Universiteit van Amsterdam

Revolve: An Evolutionary Robotics Toolkit

Elte Hupkes

Master's Thesis Computational Science

Graduate School of Informatics, Faculty of Science, Universiteit van Amsterdam
Department of Computer Science, Faculty of Science, Vrije Universiteit Amsterdam

2016

VU University Amsterdam

Revolve: An Evolutionary Robotics Toolkit

Elte Hupkes

ABSTRACT

The **R**obot **Evolve** Toolkit consists of a set of software utilities designed to facilitate evolutionary robotics research. It builds on top of the widely used Gazebo robotics simulator, providing adequate performance for simulating populations of average sized robots in real time, specifically with on-line, embodied evolution and artificial life scenarios in mind. As such it serves as an aid in evolutionary robotics research regarding the Evolution of Things [20], in which entities evolve in real-time and real-space. The toolkit is centered around the Revolve Specification, which assumes a robot body space consisting of basic building blocks that can be chained together to form a robotic organism. With the tools included, an abstract representation of such an organism can be conveniently turned into a format suitable for simulation with Gazebo. Additional utilities provide monitoring and control over the simulation environment. This thesis presents the Revolve Toolkit, applying it in practice to explore the grey area of evolutionary robotics between off-line and on-line, embodied evolution. The most prominent difference between these two types of systems is found to be the means of evaluation, which is one-off and serial in off-line scenarios whilst being continuous in on-line, embodied scenarios. The fact that the performance of robots changes over time results in different evolutionary dynamics in the latter approach.

SUPERVISOR:	A.E. Eiben
SECOND READER:	A. Visser
KEYWORDS:	Evolutionary Computing, Evolutionary Robotics, Artificial Life, Robotics Simulation

Acknowledgements

First and foremost I should thank Gusz Eiben, whose vision and passion for evolving robots is indispensable for the subject. Furthermore I thank Josh Auerbach, creator of RoboGen, which served as an inspiration for many areas of this work.

I also thank my friend Ove Danner for providing support and a voice of reason when the mountain would at times seem too high to scale. Finally I'd like to express some gratitude to my other friends, family and in particular the guys at work for being patient with my progress while reminding me that at some point, enough really is enough.

Contents

1 Introduction

The work presented in this thesis is a result of the desire to progress the field of Embodied Artificial Evolution (EAE) [18], a branch of Evolutionary Robotics (ER), as part of the grand challenges for evolutionary robotics as presented by Eiben et al. [16]. Recent technological advances in rapid manufacturing technology are providing a basis for experiments in which robots evolve in real time and real space, otherwise known as the Evolution of Things [20]. While this grand vision is aimed towards experiments involving real physical robots, the cost of such research both timewise and financially is still high. Therefore, rather than disregarding simulation entirely as something that is not part of the end result, it is embraced as a tool to bootstrap this kind of work, supplying researchers with a low cost means of realizing a proof of concept or exploring a hypothesis before committing to a larger investment. Recognizing this facilitating role, a good simulation tool should lower the threshold to perform exactly these kinds of tasks. The software toolkit outlined in this document aims to fulfill that requirement for at least a large body of potential research projects.

The primary driver for the work described in this thesis are the difficulties encountered with the simulation efforts in the work that precedes it at the Computational Intelligence (CI) group [1] of the Vrije Universiteit in Amsterdam, predominantly the work by Weel, et al. from 2014 [42]. This described a simulation of an open ended artificial ecosystem with physical robots with co-evolving minds and bodies. Implementing the Triangle of Life [17] framework, it provided a step forward towards open ended physical robot evolution.

However, the simulations, which made use of the Webots [31] robotic simulator, were

[1] `http://www.cs.vu.nl/ci/`

not without issues, as was noted in the Master's thesis [12] from which the publication followed. First of all, Webots is a commercial and closed-source platform, requiring a license for each computational unit running the software. This meant scaling the simulations towards the resources desired for running computationally expensive robotic scenarios would become increasingly expensive, a particularly pressing issue given that the simulations ran at a prohibitive approximate 11% of real time speed. The closed source nature meant it was hard to analyze and improve these performance issues, as well as the unexpected software instabilities that were found to occur. Finally, constraints in Webots limited the number of individuals that could be simulated, regardless of performance. Even though the system was designed with reusability in mind, its use as a foundation for future research project therefore became questionable. This work represents the result of the decision to take a small step back to realize an alternative foundation.

1.1 Research Goals

The software toolkit developed as part of this work, Revolve, aims to provide a basic set of tools that can be used to jump start simulating an evolutionary robotics scenario. It builds on top of an openly available robotics simulation platform, Gazebo [28], attempting to provide a good trade-off between flexibility, ease of implementation and performance.

Realizing that without an actual use case it is generally hard to create something useful, the purpose of this thesis is twofold: to describe the design and uses of the Revolve Toolkit, and to use it to answer a research question. The first part, which explains Revolve and its components, is considered in Chapter 4, following chapters 2 and 3 which provide a brief background on the subjects involved and an overview of the platforms considered to construct Revolve respectively.

This part is followed by a series of experiments, described in Section 5.4, designed to assess the performance of Revolve as a research tool both quantitatively and qualitatively, focusing on the simulation of on-line, embodied artificial evolution scenarios. The matter of computational performance is addressed first, to see what population sizes can feasibly be simulated on regular hardware. Building on these results, a number of simulations are performed to address the following research question:

– What are the differences between off-line and on-line evolutionary robotics systems?

The terminology of this question should be interpreted in the context of Section 2.5. This matter is explored by examining the outcomes of simulations of both types of scenarios that share many characteristics. More concretely, the robotic structures resulting from these simulations are compared in terms of fitness and complexity, defined by several measures such as their number of extremities and the size of their brains.

Given these results, not only do these experiments show that Revolve is up to the task for which it was designed, they also provide some insight into what happens at the 'edge' of common present-day evolutionary robotics and the envisioned Evolution of Things. To the best of the author's knowledge, this type of analysis has not yet been performed at the time of this writing.

2 Background

As Karl Sims so adequately stated in his landmark 1994 paper [38], the field of robotics suffers from a complexity versus control trade-off. In essence this means that the possibilities for creating complex physical (robotic) structures exceed the human ability of designing satisfactory control systems for such structures. Even when the desired behavior can be qualitatively described, controlling a complex physical system has been found to simply involve too many inter-playing variables to employ a top down designing approach. The same problem arises not only for control systems but also for robot hardware, as the desire for robots to perform increasingly demanding tasks in possibly hostile and unknown environments makes it hard to design a physical structure for a robot well suited to the objective. To quote another early paper on the subject: "*Interesting robots are too difficult to design*" [26].

Evolution can provide a solution to this dilemma, by borrowing from the apparent concepts with which ourselves and the other living entities around us have come to be as they are today. Instead of regarding the system as a whole, a bottom up approach is taken by comprising one of separate parameterized modules, allowing for a wide array of possible behaviors. When evolutionary processes are allowed to operate on multiple of the hereby arising 'individuals', producing new individuals that exhibit increasingly desired behavior can in many cases be accomplished by selecting the *fittest* ones (i.e. the ones currently most suitable to the task) and let them have *offspring*. This Darwinian process of refinement through selection and reproduction can be repeated indefinitely, and has for centuries been widely applied in any area involving living organisms, such as livestock and agriculture.

2.1 Evolutionary Computing

In computing, the use of evolutionary techniques is nothing new, the principal idea being as old as the computer itself [40]. In the decades following its theoretical conception, several branches emerged in what would later be known as evolutionary computing. The essence of solving a problem using evolution is to capture candidate solutions in a data structure that is capable of undergoing evolutionary operations such as genetic recombination and mutation. Such a *genotype* may directly or indirectly be expressed as a *phenotype*, terminology which has been conveniently borrowed from its biological counterparts. The phenotypic expression provides a solution that can be evaluated for a certain measure of quality, referred to as the solution's *fitness*. Selection takes place based on this value. The main advantage of this approach is that no deep understanding about the intricacies of a problem are required to arrive at a high quality solution, which may to a human observer be regarded as unintuitive and original [19]. The downside is that the way in which such a solution is reached cannot always be entirely understood, nor is it necessarily optimal [14]. Nevertheless its versatile applicability has led this principle to be successfully applied to a wide range of subjects, as aptly summarized in [20]. This article also emphasizes that the rise of Darwinian techniques in computing essentially marks a transition of evolution from real world 'wetware' to digital software, an emphasis reiterated here because of its relevance to what follows.

2.2 Evolutionary Robotics

From the field of evolutionary computing emerged in the early 1990s the field of evolutionary robotics, a term coined in 1993 [11] which can be defined as 'a method for the automatic creation of autonomous robots' [32]. An adequate overview of the emergence of and progression within this field can be found in several review papers on the subject [3, 23]. Evolutionary techniques were initially applied to find effective control systems for predesigned robots to perform a specific task, such as moving around as fast as possible while avoiding obstacles [24] or evolving a walking gait [2]. Although some works allowed for minor variations in e.g. sensor position [11], hardware limitations caused most research to refrain from also evolving robot morphologies. This did not hold back Karl Sims from experimenting with the principle in a couple of papers from 1994, one of which was mentioned earlier in this introduction [37, 38]. In his work he used a directed graph genome that could be represented as a physically realistic structure comprised of 3D primitives with a neural network brain, an approach that has served as an inspiration for the work presented in this thesis. Although the experiments were limited to simulation - the resulting creatures could not actually be constructed in real life - they did allow for exploring the interplay of coevolution of brains and morphologies. This holds significance because of the generally accepted notion in artificial intelligence that the mind and the body have a strong influence in shaping each other, a principle known as embodied cognition [3, 33]. Sims' modular approach has since then inspired several projects involving realistically constructible robots, aided by technological advances such 3-D printing. Lipson and Pollack made important progress by evolving robots consisting of basic components in simulation and verifying the resulting designs by producing the 'fittest' individuals in real life [34]. Along with these efforts, the work in simulation of Bongard and Pfeiffer [4] is cited as influential for the conception of RoboGen [1], a tool designed for educational purposes that is of particular significance to this work. Organisms in RoboGen are represented using genetic programming trees [30], phenotypically corresponding to 3D printable components for structure and electronic components for computation, actuation and sensory input.

2.3 Evolution of Things

Analogous to the transition from wetware to software, recent developments thus see evolution slowly make a transition back into the real world, this time embodied in synthetic hardware rather than organic wetware. This concept of (autonomous) robotic evolution in real-time and real-space, involving both morphologies and controllers, is otherwise known as the Evolution of Things [16, 20, 21]. Important progress in this area was recently achieved by Brodbeck et al. [9]. In this work robotic agents were designed, evolved, constructed and evaluated by a robot without any human intervention, providing all the principal components required for fully autonomous evolution in hardware.

2.4 Artificial Life

The basis of the evolutionary principle makes it interesting not only from an engineering standpoint, but also from a biological and perhaps even philosophical point of view. For a long time attempts to understand the world and the organisms within it have resorted to a reductionist approach, in which larger entities are dissected into the smaller components that make up their inner workings. It would seem however that many properties of complex systems only arise from an intricate interplay of all parts involved in the system and its environment - the whole is indeed greater than the sum of its parts. These phenomena can therefore not be sufficiently understood using a reductionist approach, much less can a satisfactory explanation be given for how they came to be. If the evolutionary process can in some way be reproduced under comprehensible terms and conditions, a much better understanding of the inner workings of biologic (eco)systems may be acquired. The branch of scientific research focusing on trying to mimic life with artificial systems is commonly referred to as Artificial Life, or ALife. Unlike many evolutionary computing projects, this branch of research is not primarily about optimization but rather about gaining an understanding of living systems by reproducing their properties in synthetic form, and as such relates closely to the subject of Computational Biology. The lack of an optimization to perform is often expressed in the absence of an explicit fitness function, and scenarios may be open ended. An early example of such an *artificial ecosystem* is Tierra [35], in which the individuals are computer programs competing for resources such as CPU and memory. Worth pointing out is the fact that the agents in this and other simulated agent based artificial ecosystems like Polyworld [43] and SugarScape [22] are not robots by any stretch of the definition but rather unspecified entities exchanging information. An example of an artificial ecosystem involving robots can be found in the work by Bredeche et al. [8] which concerns a population of simple e-puck robots that can move around within an arena. The robots can spread the genotype of their controllers only to other robots that are within a radius of 25cm. It should be noted that the lack of an explicit fitness function in this and other scenarios does not exclude the presence of *implicit* fitness, as the basis on which selection and reproduction are performed may very well favor individuals with certain traits. These traits may arise as a result of a given task or from an individual's adaptation to the environment. In case of the aforementioned paper, it turns out that the proximity requirement results in the genomes that allow for their hosts to move around fast while avoiding getting stuck to be prevalent, as it increases the potential population over which their genotype is spread.

As previously mentioned, artificial ecosystems involving physical robots are to some part of the grand vision for Evolutionary Robotics [16]. A further elaborated approach to such research is the Triangle of Life [17] framework, dividing lifespan of a robot into Birth, Infancy and Maturity, briefly summarized as the construction, learning and (potential) reproduction period of the robot. Research in this area however is still scarce, with the previously cited motivator for this thesis [42] being the only published result thus far.

2.5 Off-Line, On-Line and Embodied Evolution

The usual approach to evolutionary robotics is what could be called *disembodied* and *off-line* evolution, in which individuals are centrally generated, evaluated and reproduced in a serial manner in separate processes. While this approach may be logical in an entirely digital setting, it does not necessarily make sense when regarding real physical robots in the real world. In contrast, Embodied Evolution (EE) as introduced by Watson et al. [41] is an approach in which the evolutionary process takes place within a population of real robots. Eiben et al. further differentiate EE into systems that are *on-line* and *on-board* [6]. On-line indicates that evolution takes place during the robots' operational period, rather than the serial evaluation sketched before. On-board refers to the execution of the evolutionary algorithm taking place exclusively inside the hardware of the robots, rather than inside some external governor. It should be noted that this latter concept, while important, is of little immediate practical significance in pure simulation scenarios, where in general the simulation is carried out by a central agent. Caution should however be used to prevent the use of information in such simulations that would not be available in real life.

Several consequences arise from on-line, embodied evolution. Because the evolutionary process takes place during the operational period of a robot, its fitness value (if any) is no longer constant, but rather varies as a function of the individual's environment. These environmental changes are in large part influenced by the other robots in the same population. Whereas these robots may still be task-driven, a part of the focus is thereby directed to environmental adaptation of the individuals whilst performing this task.

The concepts of EE relate closely to ALife. Bredeche et al. [7] classify the differences and overlap, categorizing EE as being primarily objective driven (with an explicit or implicit fitness function) and ALife being primarily environment driven. There are however significant similarities, in that task completion goes hand in hand with adaptation to the environment. Throughout this work the term On-line Evolution shall be used to indicate ongoing evolution within an active population of robots (albeit in simulation), often keeping the 'embodied' aspect implicit for brevity where it applies.

2.6 Computer Models

Whereas the key interest of evolutionary robotics clearly lies with real, physical robots, experimentation with actual hardware is often time consuming and expensive and therefore only possible to a certain extent. In addition, experimentation with new technologies before or during their development may be desired. Even if the actual hardware is readily available, limiting the resources spent on it is generally beneficial. It is here that computer simulation comes into play, and unsurprisingly they have played a role in evolutionary robotics since the early days of the field, for instance in the work by Husbands from 1992 [26].

The exponential nature of the rise in computational power over the past decades has opened many windows in computational research. The new possibilities it brings along are of particular interest to the field of evolutionary computing, because the strength of evolution lies in numbers: it thrives by iterative repetition, and a larger number of computations is often the most straightforward way to a better result. The field of robotics on the other hand benefits from this advance by gaining access to faster and more accurate simulation scenarios, reducing the need for expensive and often tedious experimentation in real hardware. Both of these concerns combine where evolutionary robotics simulations are concerned: a desire for a large number of simulations on the one hand and high performance simulation of robotic structures in both fidelity and speed on the other.

The mere availability of the required silicon to address these needs however is not a solution in itself. Between a formed hypothesis and a working computational model with acceptable performance stands a time consuming process of implementation. Reducing the threshold of time between idea and experimentation is paramount in accelerating answering the wide array of open questions in the field. Although implementation code for existing research projects is increasingly made available to the the public, this code is often specifically tailored to the specific research scenario for which it was designed, making adapting or refactoring it to match another's needs a tedious process. General purpose robotics simulators on the other hand are available, commercial or otherwise, such as Webots [31] and Gazebo [28]. A broader overview is provided in Section 3.3. These packages come with a wide array of capabilities but provide little in terms of tools for evolutionary robotics.

2.7 The Reality Gap

A notorious problem in evolutionary robotics is the *reality gap* [27], referring to the behavioral differences between simulated systems and their real physical counterparts. While simplification, rounding and numerical instability lead to differences whenever computer models are involved, this effect is amplified in evolutionary systems. The reason for this is that evolution, as previously noted, will often solve its set challenges with unexpected solutions. While this is generally a favorable property, it also means that the process may eventually 'exploit' whatever modeling errors or instabilities are present, arriving at a solution that is valid only in the context of the simulation. Aside from careful calibration of the behavior of the simulator, the simplest and therefore most commonly used approach to counter this problem is to add noise to a virtual robot's sensors and actuators. While straightforward, this greatly increases the number of sensory representations of otherwise similar or identical states, slowing down the evolutionary process. More complicated approaches may involve alternating fitness evaluations between simulation and reality [3, 5, 29], but this negates the benefits of simulation in many scenarios and is infeasible when simulating entire artificial ecosystems. Crossing the reality gap is by no means a solved problem and as always one should be cautious drawing definitive conclusions from a model.

3 Related Work

This chapter will briefly assess the existing software that may be of value when performing evolutionary robotics research. In addition, Section 3.4 addresses similar experiments performed to those performed in this work.

Regarding simulation software, a distinction is made between three different categories:

– *Dynamics engines.* To perform a simulation of physical robots, software is required to calculate the effects of forces and collisions on the environment. This is a highly specialized task, for which generally an existing dynamics engine is used.

– *Robotic simulation platforms.* A variety of software packages aiding in robotics simulation is available. These generally incorporate one or more dynamics engines, and may provide a set of low and high level tools ranging from specifying a robot model and environment to providing control over its sensors and actuators.

– *Research projects and educational tools.* Existing research projects for which the source code is available may be used as inspiration or a starting point for other projects. They are commonly built either directly on top of a dynamics engine, or making use of a simulation platform.

Relevant existing software in each of these categories is addressed in the sections below.

3.1 Dynamics Engines

The popularity of realistic 3D gaming has the side effect that a wide variety of quality physics simulation engines have been created over the years which can also be used for research. One of the oldest and most widely used of such engines is the **Open Dynamics Engine** (ODE) [1], a free and open source rigid body dynamics engine that is still actively maintained and improved. A similar though more recent package is **Bullet Physics** [2], which has support for soft body dynamics as well. Bullet also promises support for GPU acceleration of dynamics, though at the time of this writing this support is anything but stable, let alone integrated in any of the packages making use of Bullet. Another example of an open source physics engine is **Newton Dynamics** [3]. While these engines were generally designed with video games in mind, they provide very functional high fidelity dynamics simulations that make them suitable for realistic simulations as well. This list is hardly exhaustive, as commercial physics engines for gaming naturally also exist, like Havok [4], which solely target video games and are therefore not considered here.

Other packages were designed from the outset as tools for realistic simulation, for instance the commercial Vortex Dynamics [5], which is often used for simulating equipment for military and industrial use. Open source examples in this category are the **Dynamic Animation and Robotics Toolkit** (DART) [6] and **SimBody** [7]. A special mention is reserved for **Voxelyze** [25], supporting soft-body dynamics through voluminous pixels or *voxels*. Voxelyze is the result of a smaller research project and therewith not nearly as mature or well maintained as the other dynamics engines in this chapter. In addition, having to calculate interactions between many small blocks make voxel dynamics computationally heavy.

The performance of each of these engines is hard to assess, as it is both a qualitative matter as well as a quantitative one. Generally some careful parameter tuning is required to get a stable simulation at minimal computational effort. Performance varies based on the use case, so having the ability to easily switch between engines, which many of the simulation platforms from Section 3.3 provide, can prove advantageous.

[1] `http://www.ode.org`
[2] `http://www.bulletphysics.org`
[3] `http://newtondynamics.com`
[4] `http://www.havok.com`
[5] `http://www.cm-labs.com`
[6] `http://dartsim.github.io`
[7] `https://simtk.org`

3.2 Research Projects and Educational Tools

Although research projects involving ecosystems with robots evolvable minds and morphologies are scarce, any evolutionary robotics project that shares some characteristics with the desired setup could in theory be adapted and used if its source code has been made available. The precursor to this project has freely available source code [8], which is relatively reusable, albeit targeted most specifically at Roombots, and comes with the disadvantages sketched earlier. A software tool released by NASA, the **NASA Tensegrity Robotics Toolkit** (NTRT) [10], provides simulation of tensegrity robots, which are structures consisting of 'tendons' and small fixed components put together in such a way that a soft but stable structure results. NTRT includes examples where the brains of these structures are evolved using off-line evolution, but is not built with morphology evolution in mind. **Ludobots** [9] is a small educational tool developed by Josh Bongard [3], using ODE for dynamics. It allows for experimentation with simple modular robots with a neural network brain, though these robots are not necessarily realistic. The tool is quite minimal and its source code is minimally documented and does not appear to be maintained. A final tool which has been touched upon in the background section is Joshua Auerbach's **RoboGen** [1] which involves modular robots that can be constructed in real life through a combination of 3D printing and attaching electronics. The robots have a neural network brain that can be evolved as well as their morphology. The morphology 'building plan' leaves room for different components than those that are used by default. Simulation scenario's in RoboGen are entirely off-line, with only one robot evaluation at a time.

3.3 Robotic Simulation Platforms

Before addressing the various simulation platforms, it is worth mentioning the Robot Operating System (ROS) as it is a term often heard in this context. ROS encompasses a range of middleware to aid in the development of robot software. Conceptually, a ROS setup consists of isolated nodes, where each node has a specific task, such as reading out sensor values or performing computations for a robot's control system. ROS provides hardware abstractions and message passing functionality between these nodes. The modularity of this approach makes it possible to, for instance, swap out communication to a physical robot with communication to a simulator without changing the rest of the setup. Many simulators come bundled with ROS integration, but ROS itself is not a simulator. Depending on the use case, ROS may be a valuable tool, especially for developing real physical robotic systems. Where pure simulation is concerned however, it can be argued that the ROS libraries add complexity and performance overhead. In addition, software integrations with ROS often lag behind

[8]`https://code.google.com/archive/p/tol-project/`

[9]`http://www.uvm.edu/~ludobots`

the main versions of simulation platforms because of their independent development.

As mentioned in the introduction, the work directly inspiring the efforts in this thesis made use of the **Webots** robot simulator [31], a commercial simulation platform powered by the Open Dynamics Engine. It comes packed with many features like human readable files for specifying scenes and models, interfaces to various programming languages and ROS integration. Its downsides were also previously touched upon: performance and stability issues, which may be hard to debug or improve because the source code of the platform has not been made available. In addition, Webots requires a costly commercial license.

Another commercial simulator is Coppellia Robotics' **V-REP** [36]. Although commercial, licenses for V-REP are available that allow use of the simulator free of charge for students and employees of educational institutions, as well as having its source code available. The simulator provides a high degree of extensibility through plugins, scripting and API bindings with several different programming languages. ROS integration is also available. Four different engines are supported for physics simulation: Bullet Physics, ODE, Vortex Dynamics and Newton Dynamics. In addition, a versatile user interface allows creating and modifying scenes and robots. A downside of V-REP is the binary format that is used for storing scene and model configurations, making it hard to alter these without the use of the platform itself.

A package similar to V-REP is Gazebo [28], which is maintained by the Open Source Robotics Foundation (OSRF) [10], the same organization behind ROS. It should therefore come as no surprise that Gazebo generally provides the most up-to-date ROS integration. Gazebo is entirely open source and the source code is available under the permissive Apache 2.0 license. Model and environment configurations are specified in an XML based format. Integration with the simulator is possible through either a publisher / subscriber based messaging API or through compiled plugins. Support is included for four dynamics engines: Bullet Physics, ODE, DART and SimBody. The simulator comes bundled with an optional user interface providing scene visualizations, control and simple editing capabilities.

The final simulator discussed here is MORSE [15], which to quote its website [11], is a 'generic simulator for academic robots'. It is built on top of Blender [12] for its rendering and dynamics simulation, which is in turn powered by Bullet Physics. MORSE communicates with Blender through Python bindings, and is itself written predominantly in Python. The combination of Blender and Python makes for a developer friendly experience, but does impose some limitations on fine grained control over simulation performance. MORSE simulations and robots are specified as a combination of Blender objects and Python code. It can be integrated using several middlewares, including ROS or a simple socket connection. Being based on Blender, MORSE always

[10] `http://www.osrf.org`
[11] `https://www.openrobots.org/morse`
[12] `http://www.blender.org`

requires an instance of Blender to run for its simulations, even if visualizations are not required. An interesting property of MORSE is its potential for multi-node simulation [13]. This distributes the calculations and sensor readings performed for each robot over a number of computational nodes, while synchronizing the actual resulting scene.

3.4 Similar Research

The background section of this thesis has addressed the past and current work in the field of Evolutionary Robotics and ALife. Off-line ER research is plentiful, whereas [42] is presently the only concrete example of an on-line artificial ecosystem evolving both morphologies and bodies (albeit in simulation). Although it is recognized in [8] that a trait such as increased speed can arise in the absence of an explicit fitness function, little is known about what happens at the 'edge' of these two scenarios - where evolution is on-line in an active population, but still governed by a central agent. The type of experiment performed in this thesis is therefore new, to the best of the author's knowledge.

[13]`https://www.openrobots.org/morse/doc/stable/multinode.html`

4 The Revolve Toolkit

The **R**obot **Evolve** Toolkit, developed as a part of this thesis, is a set of Python and C++ libraries created to aid in setting up simulation experiments involving robots with evolvable bodies and/or minds. It builds on top of the Gazebo simulator, complementing this simulator with a set of tools that aim to provide a convenient way to set up such experiments. The design of Revolve is motivated in Section 7.3. Revolve's philosophy is to make the development of simulation scenarios as easy as possible, while maintaining the performance required to simulate large and complex environments. In general this means that performance critical parts (e.g. robot controllers and parts relating to physics simulation) are written in the C++ language, which is highly performant but can be tedious to write, whereas less performance focused parts (such as world management and the specification of robots) are written in the slower yet more development friendly Python language. The bulk of the logic of a simulation setup commonly falls in the latter category, which means the experimenter will be able to implement most things quickly in a convenient language. Revolve's source code can be found at `https://www.github.com/ElteHupkes/revolve`.

This chapter starts with a description of Gazebo, and continues to describe the properties of the Revolve Toolkit in more detail. For more information about the Gazebo simulator, please refer to its website at `http://ww.gazebosim.org`.

4.1 Gazebo

As mentioned in Section 3.3, Gazebo is an open source, multi-platform robotic simulation package that is available free of charge. It provides both physics simulation and visualization of rigid body robotic structures and their environments. Abstraction wrappers are available for four well established dynamics engines, meaning that, in theory, it is possible to run simulations using any of these physics engines by changing a single parameter. The caveat is that implementation details cause behaviors to be slightly different between engines, meaning there is generally a need for parameter tweaking to get simulations to behave desirably.

In order to describe robots and environments, Gazebo uses the Simulation Description Format (SDF) [1], which allows an end user to specify anything from the texture of the terrain to the physical and visual properties of robots in an XML-based format. Because XML can be cumbersome to write for human beings, the `sdf-builder` [2] Python package was developed concurrently with Revolve to provide a thin, structured wrapper over this format that aids with positioning and alignment of geometries, and calculation of their physical properties.

What makes Gazebo particularly useful is the means by which it allows programmatic access to observing and modifying the simulation. It provides two main interfaces to do this:

- A messaging API. Gazebo comes bundled with a publisher / subscriber messaging system, in which any component can subscribe to and / or publish on so called *topics*. Many aspects of the system can be controlled using these messages, which are specified in Google's *Protocol Buffers* (Protobuf) format [3]. Because this communication happens over TCP sockets, access to this interface is quite straightforward from most programming languages.

- The plugin infrastructure. It is possible to load shared libraries as a plugin for several types of Gazebo components, providing programmatic access to the simulation using Gazebo's C++ API. As an example, one can specify a certain piece of compiled C++ code to be loaded with every robot that is inserted into the world.

Revolve makes use of both of these interfaces to provide its functionality.

[1]`http://sdformat.org/`
[2]`https://github.com/ElteHupkes/sdf-builder`
[3]`https://developers.google.com/protocol-buffers/`

4.2 The Revolve Specification

The basic goal of Revolve is to provide a set of libraries that facilitate describing robots in terms of a set of predefined parameterized base parts, converting these descriptions into a format that can be simulated by Gazebo and providing control and feedback utilities with regard to this simulation. The mechanism to construct arbitrary robots from predefined parts, known as the *Revolve Specification*, is heavily inspired by RoboGen [1] and boils down to the following principles:

- A robot consists of one or more parts p_i of type $\tau_i \in \mathcal{T}$ where $\mathcal{T}$ is the set of all possible part types. A part type specifies at least:
    - A number of *connection slots* $c(\tau_i)$. Physically speaking, these are the positions at which parts can be attached to other parts.
    - A number of inputs and outputs, each corresponding to one numerical value. Although the user is free to choose the function of these inputs and outputs, generally inputs will be used to specify sensor values, while outputs specify actuator values.
    - Zero or more numeric parameters $p(\tau_i)$, which allow for specifying variation within body part types.
- A robot is defined by a labeled tree graph $\mathcal{R}$, where each node r_j specifies at least:
    - A body part type $\tau_i \in \mathcal{T}$.
    - A list of values $\boldsymbol{\pi}_i$ for the numeric parameters $p(\tau_i)$.
    - A set of labeled node edges $\mathcal{C}$, describing how this body part is attached to other body parts. Each connection $c_i \in \mathcal{C}$ is a tuple (r_j, f, t) where $r_j \in \mathcal{R}$ defines a child node and f and t are the `from` and `to` *slots* of the connection respectively. To describe a valid robot, these slots must be supported by the respective nodes' part types. Because $\mathcal{R}$ is a tree, $\mathcal{C}$ is not allowed to form any cycles in the node graph.
    - An orientation ω encoding the orientation of the body part relative to its connection slot, or to the world if its node is the root of the tree.

Knowing the tree graph that describes a robot provides a blueprint that, combined with information on the actual physical properties of the parts, allows for construction of the robot. Note that the Revolve Specification only prescribes the syntax of this blueprint, it does not make any assumptions about these parts' physical structure or function. This blueprint syntax is captured in a Protobuf message, which is straightforward to generate from many programming languages and is easy and efficient to communicate over the network if required. Any robotic body space that maps onto the specification can in principle be implemented in - and simulated with Revolve. This allows the specification to be conveniently extended with other properties and for instance be used as a genome, as will become apparent later in this work.

4.3 Revolve Libraries

At the heart of Revolve lie a set of general purpose tools, which can be roughly separated into Python components and Gazebo C++ plugin components. A certain layering is present in the provided tools, ranging from anything between closely related to the specification to more practical tools that can be used to quickly implement an actual experiment. The most 'opinionated' part of Revolve, called `Revolve.Angle`, provides everything for a complete robot evolution experiment short of the actual body parts being used and is discussed in Section 4.4.

4.3.1 Python Libraries

The most basic Python libraries include functionality to:

– Create a *body specification*, specifying a body space in terms of *components*. A Revolve component is a Python module that wraps over the structure classes in the previously mentioned (Section 4.1) `sdf-builder` library. In addition to the physical and visual properties of the body part, to be given in terms of SDF elements, it should specify where the attachment slots are located and what sensors and / or actuators the body part implements.

– Describe actual robots in terms of this body space by making use of the Revolve Specification messages. Rather than using the Protobuf messages directly, robots can be specified in a more user Python object notation, or in the `YAML` [4] format. Tools are included to convert between these formats.

– Given a body specification, convert a robot description into an SDF representation that can be loaded into Gazebo.

– Generate a random robot body from a specification, with the possibility to provide certain constraints.

– Observe and modify the state of a running simulation in Gazebo. Revolve makes use of the `pygazebo` library [5] in order to interface with Gazebo's messaging API (mentioned in Section 4.1). The most common functionalities, such as inserting and removing robots, logging their positions and pausing or resuming the simulation are encapsulated in a `WorldManager`, which can be used as a basis on top of which more advanced system behaviors are built.

[4]`http://yaml.org/`
[5]https://github.com/jpieper/pygazebo

These previous functionalities all fail to address an important part of any functional robot: its brain. Recognizing that many different types of brains may be desired depending on the use case, an attempt has been made to make the body specification as broadly applicable as possible. That being said, Revolve does include functionality to work with one specific brain type, which is a recurrent neural network. The libraries written for this purpose allow one to:

– Specify the basic properties of a neural network, for instance the neuron types (i.e. the type of used activation function) and range limits of the weights.

– Describe a neural network in terms of its neuron types and connection weights. This again uses a Protobuf message as the central format, with converters from and to more readable formats.

– Generate a random neural network from an interface, within given constraints. A separately generated robot body can be used as an interface, turning all its specified sensor and motor values into input and output neurons respectively. One of these formats is XML-based and can be used to add the description of the neural network to the SDF contents sent to Gazebo.

Note that the actual control of a robot will have to take place inside the simulation, for which at least some code is required. The neural network plugin currently available in Revolve that does this is addressed in Section 4.3.2.

4.3.2 Gazebo Plugins

In order to actually control a robot in simulation, Gazebo has to be told what sensor values to read, what joints to control, et cetera. While it is possible in principle to provide most of these functionality through the messaging API, when it comes to controlling a robot the code is closely related to the simulation, runs often and is therefore more apt to be considered as a high performance aspect to be written in C++. Revolve supplies a base *robot controller* plugin to deal with this aspect of the simulation setup. When the SDF contents of a robot are produced for simulation, a reference to this plugin is included alongside information about its sensors, actuators and brain. Gazebo supports many types of sensors, all of which are accessed in a different fashion. Revolve wraps over a number of often used sensors and unifies them in a generic interface passed to the robot controller. The same holds for actuators, which control the joints of robots in simulation. Rather than having to specify the forces that operate on these joints, Revolve allows setting either a position or velocity target which, combined with a predefined maximum torque, resembles the interface of a real world servo motor.

The robot controller gathers the sensor and actuator information, constructs the adequate control classes and passes this information to an abstract method that loads the robot brain. By default, this method loads a controller for the recurrent neural network as described at the end of Section 4.3.1.

In addition to the robot controller, Revolve also includes a *world controller* plugin, which should be included with each loaded simulation world. While using this plugin is not strictly necessary, it includes some convenient functionality to insert robots into the world, keep track of their position and remove them.

4.3.3 The Body Analyzer

When dealing with randomly generated, mutated or evolved robots, there is often no easy way to make sure if a candidate robot is actually *valid*, that is, realistically constructible. More specifically it is often useful to know

- if the candidate has any body parts that intersect and
- what the dimensions of the candidate robot are.

To be able to answer these questions about a candidate robot, Revolve comes bundled with a standalone *Body Analyzer*. This program is nothing more than an instance of Gazebo, combined with a plugin that accepts a robot body in a network message, loads it into the world and communicates is properties back over the network. The analyzer can be run alongside a simulation experiment to decide whether or not to use a certain candidate robot.

4.4 Revolve Angle

Alongside the modules to create a wide variety of experimental setups described in the previous sections, Revolve includes a more opinionated module called `revolve.angle`, implementing a specific subset of all possible experimental setups. Its function is twofold, in that

- it allows for setting up any experiment matching these setup descriptions rapidly and
- it serves as an example on how to use Revolve.

Revolve Angle implements the following functionality:

- A genome including both a robot's body and brain.
- A conversion from this genome to a usable SDF robot.
- Evolutionary operators functioning on this genome: crossover and mutation.
- The entire RoboGen body space is included as Revolve Components, though its use is optional and other body parts may just as well be used with the genome.

This section discusses these properties of Revolve Angle.

4.4.1 Genome

The genome used by Revolve Angle is inspired by what was previously used in RoboGen. It is a single-rooted tree graph encoding the layout of the robot body, extended to incorporate the robot brain, a fully connected recurrent neural network. The genome is essentially the Revolve Specification (Section 4.2), where each node is extended with neural network properties to include:

– Two lists of neuron types and parameters, $\mathcal{N}_{\text{output}}$ and $\mathcal{N}_{\text{hidden}}$. Each is a list of tuples $(t, \boldsymbol{\pi}_{\text{neuron}})$, where $t \in \Lambda$ is a supported neuron type (see Section 5.2) and $\boldsymbol{\pi}_{\text{neuron}}$ is a list of values for the parameters specified by this neuron type. $|\mathcal{N}_{\text{output}}|$ is determined by the number of outputs specified by the body part type, $|\mathcal{N}_{\text{hidden}}|$ can vary.

– A set of paths $\mathcal{Q}$. Each path $q_i \in \mathcal{Q}$ is a tuple $(\boldsymbol{s}, w, o_1, t_1, o_2, t_2)$ with $\boldsymbol{s}$ being an ordered list positive integers $(s_0, s_1, ...)$, $|\boldsymbol{s}| \geq 0$ and w a numeric weight. Furthermore, $t_1 \in \{\text{input}, \text{output}, \text{hidden}\}, t_2 \in \{\text{output}, \text{hidden}\}$ denote the type of a neuron, with o_1, o_2 being corresponding integer offset values. Every path q ultimately encodes a connection in the neural network of the robot.

4.4.2 Phenome

The genome defined in Section 4.4.1 forms a blueprint for a robot, which can be turned into an actual robot representation only given the set of part types $\mathcal{T}$. While Revolve Angle optionally includes the RoboGen body space, a different set of body parts can just as easily be used. The same holds for the robot's brain, where the default neural network as used in RoboGen is provided, which can be changed when desired.

The RoboGen body space and neural network are addressed in sections 5.1 and 5.2 respectively.

4.4.3 Genotype to Phenotype Conversion

Construction of an actual robot from a genotype starts at the root node of the tree. The robot body is constructed by recursively traversing the tree edges attaching components at the 'from' and 'to' slots of the connection. This process aligns the body parts such that the slot normal vectors point in opposite directions, the slot tangent vectors point in the same direction and the slot surfaces touch at the slot points.

The interface of the robot's neural network results directly from parts' inputs and outputs, all of which is represented by one respective input or output neuron in the

Parameter	**Description**
$\|\mathcal{R}\|_{\text{max}}$	Maximum number of nodes
$\|\mathcal{R}\|_{\text{min}}$	Minimum number of nodes
o_{max}	Maximum number of outputs in a robot
i_{max}	Maximum number of inputs in a robot
h_{max}	Maximum number of hidden neurons in a robot
$\nu(\tau)$	Function returning the maximum number of nodes with part type τ
$\mathcal{T}_{\text{root}}$	Set of possible part types for the tree root node
$\mathcal{T}_{\text{child}}$	Set of possible part types for the tree child nodes
Λ_{hidden}	Set of possible activation functions for hidden neurons
Λ_{output}	Set of possible activation functions for output neurons

Table 4.1: Genotype restriction parameters

brain. The properties of each node's output neurons are set to match the values in its $\mathcal{N}_{\text{output}}$. For each node, $|\mathcal{N}_{\text{hidden}}|$ hidden neurons are constructed with corresponding type and parameter values.

The fully connected neural network is initialized with zero weights for all connections. For each node r_i, the path portion $\boldsymbol{s}$ of $\mathcal{Q}$ is traversed following node edges where the from slot f matches the path value s_i, until no s_i are left upon which point node r_j is reached. The weight of the network connection between $r_i(t_1, o_1)$ and $r_j(t_2, o_2)$ is then set to the weight portion w of $\mathcal{Q}$, where $r_i(t, o)$ denotes the o^{th} neuron of type t belonging to node r_i. If at any point the path cannot be traversed, either because a node edge at the from slot s_i is not available or because $r_i(t_1, o_1)$ or $r_j(t_2, o_2)$ does not exist, the path is simply ignored and no weight is set.

4.4.4 Restrictions

The genome paired with available body parts impose a few natural restrictions on a genotype, namely the values for τ and $\mathcal{V}$ as well as 'from' and 'to' slot values for the tree edges. In addition to these restrictions, a genotype is further constrained by several system parameters, which are listed in table 4.1. Paths in $\mathcal{Q}$ that do not resolve to any node / neuron combination *are* allowed, they will simply not lead to any neural connection (see Section 4.4.3).

Furthermore, the phenotype space is usually constrained to containing only robots that could realistically be constructed, i.e. robots with intersecting body parts are rejected. The body analyzer (Section 4.3.3) can be used to determine whether a generated or evolved robot matches this criterion.

4.4.5 Robot Generation

The random generation of robot genotypes is a general feature of Revolve that is further specified by Revolve Angle. The procedure to generate a random genome tree works as follows:

- A random number is drawn from $\mathcal{N}(\mu_{\text{parts}}, \sigma^2_{parts})$ to determine the targeted number of body parts for this robot.

- A random part is drawn from $\mathcal{T}_{\text{root}}$ to be used as the root of the robot.

- A random free slot on the robot body is chosen along with a part from $\mathcal{T}_{\text{child}}$ and a random slot on this part. After adding this combination to the tree, this step is repeated until either the targeted count of body parts is reached, there are no free slots available or any added body part would violate previously imposed restrictions on the body. This completes the robot body. The specification for the body parts determines the input / output interface of the robot's neural network.

- A number is drawn uniformly from $[0, h_{\max}]$ to determine the number of hidden neurons in the robot. This number is randomly distributed over the body part nodes.

- Activation functions for the hidden and output neurons in the tree are randomly drawn from Λ_{hidden} and Λ_{output} respectively. Any associated parameters are randomly initialized.

- A neural network connection weight is chosen for all possible neuron combinations $\mathcal{N} \times \mathcal{N}_{hidden} \cup \mathcal{N}_{output}$. This weight is a random uniform value from $[0, 1]$ with a probability $p_{\text{connect neurons}}$ and zero otherwise.

The generated genotype resulting from this procedure is not guaranteed to be viable, because it is generally not possible to determine any intersecting body parts as mentioned in Section 4.4.4. The body analyzer can be used to determine this fact.

4.4.6 Evolution

Revolve Angle includes a procedure of evolutionary operators in which two 'parent' robots can produce offspring through several operations on their genotype trees. This section discusses these operation in the order in which they are applied to create a child robot c from parents a and b.

4.4.6.1 Crossover

In the first step, a node a_c from a is randomly chosen to be the crossover point. A random node b_c from b is chosen to replace this node, with the condition that doing so would not violate the restrictions as given in Section 4.4.4. If no such node is available, evolution fails at this point and no offspring is produced. If such a node is found, c_1 is created by duplicating a and replacing the subtree specified by a_c with the subtree b_c.

4.4.6.2 Subtree Removal

This step involves picking a random node from c_1 and removing it and its subtree from c_1 to produce c_2. Subtree removal happens with a probability $p_{\text{remove subtree}}$, which is an experimental parameter.

4.4.6.3 Subtree Duplication

In subtree duplication, a random node d from c_2 is chosen to be duplicated, along with a random free slot on one of c_2's parts, again only provided that duplicating this node would not violate robot restrictions. The subtree specified by d is then duplicated onto the chosen free slot to produce c_3. This operation is performed with parameter probability $p_{\text{duplicate subtree}}$. If no trees could be duplicated without violating restrictions, this step is skipped and $c_3 = c_2$.

4.4.6.4 Subtree Swap

With probability $p_{\text{swap subtree}}$, a random node s_1 is chosen from c_3. Another random node s_2 is chosen provided it has no ancestral relationship with s_1 (i.e. it is not a parent or child of this node). If no such node is available the step is again skipped, otherwise s_1 and s_2 are swapped in c_3 to produce c_4.

4.4.6.5 Hidden Neuron and Neural Connection Removal

For each node in c_4, its hidden neurons $\mathcal{N}_{\text{hidden}}$ and paths $\mathcal{Q}$ are traversed. For each hidden neuron, a choice is made to either keep or delete the neuron with probability $p_{\text{remove hidden neuron}}$. The same decision is made for paths in $\mathcal{Q}$, which are removed with probability $p_{\text{remove neural connection}}$. The resulting genotype is c_5.

4.4.6.6 Parameter Mutation

Each node in c_5 has the parameter set $\boldsymbol{\pi}_{\text{body}}$ and parameter sets $\boldsymbol{\pi}_{\text{neuron}}$ for each hidden or output neuron. These parameters π are all iterated, and a new parameter π^* is generated with a random uniform draw from within the valid range of the respective parameter. The parameter is then changed from c_5 to c_6 to have the value $(1-\epsilon)\pi+\epsilon\pi^*$, where ϵ is a property that determines the mutation rate.

4.4.6.7 Hidden Neuron and Neural Connection Addition

To prevent the number of hidden neurons from tending to zero over many generations, hidden neurons are added to c_6 to create c_7. The number of newly created hidden neurons in c_6 is equal to the expected number of removed hidden neurons between c_4 and c_5. This is done to keep the number of neurons the same on average, while not keeping it strictly the same between generations. The type and parameters of newly created hidden neurons in c_7 are randomly generated. A similar process is performed for each neuron's paths in $\mathcal{Q}$, which are added in an equal number as that is expected to have been removed. It should be noted that these paths are initialized to point to valid neurons, so no broken paths are initially generated. Path weights are randomly drawn.

4.4.6.8 Part Addition

Again in order to keep robot complexity roughly the same, a new part is added with a probability proportional to the number of parts that are expected to have been removed by subtree removal, minus the number of parts expected to have been added by subtree duplication. The new part is randomly generated with both hidden neurons, neural connections and all parameters, and attached to a random free slot on the tree to produce the final robot c. Again, this step is skipped if adding a part would violate restrictions.

5 Experimental Setup

The experiments described in this work make use of the genotype and evolutionary operators as specified by Revolve Angle, using the both the body parts and neural network as they are used in RoboGen. This section discusses the properties of these body parts (Section 5.1) and neuron types (Section 5.2), as well as the experimental scenarios performed with them (Section 5.4).

5.1 Body Part Types

This section details the set $\mathcal{T}$ of body part types used in the experiments in this work. In large, these are the same body parts as used by RoboGen at the time of this writing,[1] though improvements and changes to this package are continuously made and they may no longer match. The parts are therefore detailed briefly for completeness. Detailed structure meshes suitable for use with a 3D-printer are available for these body parts, such that these robots could in principle be constructed in real life. Table 5.1 shows a brief overview of each type's properties as enumerated at the start of this section, accompanied by a short section dedicated to each part.

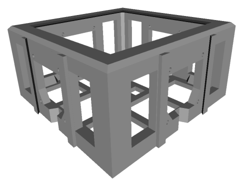
[1]`http://robogen.org/docs/robot-body-parts/`

Part	**Slots**	**Inputs**	**Outputs**	**Parameters**
Core Component	4	6	0	0
Fixed Brick	4	0	0	0
Parametric Bar Joint	2	0	0	3
Active Hinge Joint	2	0	1	0
Passive Hinge Joint	2	0	0	0
Touch sensor	2	2	0	0

Table 5.1: Body part type properties as described at the start of Section 4.4.2. The number listed underneath ‘Parameters’ specifies the part specific parameters. In addition to these parameters, each part has three color parameters (red, green and blue) which are used for visualization purposes.

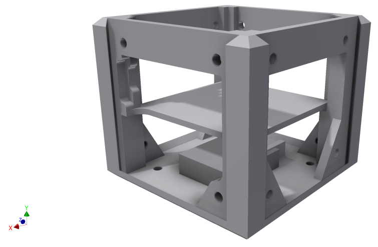
5.1.1 Core Component

This component forms the root of every robot and in real life holds a Raspberry PI powering the robot’s brain (not displayed on the image). Two sensors are included on the component: an accelerometer and a gyroscope. The x, y, z values for both these sensors result in a total of 6 input values. The component has four attachment slots: apart from the top and bottom face other parts can be attached to every side.



5.1.2 Fixed Brick

The fixed brick is a cubic component containing no sensors or actuators. Attachment slots are all faces except the top and bottom.

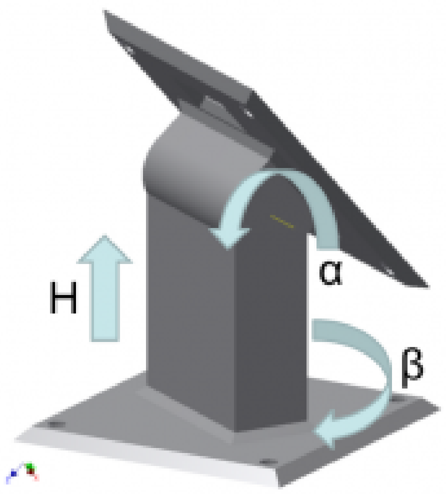


5.1.3 Parametric Bar Joint

The parametric bar joint introduces fixed angles into a robot structure. The part has two parameters:

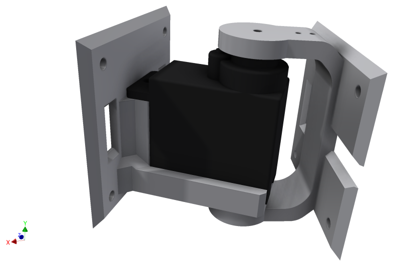
- H, the length of the joint, between 2 and 10 centimeters
- α, the tilt of the joint, between -90 and 90 degrees

The image contains a third parameter, the rotation β, which is currently constrained to 0 degrees to enforce planarity of the robots (see Section 5.3).



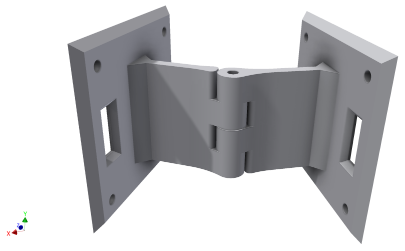
5.1.4 Active Hinge Joint

A simple hinge joint powered by a servo motor. It has one output value, which is truncated to the interval $[0, 1]$ and corresponds to a target position between -45 and 45 degrees.



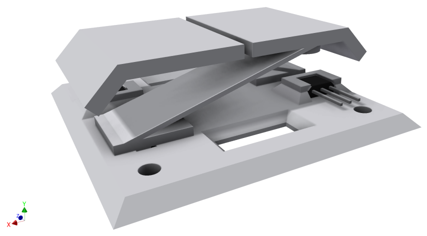
5.1.5 Passive Hinge Joint

Similar to the active hinge joint, but as the name suggests the joint is not powered by a servo but rather moves freely.



5.1.6 Touch Sensor

A two-value binary touch sensor. Each sensor covers half of the total part surface, and outputs a value that is either 0 or 1 depending on whether touch is registered.



5.1.7 Light Sensor

It should be noted that the default RoboGen body space also includes a light sensor. While this sensor is also available as a default Revolve component, at the time of this writing instabilities in the Gazebo simulator prevent it from being used in the experiments described in this work. The light sensor is therefore disabled.

5.2 Neuron Types

The neurons in the hidden and output layers of the neural network that is the robot's brain have one of three activation functions. The first two, linear and sigmoid, are common neural network activations whose parameter set $\boldsymbol{\pi}_{\text{neuron}}$ consists of a bias and a gain value. The third possible type is an oscillator neuron, whose value depends not on its input values but rather is a sinusoid depending only on the current time. The three parameters for this neuron type are the wave period, phase offset and gain.

5.3 Viability Criteria

In addition to the restrictions listed in Section 4.4.4, a few other constraints are specified in the experimental setup:

- Like in RoboGen, robots are forced to be *planar*, meaning their extremities extend only in the x-y plane, resulting in more stable structures.
- Because robots are evolved for their ability to move, only robots with at least one motor unit are considered viable.

Parameter	**Value**
$\lvert\mathcal{R}\rvert_{\text{max}}$	30
$\lvert\mathcal{R}\rvert_{\text{min}}$	3
o_{max}	10
i_{max}	10
h_{max}	10
μ_{parts}	12
σ_{parts}	5
$p_{\text{remove subtree}}$	0.05
$p_{\text{duplicate subtree}}$	0.1
$p_{\text{swap subtree}}$	0.05
$p_{\text{remove hidden neuron}}$	0.05
$p_{\text{remove neural connection}}$	0.05

Table 5.2: Parameter values shared across all experiments

5.4 Scenarios

This section enumerates the simulation scenarios that were performed making use of the robot genome and phenome as described in earlier sections. All these experiments share a set of values for previously specified parameters, which are specified in table 5.2. Six distinct scenarios are simulated in total: one to measure computational performance (Section 5.4.1), two 'baseline' experiments (Section 5.4.2) and three scenarios aimed at comparing off-line and on-line evolution (Section 5.4.3). Each simulation is repeated 30 times to get reliable results.

5.4.1 Computational Performance

The computational performance of the system in complex simulation scenarios is assessed in a series of experiments similar to scenario 3 as outlined in Section 5.4.3. In each simulation run a number of random robots is generated and inserted into the virtual world. The simulation is then started and executed for 10 seconds simulation time, measuring the real (wall clock) time required to complete this time interval. This process is repeated for populations of increasing size.

5.4.2 Baseline

In order to evaluate the efficacy of the Darwinian evolution applied in the experiments to follow, two baseline experiments were performed for comparison, which are similar in setup to the off-line evolution experiments described in Section 5.4.3. The first makes use of the fitness selection to determine which individuals survive while disabling reproduction, thereby showing the speed at which a population would increase its fitness if a selection is made out of an increasing random population. The second experiment on the other hand uses completely random survivor selection, while enabling reproduction. This allows ruling out a bias in the reproduction process, i.e. showing that a population's fitness doesn't change by merely reproducing the individuals without selecting them.

5.4.3 Evolution

In the final three experiments actual evolution is performed, and they mark a first step into the type of research that Revolve was designed to facilitate. They are simulations of three different scenarios, each moving a step in the direction of a more on-line experiment. The first two experiments are common off-line scenarios in which individuals are evaluated in isolation and a population consists of distinguishable generations. What differentiates these two scenarios is the parent selection method: in the first scenario 15 new individuals are produced before further selection takes place, whereas in the second scenario selection happens after each newly born robot. This method of parent selection is more akin to the final scenario, which is an on-line scenario in which robots coexist in the environment and are continuously evaluated and selected.

The fitness function for a robot ρ is the same in all of these scenarios, and reads

$$f(\rho) = v + 5s, \tag{5.1}$$

where v is the average path velocity of the robot over the last 12 seconds in meters per second, and s is the average straight line velocity over that same time window. The path velocity is calculated using the entire distance of the path the robot has traveled during the time window, whereas the straight line velocity only uses the length of a straight line between the point where the robot was 12 seconds ago and the point where it is now. Vertical displacement is ignored for both of these values.

The following table summarizes the three simulation scenarios by their properties:

	Scenario 1	**Scenario 2**	**Scenario 3**
Scenario type	Off-line	Off-line	On-line
Environment	Infinite flat plane.	Infinite flat plane.	Infinite flat plane.
Evaluation	One robot at a time for 12 seconds.	One robot at a time for 12 seconds.	All active robots simultaneously and continuously. Fitness is measured over a 12 second sliding time window.

Population size	Constant at 15 per generation.	Constant at 15 per generation.	Variable between 8 and 30 (see Death criterion).
Selection scheme	15 + 15: Each generation of 15 robots produces 15 children before moving on to survivor selection.	15+1: Each generation of 15 robots produces 1 child before moving on to survivor selection.	A new robot is born every 15 seconds.
Parent selection	4-tournament selection.[i]	4-tournament selection.	4-tournament selection.
Death criterion / survivor selection	Deterministically choose the 15 fittest individuals from the 30.	Determinstically choose the 15 fittest individuals from the 16.	At fixed time intervals, all robots that have a fitness less than a fraction 0.7 of the mature[ii] population mean are killed. A minimum of 8 robots is maintained to ensure variation and prevent extinction. If the population reaches 30 individuals without any individuals matching the death criterion, the 70% least fit robots in the population are killed regardless of their fitness to prevent a simulation stall.
Birth location	On the ground at the origin.	On the ground at the origin.	Random position within a circle of radius $2m$ around the origin.
Stopping criterion	After 3000 births, i.e. 200 generations of 15 individuals.	After 3000 births.	After 3000 births.

[i] In a 4-tournament selection, 4 individuals are chosen randomly from the (mature) population, the fittest of which is taken to be parent. This process is repeated to select a second parent, which is forced to be a different robot than the first parent. [ii] Maturity is reached for all individuals older than 15 seconds, which is 3 seconds insertion time for when the robot is dropped into the arena followed by 12 seconds evaluation time.

5.5 Simulation Parameters

Apart from the parameters related to the evolutionary process and experimental setup, there is a long list of parameters that can be used to alter the behavior of the the simulator and the dynamics engine. These concern the type of solver used for the dynamics, the length of one calculated time step, error correcting forces, surface friction parameters and many more. At each step of the simulation, rounding and discretization lead to errors of varying degree. It is the interplay of these variables that determines the overall performance and stability of any simulation, which is also affected by the size of the simulated structures and the forces acting upon them. Smaller step sizes generally lead to more stable simulations, at the expense of being more computationally intensive. Determining these parameters is therefore always a trade-off between the desired performance and realism. The experiments described in this chapter all use the ODE 'quick' solver, with a maximum step size of 0.003 seconds (meaning the dynamics solver performs about 333 steps of computation for each simulation second). This step size, alongside with a set of error correcting parameters was decided upon through a process of trial and error up to a point where this lead to *mostly* stable dynamics at acceptable performance. In rare cases undesired behavior is still be exhibited, such as robots breaking apart or moving solely because of error correcting forces. The effect of these errors has been tried to be kept at a minimum by detecting robots with unrealistic speeds and assigning them zero fitness.

For all these detailed parameters, the reader is referred to the experiment source code at `https://www.github.com/ElteHupkes/tol-revolve`.

5.6 Data Gathering

To analyze the outcome of the experiments, a variety of data is captured during their execution. First of all, each robot's genome and ancestry are stored when it is born, as well as its time and place of birth in case of on-line evolution. During a robot's lifetime, its position is tracked 5 times each simulation second. The off-line experiments log the fitness of each robot once, after its evaluation. The on-line experiment records the fitness values of the active robot population every 5 simulation seconds (due to the continuous evaluation, a robot's fitness value may vary over time). Whenever a robot is killed in this scenario, the position of that event is logged alongside the simulation time at which it occurs.

6 Results

This section outlines the results of the experiments described in Chapter 5. Computational performance is addressed first in Section 6.1, after which Section 6.2 presents the results of the experiments described in Section 5.4.

6.1 Computational Performance

Random populations were simulated with sizes ranging from 5 to 40 individuals, generated using the default experimental parameters for random robots as given in table 5.2. Each experiment was repeated 20 times for accuracy. The benchmark was carried out on a computer with an Intel i7-4790K CPU running at 4Ghz, with 16Gb of DDR3 RAM memory, making use of the Ubuntu 14.04.4 operating system. No explicit other foreground processes were running during the benchmark, though other (system) software was present running in the background. Since the experiments only occupied at most 3 out of the 4 hardware (and 8 virtual) cores of the CPU and these other processes were running idle, this is not expected to affect the results, which are shown in Figure 6.1.

Because the dynamics engine calculates the interactions between elements, one would expect performance to decrease quadratically with the number of bodies to simulate. Indeed, the figure shows a quadratic regression to fit the measurements quite well. The most important figure from these results is the capacity to simulate populations of robots with acceptable speed. At a little under 30 individuals a simulation can still be performed in approximately real time.

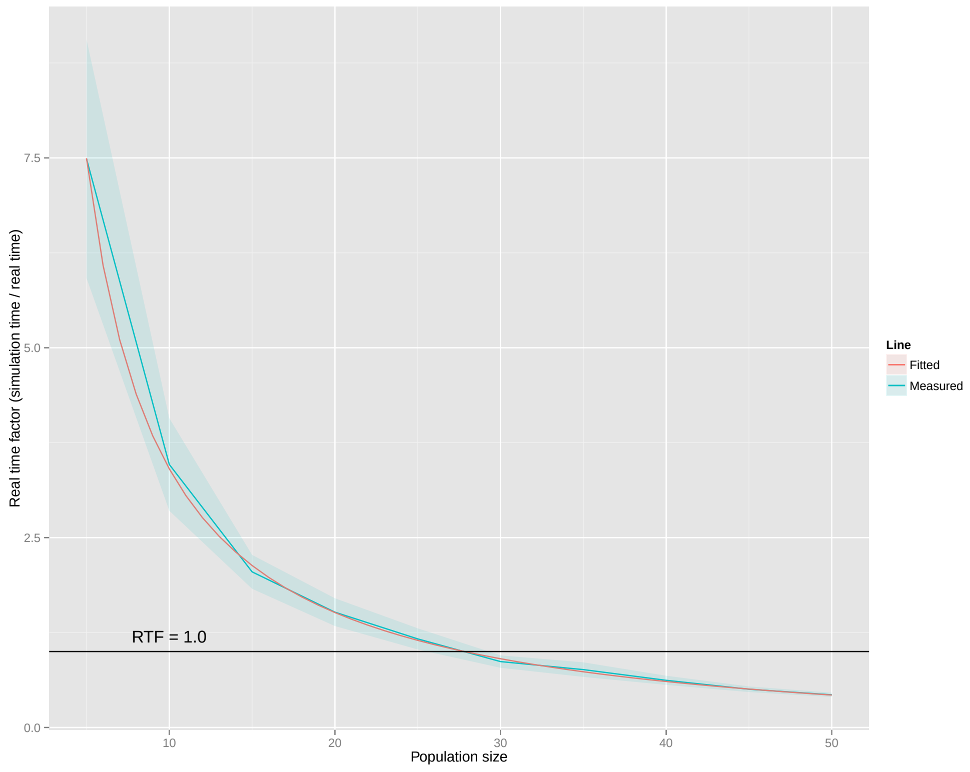


Figure 6.1: Results of the computational performance benchmark. The y-axis shows the real time factor (RTF), i.e. the amount of simulated seconds per second of real, wall clock time. The shaded area shows the standard deviation of the measurements. A fitted line is plotted corresponding to the quadratic regression $RTF = -0.26 + 34.78p^{-1} + 20.41p^{-2}$ where p is the population size. A horizontal line shows where the simulation becomes slower than real time, which in this case happens at around 28 individuals.

6.2 Evolution

This section examines the results of the experiments described in Section 5.4. The plots differentiate between five different experiments, two being the baseline experiments without reproduction or without selection, the rest referring to the three evolution scenarios.

6.2.1 Fitness

The first explored metric is the fitness of each scenario's final population, which is defined as the last generation of robots in the off-line experiments and all alive, mature robots in the on-line experiment. These results are shown in Figure 6.2.

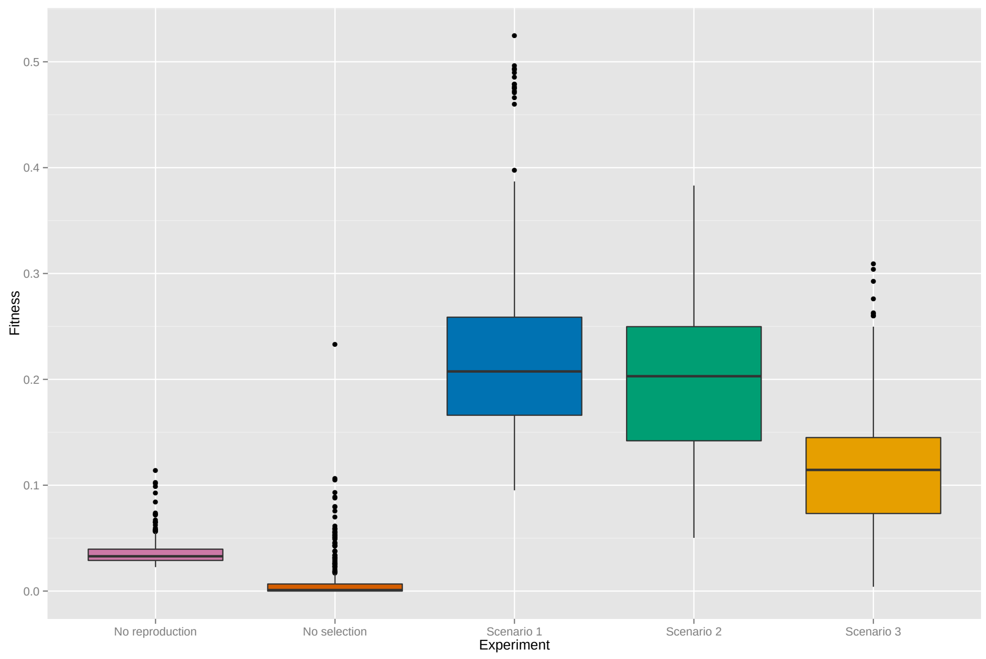


Figure 6.2: Fitness of the final population.

The baseline experiments depict the expected result. Disabling selection leads to a population of individuals with essentially zero fitness, showing that reproduction itself does not lead to higher fitness values. Selecting from random individuals does increase fitness, but only marginally. The three experiments with both selection and reproduction show a significantly higher fitness than the baseline experiments, which is relevant as it shows that a nontrivial increase in fitness can not simply be obtained as an artifact of the reproduction or selection process alone. In the remainder of the analysis the baseline experiments shall only be discussed where a baseline comparison is appropriate.

Between off-line scenarios 1 and 2 only a slight difference in fitness is observed, with a more noticeable difference in fitness variation. Scenario 3 shows a significantly lower fitness. Scenario's 2 and 3 perform parent selection without replacing the entire population, which might result in less gene variation. A lack of variation could lead to a local optimum in fitness values with a failure to improve in the long term. Figure 6.3 which shows the development of fitness in the different experiments over time, appears to suggest this effect, but no data is available to analyze these progressions past 3000 births. The fitness increase with each newly born robot declines more rapidly in scenarios 2 and 3 than it does in scenario 1, although the difference is only profound in scenario 3. However, Figure 6.4 indicates that the number of ancestors of which genes are present in the final generation is approximately equal to (scenario 2) or larger than (scenario 3) the amount in scenario 1. This suggests a larger variation, contradicting the hypothesis.

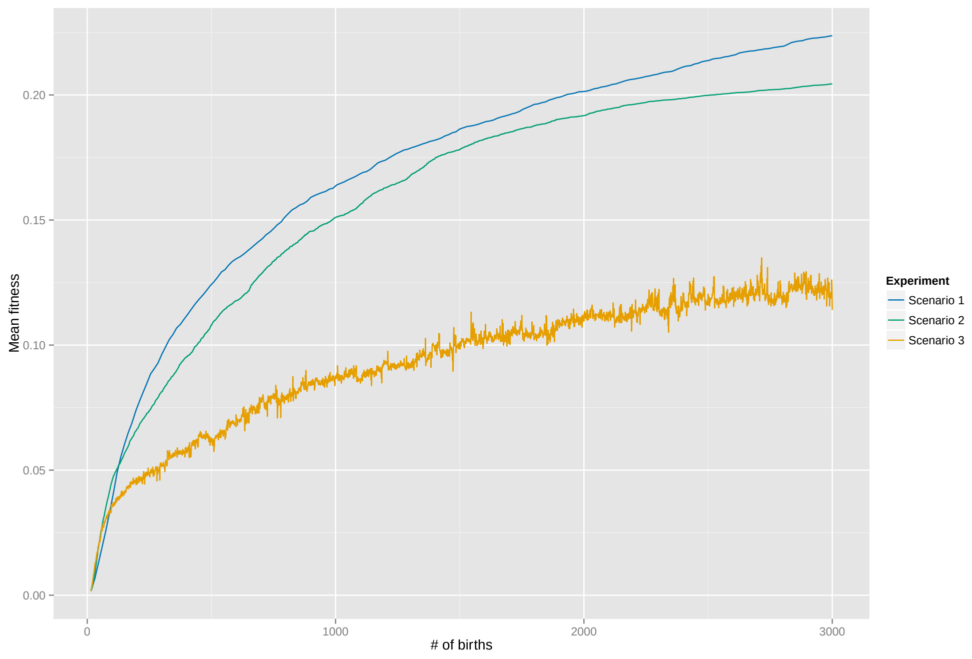


Figure 6.3: Fitness progressions averaged over all 30 runs. The number of born individuals is used as a time scale because it is uniform across all scenarios. Error bars are omitted for clarity. Note that the experimental setups guarantee a monotonically increasing function for the off-line experiments, whereas the non-constant fitness values in the on-line experiment lead to fluctuations.

The number of ancestors does not directly reflect genetic diversity however, because ancestors may be related and similar. To provide a more adequate picture, a heuristic measure is applied to quantify the genetic diversity within robot populations at each time point. This measure applies a Tree Edit Distance (TED) algorithm as described by Zhang and Sasha [44] to the genetic trees of pairs of robots that are part of the same population. The algorithm is applied with the following cost rules:

- Removing a node, adding a node or changing a node to a different type has a cost of 1.
- Attaching a node to a different parent slot has a cost of 1.

Note that this measure ignores differences in neural network contributions between nodes, as they are harder to quantify and likely not of much importance (as will be discussed further ahead). The outcome of the algorithm is included for both the final populations (Figure 6.5b) and as a progression during the experiments (Figure 6.5a). This shows an initial rapid decline of diversity in all scenarios, possibly as a result of 'bad genes' being eliminated. Diversity decline then slows down, although it decreases faster in scenario 2 than in scenario 1. This makes sense as the populations that scenario 2 uses for reproduction are very similar for each birth, which is expected to decrease variation. The same can be said about scenario 3, but the same effect cannot

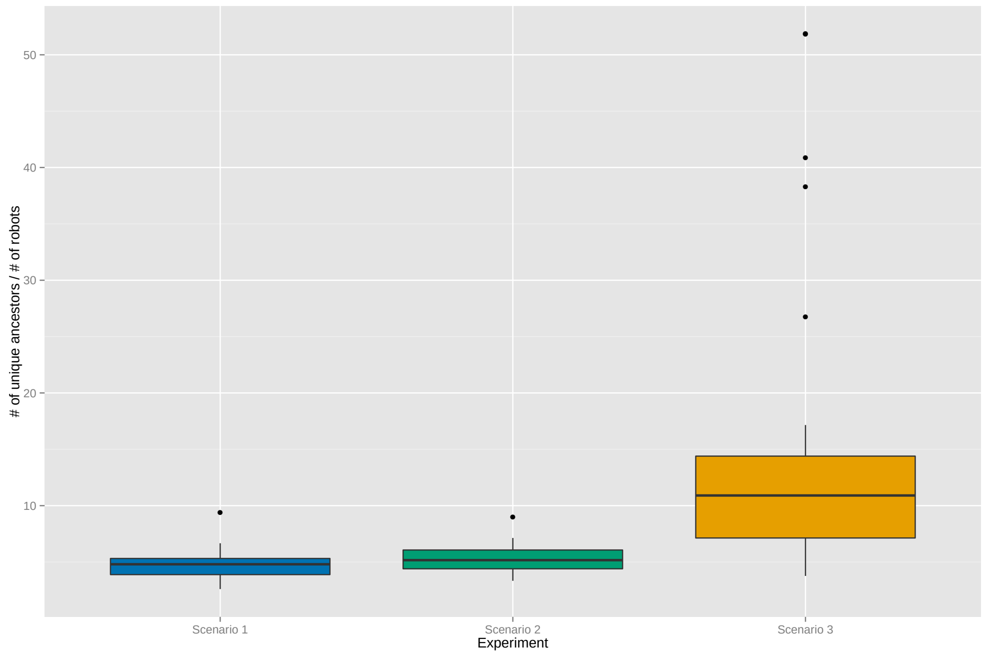


Figure 6.4: The total number of unique ancestors of the final population, divided by the number of robots in that generation, averaged over all runs.

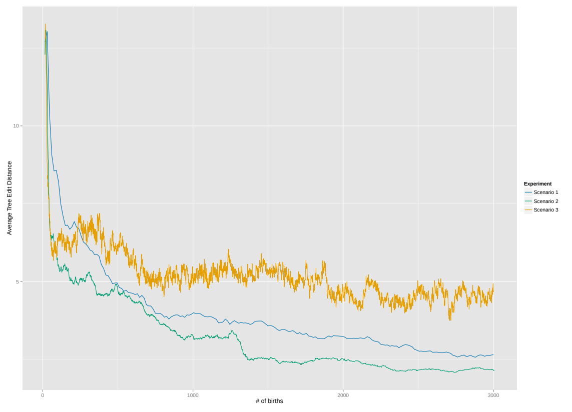
be observed there, meaning something is keeping diversity relatively high here.

To further analyze possible causes for the differences in fitness and diversity, two other metrics are examined. The first is the *rank* $\rho(r)$ of each robot r in the final population, which is defined as,

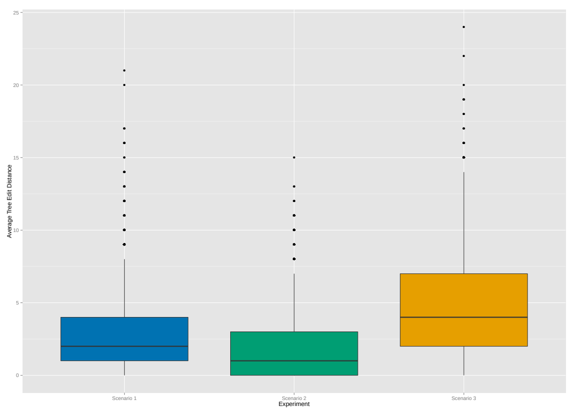
$$\rho(r) = \begin{cases} 1, & \text{if } r \text{ has no parents} \\ \max(\rho(p_1), \rho(p_2)) + 1 & \text{for robot parents } p_1, p_2 \end{cases}, \tag{6.1}$$

i.e. the starting population has a rank of 1, and a child robot always has a higher rank than both of its parents. These values are shown in Figure 6.6. The second metric aims to quantify the rate at which robots make it through survivor selection, by plotting the fraction of robots still present in the current population after n births have taken place over the number of births, which serves as a uniform time scale across all scenarios. This result, here dubbed the *retention rate*, is shown in Figure 6.7.

The average rank of robots in scenarios 2 and 3 is clearly larger than that of scenario 1, though again the difference is far more pronounced in scenario 3. The ‘family tree’ for these robots is longer, which matches the previous observation that the final populations have a larger number of ancestors. The effect is again observed in the retention rate, the lines of which follow the same (mirrored) pattern as the fitness progression lines in Figure 6.3, albeit less smooth. A low population retention can occur either if (a) newly born robots have a high probability of being fitter than existing robots or (b) fitness values fluctuate, which can only happen in the case of an



(a) Diversity progression over number of births. Error bars are omitted for clarity.



(b) Diversity of the final populations.

Figure 6.5: Genetic diversity in robot populations using Tree Edit Distance averaged over all runs.

on-line experiment, where fitness is variable. The decrease of the retention rate slope over time can be explained by (a), as an increasing population fitness sees a decline in the odds of producing a fitter individual with each birth. To confirm whether the large difference observed in scenario 3 can be explained by (b), the variation in single robot fitness is quantified in Figure 6.8, which indeed shows this variation to be substantial. The relative fitness of a single robot within the population can thus vary wildly over time, explaining the previous differences. It should be noted that it might very well be the case that the robots produced in the on-line scenario are more qualitatively desirable despite having lower quantitative fitness values. The reason for this is that the off-line robots are only guaranteed to perform according to their fitness over the duration of a single evaluation, whereas the continuous evaluation in scenario 3 encourages this performance consistently. In that sense, on-line evolution is less able to exploit artifacts of the set experimental boundaries, in this case the evaluation window. Single robot fitness fluctuations could also be caused by robot interactions, as robots might see their fitness diminish as a result of e.g. moving against each other. No data is available at this point to investigate the extent of that effect.

6.2.2 Robot Characteristics

Apart from fitness, robots have been analyzed for other phenotypic properties which are included in this section. The first characteristic is robot size, measured in number of body parts, which is shown in Figure 6.9. While no big difference in part count is observed between the three evolution scenarios, an interesting observation can be made about the size of the robots in the final generation in general. Their number of parts is significantly larger than the expected number of parts of a robot in the initial population, which is 12. Growing genotypes are a known phenomenon in Ge-

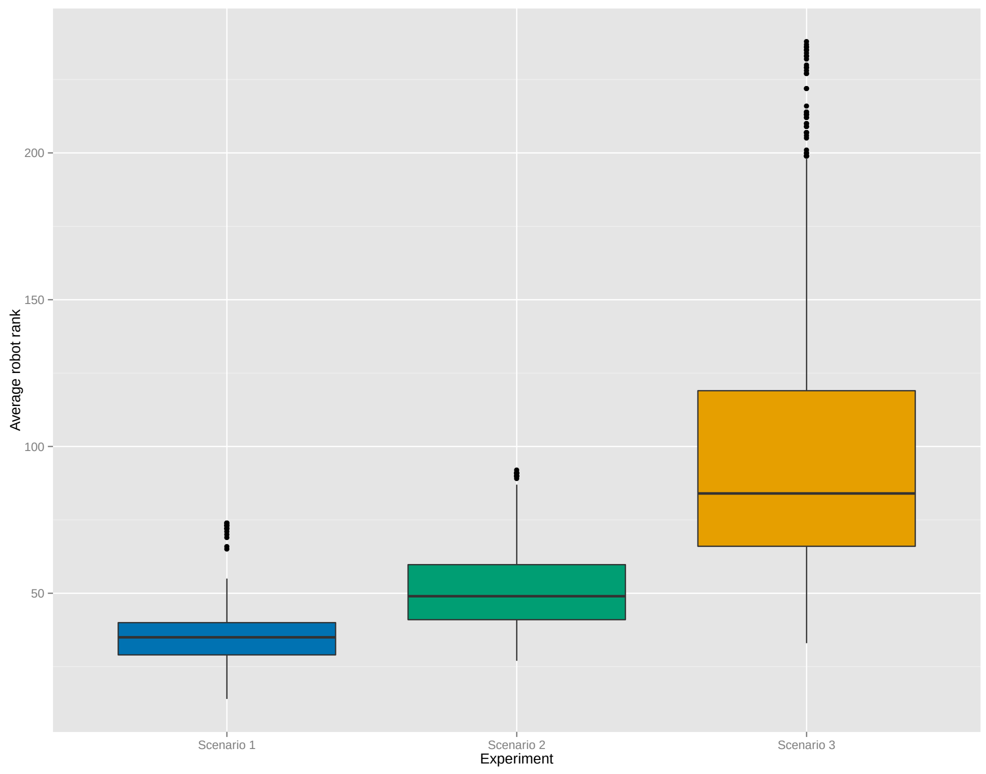


Figure 6.6: The average rank (as defined by equation 6.1) of robots in the final population.

netic Programming, referred to as *bloat* or *survival of the fattest*, the cause of which is inconclusive [19]. However, a comparison with the baseline experiments shows that the evolutionary process itself is not responsible for the phenomenon, as the growth of individuals only starts to occur once selection is enabled. It would therefore appear that the size of an individual correlates positively with its fitness. A possible explanation for this phenomenon can be found in Figure 6.10, which shows a robot from the final population of scenario 1. This robot moves by using an actuated joint in the center of the structure to push off from its left and right sides, where the distance between these two sides determines the length of the resulting 'step'. Depending on the most likely speed of the actuator it might only be capable of taking one such step during the evaluation time, which means larger robots have an advantage.

The next analyzed property is the number of joints a single robot possesses. Joints are differentiated between active joints, powered by a servo motor, and passive joints which move freely within their range of motion. Figure 6.11 shows the average joint counts within the final populations, with again no big difference between the three scenarios. A comparison with the baseline experiments seems to suggest that a larger number of joints results in higher robot fitness. However, as previously noted fitness correlates also with robot size, so it is perhaps more likely that the joint count correlates with the total size of the robot. This suspicion is confirmed by Figure 6.12, showing the

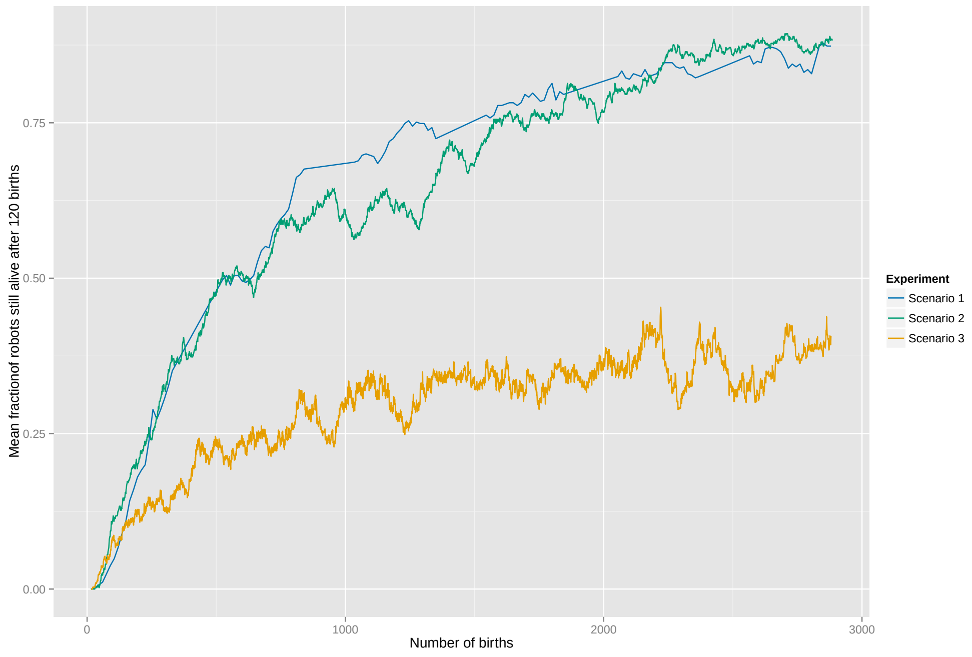


Figure 6.7: Retention rate: The fraction of robots from the active population that is still alive after $n = 120$ births plotted over the number of births for each scenario. The fractions are averages over all runs, error bars are omitted for clarity.

effect is negated by correcting for body size.

Another characteristic that has been examined is the number of *extremities* a robot has, which is of particular interest because d'Angelo et al. previously observed a relation between this number and a robot's fitness [13], albeit in a different context. In this work a correlation of a robot's speed with an increase in the number of extremities for robots with 2, 3 or 4 extremities was observed, although this speed also correlated similarly with robot size. No further effect was observed for larger numbers. Figure 6.13a shows no profound difference between the number of extremities in the three evolution scenarios. Again the larger number of extremities compared to the baseline experiments can be explained by the size of the robots rather than their benefit to fitness (Figure 6.13b). When ignoring the experiment labels a similar effect is observed as by d'Angelo et al., which can be seen in Figure 6.14a. Correcting for robot size paints a slightly different picture, as a larger number of extremities per body part actually appears detrimental to robot fitness (Figure 6.14b) beyond having almost no extremities. In line with previous conclusions it would appear that a larger number of extremities is advantageous only if they are large.

Finally some characteristics concerning the robot's sensing and brain are considered, namely the number of inputs and hidden neurons. Because the core component of each robot contains six inputs values (three dimensional values for both the accelerometer and the gyroscope), variation in number of inputs can only be achieved by adding

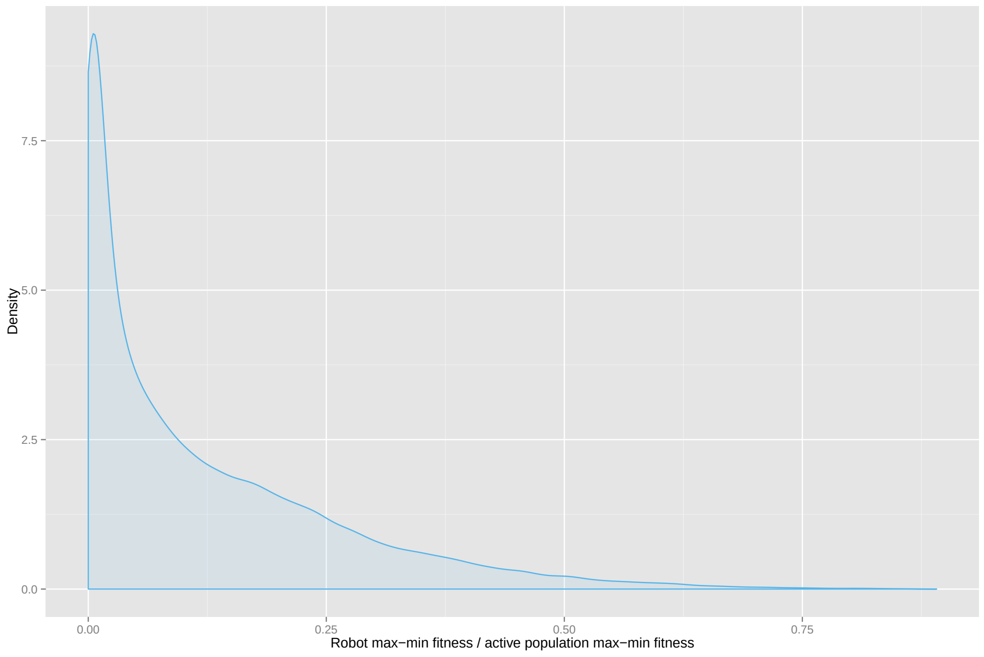


Figure 6.8: Probability density of the difference between the minimum and maximum recorded fitness of a robot as a fraction of the minimum and maximum fitness of the active populations it has been a part of, for robots in scenario 3. For instance, if a robot's minimum and maximum recorded fitness values are 0.2 and 0.3, and it was active in populations with fitness values varying between 0.1 and 0.5, it will register as a $\frac{0.3-0.2}{0.5-0.1} = 0.25$ data point.

or removing touch sensors. Furthermore the total number of input values has been limited to 10, meaning a robot can only have either 0, 1 or 2 touch sensors (each touch sensor provides 2 input values). Barring some clear evolutionary advantage to having these sensors their effect is thus expected to be limited. Figure 6.15 shows most robots in the final populations to have no touch sensors at all, although all possible numbers are present in all populations. The random populations are an exception to this, which is to be expected because body parts are added randomly to these individuals with equal probability and there is no cost to having a touch sensor. The 'no selection' baseline hints towards the reproduction process being biased towards removing touch sensors. This is unsurprising, because the limit on number of inputs favors tree crossover and duplication for genotype subtrees without sensors. The difference in variation in number of touch sensors might therefore arise simply as a result of robot rank (Figure 6.6). Whether or not sensors have any influence on robot fitness remains inconclusive.

The number of hidden neurons is another property that has been limited to a relatively low value (10) and is therefore expected to be biased down following an argument similar to the one in the previous paragraph, because hidden neurons duplicate with

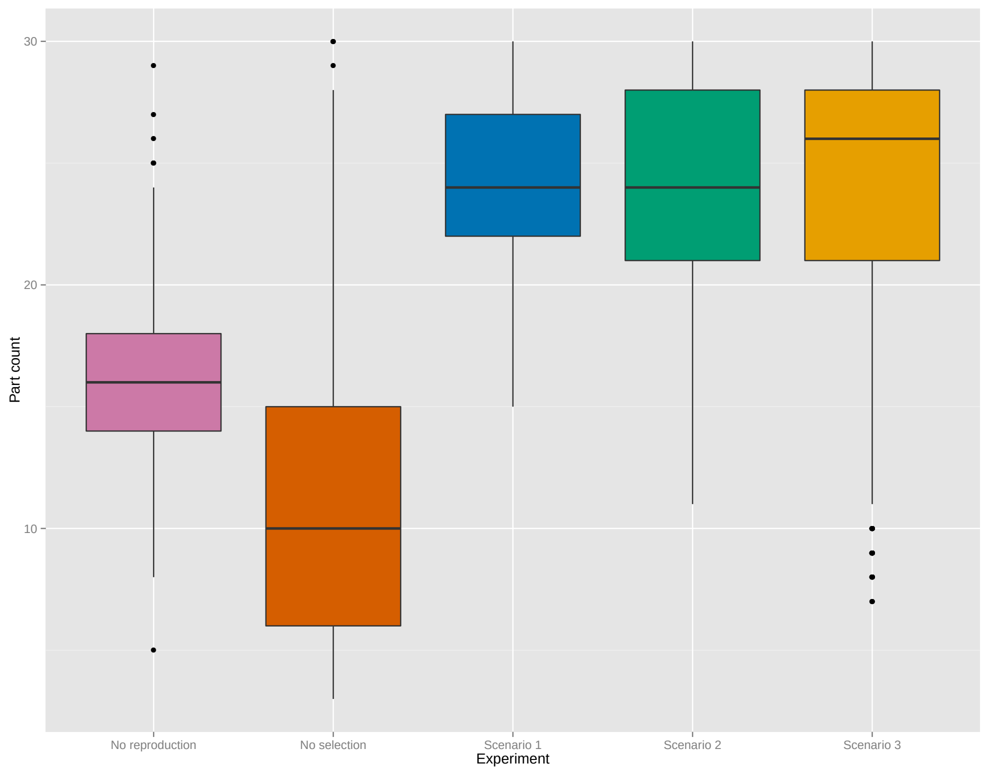


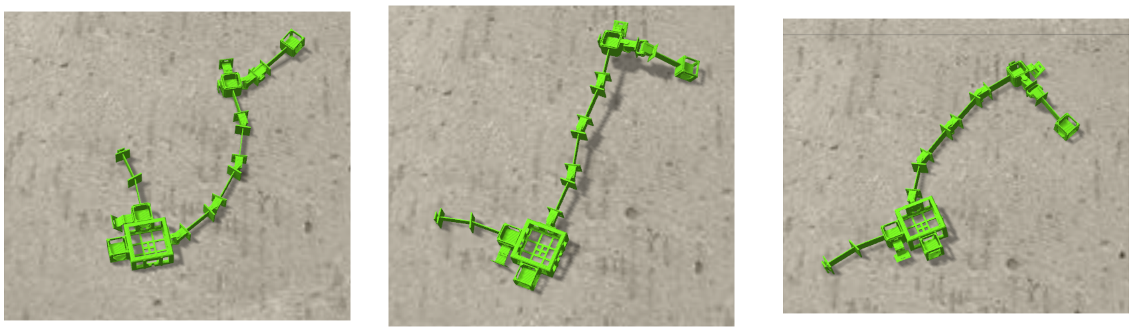
Figure 6.9: The average number of parts for a single robot in the final population of each experiment.

the tree nodes they are assigned to. This bias is confirmed by Figure 6.16 and does not appear to be countered by evolution. In fact the number of hidden neurons is distributed perfectly around the values following from initial generation in the random population, leading to the conclusion that in the current setup hidden neuron presence does not influence fitness.

6.2.3 Robot Morphologies

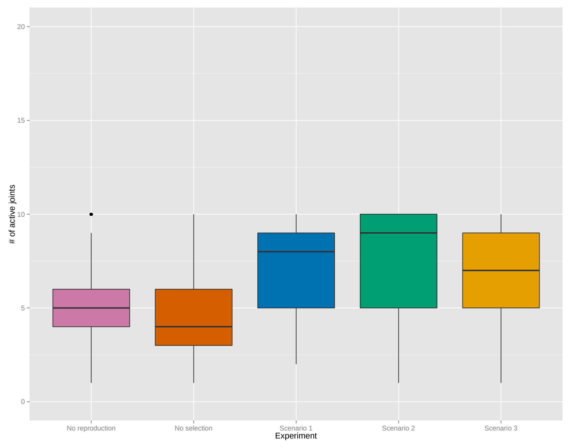
Apart from quantitative characteristics it is interesting and important to observe the evolved organisms qualitatively. This section therefore displays the fittest individuals taken from across all runs for each of the three scenarios, as well as some of their ancestors. The displayed ancestry trees only span a couple of levels, because showing the full ancestry trees would obscure the details of the images. A full ancestry tree without images is shown in Figure 6.17 as an example.

Figures 6.18, 6.19 and 6.20 show partial ancestry trees of a robot from the final populations of scenario 1, 2 and 3 respectively. A few similarities are observed between the portrayed individuals and ancestry trees. As was previously suggested by the data, the robots have large appendages. Scenario's 1 and 3 are very similar in that they use

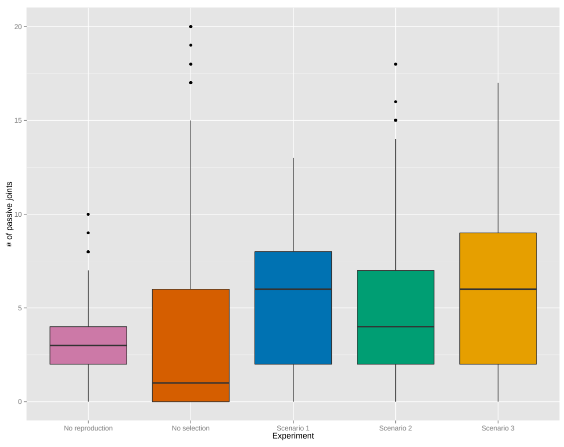


(a) (b) (c)

Figure 6.10: A robot from one of the final populations of scenario 1 in three consecutive stages of its gait.



(a) Active joints (motors).



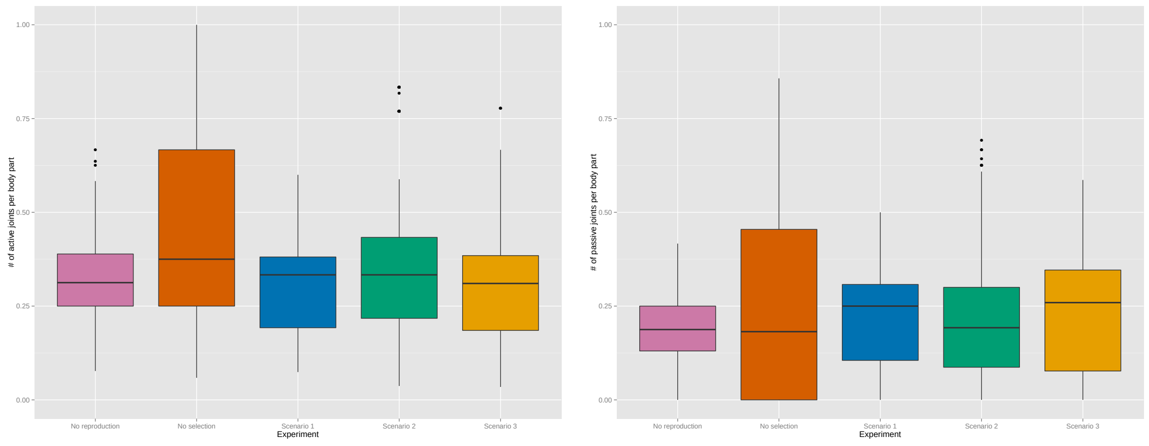
(b) Passive ('free') joints.

Figure 6.11: The average number of joints for a single robot in the final population of each experiment, differentiated between active and passive joints.

a long 'tail' for movement. It should be noted that, because of structural stresses, the ability to physically build this kind of structure is questionable, something which is discussed further in Section 7.2.2.

At their conception, body parts are assigned a random color which does not undergo mutation throughout the evolutionary process, which makes it possible to visually track which parent a body part originates from. This property reveals that the displayed individuals only have 'genes' from a few of their original parents, even if they descend from a larger number. The scenario 2 individual is even mostly a recombination and duplication of parts from one single ancestor. The individuals from scenarios 1 and 3 appear to be a uniform mix of body parts from several parents.

The partial ancestry trees show the last 5 reproduction steps resulting in the final robot. In each of the scenarios all individuals involved in these final steps are very



(a) Active joints (motors). (b) Passive ('free') joints.

Figure 6.12: The average number of joints per body part for a single robot in the final population of each experiment, differentiated between active and passive joints.

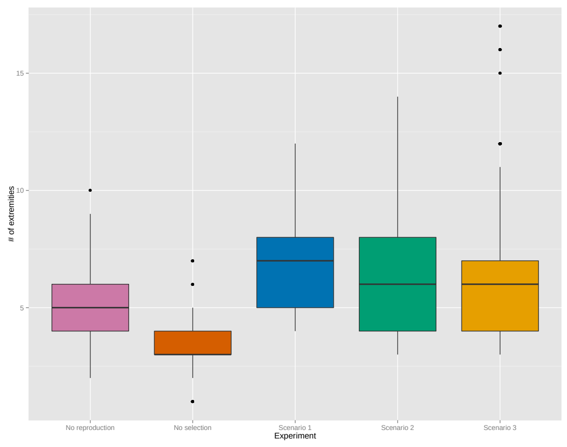
similar, a result which is in line with the low diversity observed in the previous section.

6.2.4 In Summary

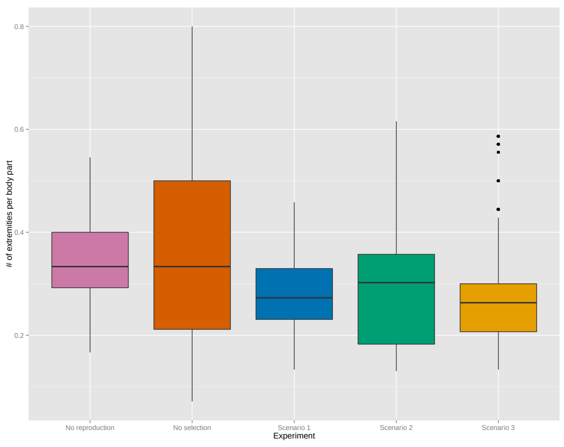
A number of metrics were investigated in this section to compare the various simulation scenarios. Table 6.2 provides an overview of these metrics. In addition, some robot morphologies and ancestry trees were examined in section 6.2.3. Robot fitness is found to differ significantly between the off-line and on-line scenarios, which can at least partially be explained by the increased diversity present in the on-line experiment, causing slower convergence. This increased diversity is likely a result of fitness variability in the on-line population, influencing rank and retention rate. Other metrics show no clear difference between the scenario types that cannot be explained by other variables such as fitness, robot size or rank. Disregarding the scenario type, the phenotype space appears to prefer large robots, in particular robots with large extremities, possibly because they can take large 'steps' within the evaluation time. This hypothesis is supported by the morphology examinations, which show some robots with large extremities. The examined ancestry trees of these individuals furthermore shows the ancestors in the final generations leading up to these robots to be very similar.

Metric	**Description**
Fitness	The fitness of a robot, as given by Equation 5.1. Figures 6.2 and 6.3 show final population fitness and fitness progression over time respectively.
Ancestors	The number of unqiue ancestors of a single robot, given averaged over population size in Figure 6.4.
Diversity	Genetic diversity measured in Tree Edit Distance, shown in figure 6.5.
Rank	The rank ρ (Equation 6.1) of a robot is defined by the length of its 'bloodline', i.e. the maximum number of steps to the root of its ancestry tree. Results are shown in Figure 6.6.
Retention rate	The fraction of robots that is still alive after a certain number of births n. Shown for $n = 120$ in Figure 6.7.
Fitness variability	The expected variability in fitness value for a single robot in an on-line population. Shown in Figure 6.8.
Number of parts	A robot's average number of components, i.e. body parts, given in Figure 6.9.
Number of joints	The average number of movable parts, i.e. joints, for a single robot, differentiated between joints with and without motors. Given as a total count (Figure 6.11) and relative to a robot's number of parts (Figure 6.12).
Number of extremities	The number of 'leaf' bodyparts in a robot, shown both absolute numbers and relative to robot size in Figure 6.13a. In addition, this number is plotted against robot fitness in figure 6.13.
Number of touch sensors	The average number of touch sensors in a robot, Figure 6.15.
Number of hidden neurons	The number of hidden neurons in a robot's brain, given relative to robot size in figure 6.16.

Table 6.2: Metrics investigated in section 6.2.

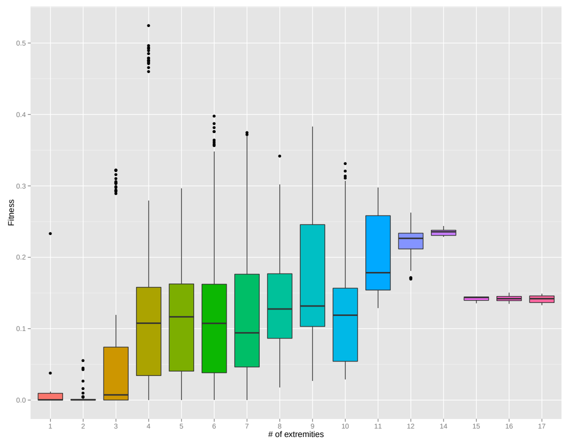


(a) Absolute numbers.

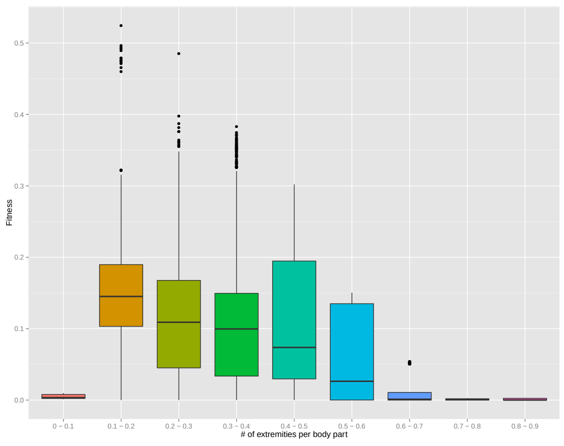


(b) Relative to robot size.

Figure 6.13: The average number of extremities for a single robot in the final population of each experiment, both in absolute numbers and relative to robot size. An extremity is defined as a chain of one or more body parts that has exactly one leaf node in the body tree.



(a) Absolute numbers.



(b) Relative to robot size.

Figure 6.14: Robot fitness with respect to to the average number of extremities, regardless of experiment.

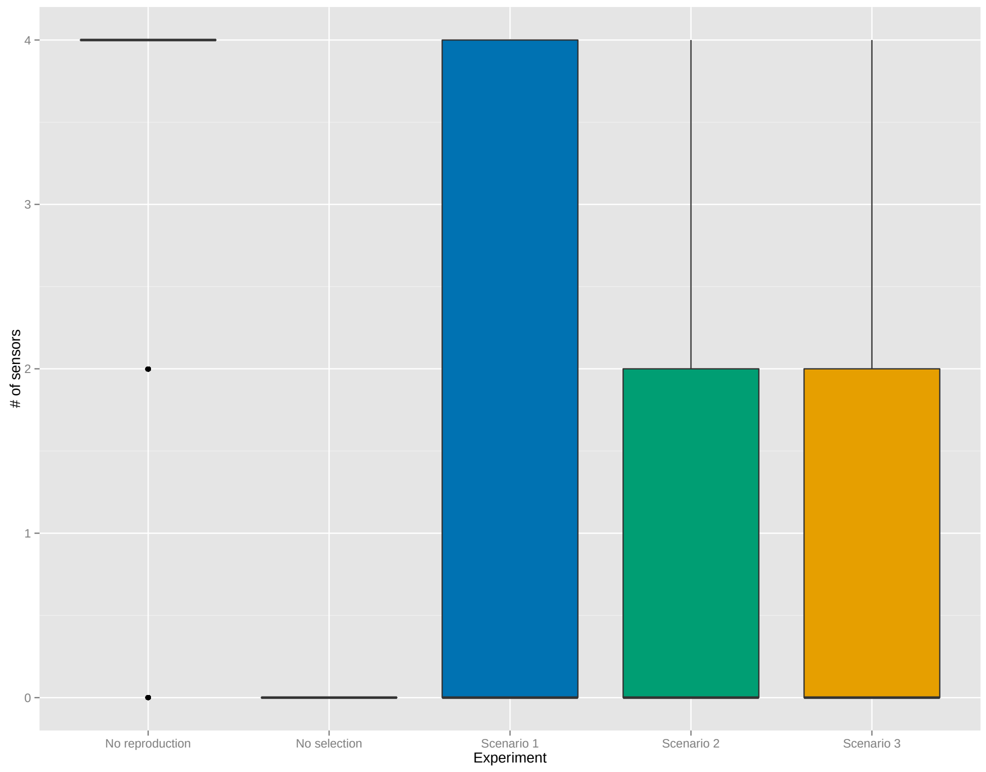


Figure 6.15: The average number of touch sensors for a single robot in the final population of each experiment.

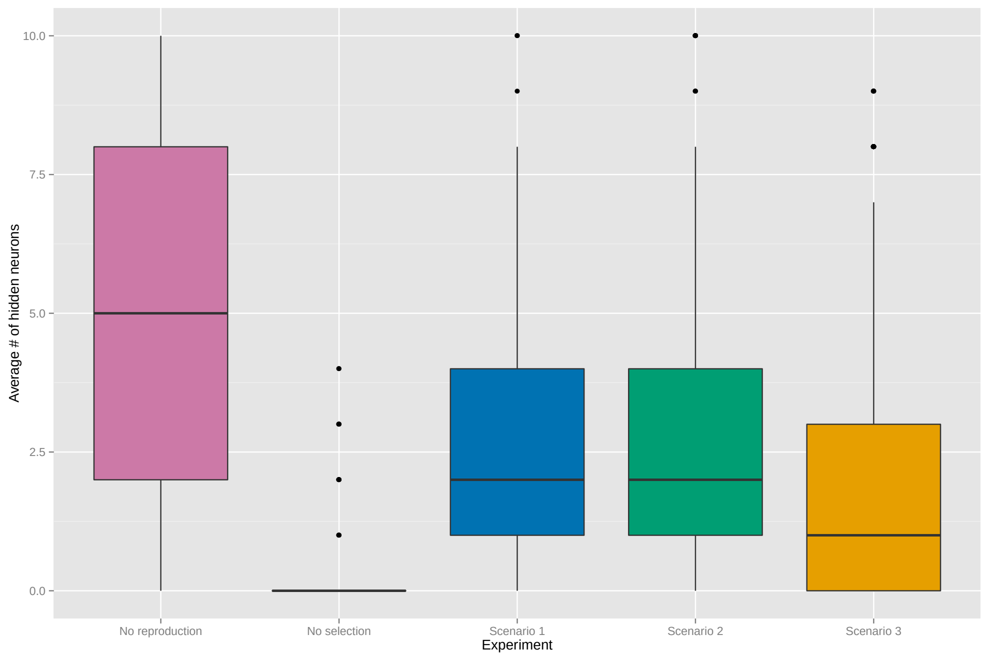


Figure 6.16: The average number of hidden neurons per body part in the brain of a single robot in the final population of each experiment.

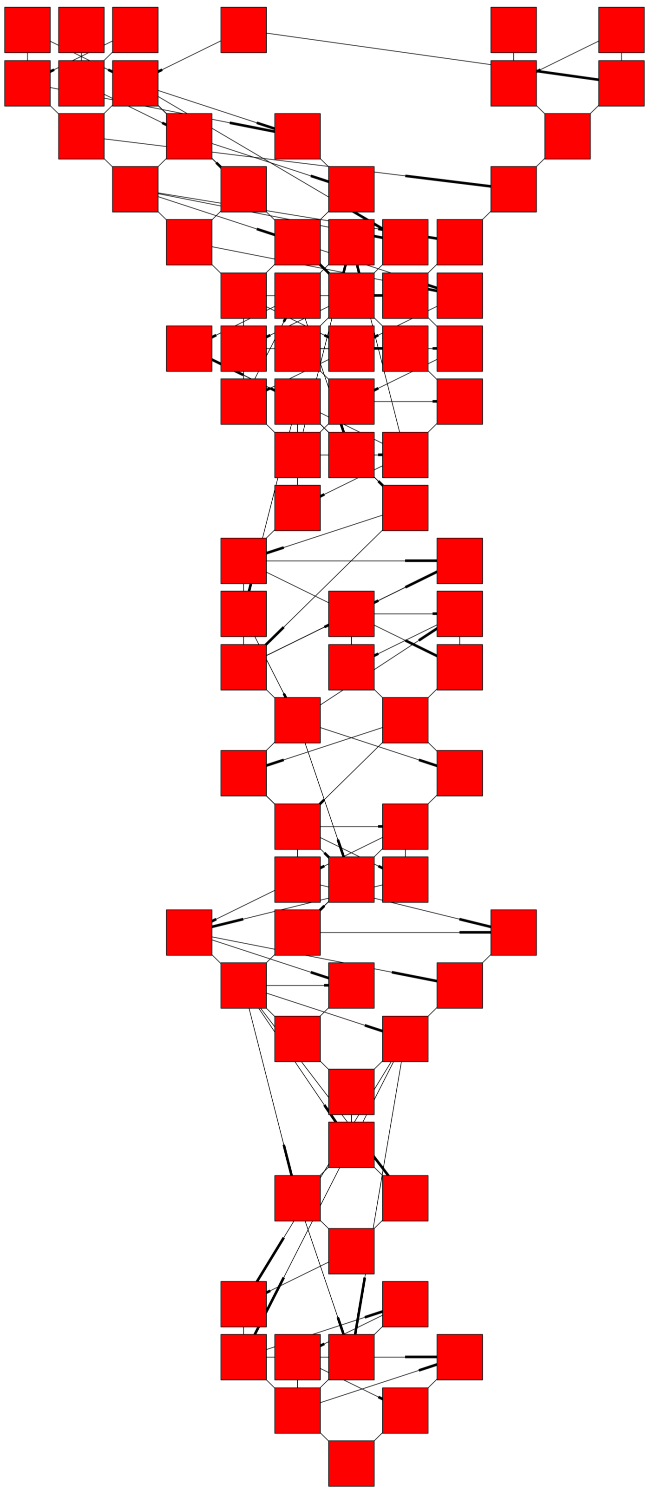


Figure 6.17: Complete ancestry tree of a single individual in the final population of scenario 1. The tree is displayed in ascending order, with the individual of interest at the bottom.

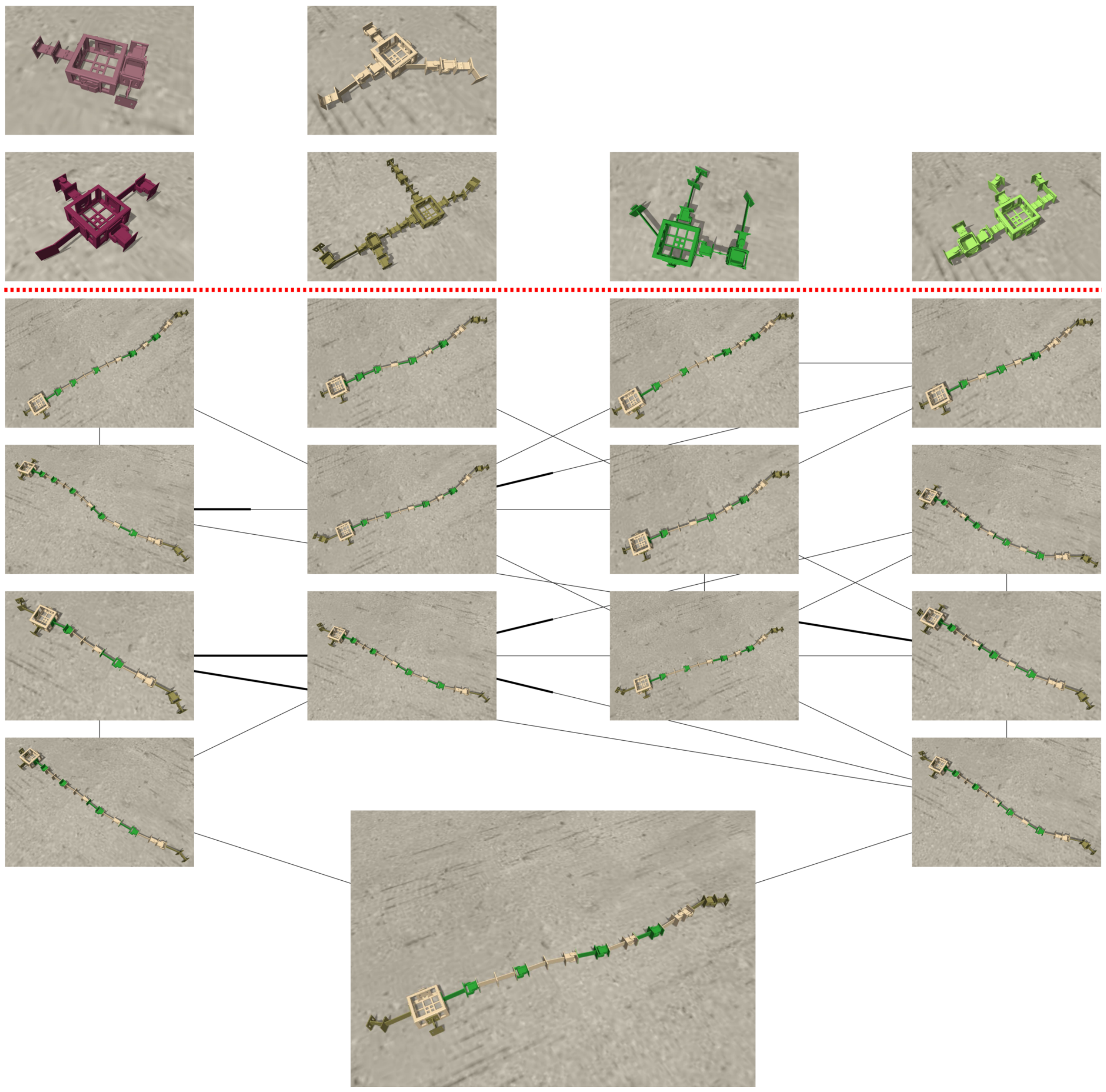


Figure 6.18: Partial ancestry tree for one individual from the final population of scenario 2, shown at the bottom. The individuals from the initial generation that were part of this individual's ancestry are shown above the dotted line.

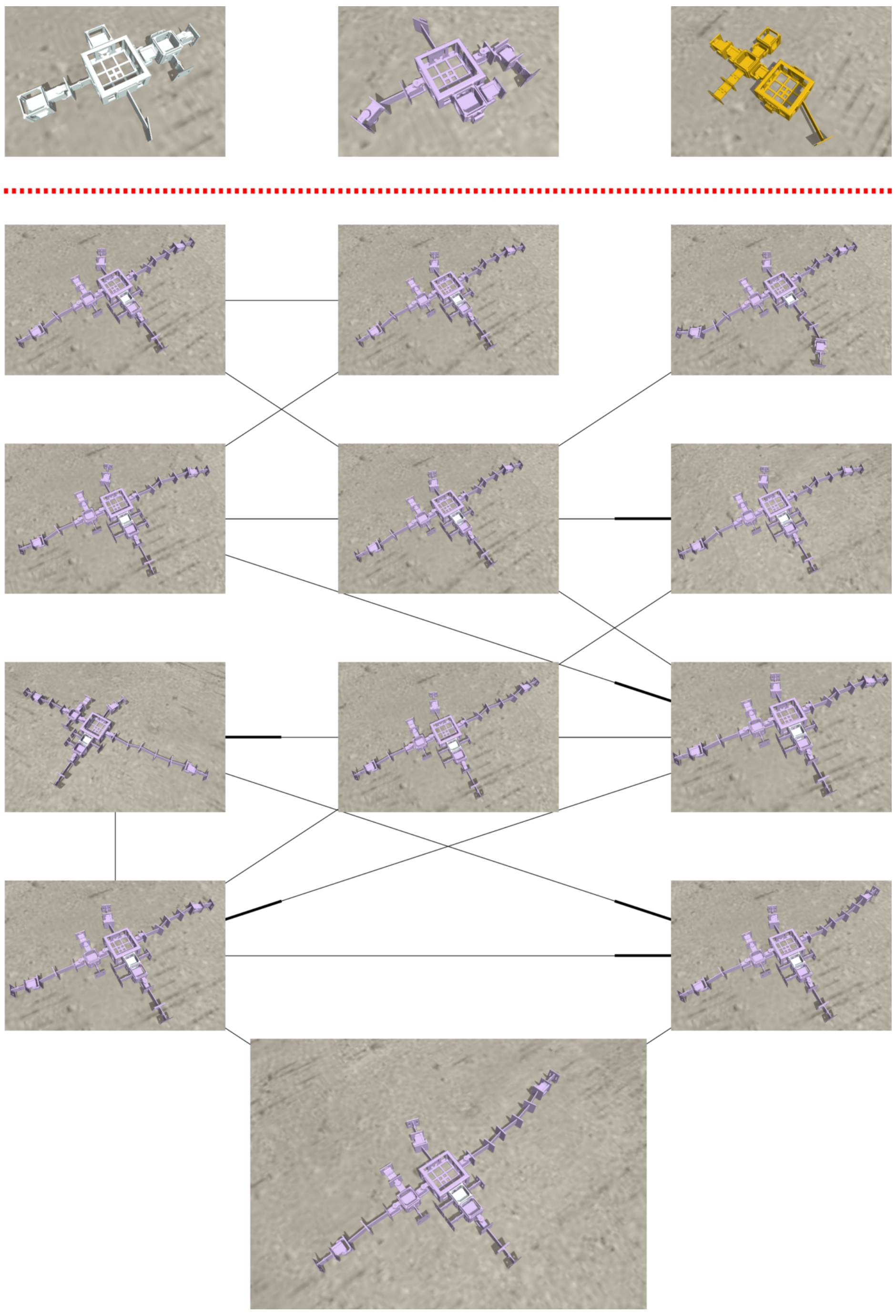


Figure 6.19: Partial ancestry tree for one individual from the final population of scenario 2, shown at the bottom. The individuals from the initial generation that were part of this individual's ancestry are shown above the dotted line.

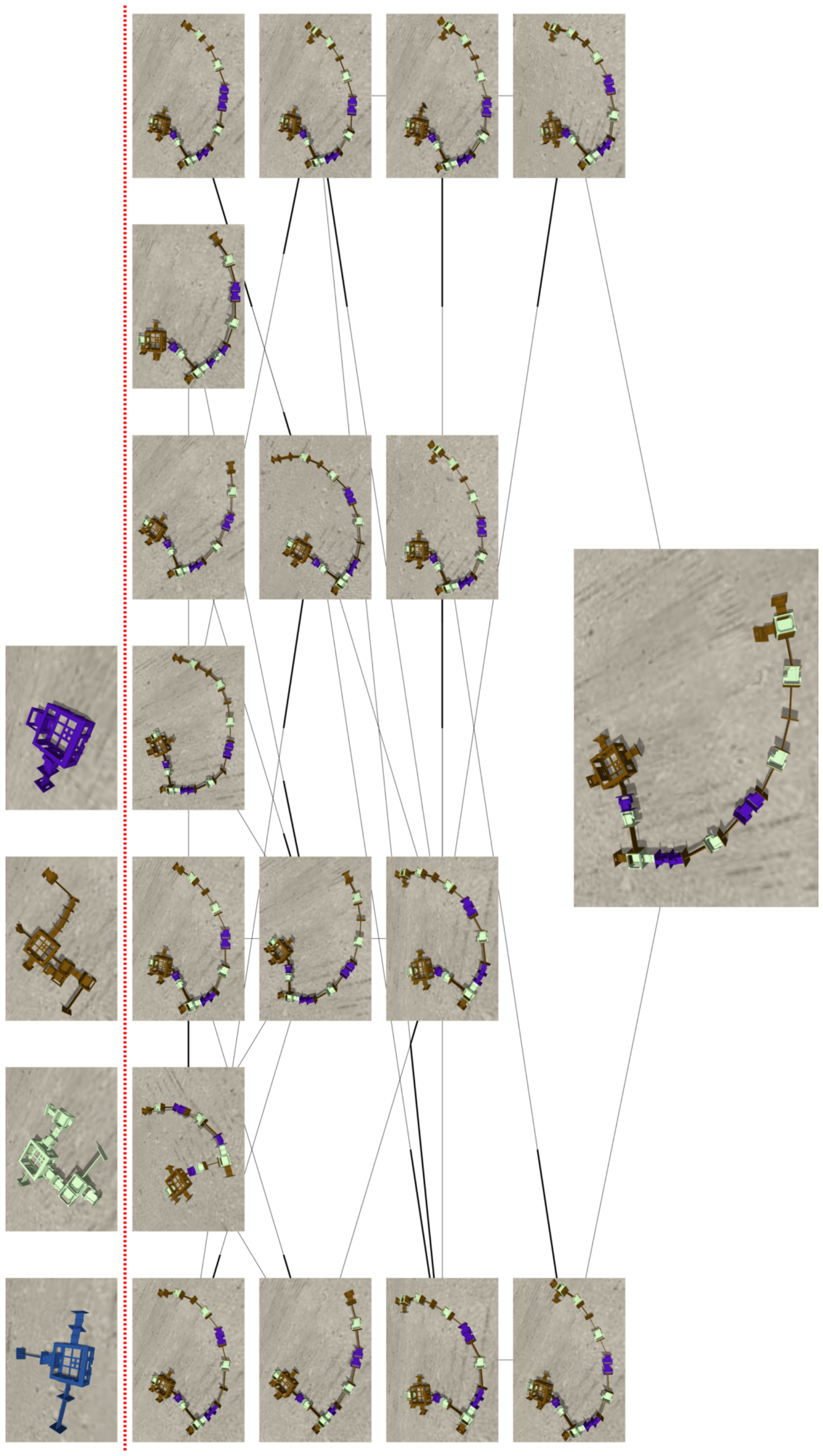


Figure 6.20: Partial ancestry tree for one individual from the final population of scenario 3, shown at the bottom. The individuals from the initial generation that were part of this individual's ancestry are shown above the dotted line. The image is rotated to fit the page.

7 Conclusions and Discussion

When developing software for any purpose a trade-off always has to be made between time in development and time in execution. The middle ground is essentially an infinite plane, making it hard to judge any end result by metrics other than how successful its use has proven to be in retrospect. This is predominantly a qualitative matter. For a research tool such as Revolve, this means its quality can really only be assessed once it has been employed in research projects which are not the product of the original author. Nevertheless the purpose of this chapter is to assess Revolve by the goals set out in Section 1.1. This is done rather qualitatively in Section 7.1, which judges the utility of Revolve in Evolutionary Robotics research. The next section, 7.2, discusses the results of the experiments performed in this work. Section 7.3 concludes the chapter with a motivation of the design and implementation of Revolve.

7.1 Utility

7.1.1 Experimental Design

In the process of determining the final experiments to be used in this thesis and during the development of Revolve, many experimental scenarios were implemented. It was found to be easy to setup an entirely new scenario or drastically change the experimental parameters, which was one of the key goals of Revolve. Again the caveat here is that some knowledge of the system is required for anyone other than the author, and documentation is currently lacking (more on that in Section 8.1). However, during its development, a team of students following the Computational Intelligence course at the University of Amsterdam used an early version of Revolve to implement evolutionary learning using the NEAT [39] algorithm. The successful completion of this project (the work of which is unpublished) speaks to the potential of the toolkit.

7.1.2 Computational Performance

A more quantitative assessment can be made of the computational performance of Revolve, using the results of the benchmark found in Section 6.1. It was found that dynamics in a population of around 28 robots of average complexity could be computed in real time at sufficient stability. Whether or not this performance is acceptable (and exactly what qualifies as 'sufficient stability') remains highly dependent on the use case. For experiments with similar complexity as the ones performed and discussed in this work, it is safe to say that they can be feasibly performed on normal hardware. As Revolve is piggybacking on existing software packages which are in active development, this is only expected to improve in the future.

7.2 Experiments

7.2.1 Off-Line versus On-Line

Section 6.2 attempts to reveal the differences between off-line and on-line (embodied) evolution by running similar experiments of both types. The comparison remains tricky as the scenario types are fundamentally different, predominantly with regard to the way in which robots are evaluated. The main conclusion from this chapter is that the observed differences in robot fitness can be explained by the continuous evaluation present in on-line evolution. Effects that can be explained as part of robot interaction or the environment are either not present or too subtle to be noticed over this larger effect. It should be noted that no specific emphasis has been put on the environment of the robots in this work, nor have the robots been explicitly equipped to enable complex interactions. Other sensors and learning might have a significant impact on these effects, as discussed in Section 8.2.

Because of the more realistic conditions under which on-line robots operate, it is very well possible that their behavior is more robust. Indeed a constant fitness value for individuals, as is commonly the result of off-line experiments, is not a very realistic prospect. Steps can be taken to make off-line results more 'on-line-like' however, like repeating fitness evaluations over varying evaluation times. This would increase the time required for (simulated) evaluation - but this is even more so the case with on-line evolution, which is computationally more expensive for the same task. It is exactly the difference in task description that differentiates on-line evolution experiments, which gives rise to the question whether further evaluation of the exact differences is even useful, or the scenarios should just be investigated in their own regard.

7.2.2 The Reality Gap

Despite the conceptual nature of the described experiments, realism is an important goal in simulating on-line embodied evolutionary systems, and an aspect which is currently lacking in this work. Although the robotic lifeforms are theoretically constructable through 3D printing and some amount of manual assemblage, none of the robots resulting from the experiments have been constructed in real life at this point due to lack of time and resources. Qualitative analysis of some of the individuals already revealed some exploitation of simulation artifacts such as error correcting parameters in order to improve speed, and there are several other areas in which the reality gap might manifest itself. It is possible that some of the organisms have structural limitations that prevent them from being physically constructed that are not an issue in simulation. For instance, forces working on some of the observed 'long arms' might cause them to physically break in real life. Several other areas where steps could be taken to improve the realism of robots such as calibration of frictional parameters and sensor noise are also out of the scope of this work and part of the future recommendations outlined in Section 8.2. As a result, any conclusions from this work should be interpreted conceptually first and foremost.

7.3 Motivating the Design of Revolve

The main design decision when conceiving Revolve was a choice between either (a) building on top of a dynamics engine directly, (b) modifying the code of an existing research project or (c) using a simulation platform. Out of these (b) and (c) are more viable options because they take away a large part of the bootstrapping process, and in addition ensure improvements and fixes to the underlying infrastructure regardless of development of the toolkit. Investigations were performed with the NASA Tensegrity and RoboGen source codes, running benchmarks and trying to realize simple artificial ecosystems using the existing code base. There was a particular focus on RoboGen as an attractive candidate for a proof of concept, given that its robot body space is easily constructed using 3D printing, and is subject to an ongoing real-life calibration process. During the setup of simple scenarios however, it was found that the RoboGen software suite was too much tailored to its serial, off-line evolution to be conveniently refactored to the new use case. In addition all code would have to be written in the C++ language, which provides high performance at the expense of being verbose and some times tedious to develop. In many cases the high performance would be attained in areas where it was not at all needed and a more convenient setup would be preferred. While the choice for the RoboGen body space as a proof of concept remained, the decision was made to build the Revolve Toolkit on top of a general purpose simulation platform instead.

Out of the simulation platforms discussed in Section 3.3, only Webots was discarded beforehand because of the negative experiences addressed earlier. MORSE appeared to be a suitable candidate, but lacked the ability for high performance C++ integration that Gazebo and V-REP provided, as well as the ability to work with a choice of physics engines. An online paper exists performing a comparative analysis between the last two remaining platforms, ruling in favor of V-REP by a slight margin. However, the use case examined in this paper differs from the type of research in this thesis. In addition, a much older version (2.2) of Gazebo was used than was available even at the time this paper was written, in order to simplify ROS integration. The methodology used to compare performance is furthermore flawed in that it compares CPU usage rather than simulation work performed over time. Taking away these points, the bottom line is that V-REP and Gazebo are very similar platforms in terms of features. The eventual choice for Gazebo is motivated by its non-commercial nature, its large online community and the XML format it uses to describe models, which simplifies creating dynamic robot morphologies from external applications. That being said, V-REP would likely also have been very suitable as a platform. While Revolve has been written with Gazebo in mind, large parts are simulator agnostic and could potentially be used for creating a similar platform for use with V-REP.

For completeness one of the early performance benchmarks used to assess RoboGen was implemented in Revolve for comparison after its development. In this benchmark, a population consisting of a varying number of spider-like robots is placed on a grid and simulated for a fixed time, measuring the real time it takes to perform this simulation. Setting up such a scenario is rather straightforward in Revolve, as it was designed for these kinds of tasks. For the RoboGen code, a 'naive' approach is taken to obtain the benchmark, in which the standard robot evaluation task is modified to copy the target robot multiple times. This is still a far cry from something that can be used to produce other experiments as well, as addressed in the previous paragraph. The RoboGen benchmark was later altered to support the latest version of the software suite available at the time (which underwent some performance optimizations since the first version of the benchmark), after which both benchmarks were executed with the same system setup as was used in Section 6.1. These results are presented in figure 7.1, which shows that for this specific test RoboGen performs better for small populations, whereas performance is similar for larger population sizes. The relevance of this benchmark should not be overstated, because important aspects such as optimization, calibration and development speed are not taken into consideration. It is merely included to show that the increased development ease of the Revolve platform for these kinds of scenarios does not necessarily mean performance is adversely affected.

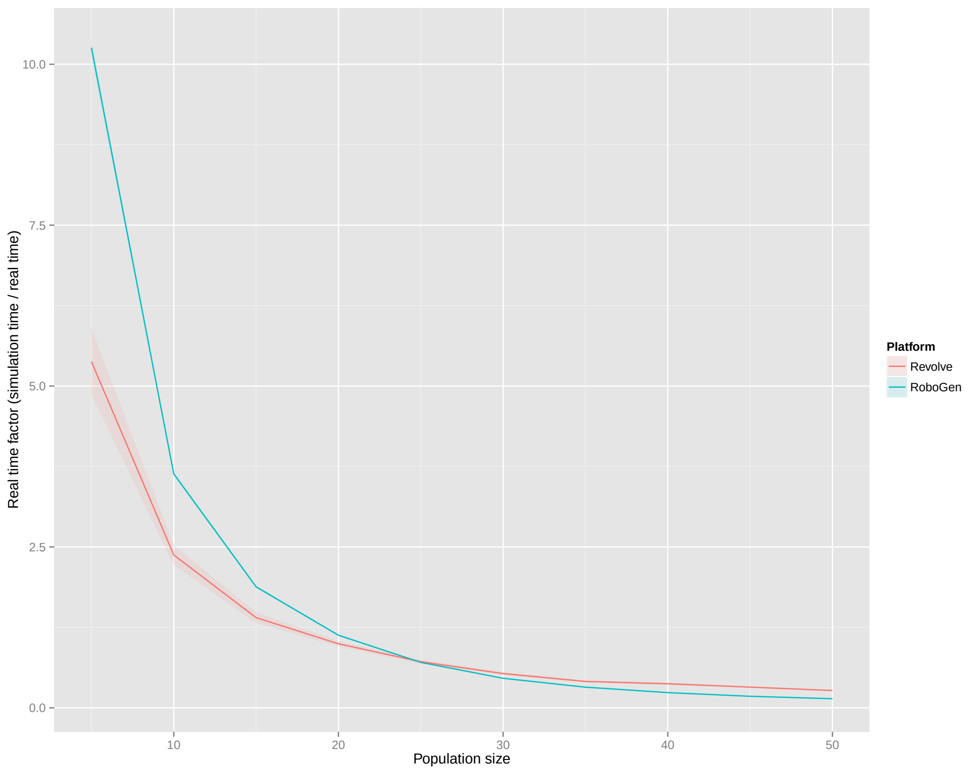


Figure 7.1: Results of the comparison benchmark described in section 7.3, in which populations consisting of spider-like robots placed on a grid were simulated in both Revolve and RoboGen. The y-axis shows the real time factor (RTF), i.e. the amount of simulated seconds per second of real, wall clock time. The shaded area shows the standard deviation of the measurements, each of which was repeated 30 times.

8 Recommendations and Future Work

This section outlines some recommendations for future research relating to the work in this thesis. It starts by describing the state and challenges to overcome with respect to the Revolve Toolkit in Section 8.1, followed by recommendations regarding actual research in Section 8.2.

8.1 State of Revolve

At the time of this writing the functionality available as part of the Revolve Toolkit should be considered a first version which, while already usable for research, leaves many desired improvements. The most pressing of these matters is the compatibility with the Gazebo. During the development of Revolve, several issues of varying severity were encountered that could only be resolved within the simulator code itself. These issues have been reported to the Gazebo developers, but in the mean time a fork of Gazebo has been created with fixes to these issues[1]. This fork is meant as a temporary solution until they are fixed in the main Gazebo software, making sure this happens is important to be able to use up-to-date versions of Gazebo with Revolve in the future. It was noted in Section 5.1 that light sensors cannot currently be used in many scenario types, which is a result of a stability issue with Gazebo when many individuals are added or removed to an environment in a short time span. Fixing this issue is of some importance as vision in general and light sensors in particular are often interesting

[1]This fork is available at `https://bitbucket.org/ElteHupkes/gazebo/branch/gazebo6-revolve`.

tools to employ in robotics research.

As for Revolve itself, the potential improvements are numerous, as is to be expected of a software package which targets a wide array of possible research. The urgency of these improvements tends to depend on the specific use case, and it is therefore hoped that they will be incorporated during future Revolve research projects. Documentation is a definite point that deserves attention. While the Revolve code contains inline documentation, the use of the toolkit is not currently described in any way that can be considered particularly accessible. Having introductory descriptions and tutorials available is conditional to making Revolve the tool it has set out to be, regardless of the functionality it provides, and therefore should be prioritized.

It should be noted that no framework or toolkit can likely ever be an end-all be-all solution to one's research needs, because a trade-off between performance, accuracy and rapid prototyping will always be a part of the scientific process. As it stands, the process of acquiring a simulation with acceptable stability and performance is very much dependent on parameter tuning of the end user. Revolve should possibly provide some guidance in this matter, be it through documentation for its most used parts or through tools for stability analysis and heuristics.

8.2 Future Research

With regard to the research performed in the context of this thesis, there are several areas in which it can be improved upon. Because the work takes place entirely in simulation, the risk of a reality gap is significant. Extending the experimental procedures to include real-life validation in some way is therefore an important next step. Acquiring a clear picture of what the possible differences with reality entail would be helpful in generalizing the results to physical robotics. Apart from mapping out these differences, a number of other steps could be taken to minimize the reality gap, such as sensor and actuator calibration, and the utilization of sensor and actuator noise to improve robustness. The effect that sensors have on robot performance has not been investigated in this work and might also be of interest, in particular in conjunction with real, physical robots where optimizing their use may be desirable. Most of the analysis in Section 6 has focused on morphology, whereas the co-evolution of brains with these morphologies is of definite interest.

When making use of the evolutionary processes described in Section 4.4.6, there is a wide range of parameters to consider. The parameter list used in this work is a result of both default values taken from RoboGen and a process of trial and error. Performing a more thorough parameter search over a set of small simulations might provide for different dynamics.

To further investigate the differences between the off-line and on-line systems found in Section 6.2, it may be worth designing and simulating a system that bridges scenarios 2 and 3 from Section 5.4. Because of the suggested significance of continuous, dynamic fitness evaluation, extending scenario 2 with variable simulation times at the same sliding window size seems logical, because it would potentially encompass different stages of a robot's gait cycle. Periodically selecting either all or a subset previously evaluated robots from the active population for re-evaluation at a different simulation time length would be a sensible part of this approach. In addition, scenario 2 could be equipped with a selection process similar to that of scenario 3, resulting in a variable population size.

Because Revolve was designed explicitly with future research in mind, recommendations for such projects are numerous. As a first step proof of concept, the scenarios of Section 5.4 fall just short of the envisioned on-line embodied evolution. Repeating an experiment similar to the precursor to this work by Weel et al. [42] seems to be a logical next step. Setting up such an experiment should be straightforward with the help of Revolve, although finding parameters that provide interesting dynamics in such a system is a whole new research project in itself. Attempts towards this goal were in fact made as part of developing Revolve, but were eventually decided to be out of scope, partially as a result of lack of progress in this area. Something to be wary of in this context is the 'bootstrapping problem', a term used to describe the failure of a system to evolve into interesting dynamics simply because there are no dynamics to begin with. Robot learning could be developed and integrated into Revolve as a potential solution to this problem. This would enable robots to make more rapid and efficient use of any sensors they have, which is expected to have an impact on the influence of robot interactions and the environment in which they operate. Varying environmental properties is also an interesting line of research, in conjunction with for instance evaluating the robustness and adaptability of robots and robot populations.

References

[1] Joshua Auerbach, Deniz Aydin, Andrea Maesani, Przemyslaw Kornatowski, Titus Cieslewski, Grégoire Heitz, Pradeep Fernando, Ilya Loshchilov, Ludovic Daler, and Dario Floreano. “RoboGen: Robot Generation through Artificial Evolution”. In: *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*. EPFL-CONF-200995. The MIT Press. 2014, pp. 136–137.

[2] Randall D Beer, Hillel J Chiel, and Leon S Sterling. “Heterogeneous Neural Networks for Adaptive Behavior in Dynamic Environments”. In: *Advances in Neural Information Processing Systems*. 1989, pp. 577–585.

[3] Josh C Bongard. “Evolutionary Robotics”. In: *Commun. ACM* 56.8 (Aug. 2013), pp. 74–83. ISSN: 0001-0782. DOI: `10.1145/2493883`. URL: `http://doi.acm.org/10.1145/2493883`.

[4] Josh C Bongard and Rolf Pfeifer. “Evolving Complete Agents Using Artificial Ontogeny”. In: *Morpho-functional Machines: The New Species* (2003), pp. 237–258. DOI: `10.1.1.26.4442`.

[5] Josh Bongard, Victor Zykov, and Hod Lipson. “Resilient Machines through Continuous Self-Modeling”. In: *Science* 314.5802 (2006), pp. 1118–1121.

[6] Nicolas Bredeche, Evert Haasdijk, and Agoston E Eiben. “On-Line, On-Board Evolution of Robot Controllers”. In: *Artifical Evolution*. Springer, 2009, pp. 110–121.

[7] Nicolas Bredeche and Jean-Marc Montanier. “Environment-Driven Open-Ended Evolution with a Population of Autonomous Robots”. In: *Evolving Physical Systems Workshop* (2012), pp. 7–14.

[8] Nicolas Bredeche, Jean-Marc Montanier, Wenguo Liu, and Alan FT Winfield. “Environment-Driven Distributed Evolutionary Adaptation In a Population of Autonomous Robotic Agents”. In: *Mathematical and Computer Modelling of Dynamical Systems* 18.1 (2012), pp. 101–129.

[9] Luzius Brodbeck, Simon Hauser, and Fumiya Iida. “Morphological Evolution of Physical Robots through Model-Free Phenotype Development”. In: *Plos One* 10.6 (2015), e0128444. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0128444`. URL: `http://dx.plos.org/10.1371/journal.pone.0128444`.

[10] Ken Caluwaerts, Jérémie Despraz, Atıl Işçen, Andrew P Sabelhaus, Jonathan Bruce, Benjamin Schrauwen, and Vytas SunSpiral. “Design and Control of Compliant Tensegrity Robots Through Simulation and Hardware Validation”. In: *Journal of The Royal Society Interface* 11.98 (2014), p. 20140520.

[11] Dave Cliff, Phil Husbands, and Inman Harvey. “Explorations in Evolutionary Robotics”. In: *Adapt. Behav.* 2.1 (1993), pp. 73–110. ISSN: 1059-7123. DOI: 10.1177/105971239300200104. URL: http://dx.doi.org/10.1177/105971239300200104.

[12] Emanuele Crosato. “A Robotic Ecosystem with Co-Evolvable Minds and Bodies”. Master’s Thesis. Vrije Universiteit Amsterdam, 2014.

[13] Massimiliano D’Angelo, Berend Weel, and Agoston E Eiben. “HyperNEAT Versus RL PoWER for Online Gait Learning in Modular Robots”. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2014, pp. 777–788.

[14] Kenneth A De Jong. “Are Genetic Algorithms Function Optimizers?” In: *PPSN*. Vol. 2. 1992, pp. 3–14.

[15] Gilberto Echeverria, Séverin Lemaignan, Arnaud Degroote, Simon Lacroix, Michael Karg, Pierrick Koch, Charles Lesire, and Serge Stinckwich. “Simulating Complex Robotic Scenarios with MORSE”. In: *SIMPAR*. 2012, pp. 197–208. URL: http://morse.openrobots.org.

[16] Agoston E Eiben. “Grand Challenges for Evolutionary Robotics”. In: *Frontiers in Robotics and AI* 1.June (June 2014), pp. 1423–1451. ISSN: 2296-9144. DOI: 10.3389/frobt.2014.00004. URL: http://journal.frontiersin.org/article/10.3389/frobt.2014.00004/abstract.

[17] Agoston E Eiben, Nicolas Bredeche, Mark Hoogendoorn, J Stradner, Jon Timmis, AM Tyrrell, and Alan FT Winfield. “The Triangle of Life: Evolving Robots in Real-Time and Real-Space”. In: *Advances in Artificial Life, ECAL 2013* (2013), pp. 1056–1063. DOI: 10.7551/978-0-262-31709-2-ch157. URL: http://mitpress.mit.edu/sites/default/files/titles/content/ecal13/978-0-262-31709-2-ch157.pdf.

[18] Agoston E Eiben, Serge Kernbach, and Evert Haasdijk. “Embodied Artificial Evolution”. In: *Evolutionary Intelligence* 5.4 (2012), pp. 261–272. ISSN: 1864-5909. DOI: 10.1007/s12065-012-0071-x. URL: http://link.springer.com/10.1007/s12065-012-0071-x$%5Cbackslash$npapers3://publication/doi/10.1007/s12065-012-0071-x.

[19] Agoston E Eiben and James E Smith. *Introduction to Evolutionary Computing*. Vol. 53. Springer, 2003.

[20] Agoston E Eiben and Jim E Smith. “From Evolutionary Computation to the Evolution of Things”. In: *Nature* 521.7553 (2015), pp. 476–482. ISSN: 0028-0836. DOI: 10.1038/nature14544. URL: http://www.nature.com/doifinder/10.1038/nature14544.

[21] Agoston E Eiben and Jim E Smith. “Towards the evolution of things”. In: *ACM SIGEVOlution* 8.3 (2016), pp. 3–6.

[22] Joshua M Epstein and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. Brookings Institution Press, 1996.

[23] Dario Floreano, Phil Husbands, and Stefano Nolfi. “Evolutionary Robotics”. In: *Springer handbook of robotics*. Springer, 2008, pp. 1423–1451.

[24] Dario Floreano and Francesco Mondada. “Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural Network Driven Robot”. In: *Proceedings of the third international conference on Simulation of adaptive behavior: From Animals to Animats 3*. LIS-CONF-1994-003. MIT Press. 1994, pp. 421–430.

[25] Jonathan Hiller and Hod Lipson. “Dynamic Simulation of Soft Multimaterial 3D-Printed Objects”. In: *Soft Robotics* 1.1 (2014), pp. 88–101.

[26] Philip Husbands and Inman Harvey. “Evolution Versus Design: Controlling Autonomous Robots”. In: *AI, Simulation and Planning in High Autonomy Systems, 1992. Integrating Perception, Planning and Action., Proceedings of the Third Annual Conference of.* IEEE. 1992, pp. 139–146.

[27] Nick Jakobi, Phil Husbands, and Inman Harvey. “Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics”. In: *Advances in artificial life*. Springer, 1995, pp. 704–720.

[28] Nathan Koenig and Andrew Howard. “Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator”. In: *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on.* Vol. 3. IEEE, pp. 2149–2154.

[29] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. “The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics”. In: *Evolutionary Computation, IEEE Transactions on* 17.1 (2013), pp. 122–145.

[30] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Vol. 1. MIT press, 1992.

[31] Oliver Michel. “Webots: Professional Mobile Robot Simulation”. In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42. URL: `http://www.ars-journal.com/International-Journal-of-%20Advanced-Robotic-Systems/Volume-1/39-42.pdf`.

[32] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT press, 2000.

[33] Rolf Pfeifer and Josh Bongard. *How the Body Shapes the Way We Think: A New View of Intelligence*. MIT press, 2006.

[34] Jordan B Pollack and Hod Lipson. “Automatic Design and Manufacture of Robotic Lifeforms”. In: *Nature* 406.6799 (Aug. 2000), pp. 974–978. ISSN: 00280836. DOI: `10.1038/35023115`. URL: `http://www.nature.com/doifinder/10.1038/35023115`.

[35] Thomas S Ray. “An Approach to the Synthesis of Life”. In: (1991).

[36] E Rohmer, SPN Singh, and M Freese. “V-REP: a Versatile and Scalable Robot Simulation Framework”. In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. 2013.

[37] Karl Sims. “Evolving 3D Morphology and Behavior by Competition”. In: *Artificial Life* 1.4 (1994), pp. 353–372. ISSN: 1064-5462. DOI: 10.1162/artl.1994.1.4.353. URL: http://dx.doi.org/10.1162/artl.1994.1.4.353$%5Cbackslash$nhttp://www.mitpressjournals.org/doi/pdf/10.1162/artl.1994.1.4.353$%5Cbackslash$nhttp://www.mitpressjournals.org/toc/artl/1/4.

[38] Karl Sims. “Evolving Virtual Creatures”. In: *Siggraph ’94* SIGGRAPH ’.July (1994), pp. 15–22. ISSN: 10645462. DOI: 10.1145/192161.192167. URL: http://dl.acm.org/citation.cfm?id=192167$%5Cbackslash$nhttp://portal.acm.org/citation.cfm?doid=192161.192167.

[39] Kenneth O Stanley and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[40] Alan M Turing. “Computing Machinery and Intelligence”. In: *Mind* 59.236 (1950), pp. 433–460.

[41] Richard A Watson, Sevan G Ficici, and Jordan B Pollack. “Embodied Evolution: Distributing an Evolutionary Algorithm in a Population of Robots”. In: *Robotics and Autonomous Systems* 39.1 (2002), pp. 1–18.

[42] Berend Weel, Emanuele Crosato, Jacqueline Heinerman, Evert Haasdijk, and Agoston E Eiben. “A Robotic Ecosystem with Evolvable Minds and Bodies”. In: *2014 IEEE International Conference on Evolvable Systems* (2014), pp. 165–172. DOI: 10.1109/ICES.2014.7008736.

[43] Larry Yaeger, Virgil Griffith, and Olaf Sporns. “Passive and Driven Trends in the Evolution of Complexity”. In: *arXiv preprint arXiv:1112.4906* (2011).

[44] Kaizhong Zhang and Dennis Shasha. “Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems”. In: *SIAM journal on computing* 18.6 (1989), pp. 1245–1262.