the phenome in this case being the NN's values. Obviously the size of the genome must correspond to that of the neural network, not necessarily directly but certainly in terms of quantity of input and output to that mapping function.

The product of the evolution of a certain robot will not be compatible with a robot of a different kind. A different amount of sensors and actuators on the robot requires a different quantity of controller input and output connectors. Furthermore, an equal amount but different kinds of sensors and actuators would give a compatible controller in terms of I/O, but that controller would be unable to perform meaningful control in this unfamiliar environment. Finally, a correct number of genes for a specific controller but with a different mapping function would yield a useless controller.

There are many ways to map a genome to a controller. In biology the mapping from genotype to phenotype or in other words the way the genes are expressed, is a complex and not well understood process. It has also been, and still is, a subject of discussion in the AI community as it has been shown how relevant a proper mapping may be to evolving good solutions [6].
A simple and straightforward way of doing the mapping is shown in the figure above: the mapping from genome to neural network is direct (no interpretation of the values is done) and there is one value in the genome for every value in the network. For this work, a similar simple direct mapping method was used.

### 3.4.2 The Genetic Algorithm's parameters

The GA implementation that is used in this system comes from Elio Tuci’s research work, and allows the experimenter to vary its operative parameters. Below is a presentation of those parameters along with a short description of what they do and the values that are used in the experiments. Some values are defined by the experimenter at experiment time, while others are fixed to values that were suggested by Elio Tuci.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Value used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>population_size</td>
<td>Defined at</td>
<td>Amount of individual solutions that make up the population.</td>
</tr>
</tbody>
</table>

---

Figure 5: Controller parameters that have evolved in the incubator are then tested in the simulator.
num_genes Defined at experiment time. Number of genes that make up an individual.

prop_sel_out 0.2 Proportion of the population that will be purged out of the population at the end of every generation without being allowed to reproduce.

num_elite 3 Number of elite individuals that will be passed down to the next generation, unmodified. They are still eligible for reproduction.

prob_crossover 0.3 Probability, for any given individual, to reproduce.

prob_mutation 0.15 Probability, for any given gene except those of the elite, to be mutated when passed down.

highest_gene_value / lowest_gene_value 1.0 / 0.0 Respectively the highest and lowest continuous value that a gene can take.

r_rand Semi-randomly initialised at experiment time. A random number generator.

Table 1: The genetic algorithm’s parameters.

In the algorithm used in this work, the mutation of the value of a gene is done by adding to its value a Gaussian random variate with mean 0 and standard deviation 0.1, using the GNU Scientific Library (GSL)\(^\text{13}\).

Crossover is done by using a single “cutting point” chosen randomly again using the GNU Scientific Library, and then by building a child genome by combining the genetic sequence at the left of this point from one parent, and the genetic sequence at the right of this same point from the other parent.

3.5 Using the system

Below is a presentation of the Use Cases (UC) of the system roughly based on the RUP\(^\text{14}\) method. The system has five use cases:


\(^{14}\) IBM Rational Unified Process
“Launch experiment”, to start an instance of an experiment;

“Interrupt experiment”, to halt an ongoing experiment;

“Resume experiment”, to resume an experiment that has been interrupted;

“View experiment”, to visualise the results of an experiment;

“Create experiment”, to create and define a new type of experiment such as “Light Finding” or “Incremental Box Pushing”.

It is designed for three users:

- Experimenter, who sets an experiment's parameters and launches it;
- Experiment Viewer, who views the experiments' results;
- Experiment Creator, who creates and defines experiments using C++ programming\(^\text{15}\).

**Launch Experiment**

**Brief description**

The experimenter sets up an experiment and then launches it. The experiment may be interrupted at any time, or left to run till the end. The duration of a typical experiment ranges from a few tens of minutes to several hours.

**Flow of events**

The experimenter chooses an experiment type, defines the parameters, and saves them in a file and then launches the application.

While the application is running, they may watch the output and according to the prospects, decide whether to

---

\(^{15}\) In most project analysis methods, the distinction between the developer and the user of a system is such that the user should not be required to modify the system. Nevertheless, it does make sense in this case considering that 1) creating a meaningful and interesting experiment requires processing that simply could not be expressed in a higher-level, more user-friendly user interface and language, 2) the user of the system is assumed to be computer-literate, Artificial Intelligence being a computer sciences discipline.
- let it run further;
- check the behaviour of the robots in the simulator;
- abort the experiment.

For checking the behaviour, they can either interrupt the run or not. If the experiment is not running on the same machine, they copy the output file to their local (3D display-enabled) machine, and launch the UC “View experiment”. After that, they may choose between

- resuming / letting the experiment run;
- changing the parameters;
- aborting.

If the experiment is never aborted, it runs until the maximum allowed number of generations is reached.

**Pre-conditions and input**

The file `.experiment.parameters` is edited in order to choose an experiment type and tune its parameters. Below is a listing of these parameters along with their corresponding description, their allowed values, and their arity (second column, with the heading "#"), The arity is the number of times the corresponding parameter may occur. An arity of 0 indicates an optional parameter.

<table>
<thead>
<tr>
<th>Name</th>
<th>#</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>experiment_name</td>
<td>1</td>
<td>&lt;user defined&gt;</td>
<td>Description of the experimental setup. Will be used for naming output files.</td>
</tr>
<tr>
<td>experiment_type</td>
<td>1</td>
<td>SameSide, SameSideFacingBox, LightFinding, ObstacleAvoiding, IncrementalBoxPushing,</td>
<td>Name of one of the existing experiments.</td>
</tr>
</tbody>
</table>

Dalarna University
Röda vägen 3S-781 88
Borlänge Sweden
Tel: +46(0)23 7780000
Fax: +46(0)23 778080
http://www.du.se
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_iterations</td>
<td>1 [0, ...[</td>
<td>Number of iterations, or time steps for which the simulation is allowed to run. If set to 0, the experiment will use its default value.</td>
</tr>
<tr>
<td>num_generations</td>
<td>1 [1, ...[</td>
<td>Maximum number of generations for which an experiment is allowed to run.</td>
</tr>
<tr>
<td>dump_elite_genome_frequency</td>
<td>1 [1, ...[</td>
<td>The genome of the elite individual of the population is dumped to the output file every n generations.</td>
</tr>
<tr>
<td>dump_pop_genome_frequency</td>
<td>1 [1, ...[</td>
<td>Same as above, but for the genomes of the whole population.</td>
</tr>
<tr>
<td>population_size</td>
<td>1 [1, ...[</td>
<td>Number of individuals in the population.</td>
</tr>
<tr>
<td>num_robots</td>
<td>1 [1, ...[</td>
<td>Number of robots used in the simulation. Every robot contains a copy of the individual solution currently being evaluated.</td>
</tr>
<tr>
<td>robot_controller_type</td>
<td>Perceptron, CtrnnML, CtrnnML_iho, CtrnnML_irho, CtrnnFR, RemoteController</td>
<td>Type of controller being evolved. Respectively one-layer perceptron, continuous time recurrent neural network multi-layered, continuous time recurrent neural network multi-layered with recurrent hidden layer, continuous time recurrent neural network fully recursive, remote controller. Note that using the remote controller would not make sense.</td>
</tr>
</tbody>
</table>
The robot has 15 available slots.

The number of occurrences of this parameter in the configuration automatically defines the number of IR sensors used.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir_position</td>
<td>1..15</td>
<td>Slots in which to fit an IR sensor.</td>
</tr>
<tr>
<td>al_position</td>
<td>1..8</td>
<td>Same logic as for IR_positions.</td>
</tr>
<tr>
<td>camera_num_sectors</td>
<td>1</td>
<td>Number of sectors the camera's field of vision is divided in. 0 means no camera at all.</td>
</tr>
<tr>
<td>camera_sector_offset</td>
<td>1</td>
<td>Offset of the sectors of the camera in relation to the direction of the robot, expressed as a fraction of the angle of a single sector. In practice, only the values 0 or 0.5 are useful.</td>
</tr>
<tr>
<td>camera_visible_colour</td>
<td>1..5</td>
<td>Colours that the camera will be able to detect.</td>
</tr>
<tr>
<td>proxi_camera_num_sectors</td>
<td>1</td>
<td>Same logic as for camera_num_sectors.</td>
</tr>
<tr>
<td>proxi_camera_visible_colour</td>
<td>1..5</td>
<td>Same logic as for camera_visible_colour</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>robot_has_light</td>
<td>1</td>
<td>Presence (1) or absence (0) of a light on the robot.</td>
</tr>
<tr>
<td>robot_light_colour</td>
<td>0..1</td>
<td>Colour of the robot's light. Only required if robot_has_light is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Red,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Green,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blue,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yellow,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>White</td>
</tr>
<tr>
<td>num_net_hidden</td>
<td>0..1</td>
<td>Number of hidden layers in the neural network, if the chosen controller is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indeed a neural network.</td>
</tr>
<tr>
<td>arena_width</td>
<td>1</td>
<td>Width and length of the arena used in the simulations.</td>
</tr>
<tr>
<td>arena_length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arena_num_lights</td>
<td>1</td>
<td>Number of lights in the arena.</td>
</tr>
<tr>
<td>arena_light_colour</td>
<td>1</td>
<td>Colour of the lights in the arena.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Red,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Green,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blue,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yellow,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>White</td>
</tr>
<tr>
<td>has_remote_controlled_robot</td>
<td>1</td>
<td>Presence or absence of a remote controlled robot in the arena during the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>visualisation of the experiment.</td>
</tr>
<tr>
<td>viewpoint_x</td>
<td>1</td>
<td>Position and direction of the camera to be used during the visualisation of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the experiment.</td>
</tr>
<tr>
<td>viewpoint_y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>viewpoint_z</td>
<td></td>
<td>X, Y, Z components of the position and height, pitch and roll of the direction.</td>
</tr>
<tr>
<td>viewpoint_h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>viewpoint_p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>viewpoint_r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>num_bricks</td>
<td>1</td>
<td>Number of bricks in the arena.</td>
</tr>
<tr>
<td>brick_length</td>
<td>0..1</td>
<td>Bricks' dimensions.</td>
</tr>
<tr>
<td>brick_width</td>
<td></td>
<td>Only required if num_bricks is not null.</td>
</tr>
<tr>
<td>brick_height</td>
<td></td>
<td></td>
</tr>
<tr>
<td>num_boxes</td>
<td>1</td>
<td>Number of boxes in the arena.</td>
</tr>
<tr>
<td>box_length</td>
<td>0..1</td>
<td>Boxes' dimensions and mass.</td>
</tr>
<tr>
<td>box_width</td>
<td></td>
<td>Only required if num_boxes is not null.</td>
</tr>
<tr>
<td>box_height</td>
<td></td>
<td></td>
</tr>
<tr>
<td>box_mass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>num_lightboxes</td>
<td>1</td>
<td>Number of light-equipped boxes in the arena.</td>
</tr>
<tr>
<td>lightbox_length</td>
<td>0..1</td>
<td>Dimension and mass of the light-equipped boxes.</td>
</tr>
<tr>
<td>lightbox_width</td>
<td></td>
<td>Only required if num_lightboxes is not null.</td>
</tr>
<tr>
<td>lightbox_height</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lightbox_mass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lightbox_colour</td>
<td>0..1</td>
<td>Colour of the lightbox.</td>
</tr>
<tr>
<td></td>
<td>Red</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Green</td>
<td>Only required if num_lightboxes is not null.</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td></td>
</tr>
</tbody>
</table>
The Experiment Creator user may have defined any number of extra parameters of any type, as suited for the needs of the particular experiment.

**Table 2: Experimental setup parameters.**

Once all parameters have been set as necessary in the file and that the file has been saved in the working directory, the user launches the experiment with the following command:

```
./rim -x
```

**Output and Post-conditions**

During the evolutionary run, the file `./results/<experiment_name>` (where the experiment_name is the one defined in the input parameters) is populated.

The output file contains a header that lists all the experiment's parameters and values that were defined earlier (such as the experiment's type, the arena's configuration, and so on).

The header is followed by a series of logs that trace the evolutionary process by recording, at every given number of generations, the state of the population (it is maximum and average fitness for example), along with the complete genotype of the best performing individual at that generation, as well as the genotype of the whole population. The genotype of the whole population typically is recorded less often than that of the best individual. When resuming an evolutionary run from an output file, the new run will have to start at a generation for which the genotype of the whole population was recorded.

Below is an excerpt of an example of such an output file (highlighting was added for readability).
experiment_type=IncrementalBoxPushing

 [...]
 al_position=6
 numGenes=46

 **Generation=0**
 Max_fitness=258.363953
 Min_fitness=0.000000
 Upper_quartile_fitness=36.776901
 Lower_quartile_fitness=0.000000
 Mean_fitness=33.152508
 Success_rate=0.100000
 Elite_genome:
 0.01277071894899 0.150179311633110 0.837599515914917 [...]
 [...]
 Population_genomes:
 0.01277071894899 0.091394409537315 0.837599515914917 [...]
 0.988728761672974 0.238151445984840
 0.988728761672974 0.238151445984840

 **Generation=100**
 Max_fitness=432.763910
 [...]

---

**Interrupt Experiment**

**Brief description**

This UC allows for the experimenter to interrupt a running experiment in a way that will allow it to be resumed subsequently. It is useful in the cases where the experiment's parameters need to be tweaked without starting the whole evolution all over again or for occasions where the computer running the experiment (which can be several hours long) needs to be shut down.

**Flow of events**

The experimenter sends a termination signal to the application. The signal is captured by the application which does not abort immediately, but continues running until the end of the population's current generation.
**Pre-conditions and input**

An experiment is running on a host to which the user has access. They send a signal using the system's signalling mechanism. (For example by pressing <CTRL> + “c”).

The user may then modify the experiment's parameters in the *output* file (and not in the original input file). That is the file that will be used when reloading the experiment before resuming it.

**Output and Post-conditions**

The experiment will not be running anymore and its output file will have the same format as described in the output section of the UC “Launch experiment”, except that the genes of the whole population at the last running generation before abort will be appended to the file so that the UC “Resume experiment” can use it as input.

**Resume Experiment**

**Brief description**

This UC allows for the experimenter to reload and resume a previously interrupted experiment.

**Flow of events**

The experimenter selects the output file of a previously interrupted experiment, and passes it to the application which resumes the evolution.

**Pre-conditions and input**

The output file from the experiment which is to be resumed is present with the correct format (see UC “Interrupt experiment”).

The user issues the following command:

```
rim -r <experiment output file name>
```

**Output and Post-conditions**

The experiment will resume and its output file will have the same format as described in the output section of the UC “Launch experiment”.

---

Dalarna University
Röda vägen 3S-781 88
Borlänge Sweden

Tel: +46(0)23 7780000
Fax: +46(0)23 778080
http://www.du.se
View Experiment

Brief description

This UC allows the experiment viewer to visualise the results of an experiment. The user has the possibility of viewing a simulation run using the controller of the elite individual of any logged generation, or to view a graph plotting the evolution of the fitness of the population throughout all the successive generations.

Pre-conditions and input

A correct output file of the experiment to be viewed must exist. See previous UC “Launch experiment” for more details.

The simulation is launched by issuing the following command:

```
./rim -v <experiment output file name> [-g <generation number>]
```

The -g option allows choosing the elite genome of which logged generation to view in the simulation. If a generation number is given that does not exist then the next higher generation number is used. If no value is given, the last logged generation is used.

Output and Post-conditions

The simulator opens a 3D viewing window and runs at the maximum possible speed. (A very rough control of running speed can be obtained by resizing the window, or triggering the display of shadows and textures). Further information is output to the console.

During the run, the following key presses are captured and interpreted as following:

```
<CTRL> + p  pause the simulation
<CTRL> + s  toggle shadows
<CTRL> + t  toggle texture on the ground
<CTRL> + w  write the 3D frames to files
<CTRL> + v  display the current viewpoint's coordinates in the console
<CTRL> + x  quit
```
The following key presses are recognised when a remote-controlled robot is present in the simulation:

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up and down arrow</td>
<td>Pilot the remote-controlled robot forth and back</td>
</tr>
<tr>
<td>Left and right arrow</td>
<td>Steer the remote-controlled robot</td>
</tr>
<tr>
<td>l</td>
<td>Toggle the robots’ light on and off</td>
</tr>
<tr>
<td>i</td>
<td>Toggle the robots’ IR beams on and off</td>
</tr>
</tbody>
</table>

The output of the plotting of the evolution is saved to a file the name of which was given by the user. Any existing file with the same name is overwritten.

Flow of events

First alternative flow: The user launches a simulation visualisation of the chosen experiment. The experiment is loaded, which means all parameters are loaded and their values are used to initialize the simulation, and the elite genome is used to initialize the controller of every robot present in the simulation.

The system then launches the simulation, at which point the user may toggle some graphical output options and take control of any remote-controlled robot they would have defined in the experiment's parameters.

Second alternative flow: The user feeds the output of the evolution of a population during a particular experiment to a graph plotting tool. The system saves the graphic in a GIF file which the user can then open and visualise.

Create Experiment

Brief description

The user Experiment Creator will be able to write a C++ class that redefines the existing Experiment class. This new experiment will have its own specific parameters, environment set-up methods, and population evaluation methods. The experiment is then made available to the user Experimenter.

Pre-conditions and input

When the Experimenter user needs to undertake a new type of experiment or to modify an existing experiment they then trigger this use case.
The file `./experiment.h` is present in the project's root directory.

**Output and Post-conditions**

The application has been successfully recompiled to take the new experiment into account, thus making it available to the Experimenter user.

**Flow of events**

The user creates two new files `./experiments/<experiment_name>.h` and `./experiments/<experiment_name>.cpp`, inputs the necessary C++ code, modifies the files `./main.cpp`, `./experiment.cpp`, and `./experiment.parameters` as needed (to take into account the new experiment type that is now available), and recompiles the application.

The new experiment class must at least reimplement the methods

```cpp
void resetPositions( void );
void computeFitness( void );
```

The first one is responsible for resetting the position of the objects (robots, lamps, obstacles, ...) at the beginning of every evaluation during simulations.

The second one, crucial to the evolutionary process, is responsible for assigning a fitness level to each individual. It is up to the Experiment Creator user to define how this value is computed. It may take into account the relative or absolute position and orientation of the objects in the arena, the state of lamps, or whatever else exists in the simulated world.

The other methods of the experiment base class may be redefined in case the user wants to make new experiment parameters available (by adding them to the `./experiment.parameters` input file), or in case special actions or computations must be performed at various points of the experiment. The source code's comments should be consulted for more explanations.

Experiment Creator chooses exact values, allowed ranges, or default values for a set of parameters. Those parameters include the amount of robots, the amount, dimension, and weight of obstacles and boxes, the population size, and a set of other variables described in more detail in the UC “Launch experiment”.

---

Boris VAN LIERDE

Degree Project

ExxxxxD

October, 2011

---

Dalarna University  
Röda vägen 3S-781 88  
Borlänge Sweden  
Tel: +46(0)23 7780000  
Fax: +46(0)23 778080  
http://www.du.se  

38
Experiment Creator defines a fitness function which will undertake measurements in a simulation, and any initialisation steps to be performed at the beginning of an experiment, of a generation, or of an individual's creation or evaluation.

Once all is set up, the experiment's source code is compiled and added to the system by recompiling the application.
4 Experiments

Once the system is built, the role of the experimenter in ER is ideally limited to defining an objective to attain or a problem to solve and expressing it in the form of a fitness function that will drive the evolutionary algorithm towards a successful solution.

Below a series of experiment sets are presented. The first two sets were lead to test the system itself and to demonstrate its ability to evolve behaviours, while the following ones were aimed at developing box pushing behaviours.

4.1 Light finding

4.1.1 Method and setup

The objective of this experiment set was to evolve a behaviour that leads the robot to bringing itself to the nearest light. The method for doing so was placing a single camera-equipped robot and a single light at random positions in the arena and then running an evolutionary process with a fitness function that rewards proximity of the robot to the arena's light.

More precisely, after each iteration (i.e. one discreet time step in the simulated arena), the robot controller's fitness was increased according to the rule

$$\dot{f} = f + \frac{1}{d^2}$$

where $f$ is the fitness and $d$ is the distance between the robot and the light.

Every solution was evaluated 16 times, meaning it was run 16 times in the simulator, and each evaluation was 256 iterations, or discreet time steps, long.

At the end of the evolution, or whenever the population's fitness reached a plateau, the best solution was visually evaluated using the 3D viewer. During the visualisation either the same setup as was used during the evolution (one randomly placed light and one robot) or a setup where the arena's light was replaced by a remote-controlled robot equipped with an on-board light were put in place. In the later case the evolved robot's ability to follow that remote-controlled robot was tested.
Below are the detailed values of the parameters used for this experiment, leaving out unimportant, unused and null parameters. The semantics of these parameters were explained in an earlier section.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>population_size</td>
<td>20</td>
</tr>
<tr>
<td>num_generations</td>
<td>1001</td>
</tr>
<tr>
<td>num_iterations</td>
<td>256</td>
</tr>
<tr>
<td>arena_num_lights</td>
<td>1</td>
</tr>
<tr>
<td>arena_light_colour</td>
<td>white</td>
</tr>
<tr>
<td>num_robots</td>
<td>1</td>
</tr>
<tr>
<td>has_remote_controlled_robot</td>
<td>0 / 1</td>
</tr>
<tr>
<td>robot_has_light</td>
<td>1</td>
</tr>
<tr>
<td>robot_light_colour</td>
<td>yellow</td>
</tr>
<tr>
<td>camera_num_sectors</td>
<td>4</td>
</tr>
<tr>
<td>camera_offset</td>
<td>0.5</td>
</tr>
<tr>
<td>camera_visible_colour</td>
<td>white</td>
</tr>
<tr>
<td>ir_position</td>
<td>15</td>
</tr>
<tr>
<td>robot_controller_type</td>
<td>Perceptron</td>
</tr>
<tr>
<td>num_net_hidden</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Parameter values for the light finding experiment.

### 4.1.2 Results

Results were conclusive and quick to come. After only about six generations, a successful behaviour emerged. The evolved controller was able to lead the robot to the light and have it reside in its proximity, as well as to make it follow another robot equipped with an on-board light. For this the extremely simple fitness function presented above was sufficient.

Below the graph of the evolution of the population's fitness level across the generations for one of the evolutionary runs and a series of screen shots showing the behaviour of a robot using the evolved solution are presented.

On the graph, the maximum fitness reaches a small peak at generation 4, which corresponds to a behaviour with which the robot reaches the light by following a sub-optimal whirling trajectory, made of long eccentric loops. At generation 6, the trajectory becomes straighter, without any unnecessary turns, even though those turns that are taken still are quite wide. All subsequent controllers that obtain a higher score manage to do so by enhancing their behaviour from the moment the light is reached to
the end of the evaluation: they describe tighter, flatter loops under the light instead of circling widely around it. None of them seem to develop the habit of staying still under the lamp once it is reached.

Randomness also plays an important part in the fitness's fluctuations from generation to generation by placing the robots more or less close to the arena light at the beginning of the evaluation.

In the screen shots below, the robot starts the run randomly placed, in this occurrence facing away from the light. It then does a rather wide left turn and advances towards the light. The behaviour shown here corresponds to the best solution of the population at generation 30.
It is worth noting that it is possible to use the robots for generalisations of the tasks for which they were originally evolved, such as following another robot rather than finding a light, as was shown here. In the same way, a box-pushing behaviour could help solving various load transportation problems.

### 4.2 Collaborative undirected rectilinear box pushing

This experiment was run with two robots and used a more complex designed fitness function. It was both a first try at the “fitness function designing” method, and a test for the capabilities of the system to evolve collaborative behaviour, even though collaborative behaviour is not the goal of this work.

The behaviour that is sought is “undirected rectilinear box pushing”, meaning pushing a box with no particular given goal, but in a straight line. Once the box starts moving, it is expected to stay on its
initial course as much as possible. This can be seen as sub-behaviour of more general box pushing, even though that is not necessarily the case.

Several parameters and fitness functions were tested with this “designed fitness function” method. The process of tweaking the parameters and adding “incentives” in the form of subcomponents to the fitness function proved to be tedious and rather fruitless.

4.2.1 Method: designing the fitness function

The “designed fitness function” method refers to the act of adding subcomponents to the fitness function so as to reward an individual that accomplishes a sub-task of the whole task. The idea being that if a sufficient number of sub-tasks are rewarded, the robots will eventually accomplish all of those sub-tasks. An even greater reward is also handed out for the completion of the whole task, assuming that the robots will then learn to perform the sub-tasks in the correct order. Another assumption is that the defined sub-tasks indeed cover all the aspects of the entire task.

Here the subcomponents of the fitness function are presented. Note that in this set of experiments both robots are controlled by copies of the same controller; they are identical twins. The fitness of a solution is thus attributed to the group's result, not to an individual robot.

The total fitness of the solutions is proportional to:

- whether both robots are in a good position for pushing the box (the possible amount of extra points earned in this fitness subcomponent are \{0, 1, 2\});
- whether the robots are facing the box ( range of extra points: \[0, 2\pi\]);

Figure 8: First subcomponent. Robots in their positions relative to the box.

Figure 9: Second subcomponent. The angle between the robots' direction and the box's closest face.

Figure 10: Third subcomponent. The relative direction of the box between two successive time steps.
The first two fitness subcomponents were designed with the idea of having the robots position themselves optimally in order to push the box.

The first subcomponent is quantified by first dividing the area around the box in six areas, a, b, c, d, e, and f (see Figure 8). The solution is given two extra points when one robot is in area a, b, c or d and the other robot is respectively in area b, a, d or e. In other words, the two points are given when both robots are on one of the long sides of the box and in adjacent areas, which would represent the ideal configuration for pushing the box. Or at least it seems to be the ideal configuration, providing good control over the box's movements and avoiding interference between the two robots' trajectories. The solution is given one point when both robots are in the same area and no extra point in any other situation.

The second fitness function subcomponent is the sum of both robots' orientation in relation to the normal of the box's nearest side. An angle of π (facing away from the box) yields no points, and an angle of 0 degrees (facing the box) yields π points.

The last subcomponent is designed to encourage the box travelling in a straight line. It is the difference between the direction of the box at the time-step t and its direction at time-step t-1. So if at time-step t-1 the box is oriented at an angle α and at time-step t at an angle β, the solution will be given π-|β-α| extra points.

Note that no incentive for the box to actually travel was included. This and the nature of the three other subcomponents will have unexpected results, as shown later on.

4.2.2 Setup

For this set of experiments, robots with three front-pointing IR sensors, a camera able to see the colours green and yellow (corresponding to the colour of the light on the robots themselves and to the colour of the light on the box respectively) and controlled by a multi-layered CTRNN were used. Below are the values given to the parameters of the experiment, again leaving out less important, unused and null values.
num_iterations 600

num_lightboxes 1
lightbox_length 0.1
lightbox_width 0.4
lightbox_height 0.1
lightbox_mass 0.035
lightbox_colour green
num_robots 2

robot_has_light 1 (true)
robot_light_colour yellow
camera_num_sectors 8
camera_visible_colour green, yellow
ir_position 14, 15, 0

Every solution was evaluated nine times, with the robots being initially placed in a different particular combination of positions relatively to the box at every evaluation. The combinations are shown in the figure below.

Figure 11: Initial positions of the robots at each of the solution's nine evaluations.
The aim of these alternating positions is the production of adaptive solutions by confronting the population to various initial situations. (Even though the first and the last combinations seem equivalent, they are not because of the position of the arena's light relative to the box and the robots).

4.2.3 Results

As can be seen on the graph above, the algorithm fails to engage in an increase of the population's fitness after a certain point around generation 250. At that point, the advantage of having a way of visualising the robots' behaviours with the 3D tool becomes obvious, as there is no way the quality of a solution could be evaluated only from the evolution of its fitness. It can indeed reach a high level of fitness early on, such as around generation 250 in this case, and then stagnate. Furthermore, there is no way to know how many of the necessary tasks or which tasks are not completed. The surge around generation 250 could correspond to a near-perfect general behaviour, with only one thing missing for the robot to actually complete the given task, but it could just as well correspond to a behaviour that allows the robot to accomplish just one of the subtasks.

But running the simulator in visualisation mode on this data set quickly shows that no good solution was yielded. What happens is that both robots head straight to each other, regardless of the position of the box, and push each other till the end of the run. While it is not at all what was expected, this behaviour does have the advantage of yielding an almost constant level of fitness for the solutions that implement it.
Indeed, it does correspond to the three designed fitness function subcomponents. Being next to each other almost guaranties that both robots are in the same area, thus yielding in most cases at least 1 point of fitness. Facing each other, chances are none of the robots are facing the box directly, but the sum of their orientations in relation to the box is constant. And finally, ignoring the box and leaving it untouched guaranties that a skewed trajectory will not diminish the return of the fitness function. Thus having the box stay still maximizes the return of the third subcomponent.

Thus, all three of the fitness function's subcomponents are satisfied without the task being anywhere close to completed.

This shortcoming wasn't unveiled until after the evolution produced robots that would simply avoid touching the box, and be rewarded with high scores. Obviously this is a simple problem with an easy solution (adding a fourth fitness subcomponent that rewards the distance travelled by the box), but it illustrates one of the risks of “engineered” fitness functions which is introducing bias in the solution search.

The problem with this approach is that it requires human à priori knowledge, which is precisely what ER seeks to do without. Indeed, there are two potential problems with using human knowledge for assisting the evolution of controllers. The first is that future real-world applications of robotics are most likely to consist in situations not well known by humans, for example because of the environment in which they take place. The second, perhaps more worrying, is that the way humans evaluate the problem introduces a bias that might limit the solution space that is explored by the evolution.

4.3 Directed box pushing, minimal fitness function

This set of experiments and the following ones were conducted in order to develop a directed box-pushing behaviour. This set in particular tries to do so with minimal work on the fitness function.

Considering the ease with which the light-finding behaviour was evolved, one wonders if it would be as easy to evolve a box-pushing behaviour, possibly at the cost of a longer evolutionary period.

To verify that possibility, a simple experiment was set up in which the fitness of the solution being evaluated is incremented at every time-step by an amount proportional to the box's proximity to the
goal (the arena's light), independently of the robot's position or activity. At each generation, every solution was evaluated sixteen times.

At first the evolution was run several times with a population size of just ten, without success. It was then run with a population of twenty. The results of those runs are illustrated with the experiment described below as an example.

### 4.3.1 Setup

<table>
<thead>
<tr>
<th>Genetic Algorithm</th>
<th>population size</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>num_generations</td>
<td>8000</td>
</tr>
<tr>
<td></td>
<td>num_iterations</td>
<td>256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arena setup</th>
<th>arena_num_lights</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>arena_light_colour</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td>num_lightboxes</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lightbox_length</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>lightbox_width</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>lightbox_height</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>lightbox_mass</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>lightbox_colour</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>num_robots</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Robot configuration</th>
<th>camera_num_sectors</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>camera_visible_colour</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>ir_position</td>
<td>13, 15, 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Robot controller configuration</th>
<th>robot_controller_type</th>
<th>CtrnnML_irho</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>num_net_hidden</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5: Parameter values for the directed box pushing with minimal fitness function experiment.
4.3.2 Results

As can be seen, there was no clear upwards trend in the fitness of the population. (The spike around generation 5500 is an anomaly probably due to the box being placed by chance very close to the light for several evaluations of a particular solution).

When running the results in the simulator with the 3D viewer, it could be seen that the robots were totally useless, unable even to find the box.

This result was to be expected with a simplistic fitness function for such a complex task, and is a perfect illustration of the bootstrapping problem.

In reaction to that, alternative methods are explored in the following experiment sets, starting with the intuitive solution of designing the fitness function.

4.4 Directed box pushing, designed fitness function

A problem occurs when more complex tasks are sought to be accomplished, as seen in the previous experiment set in section 4.2 Collaborative undirected rectilinear box pushing and as discussed in the introduction to the bootstrapping problem. If a task requires many intermediate steps, then there is
only an extremely small probability the evolutionary algorithm will find a solution. One way to solve this problem is to add sub-components to the fitness function as was done before.

For completeness this approach is tried once again in the context of directed box-pushing with a still rather simple designed fitness function.

4.4.1 Setup

The fitness function in this set of experiments rated individuals based on the following fitness criteria:

- the proximity of the robot to the box, to encourage the robot to find the box;
- the proximity of the robot to the lamp, to encourage the robot to move towards the light;
- the proximity of the box to the light, with this criteria squared to strongly reward the box being close to the light.

Here are the values given to the more important parameters for the experimental run that is presented as an example in the results section:

<table>
<thead>
<tr>
<th>Genetic Algorithm</th>
<th>population_size</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>num_generations</td>
<td>8000</td>
</tr>
<tr>
<td></td>
<td>num_iterations</td>
<td>500</td>
</tr>
<tr>
<td>Arena setup</td>
<td>arena_num_lights</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>arena_light_colour</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td>num_lightboxes</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lightbox_length</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>lightbox_width</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>lightbox_height</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>lightbox_mass</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>lightbox_colour</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>num_robots</td>
<td>1</td>
</tr>
<tr>
<td>Robot configuration</td>
<td>camera_num_sectors</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>camera_visible_colour</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>ir_position</td>
<td>13, 15, 1</td>
</tr>
<tr>
<td>Robot controller configuration</td>
<td>robot_controller_type</td>
<td>CtrnnML_irho</td>
</tr>
<tr>
<td></td>
<td>num_net_hidden</td>
<td>4</td>
</tr>
</tbody>
</table>

Dalarna University
Röda vägen 3S-781 88
Borlänge Sweden

Tel: +46(0)23 7780000
Fax: +46(0)23 778080
http://www.du.se
4.4.2 Results

Here is a graph of a typical run using the described method and parameter values.

![Graph showing fitness over generations.]

Figure 14: Directed box pushing, designed fitness function.

Again, there is no upwards trend in the fitness, although there does seem to be more fluctuations in the fitness levels than in the previous set of experiments which had an even more naïve fitness function. Unfortunately these small fluctuations are due to chance (the positioning of the robot and the boxes relative to the lamp), and some robots randomly approaching either the box or the lamp. This was confirmed by checking some of the controllers in the 3D viewer.

Running the experiment for several thousand more generations did not help.

4.5 Incremental box pushing

4.5.1 Solving the bootstrapping problem with competition

Floreano and Nolfi [xvi] offer three solutions for solving the bootstrapping and complexity escalation problem, of which one - competitive evolution - is of particular interest to this work.
The idea is to have two competing populations with opposite objectives. They both start out as simple entities, and are then driven into higher levels of complexity because of the pressure applied by the other population. A popular example is the hunter and prey scenario\textsuperscript{16}.

The competitive evolution approach is one of great elegance, since the complexity of a task arises naturally, and is dealt with naturally. (The word "naturally" is used in the sense of the nature, the ecology, of the artificial system.) Furthermore, the fitness function may be reduced to its simplest expression while still conserving its efficiency. This eliminates human bias as much as can be.

But there is the obvious caveat that there has to be two populations with a common but inverse goal for it to occur. Having two competing populations does not happen in all ER experiences, and is actually quite rare in the potential applications of ER. More often, the evolved robots will be required to complete either an autonomous task (i.e. they are alone in a human-inaccessible location), or a collaborative task (i.e. they are part of a bigger system that also includes humans, machines, or other robots, and requires collaboration with some or all of those entities, such as in a factory).

It is at this point, in an attempt to generalize the competitive evolution model, that the idea of competing against the environment came up. The environment is an indispensable actor in the evolution of a robot. So why not give it a more active role in this evolution? It could, much like a competing species in competitive ER, present ever more complex challenges to the robots, pushing them by doing so into evolving higher and higher complexity.

The following section presents a scheme for implementing a system that makes use of a changing environment.

\textbf{4.5.2 A changing environment}

First of all, a disambiguation between the terms \textit{evolving} and \textit{changing} is necessary. Hereafter, the word \textit{evolution} will be used to mean a process of morphological or behavioural change through the generations of a population in an ecology (such as the evolution implemented by a GA), whereas \textit{change} will designate any phenomenon, artificial (i.e. hard coded or observer induced) or natural (i.e. that flows from the rules of an ecology, artificial or not). Evolution is thus a subset of change.

\textsuperscript{16} In that scenario, the prey population grows better at fleeing the hunters, which themselves progressively become better at chasing the prey.
4.5.2.1 Levels of difficulty

The level of difficulty of any task is correlated to the initial set-up of the environment. A more complex environment likely makes any task more difficult. In the context of box-pushing, a robot placed behind a box that is well positioned in relation to the goal will find things easier than a robot that is randomly positioned and has to find the box before pushing it, for example.

Following that line of thought, the evolutionary process brought to confront an environment that is made progressively more difficult as the robot evolves. Technically, the different complexity levels were defined as different arena configurations of increasing complexity.

Before an evolutionary run is started, the environment is set to a low level of complexity (level 0). Then the evolution is launched, and as soon as the population reaches a certain success rate (more about that later) in its task, the environment changes its configuration to a higher, more challenging level of difficulty. The change from one level to the other is done in discreet steps. Below the levels that were defined are presented and illustrated.

Note that from now on, the goal that the box has to reach will be materialized by a lamp placed in the arena and visible to the robot.

- Level 0 (Figure 15, a): The robot, the box and the goal are aligned. The robot is facing the box as well as the goal and is touching the box. The light is either at a fixed position (in the early versions of this experiment), or at a random distance from the box in the range [1, 2].

Figure 15: Five successive configurations of the environment, of increasing complexity.
(Figures are not to scale.)
- Level 1 (Figure 15, b): Same as level 0, but the robot is facing a random direction, at an angle $\alpha$ in the range $[0, 360]$ degrees.

- Level 2 (Figure 15, c): Same as level 1, except the robot's lateral position is offset relatively to the robot-box-light axis by a distance $d$ randomly selected in the range $[-2r, 2r]$ where $r$ is the radius of the robot.

- Level 3 (Figure 15, c): Same as level 2, except the range is increased to $[-4r, 4r]$.

- Level 4 (Figure 15, c): The range is extended to the box's lateral dimension.

- Level 5 (Figure 15, d): The robot is anywhere behind or on the sides of the box, but as before, always touching the box and randomly rotated.

- Level 6 (Figure 15, e): The box and the robot are randomly rotated and randomly placed in the arena.

Note that in the first few experiments presented below, the level 5 as it is defined here is not used in during the evolution. The environment's complexity level goes directly from level 4 to level 6 (a random disposition). Level 5 was inserted later, in a process described below. Note that there is still a very large increase in difficulty between level 5 and level 6.

### 4.5.2.2 Increasing the complexity of the environment according to the rate of success

When should the environment's complexity level be incremented?

The level incrementation as it is defined here is a discrete and artificial process, and raises the question of at what moment in the evolution should it occur. The values given by the evaluation of the population (by the fitness function) could be taken into consideration, and targets could be set in terms of minimum, mean and maximum fitness.

The problem with that idea is that while the fitness function does work in discriminating different individuals, it brings no information on the actual completion of the task (i.e. bringing the box under the light), since the fitness level corresponding to the completion of the task by the robot is unknown.

Therefore another measure was introduced that indicates the number of times a population successfully brings the box to the goal. It is expressed as a percentage of successful evaluations across all evaluations. When a certain threshold is reached, the complexity level is incremented.
More precisely, this percentage expresses the number of times the box finds itself under the goal at the end of an evaluation, across all evaluations of all individuals for every given generation.

### 4.5.3 Results

Below are the graphs of the results obtained in two early runs of this new method. The two experiments presented differ only in the placement of the light (the goal) in the arena. In the first level of the first experiment presented, the light was set at a fixed distance from the box, whereas in the second experiment the light was set at a random distance.

![Graph](image.png)

**Figure 16: Incremental Box Pushing, version 3.**

*Maximum and mean fitness, success rate, and level, using the incremental box-pushing approach.*

*In blue, the success rate of the population at the given generation, in pink, the levels of complexity of the environment. The success rate is expressed as a proportion of the graph’s total height, i.e. the top of the graph would be a 100% success rate. The complexity level is arbitrarily set to one tenth of the total height times the level, i.e. the top of the graph would represent level 10.*

Note that the level increments do not seem to correspond to high success rates, judging from the graph. This is due to the fact that the system only logs the state of the population intermittently and the generation at which the high success rate occurred happened to not be logged.
Figure 17: Incremental Box Pushing, version 5.

From the graphs above it seems the mechanism is fairly successful.

It can be noted that the second experiment took significantly more time to reach high success levels than the first one. This is because, as stated above, the original experiments had the lamp set at a fixed distance. What would happen is that the robots would quickly learn to push the box for a certain, constant amount of time and then stop, without consideration for the actual proximity of the box to the light. This strategy worked well up to level three (where the robot was positioned in a relatively advantageous situation), but did not allow it to evolve useful skills for the higher levels, nor did it produce robots that were able to generalise their behaviour to a randomly placed goal.

The success rate (in blue, represented as a percentage of the total height of the graph), climbs until it reaches a threshold (set at 15% here), at which point the level (in pink) is incremented by one. Nevertheless, there is a problem in the evolution that does not show on the graph. The robot tends to “forget” some behaviours, or it tends to exaggerate some others. In those cases it becomes unable to solve problems that it has met on earlier levels. Forgetting good simple behaviours (such as aligning itself to the box and the goal before pushing) can lead to performance decrease, as seen a few hundred generations before 4000 on the graph of the second experiment. Note that the robots do not go beyond level 3 (out of the 5 available).
This is due to the unnatural situation of the robot being confronted repeatedly to identical environments. Because of on one hand the lack of incentive to conserve older behaviours and on the other hand the incentive to specialise in the environment, the robot tends to forget previous behaviours in favour of the latest ones. This phenomenon of over-fitting is well known in the fields of artificial intelligence.

One solution to over-fitting is diversifying the training data. That is what was done in the next version of the experimental setup: each robot at each generation was evaluated against all of the previous levels as well as against the current level.

As illustrated in the graphs below, evaluating against previous levels seems to help a lot. Robots reach level 5 after only a few hundred generations (roughly around generations 400 and 2200 respectively in the cases depicted above). When they reach level 5, corresponding to a random disposition of the robots and boxes, the GA shows its limits. That is why an intermediate level (described in the previous section) was added.

Figure 18: Incremental Box Pushing, version 6.

Adding a mechanism for evaluating against previous levels.
But again, when evaluating them in the 3D viewer, the behaviours were not quite as good as their fitness and success rate suggest. Although they performed well in most cases, there are situations where they got stuck in “bad habits”.

An explanation for this requires considering different elements of the system in relation to each other. Namely, the success rate at which the level is incremented, the population size, and the number of individuals selected in the “elitism” process of the GA.

Indeed, the success rate was defined as the percentage of successful evaluations across the whole population. Bear in mind that each individual got to undergo several evaluations, so that it could be tried against different situations. Because of that, it could happen that there were generations where the necessary success rate was reached, even though none of the individuals completed successfully all of its evaluations.

For example, if there are 20 individuals and 10 evaluations per individual, and a 20% success rate is required as a condition for incrementing the complexity level, then it is enough that 40 individuals complete just one evaluation each.

Of course, the evaluations that are most likely to be completed are the easiest ones. The consequence of this is that suboptimal individuals are selected as the elite of the generation, undermining the
chances of success of future generations. So this is a problem in the particular setup of this experiment.

A solution to this could have been to add an extra condition for level increment. Say, “There must be at least \( e \) individuals that complete \( all \) trials, where \( e \) is the number of individuals chosen as elite”.

But the very reason why this incremental complexity approach was implemented was to diminish human intervention, at least at the level of the fitness function definition. As will be discussed in the conclusions, the hope is to be able to remove human intervention from the definition of the levels and from the level incrementation process. And adding further criteria to that process does not go in that direction.

What could be tried instead was augmenting the population size in the hope that a bigger population would be more likely to yield individuals able to succeed in all of their evaluations. This again is human intervention, but less heavy-handed, as it is just a tweak in the parameters of the GA itself to avoid a border effect due to the small size of the population. Furthermore, this kind of adjustment could again potentially be automated.

The next experiments presented below used an extra, intermediate environment complexity level and a bigger population size.
Figure 21: Incremental Box Pushing, version 6.
The same as above, with a population of 60.

The bigger population size did yield robots that presented better behaviours, but at a cost. In the first graph, the last level was reached after more than 5000 generations for a population size of 40, despite the addition of an intermediate level. This is about 25% more time for a population 100% larger. In the second graph (200% larger population), level 4 was not reached after 8000 generations, at which point the experiment was interrupted.

What about augmenting the minimum success rate? Again, the resulting behaviours were more solid, but at the cost of a longer genesis\(^\text{17}\), as can be seen in the graphs below.

\(^{17}\) This is of course a common dilemma in artificial evolution.
Figure 22: Incremental Box Pushing, version 6.
Population of 40, minimum success rate for incrementing level: 20%.

Figure 23: Incremental Box Pushing, version 6.
Population of 40, minimum success rate for incrementing level: 20%.

So after considering these results, it seems like this approach of incremental difficulty levels could be quite efficient, after some tuning of the complexity levels and of the level incrementation procedure. This tuning would ideally be automated, as discussed in the “Further work” section of the conclusions.
5 Conclusions

5.1 Synthesis

It was shown that Evolutionary Robotics, for all the benefits they could yield, is extremely sensitive to the bootstrapping problem. Solutions requiring the experimenter to tailor the fitness function to the task at hand, regardless of their chances of success, have the huge disadvantage of introducing human bias in the evolutionary process, thus potentially limiting its “creativity”.

Competitive evolution is an elegant and powerful solution to the bootstrapping problem, albeit limited in applicability to competing multi-species environments, and thus to certain types of problems.

Therefore, an attempt was made to generalize it by extracting its fundamental concept: competitive change.

Another positive aspect to the changing environment idea is that it could offer automation and co-evolving possibilities. Indeed, as was repeatedly stated, the whole point of ER is to assist the emergence of self-organized intelligent structures, and having to manually configure any aspect other than the initial setup, even indirectly related to the robot itself, defeats the purpose.

Below, the next steps towards a potential automated, continuous and natural process based on the incremental approach are discussed.

5.2 Further work

5.2.1 Recreating the dynamics of a competitive evolution

Considering competitive evolution, it can be observed that the driving force of the evolution is the arms-race between the two species. In the absence of competition, there is no incentive for a population to evolve. Evolving actually is costly for a population in terms of efficiency, since potentially weaker individuals which bring down the overall success level must be created in order to find stronger ones. These two factors together create a dynamic balance that has the side-effect of a smoothly increasing complexity. And that effect can be used to the benefit of the experimenter.
Now how can this dynamic balance of forces and gradual evolution be recreated? Two things are necessary:

- a common goal for both species, but with one species trying to minimize this goal, and the other one trying to maximise it;

- a process of gradual change, with a notion of cost associated to change in order to limit the risk of one of the two sides too easily taking the upper hand, thus eliminating the competition which is meant to be the driving force of the evolution.

The common goal must be quantifiable and the outcome of its evaluation for one species must always be the opposite of the outcome of the evaluation for the other species. For example in the predator and prey scenario, the number of preys eaten by the predators is the common goal, where predators try to maximize it and preys try to minimize it. In the context of this work, the common goal is the number of times the box ends up under the lamp, where the robot tries to maximize this, and the environment tries to minimize it.

The gradual change process with associated notion of cost would be different for the robot and for the environment. For the robot, the process utilized would of course be the GA acting on the robot's controller. For the environment, a method is proposed in the next section.

5.2.2 Gradual and automatic complexity increase

For the environment, another GA or any more straightforward process could be used, but the difficulty comes from defining the cost of a solution.

Nevertheless, considering the way the environment complexity levels were defined in the last set of experiments, it appears that all the geographical arrangements of the various objects in the arena could be expressed with numerical arguments. For example, the rotation of the robot in relation to the side of the box. This rotation could be defined as the angle between the robots' direction and the normal of the side of the box, and this angle could be associated to a cost. So setting the robot with a greater rotation in relation to the box (which makes things more difficult for the robot) would have a greater cost for the environment. Similarly, the distance of the robot from the box could be added to the cost of the setup. Then obstacles might be added, the amount, shape, position of which may be similarly factored into the total cost of a setup.
Figure 24: Configurations with a cost.

a) A simple environment configuration where all geographical parameters are set to 0;

b) A more complex and costly environment configuration where angles $\alpha$, $\beta$ and $\gamma$ and distances $d$ and $e$ are non-null.

A highly complex configuration with random positions and rotations and numerous obstacles would thus be associated with a high cost for the environment, so such configuration would not occur before it was the only way of restraining the success of the robot.

Instead, the environment would start with a simple cheap configuration. From there, a population could evolve a solution able to reliably fulfil its task in this simple setup. The environment would then slightly step up its complexity in order to defeat this newly evolved population, while trying to keep its costs low. And the scenario would repeat.

In that way, the robot could potentially be brought to a high level of sophistication by the environment in a natural and automatic way. And thus with minimal human intervention, a phenomenon of natural competition between the population and the environment could potentially produce high quality solutions.
References


