University of Edinburgh
Division of Informatics

Evolving Robust Control Strategies for Simulated Animats

4th Year Project Report
Artificial Intelligence and Software Engineering

Seyed Mohammadali Eslami

March 31, 2009

Abstract

Recent research in the field of robotics has seen an increase of interest in the study of modular, self-reconfiguring robotic systems. In addition, improvements to automatic fabrication methods and rapid prototyping technologies have given researchers the ability to experiment with different robot morphologies quickly and with little cost. The problem of high-level, task-oriented control of these robots however, has remained to be a challenge.

This project introduces a framework to tackle the aforementioned problem. Control strategies for robot morphologies are automatically found using evolutionary computational techniques. The optimisation algorithms are designed to exploit the specification language used to generate the morphologies, by creating hierarchical groupings of the morphologies' joints and motors.

The fitness of the best solutions found for various types of morphologies is recorded and the data is analysed to identify if correlations can be found between certain features of the morphologies and their abilities to perform various tasks.

Acknowledgements

I would like to thank my supervisor, Dr. Subramanian Ramamoorthy, for his guidance throughout the course of the project.

Contents

1. Introduction

Recent research in the field of robotics has seen an increase of interest in the study of modular, self-reconfiguring robotic systems. These self-reconfiguring robots can be completely autonomous, and have the ability to change their shapes by rearranging the connectivity of their parts. Figures 1.1 and 1.2 show MTRAN III [14] and Molecubes [21], which are two examples of such robotic systems.

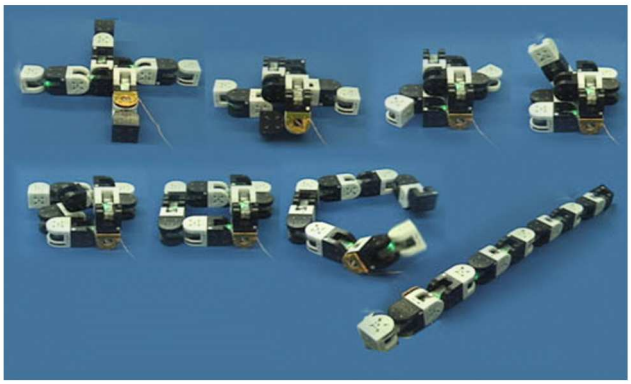

Figure 1.1: The MTRAN III hybrid chain and lattice self-reconfiguring system, Satoshi Murati et al (2005) [14].

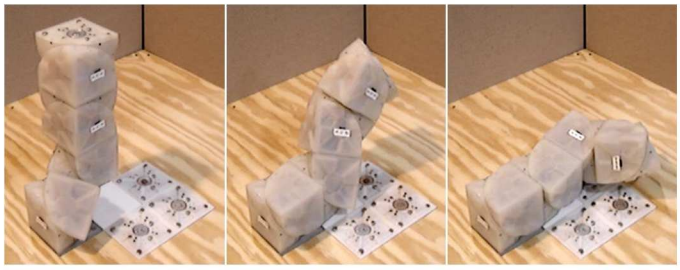

Figure 1.2: The Molecubes chain self-reconfiguring system, Hod Lipson et al (2005) [21].

In addition, improvements to automatic fabrication methods and rapid prototyping technologies have given researchers the ability to experiment with different robot morphologies quickly and with little cost. These methods use commercial rapid prototyping ("3D printing") technologies to create physical instances of desirable robot morphologies. Figure 1.3 shows images of the work by Hod Lipson and Jordan Pollock on such automatic methods [15].

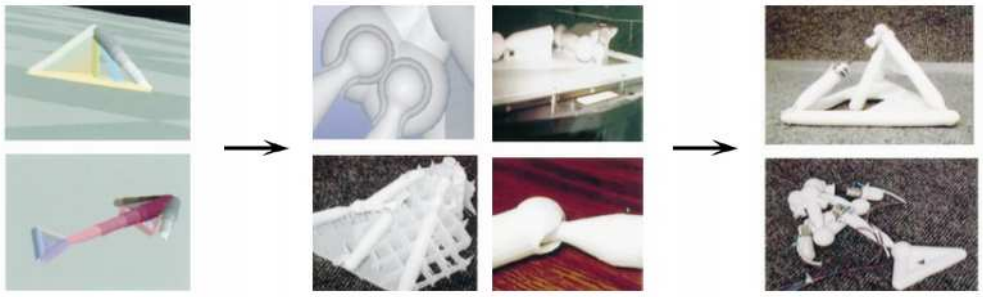

Figure 1.3: The instantiation of a simulated robot to a physical one. Hod Lipson and Jordan Pollack (2000) [15].

The problem of high-level, task-oriented control of these robots however, has remained to be a challenge [20]. The reasons for this are threefold. First, researchers face a huge space of possible robot morphologies from which they can choose from. These robots can reconfigure to take almost any shape, or such morphologies can be created from scratch. However, it can often be unclear what the most suitable configuration for achieving a given task is. This problem can be further exacerbated when the robots are expected to perform a *range* of tasks with a single morphology.

Second, is the fact that these robot configurations tend to have many more degrees of freedom than traditional, hand-built robots. These extra degrees of freedom make finding good control strategies for the robots much more difficult.

Third, is the fact that since many of these robots bear no resemblance to creatures we are familiar with in nature, designing control strategies for them by hand can often be difficult. In these cases, even the use of patterns seen in nature as a heuristic for searching the space of control strategies, becomes unfeasible.

If is of interest to see if solutions to these problems can be found. By providing researchers with a framework which allows suitable control strategies to be found for any robot morphology, and by providing a baseline comparison of different morphology types' aptitudes at various tasks, advances can be made more quickly in this field.

1.1 Existing Work

A number of different strategies been employed to tackle the aforementioned problems over the years.

Optimisation algorithms of various types have been used to find control strategies for *fixed* robot morphologies. Neural network controllers have been used to animate dolphin, car and human animats (*animat* is a contraction of *anima-material*, meaning 'artificial animal'. In this text, we use the term to refer to simulated robots) [11]. More recently, Reeve and Hallam use similar techniques to control quadruped animats [18].

Genetic algorithms have also been employed to learn controls strategies for such simulated robots [12]. In a very well known paper, Bongard, Zykov and Lipson present genetic algorithms that not only learn such control strategies, but also adjust their strategies in the event of physical failures [6].

In one of the most widely-cited papers in this field, Sims uses evolutionary computing techniques to learn animat morphologies *simultaneously* with their control strategies [19]. More recently, similar techniques have been used to evolve walking and block-throwing creatures [8]. In his PhD thesis, Miconi presents a thorough investigation of the effectiveness of evolutionary techniques in evolving animat morphologies [16].

In a separate area of research, academics have also been looking at nature to find morphologies which are suitable for the tasks they have in mind for their robots. These biologically inspired robots include the insect-like robots for robust locomotion [7], starfish-like robots [6] and limbless, snake-shaped robots for navigation in tight spaces [9].

Observations from nature have also been used to find controls strategies for robots. A good overview of such work is given in [5]. More recently, Auke Jan Ijspeert et al examine amphibious salamanders and use biological observations to create robots which can seamlessly switch between swimming and walking gaits [13].

1.2 Overview

This project introduces a framework which allows for some of the problems described in §1 to be overcome. First, a robot specification language is presented which limits the range of morphologies to one which is more likely to exist in nature, and reduces the space of robot morphologies to one that is more likely to produce desirable robots.

Control strategies for the morphologies are automatically found using evolutionary computational techniques. The optimisation algorithms are designed to exploit the specification language used to generate the morphologies, by creating hierarchical groupings of the morphologies' joints and motors.

The fitness of the best solutions found for three types of morphologies is recorded and the data is analysed to identify if correlations can be found between certain features of the morphologies and their abilities to perform various tasks.

In particular, this project evaluates the following hypotheses:

1. That there exists a correlation between the set of specified test tasks and the morphologies of the animats that excel at those tasks.
2. That some morphologies will be specialists (they will be excellent at only a few tasks), and others will be generalists (they will be good at all tasks).
3. That there exists a correlation between the features in the animats' morphologies and the tasks they specialise in.

In addition, this project proposes the following:

1. That the results of the tests performed using the framework will allow researchers to identify which morphologies are most suited to the requirements of their robot.
2. That these results will give us insights into the patterns we witness in nature.
3. That the hierarchical control schemes can be used with genetic algorithms to find suitable control strategies for a wide range of morphologies.

1.3 Influence

This project makes contributions to several areas within the field of robotics. First, it extends upon existing morphology specification languages and describes the extended language in detail. Second, it presents novel techniques which use these graphical models to create realistic control strategies for the morphologies. Third, it provides an analysis of different morphology types' abilities to perform various tasks.

In addition, the software platform created for the purposes of this project provides a reliable and easy to use framework for 3D robot simulation.

1.4 Summary of Results

The study was very successful, upholding the first two hypotheses and demonstrating that hierarchical control schemes can used with genetic algorithms to find suitable control strategies for a wide range of morphologies. However, no

correlations could be found between the features in the animats' morphologies and the tasks they specialised in.

1.5 Dissertation Outline

The following chapter, *Methodology*, describes the algorithms and techniques used to train the animats and provides an overview of the design of the simulation application. The *Experimental Setup* chapter describes the experiments that have been run to evaluate the hypotheses. The *Results* chapter presents the results of these experiments and reconsiders the hypotheses. Finally, the *Conclusion* summarises the project outcomes and proposes ideas for further work.

2. Methodology

In this chapter, the design decisions that were made to allow for the evaluation of the project's hypotheses are discussed. The graphical models used to instantiate the families of animat morphologies are described, and novel algorithms are presented to control the morphologies' motors. The genetic algorithms which are used to find optimal control strategies for the morphologies are defined. Finally, an overview of the technical aspects of the project's implementation is given.

2.1 Morphologies

Due to the nature of the project's experimental setup (see §3), a large number of animats with varying morphologies have to be created in the physics simulation environment. Whilst these animats were manually defined in the code in the initial attempts, it quickly became clear that this process would eventually have to be automated.

To this end, an animat specification language based on the work by Sims [19] was designed. The animat specifications are interpreted at run-time and automatically converted to physical actors in the simulation environment.

Each morphology specification is described by three parameters: a directed (cyclic) graph, a pointer to the root node in the graph and a depth limit. The graph describes the type and shape of the animat, whereas the depth limit specifies its complexity and size. Figure 2.1 shows two morphology specifications and their corresponding animats.

The specification graph is traversed by a depth-first-search algorithm. The algorithm operates by following the edges in the graph, and uses the information about its current position on the graph to create the animat's parts in the physics environment.

The cuboids are spawned in precisely the correct locations in three dimensional space to allow neighbouring parts to be attached to each another with joints. The position and orientation of each part is a function of the position and orientation of its parent, as well as the number of its siblings. For parent nodes with an odd number of children, the parts are placed in a way which maximises the amount of space spanned by the parent and the children. Parent nodes with an even number of children, however, are treated differently. Due to the design of the animat control mechanisms (see §2.2), one of the children is chosen to extend the parent's orientation, and the others are distributed around this primary child. If

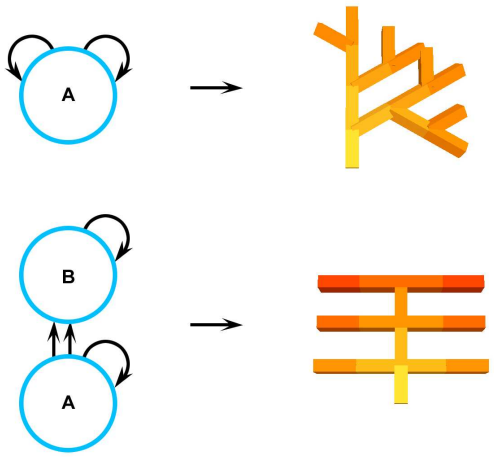

Figure 2.1: Two graphical models and their corresponding animats. The top model has been instantiated with a depth-limit of 4, and the bottom one with a depth-limit of 2.

a spine is found to have more than point at which an even number of parts are attached to it, the distribution of the parts is automatically balanced. Figure 2.2 compares the position of the parts for parents with even and odd numbers of children and provides an example of an automatically balanced spine.

The cuboids are connected to each other with six-degree-of-freedom joints, however limits are placed on the joints to mimic those of real robots and animals. Additionally, three linear motors are placed on the joints to provide vertical (raising), horizontal (swinging) and twisting movement capabilities to each part.

Extra measures are taken to ensure that the produced animats are as life-like as possible. The size of the cuboids generated by the graph-traversal algorithm are adjusted to reflect their distance from the animat's head, ensuring that the size of the parts progressively gets smaller towards the tips of the animat's limbs.

2.2 Control Strategies

The specification language used to create the morphologies is exploited to create concise descriptions of the animats' control strategies. Once an animat has been instantiated, a second traversal is made on the specification graph to identify *spines* in its morphology. In this context, spines are defined to be sequences of

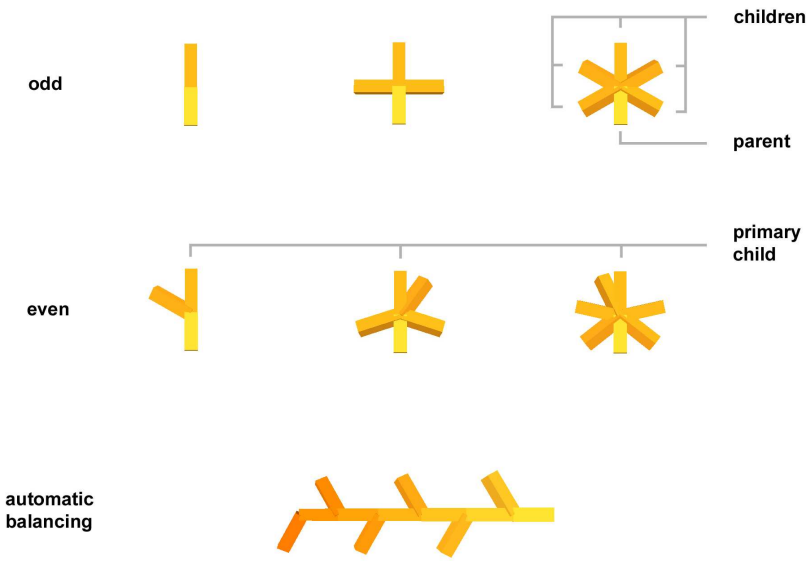

Figure 2.2: A few examples of automatically generated joints. The top row shows how an odd number of child limbs are connected to a parent limb. The middle row shows how an even number of child limbs are connected to a parent limb. The final image shows how uneven limbs are distributed along the animat's spine.

parts in an animat's morphology which extend in the same direction in space.

This is done using a depth-first-search algorithm similar to the one used to create the morphologies in the first place. Figure 2.3 shows how spines are extracted from the animat morphologies with a simple example and Figure 2.4 shows the anatomy of an individual spine.

The motors corresponding to the joints in each spine are grouped into spine controllers in the animats' 'brains'. Instead of controlling each one of the motors independently, the animats now only send control signals to these controllers.

The controllers manipulate each motor with a sinusoidal movement in the vertical (raise) and horizontal (swing) directions. The signals sent to the motors in each joint are phase-shifted by a value proportional to the joint's rank in the spine. The parameters of the sinusoidal motion and the size of the phase-shift are the only parameters determined by the animat's 'brain'. The sinusoidal motion in each direction (vertical and horizontal) is mixed using a real-valued variable between zero and 1. Figure 2.5 shows the relationship between the motors and the signals sent from the animats' brains. Appendix D lists the section of the code which performs some of these functions.

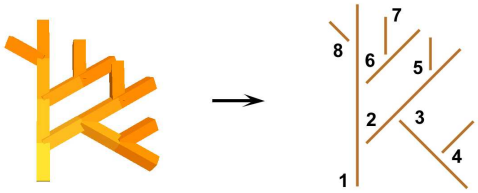

Figure 2.3: Spines are automatically extracted from the morphology of a tree-shaped animat.

2.3 Finding the Best Control Strategies

The parameters for the animats' spine controllers are tuned to maximise their performances at different tasks. To do this, this project uses a family of search and optimisation methods called genetic algorithms (GAs). These algorithms are considered to be a sub-class of evolutionary algorithms, and use techniques inspired by evolutionary biology such as inheritance, mutation, selection and crossover to find solutions to search problems [17] [10].

Algorithm 1 outlines the workings of a simple GA. To use any GA for search and optimisation problems, two different types of decisions need to be made. The first, is to choose a representation for the optimisation problem which the GA can work with, and the second is to choose a suitable set of parameters and operators for the GA. The following sections will describe the choices made for the genetic algorithms used in this project.

2.3.1 Genetic Algorithm Representation

Two additional formalisms are made in order to adapt a search or optimisation problem to the generic genetic algorithm. First, a fitness measure is defined, which is used to identify the relative fitness of the possible solutions to the problem. This measure is often referred to as the problem's *fitness function*. Second, a suitable encoding of the possible solutions is defined to a data structure which can be manipulated by the GA. It is this encoding that is mutated and crossed in a semi-informed way to reach the solution. An instance of this encoding is referred to as a *chromosome*.

In this project, genetic algorithms are used to optimise the animats' performances

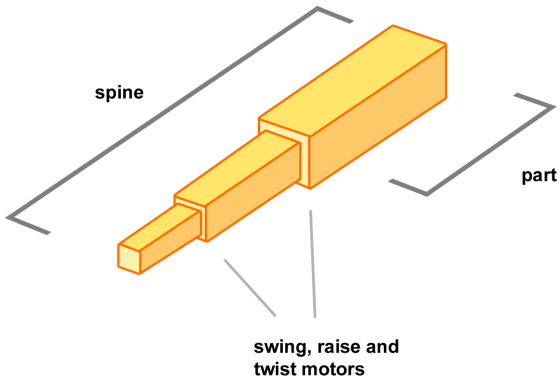

Figure 2.4: The anatomy of a spine. Animats are composed entirely of spines, and spines are in turn made by connecting limbs together with joints and motors.

Alg. 1 Simple Genetic Algorithm.

Input: fitness function, chromosome representation and operators
Output: fit population

```
1   begin
2       choose initial population
3       evaluate the fitness of each individual in the population
4       while fitness is not sufficiently high do
5           select best-ranking individuals to reproduce
6           breed new generation through crossover and mutation
7           evaluate the individual fitnesses of the offspring
8           replace worst ranked part of the population with offspring
9       end
10  end
```

at a number of different tasks (see §3.4). A separate genetic algorithm is run to train the animats' control parameters for each task. For this reason, distinct fitness functions are defined for each task.

The fitness of any given set of spine controller parameters (or control strategy) at a task is defined to be the fitness of an instantiation of an animat with that control strategy at that task. Special scenes have been designed to calculate these fitnesses for each task.

The scenes are designed to be variants of a simple rectangular 'pen' with a surrounding parameter. The animats are spawned in the centre of this rectangular

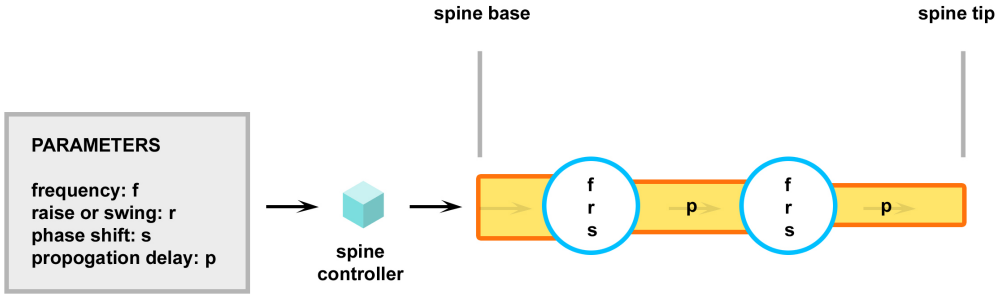

Figure 2.5: The relationship between the motors in a spine and the signals relayed from the animats' brains.

pen. The animat's brain then begins to send signals to the spine controllers based on the parameters encoded in its chromosome.

A supervisor thread keeps track of the amount of time the simulation has been running. Once the time limit has been exceeded, the animat's 'score' at that tasked is calculated and returned as the chromosome's fitness.

Figure 2.6 shows an aerial view of the simulation environment. For more details on the scenes designed specifically for each task, see §3.4.

A second formalism has to be made on the GA's representational structures. Specifically, an encoding scheme for each control strategy as a chromosome is designed. Due to the way the animats are controlled in the implementation (through 4 parameters for each one of the spine controllers), this can be done very easily. Once the spines are extracted from the animat, the control strategy can be represented as a sequence of groups of 4 positive reals, each corresponding to one of the spines. Figure 2.7 shows the relationship between a chromosome and the corresponding animat's control strategy.

2.3.2 Genetic Algorithm Parameters

In addition to representational decisions, the parameters of the genetic algorithms also have to be chosen. No formal procedure was used to make these decisions, rather the parameters that proved to be the most promising in initial tests were used for the final experiments.

The parameters have been chosen in a way which ensures that reasonably good solutions are found, in as little time as possible (as opposed to finding great solutions after a long period of time). These preliminary solutions could be

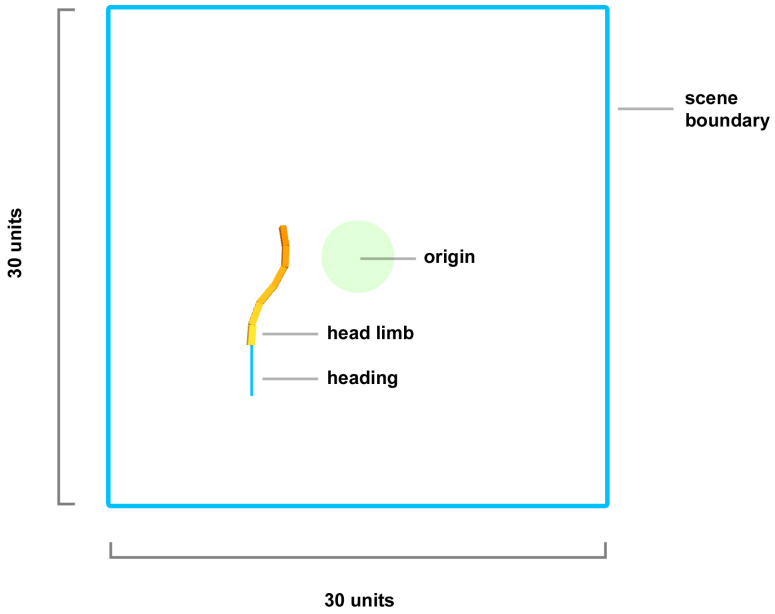

Figure 2.6: The layout of the simulation environment. The animats are always spawned in the centre (origin) of the scene.

refined with more fine-tuned algorithms, but this was not within the scope of this project.

To this end, populations of size 50 are evaluated for a total of 10 iterations for each task. By having a relatively small number of iterations and a large population size, a more random search in the space of control strategies is performed, which leads to solutions faster.

To further increase the rate at which solutions are found, two extra measures are taken. First, the chromosome selection method is augmented with *elitism*, which ensures that the fittest chromosome from the previous population is repeated at least n times in the new population. Extremely strong elitism was used in this implementation ($n = 20$).

Finally, the standard fitness-proportionate selection method is augmented to increase the selection pressure on fitter chromosomes. This is done by performing fitness-proportionate (roulette-wheel) selection on *a function* of the fitnesses. Specifically, the fitnesses are raised to the power of k (in this implementation, k is set to 5). This increases the probability of fitter chromosomes being chosen for reproduction.

Since the chromosomes are merely represented as a fixed-length sequence of genes (real numbers), basic mutation and crossover operators can be used. Every chro-

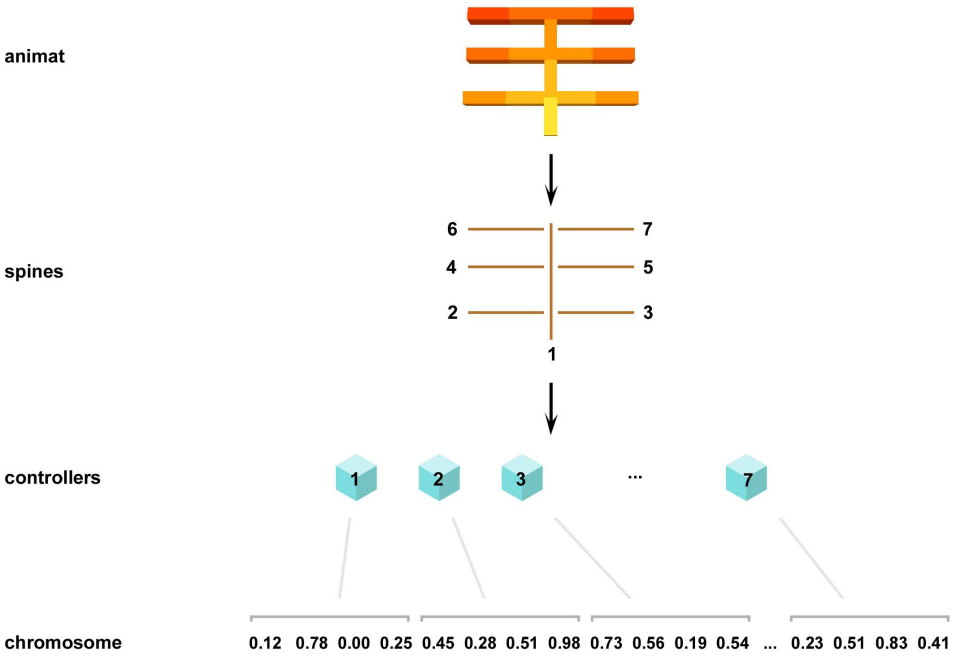

Figure 2.7: The relationship between a chromosome and the corresponding animat's control strategy. Once the animat's spines have been identified, each one of the chromosome's values (genes) is used as an input to one of the spine controllers.

mosome which progresses to the new population is mutated. Each one of the chromosome's genes are mutated with a fixed probability (in this implementation, $p_m = 0.5$). The mutation procedure consists of a random increase or decrease of the gene's current value (which must be within 10% of the gene's old value).

When two chromosomes are chosen for breeding, a single random point in the chromosomes' length is chosen. The children are created by combining the parents' genes from before and after the chosen crossover point.

2.4 Technology

The experimentation program is implemented as a single Microsoft Windows application. The experiment parameters and program settings are fed into the application via a settings file (in a proprietary comma separated file format – see

Appendix D). The results of the experiments are stored as a series of chromosome serialisations, each corresponding to the fittest chromosome in every generation.

The same application is used to inspect the fittest chromosomes for each morphology at each task. This is done by choosing the appropriate settings in the application's configuration file. The camera, the environment and the animats themselves can be manipulated via keyboard and mouse input in real-time.

The application can run in three different speed settings. The normal speed setting runs the experiments in 1× real-time. The medium speed setting runs the experiments as fast as the hardware allows (without any artificial time delays in the physics simulations). The fastest speed setting additionally turns off all rendering procedures, which results in a dramatic increase in speed.

Videos of the application's simulation environment and the user interactivity features can be seen at `http://www.arkitus.com/ERCS`.

2.4.1 Implementation

The application is written in the C++ programming language and external libraries are used for the physics calculations and rendering. Specifically, a third-party 3D physics engine, NVIDIA PhysX [1] is used to recreate the realistic physics environment. PhysX is well-known in the games industry and features support for soft and rigid-body dynamics, joints, motors and forces. The PhysX drivers are required to run the application and to compile it. The PhysX software development kit (SDK), on the other hand, is only required for compilation.

The Open Graphics Library (OpenGL) [2] is used to render the animats and their environments. The Simple DirectMedia Layer (SDL) library [3] is used for window management and for mouse and keyboard input. Figure 2.8 displays the interactions between the application's libraries at a high-level.

The code structure is similar in many ways to any modern 3D game. Once all the libraries have been loaded and the scene has been created, an instance of the `SceneController` super-class is chosen and activated. `SceneControllers` are classes which have the ability to manipulate the scene by adding or removing physical objects from it. In this implementation, a `GASceneController` is chosen as the active scene upon initialisation. The `GASceneController` runs the genetic algorithm in a separate thread and periodically inserts animats into the scene and measures their performance.

Each instance of a genetic algorithm is defined by arguments of the following types: `Chromosome`, `Mutator`, `Crosser` and `FitnessEvaluator`. The `Chromosome` is instantiated as a `ConditionalLocalBrain`, with suitable `ConditionalBrainMutator` and `ConditionalBrainCrosser` operators to match. The `FitnessEvalu-`

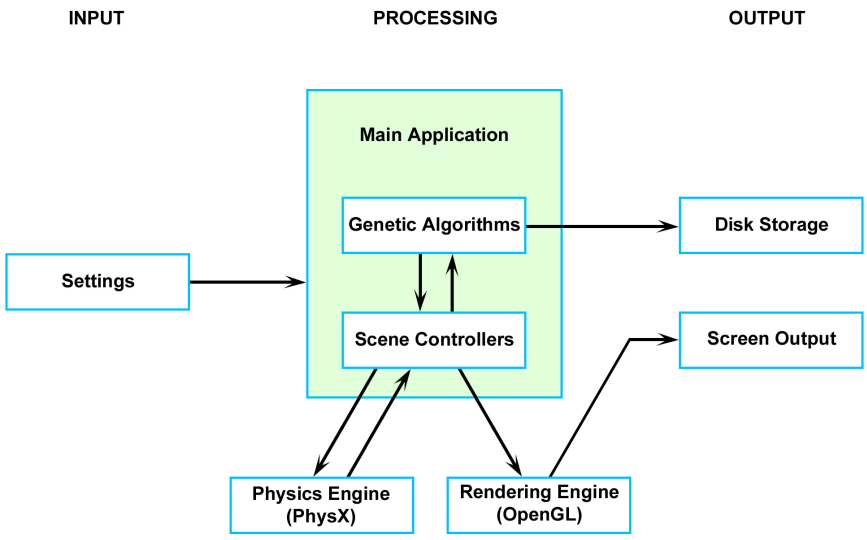

Figure 2.8: A high-level view of the interactions between the application's various libraries.

`ator` is a subclass of `GASceneController` itself (e.g. `LevelLocomotionController`), as it has the ability to measure the various attributes of the scene. The `FitnessEvaluator` variable is what is changed when the animats are being trained for different tasks.

The application is composed of over 110 separate classes across as many files, with a total of just under 13,000 lines of code. At a high level, the code is split up into 8 namespaces:

- `animats`: Contains the classes which define the animat morphologies and their control strategies, as well as the GA operators used to manipulate these control strategies.
- `datapackets`: Contains the classes which are used to define the protocols which the core application uses to communicate with the rendering and physics engines.
- `geneticalgorithms`: Contains the GA code itself, as well as the superclasses (chromosomes, crossover methods, mutation methods, selection methods, fitness evaluators) used by the genetic algorithm.
- `input`: Contains classes which deal with mouse and keyboard user input.
- `rendering`: Contains classes which handle the rendering aspects of the

application and communicate with OpenGL and SDL.

- `scenecontrollers`: Contains the scene controllers which run the experiments and define each one of the test tasks.
- `simulation`: Contains the classes which handle the simulation aspects of the application and communicate with PhysX.
- `utilities`: Contains the classes that save and load data to and from the disk, the standard output and the screen buffer, as well as the settings file parser and the random number generation functions.

The `main()` function is located in a file outside these name-spaces (`simulation_main.cpp`). The routines in this file load and initialise the libraries, initialise the application itself, run the main application loop and connect the renderer and simulator to each other.

A simple Python script is used to run the 72 experiments in succession (see Appendix D). The script records the results and logs and saves them in a directory structure for later viewing. The main application has occasionally been seen to crash in the physics routines. For this reason, the script has been written to accommodate for crashes by restarting the current experiment if one is detected. A secondary Python script is used to extract the fitnesses from the directory structure and to convert them into to a MATLAB-friendly format. A MATLAB script is used to normalise the data, to analyse it and to generate graphs.

2.5 Chapter Summary

In this chapter, the design decisions that were made to allow for the evaluation of the project's hypotheses were discussed. The graphical models used to instantiate the families of animat morphologies were described, and novel algorithms were presented to control the morphologies' motors. The genetic algorithms which are used to find optimal control strategies for the morphologies were defined. Additionally, an overview of the technical aspects of the project's implementation was given.

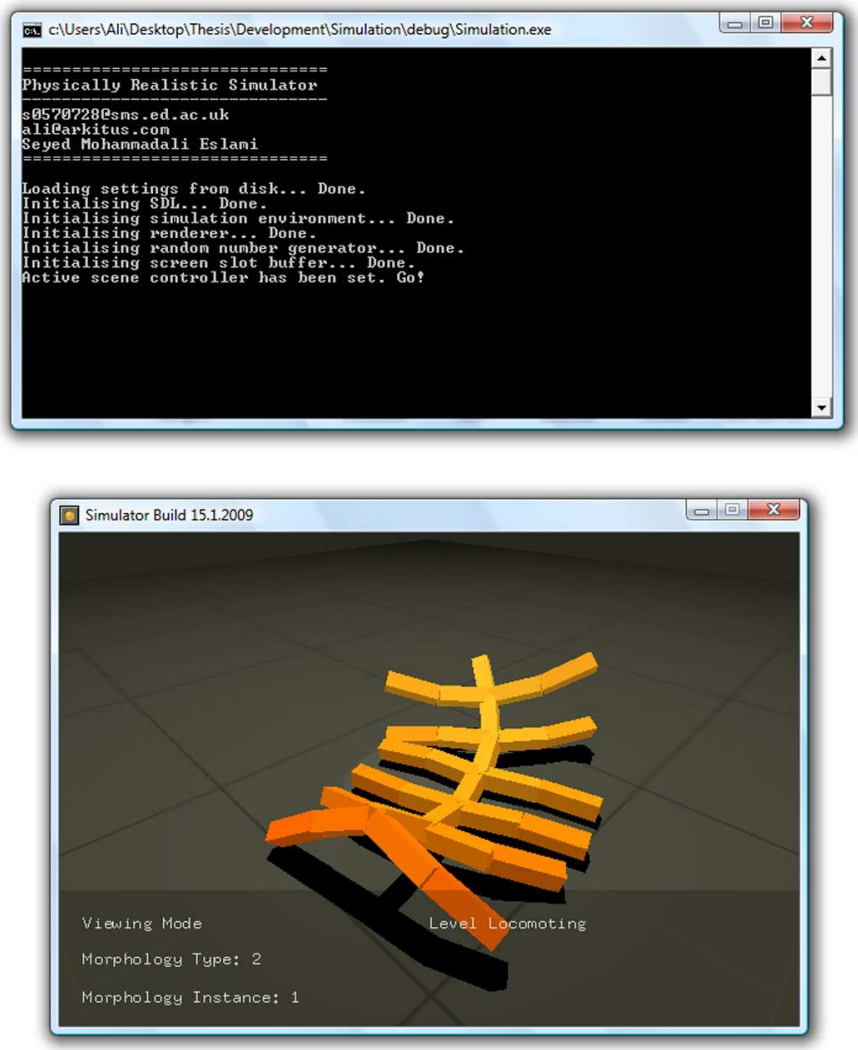

Figure 2.9: The physically realistic simulation environment and its rendering engine. Videos of the application and the user interactivity features can be seen at `http://www.arkitus.com/ERCS`.

3. Experimental Setup

In this chapter, the set of experiments which have been run for this project are described. First, the hypotheses which the project sets out to evaluate are listed. The three families of animat morphologies (snakes, starfish and insects) which have been chosen for the experiments are described. Finally, a suitable set of tasks (level locomotion, rotation, slippery locomotion, weighted locomotion, unlevel locomotion and rough locomotion) at which the morphologies' performances are measured, is presented.

3.1 Hypotheses

The primary goal of this project is to identify if correlations can be found between an animat's morphology type and its physical capabilities. In addition, the data gathered from the tests will be analysed to give us a more accurate sense of the correlations between specific animat features (such as complexity, length, degree of symmetry and so on) and the specific task features (such as whether the tasks are power-oriented, speed-oriented, coarse or fine-grained and so on). Formally, the project will collect data that will evaluate the following hypotheses.

1. That there exists a correlation between the set of specified test tasks and the morphologies of the animats that excel at those tasks.
2. That some morphologies will be specialists (they will be excellent at only a few tasks), and others will be generalists (they will be good at all tasks).
3. That there exists a correlation between the features in the animats' morphologies and the tasks they specialise in.

3.2 Overview

A number of experiments have been run to test the validity of the hypotheses mentioned above. Animats with different morphologies have been trained at a range of tasks using the genetic algorithms discussed in Chapter 2. The GA's success at training the animats to perform these tasks has been recorded. The data from these experiments will be discussed and analysed in the Chapter 4. The following sections will describe the morphologies and tasks which have been examined in the experiments.

3.3 Morphologies

In total, 12 different morphologies have been examined. The morphologies have been chosen from 3 different categories (snakes, starfish and insects) to examine as wide a range of shapes as possible. 4 different instances have been generated from each morphology-type by varying various parameters in each type's graphical model.

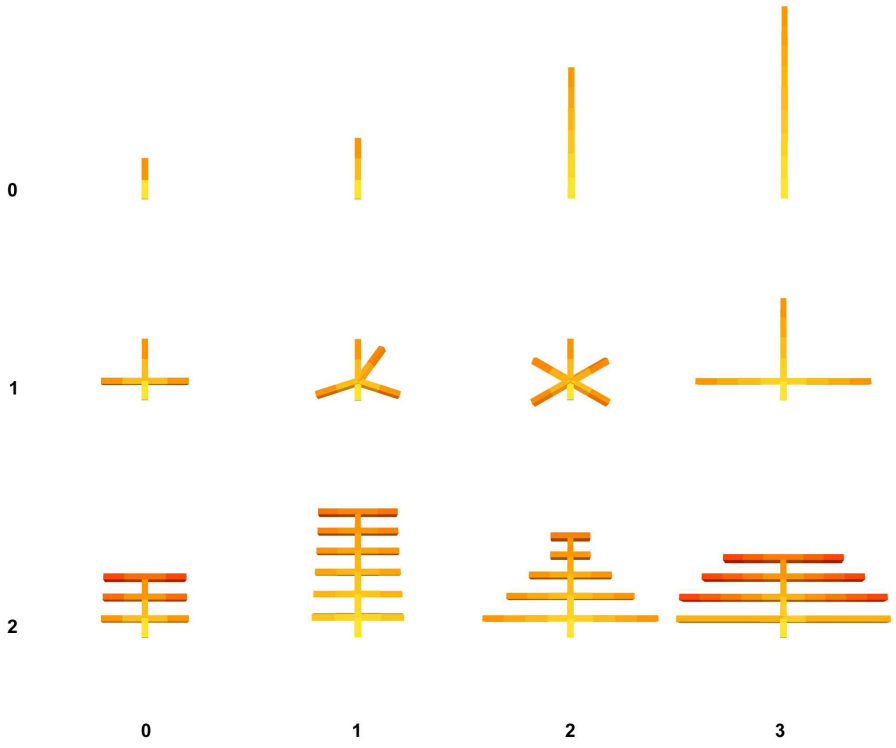

Figure 3.1: The 12 morphology instances examined in the experiments. The top row are instances from the snakes category, the middle row from the starfish category and the bottom row from the insects category.

Figure 3.1 shows the 12 morphology instances examined in the experiments. The top row are instances from the snakes category, the middle row from the starfish category and the bottom row from the insects category. The following sections will describe how each instance is created.

3.3.1 Snakes (Type 0)

This category of morphology instances are created by joining together a sequence of parts which extend in one direction in three dimensional space. The only difference between the different instances in this category is the length of this sequence.

Figure 3.2 shows the family of graphical models used to create these instances. The only parameter which can be altered in this family is d, the model's depth. By using different values for d, the length of the instance can be changed. Table 3.1 lists the values chosen for this parameter for the different snake instances.

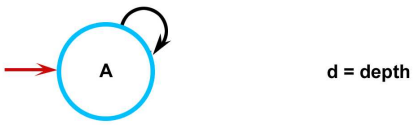

Figure 3.2: The family of graphical models used to create the snake instances.

Snake Instances	
Index	d (depth)
0	1
1	2
2	5
3	8

Table 3.1: The parameters chosen for the different snake instances.

3.3.2 Starfish (Type 1)

This category of morphology instances are similar to the category of snakes, and can be seen as a number of snakes connected to each other at a single focal point. Different instances of this group can be created by connecting different numbers of limbs with different lengths.

Figure 3.3 shows the family of graphical models used to create these instances. Two parameters can be altered in this category's model, the depth (d) and the number of edges from node A to node B (n). By choosing different values for d, the length of each limb can be changed and by using different values for n, the number of limbs can be changed. Table 3.2 lists the values chosen for these parameters for the different starfish instances.

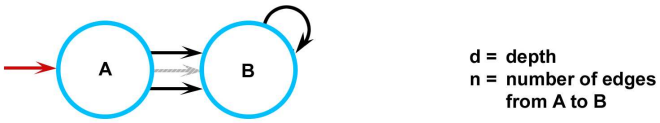

Figure 3.3: The family of graphical models used to create the starfish instances.

Starfish Instances		
Index	d (depth)	n (number of edges from A to B)
0	2	3
1	2	4
2	2	5
3	4	3

Table 3.2: The parameters chosen for the different starfish instances.

3.3.3 Insects (Type 2)

This category of morphology instances are similar to insect morphologies that can be seen in nature.

Figure 3.4 shows the family of graphical models used to create these instances. Two parameters can be altered in this type's model, the depth (d) and the number B nodes (n). By choosing different values for d, the length of each limb can be changed and by using different values for n, the minimum length of each limb can be changed. Table 3.3 lists the values chosen for these paramate rs for the different insect instances.

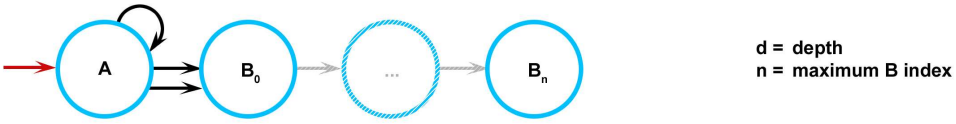

Figure 3.4: The family of graphical models used to create the insect instances.

3.4 Tasks

The performance of each of these instances has been tested at a range of tasks. These tasks have been chosen to be as varied as possible, whilst remaining representative of the kind of tasks the animats may be expected to perform in the real

Insect Instances		
Index	d (depth)	n (maximum B index)
0	2	1
1	5	1
2	3	0
3	4	4

Table 3.3: The parameters chosen for the different insect instances.

world. Table 3.4 lists the tasks and the qualities each was believed to highlight in the morphology instances.

Test Tasks			
Index	Name	Description	Defining Success Factor
0	Level Locomotion	Walking Straight	Accuracy
1	Rotation	Rotating on the Spot	Accuracy
2	Slippery Locomotion	Walking on a Slippery Surface	Robustness
3	Weighted Locomotion	Pulling a Weight	Power
4	Unlevel Locomotion	Climbing Slopes	Power
5	Rough Locomotion	Navigating Rough Terrain	Robustness

Table 3.4: The 6 tasks each morphology instance is tested at.

3.4.1 Level Locomotion (Task 0)

In this task, the animat's ability to walk in a straight line is measured. In order to do this, the animat is spawned in the centre of the test scene. Four probe boxes are also spawned in the scene – one in each cardinal direction relative to the animat's initial heading (see Figure 3.5).

The animat is allowed to move (execute its control strategy) for 15 simulated seconds, at which point its fitness is calculated and the scene is reset. The fitness F of the ith control strategy x_i, is calculated using the following formula:

$$F(x_i) = d(0, b_k) - \alpha D(P(x_i)_{t=t_{max}}) - \beta R_C(P(x_i)_{t=t_{max}}) \tag{3.1}$$

where

$$D(x) = \min \begin{cases} d(x, b_0) \\ d(x, b_1) \\ d(x, b_2) \\ d(x, b_4) \end{cases} \tag{3.2}$$

$d(x, y)$ is the Euclidean distance of object x and object y in three dimensional space, $d(0, b_k)$ is the distance of box k from the origin, $R_C(x)$ is the *cumulative* amount in radians the projection of object x's heading on the ground plane has rotated, α and β are arbitrary mixing coefficients chosen through trial and error and $P(x_i)_{t=t_{max}}$ is the physical object corresponding to the animat at time t_{max}.

This fitness function ensures that the animats are not forced to walk in any particular direction, as motion in any of the four cardinal directions is equally rewarded. The R_C term ensures that the animats' heads are kept relatively still as they move, which makes the resulting gaits more realistic.

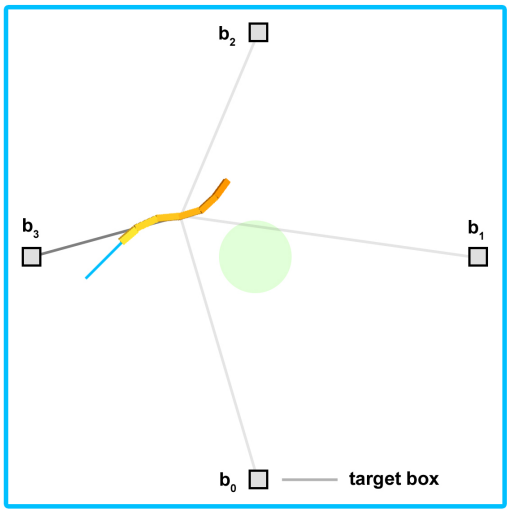

Figure 3.5: The locomotion scene setup.

3.4.2 Rotation (Task 1)

In this task, the animat's ability to rotate on the spot is measured. In order to do this, the animat is spawned in the centre of the test scene. The animat is allowed to move (execute its control strategy) for 5 simulated seconds, at which point its fitness is calculated and the scene is reset. The fitness F of the ith control strategy x_i, is calculated using the following formula:

$$F(x_i) = R(P(x_i)_{t=t_{max}}) \tag{3.3}$$

where $R(x)$ is the amount in degrees the projection of object x's heading on the ground plane has rotated relative to the initial orientation. Note that this is not the same as $R_C(x)$, as it does not calculate the cumulative amount, but the final difference in rotation. This fitness function ensures that the animats make a smooth rotation in a single direction.

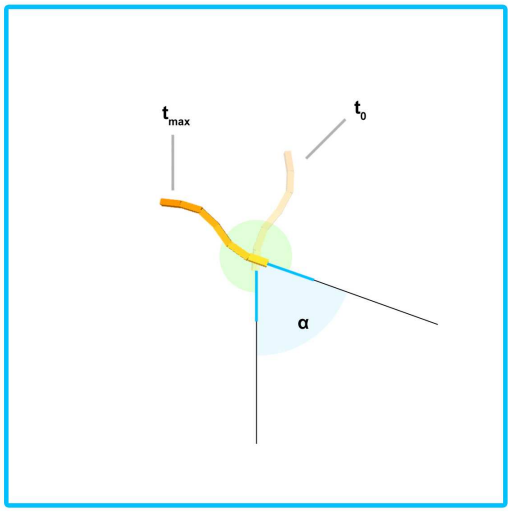

Figure 3.6: The rotation scene setup.

3.4.3 Slippery Locomotion (Task 2)

This task is identical to level locomotion (task 0), with the only difference being the physical properties of the simulation environment. The static and dynamic friction coefficients (μ_s and μ_k) for the scene's ground plane are reduced 80% (from 0.5 to 0.1) to simulate slippery terrain.

3.4.4 Weighted Locomotion (Task 3)

This task also uses an identical fitness function to that of level locomotion (task 0). However, an additional weight (a fixed-size cube) is connected to each animat's tail with a fixed-length damper spring joint.

3.4.5 Unlevel Locomotion (Task 4)

This task also uses an identical fitness function to that of level locomotion (task 0). However, 4 slight inclinations (10 degree uphill) are placed between the centre of origin and the target boxes.

3.4.6 Rough Locomotion (Task 5)

This task also uses an identical fitness function to that of level locomotion (task 0). However, the terrain surrounding the origin is distorted with 9 additional boxes set up in uneven positions and orientations.

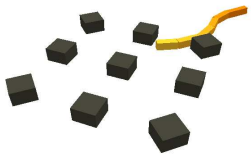

Figure 3.7: A screenshot from the rough locomotion task scene.

3.5 Analysis

In order to obtain the performances of each morphology at the given set of tasks, all possible instance-task combinations have been examined. Each morphology is trained to perform the 6 tasks, and the final scores are recorded. This leads to a total of $12 \times 6 = 72$ individual GA experiments.

The scores obtained by the animats from each task are analysed. The mean of each type's instances' scores at different tasks is computed. A statistically significant variance in the types' performance profiles over the range of tasks would suggest that the morphology types have different areas of strength and weakness. On the other hand, if the different morphology types are found to obtain relatively similar scores, it can be said that no correlations have been found between the types and their abilities to perform the tasks.

The variance of each type's scores is also calculated. A statistically significant difference in the three types' variances would suggest that some of the types are

generalists, and others are specialists. On the other hand, if the variances are found to be roughly the same, this claim cannot be made.

Next, the instances' scores are plotted against their features (such as length and complexity). If correlations can be seen between a feature and the scores, it can be said that the feature explains the animats' performances. On the other hand, if the scores are found to be scattered randomly, or if they remain completely constant, it would suggest that the feature in question has little role in explaining the variance in the animats' performances.

Finally, the animats' performances are plotted in a high-dimensional space in which each dimension corresponds to a single task. This plot is then examined to see if patterns can be found. If the performances of the different types are found to cluster in different regions, it would be further evidence for the existence of correlations between the types are their abilities. If, on the other hand, the points are randomly scattered, no such conclusion can be made.

3.6 Chapter Summary

In this chapter, the set of experiments which have been run for this project were described. First, the hypotheses which the project sets out to evaluate were listed. The three families of animat morphologies (snakes, starfish and insects) which have been chosen for the experiments were described. Finally, a suitable set of tasks (level locomotion, rotation, slippery locomotion, weighted locomotion, unlevel locomotion and rough locomotion) at which the morphologies' performances are measured, was presented.

4. Results and Evaluation

In this chapter, the results of the experiments described in Chapter 3 are presented. A subset of these results is discussed in detail and analysed. Finally, the hypotheses put forward in the previous chapter are re-examined in light of the experimentation results.

4.1 Control Strategies

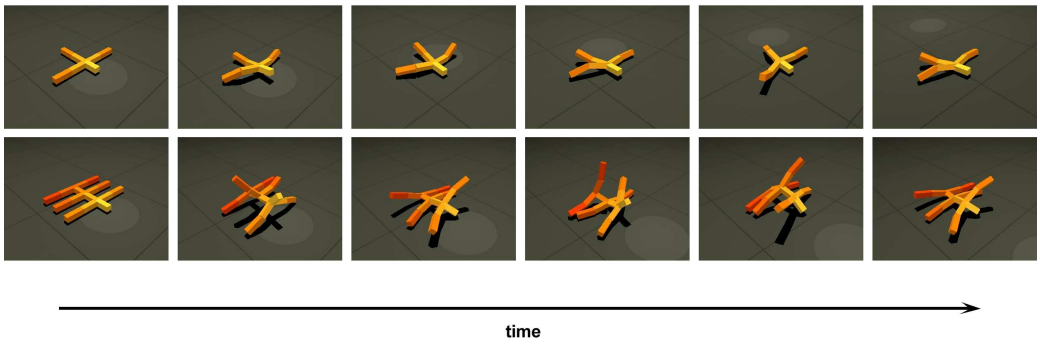

Figure 4.1: Sequence of screenshots showing two animat instances executing their best control strategies for level locomotion. Notice how the instances have learnt strategies which move them in different directions. For videos of these gaits, visit `http://www.arkitus.com/ERCS`.

The genetic algorithms were able to find control strategies for each task surprisingly well. Even with the relatively small number of iterations used in the experiments, most of the animats were able to find very fit gaits. In particular, the animats were found to excel at the first two tasks (level locomotion and level rotation).

Figure 4.1 displays a sequence of screenshots from the level locomotion control strategies learnt for a starfish (type 1, index 0) and an insect (type 2, index 0) instance. It can be seen in the screenshots that both instances successfully move away from the origin (indicated by the grey circle on the ground plane), whilst maintaining a relatively stable head orientation.

The snake instances all learnt very fit gaits for level locomotion. What was surprising, was that the snakes all adopt a 'sidewinding' gait, similar to the gaits adopted by Crotalus cerastes (a species of central-American pitviper snakes).

This form of movement allows the animat the move in a direction which is perpendicular to its spine's extension. The snakes fared much worse at the rotation task, however. The best rotation gaits found for these instances were considerably slower than those found for different morphology types.

The starfish instances performed very well at most of the tasks. The rotational symmetry of their morphologies allowed them to outperform the other instances at the level rotation task. The starfish also learnt to use their limbs to pull themselves forward and their 'tails' to align their motion in the level locomotion task.

Whilst the solutions found for the larger insect instances were not very competitive, the smaller instances of that type found very good solutions for the locomotion and rotation tasks. In the case of the level locomotion task, two different types of strategies were learnt by the insect instances. In the cases where the instance's primary spine is shorter than the length of its legs, the insect learns to use its legs to drive itself forward. When the spine is longer than the legs, however, a sidewinding gait very similar to that of the snakes is found. The insect instances all made use of their legs in the rotation task, however.

The control strategies found for the remainder of the tasks for each type, were very similar to those found for level locomotion for the respective types. In the case of task 2 (slippery locomotion), the animats were found to choose gaits which lift them from the ground and reduce the number of contacts points they have with it any given point in time. In the case of task 4 (unlevel locomotion), the speed of the animats' motions was seen to slightly increase.

For videos of the evolved control strategies, visit `http://www.arkitus.com/ERCS`.

4.2 Normalisation

The scores obtained from the 72 experiments detailed in Chapter 3 have been recorded and can be seen in Tables A.1, A.2 and A.3. Due to the fact that different quantities are measured in the task scenes, the score of an instance from one task cannot be directly compared to the score of that instance from another task.

In order to overcome this problem, the scores are normalised. The normalised fitness F'_{ijk} of the jth instance of the ith type at task k is set to be:

$$F'_{ijk} = \frac{F_{ijk} - \min_k}{\max_k - \min_k} \tag{4.1}$$

where F_{ijk} is the un-normalised fitness, $\min_k$ is the lowest fitness of any instance at task k and $\max_k$ is the highest fitness of any instance at task k:

$$\min_k = \min_{a,b} F_{abk} \tag{4.2}$$

$$\max_k = \max_{a,b} F_{abk} \tag{4.3}$$

This normalisation procedure ensures that all the fitnesses are now between 0 and 1, with the morphology with the normalised fitness of $F'_{ijk} = 1$ being the fittest morphology at task k. See Tables A.1, A.2 and A.3 and Figures C.1, C.3 and C.5 for the normalised scores for each experiment.

The resulting graphs can be analysed to identify trends in the data, many of which match our observations in the natural world.

Figure 4.2 displays the performances of type 0 (snake) instances at the range of tasks described in §3.4. The task the snakes perform worst at is number 3 (weighted locomotion), and the task these instances are most suited to is task number 5 (rough locomotion). However, the graphs show that the snake instances perform below average on most of the tasks (their scores are smaller than 0.5).

Figure 4.3 displays the performances of type 1 (starfish) instances at the same range of tasks. These instances perform extremely well at some tasks, and fairly poorly at others. The task the starfish are most suited to is number 1 (level rotation), and the task they perform worst at is number 4 (unlevel locomotion). The starfish instances' good performances at the level rotation task is likely to be due to the rotational symmetry of their morphologies.

Figure 4.4 displays the performances of type 2 (insect) instances at the same range of tasks. As with the type 0 instances, the task these instances are most suited to is number 5 (rough locomotion), and the task they perform worst at is number 3 (weighted locomotion).

One interesting observation that can be made is that the performance characteristics of type 0 (snake) and type 2 (insect) instances are very similar, with the type 2 instances performing only slightly better at some of the tasks. This is an indicator that, due to the experiments' relatively small number of iterations, the insect instances' control strategies were mainly driven by their primary spine. This would lead to similar performance characteristics, as the type 0 and type 2 instances share a central spine from which the legs extend outwards. It is expected that by increasing the number of iterations each genetic algorithm is allowed to execute, the difference between the type 0 and type 2 scores can be made more visible.

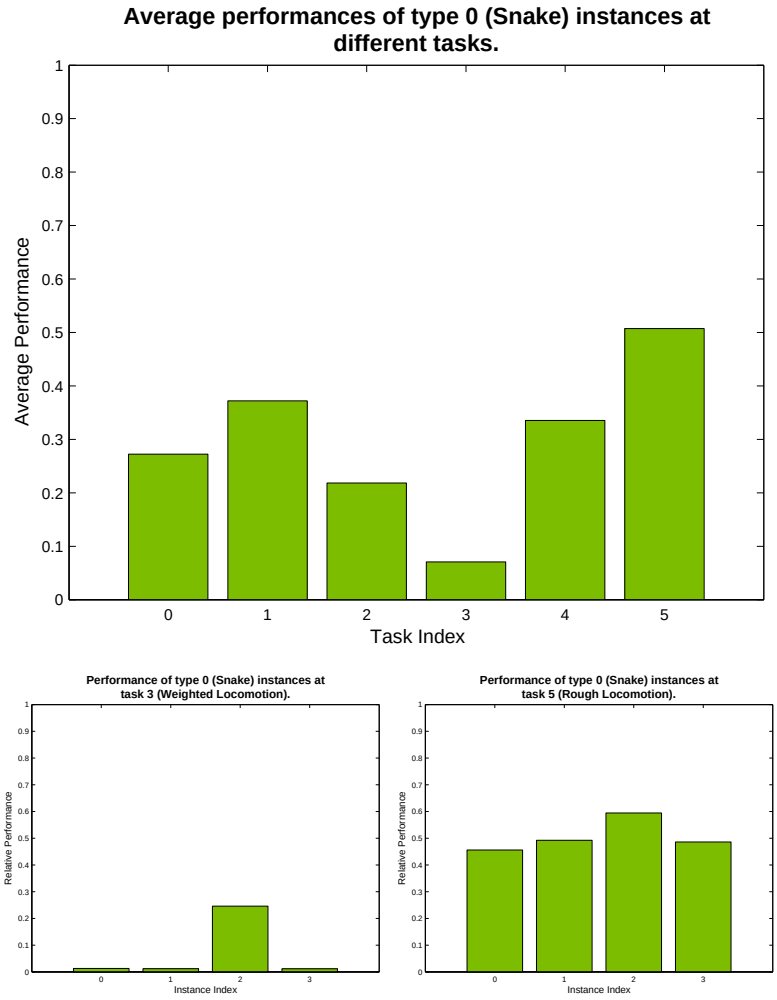

Figure 4.2: Top: Performances of type 0 (snake) instances at various tasks. Bottom left: the instances' performances at task 3 (worst). Bottom right: the instances' performances at task 5 (best).

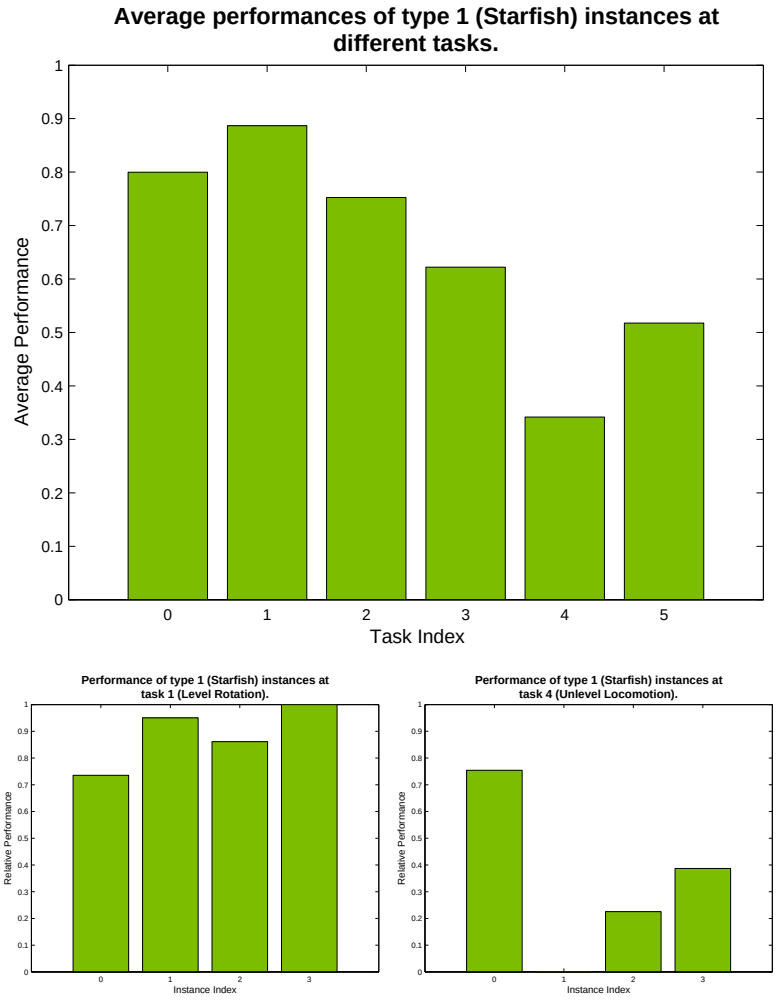

Figure 4.3: Top: Performances of type 1 (starfish) instances at various tasks. Bottom left: the instances' performances at task 1 (worst). Bottom right: the instances' performances at task 4 (best).

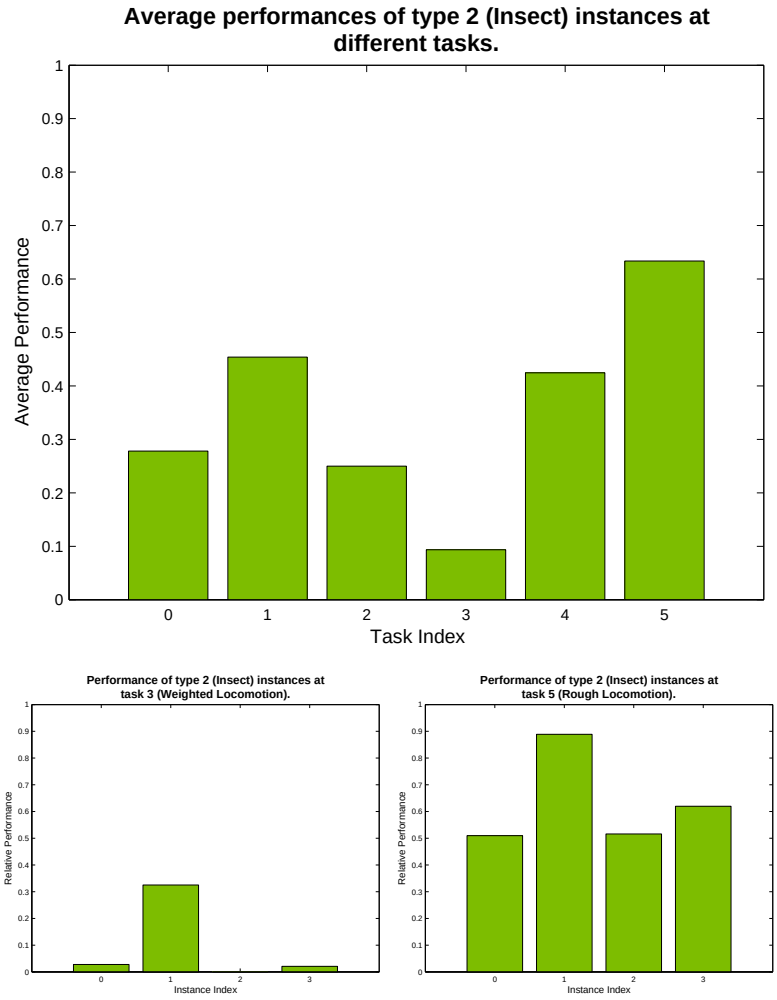

Figure 4.4: Top: Performances of type 2 (insect) instances at various tasks. Bottom left: the instances' performances at task 3 (worst). Bottom right: the instances' performances at task 5 (best).

On average, the type 1 (starfish) instances obtain the highest scores over the range of tasks, with the type 2 (insect) and type 0 (snake) instances coming in second and third place, respectively. In 5 out of the total of 6 tasks, the instance with the highest score at that task was a type 1 (starfish) instance. In only 1 task (unlevel locomotion), the highest ranking instance was a type 2 (insect) instance.

These results validate the first hypothesis presented in §3.1 (that there exists a correlation between the set of specified test tasks and the morphologies of the animats that excel at those tasks). The results show that each morphology has specific strengths and weaknesses, and that the best ranking morphologies at the tasks differ from task to task.

4.3 Specialisation or Generalisation

The next step was to calculate the variance of each morphology type's performances on the range of tasks. This value can be used to identify if a given morphology type is a specialist (it performs much better in one test than the others) or a generalist (it performs equally well at all tasks). Table 4.1 shows the results of these calculations.

The variance of each instance's score at the range of tasks.		
Type Index	Instance Index	Variance
0 (Snakes)	0	0.0371
	1	0.0491
	2	0.0353
	3	0.0629
1 (Starfish)	0	0.0252
	1	0.2109
	2	0.0911
	3	0.0582
2 (Insects)	0	0.1226
	1	0.0990
	2	0.0441
	3	0.0680

Table 4.1: The variance of each instance's score at the range of tasks.

The average variance for type 0 instances is 0.0461, for type 1 instances is 0.09635 and for type 2 instances is 0.083425. The large variance in the type 1 instances' performances at the different tasks can be seen in the results above. These numbers suggest that the type 1 instances can bee seen as specialists and the type 0 instances can be seen as generalists.

These results validate the second hypothesis presented in §3.1 (that some morphologies will be specialists, and others will be generalists).

4.4 Feature-aptitude Correlation

Values for a number features of the instances' morphologies are calculated (or chosen), including values for complexity and size. Each feature is plotted against the scores obtained by an instance of a morphology with the corresponding feature-value, from each task. These plots can be used to identify if correlations can be found between a morphology's features and its ability to perform different tasks.

Tables B.1, B.2 and B.3 show the feature-values for each instance.

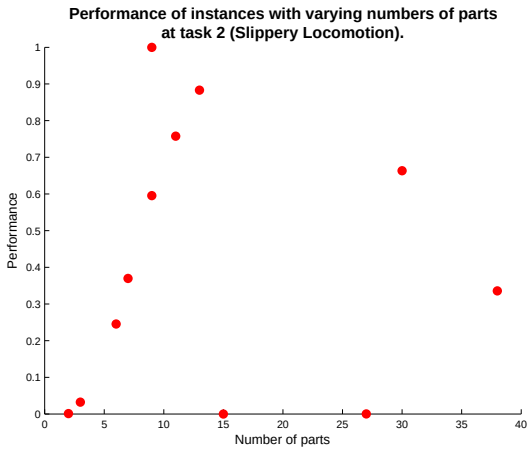

Figure 4.5: Performance of instances of varying size at task 2. A strong linear correlation can be seen between the instances' performances and their number of parts (in instances with a small number of parts). However, the instances with more than 15 parts have highly scattered performances.

Overall, no significant correlations were found between the animats' features and their performances at the different tasks. Figure 4.5 shows the performance of instances with varying numbers of parts at task 2 (slippery locomotion), and Figure 4.6 shows the performance of instances with varying spine lengths at task 5. Figures C.27 to C.32, C.33 to C.38 and C.39 to C.44 show the rest of the feature vs. aptitude plots.

These results invalidate the third hypothesis presented in §3.1 (that there exists a correlation between the features in the animats' morphologies and the tasks they specialise in).

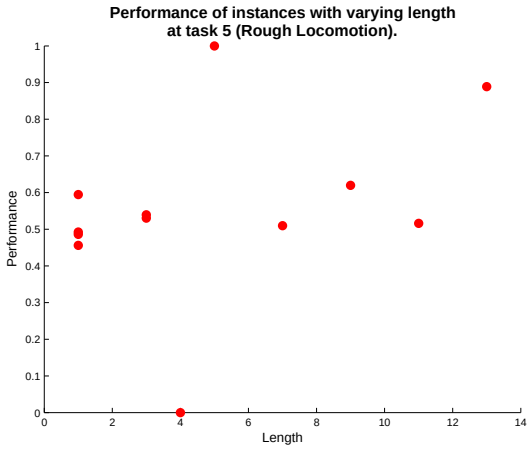

Figure 4.6: Performance of instances of varying size at task 5. The horizontal correlation suggests that the morphologies' lengths are not capable of explaining the variance in the performances.

4.5 Morphology-task Correlation

Finally, the performance of each of the instances is plotted in a high-dimensional space in which each dimension represents a single task. It is of interest to see if the positions of the instances of any given morphology are seen to cluster in this high-dimensional space.

To make the visualisation of this high-dimensional space easier, the performances are plotted, in threes, in a three dimensional space and then projected into a two dimensional plane. In this work, Simplices are used to make these projections. Figure 4.7 displays the plane (white triangle) onto which the fitnesses are projected (red dot).

Additionally, the radius of the dots which result from these projections are adjusted to show the points' original distances from the origin. Dots with larger radii perform well across the three tasks being considered (on average), whereas the dots with smaller radii have poorer average performances.

All possible combinations of three tasks out of the total of six are examined. The resulting plots can be seen in Figures 4.8 to C.26.

Figure 4.8 shows the relative performance of the 12 morphology instances at tasks 0 (level locomotion), 1 (level rotation) and 2 (slippery locomotion). It can be clearly seen that the type 2 (starfish) instances cluster in the centre of the Simplex, which shows that they have performed equally well at the three tasks. The large size of the circles shows that they have performed *well* at the tasks, not poorly. Two of the type 0 (snake) instances have specialised in the level rotation

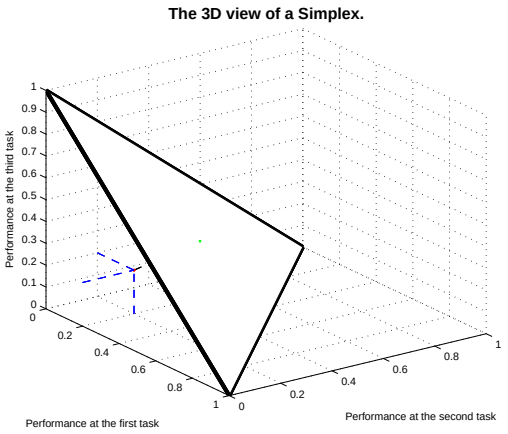

Figure 4.7: The 3D view of a Simplex. The fitness (red dot) is projected onto the plane (white triangle). The projections are then copied from the plane onto a two dimensional plot.

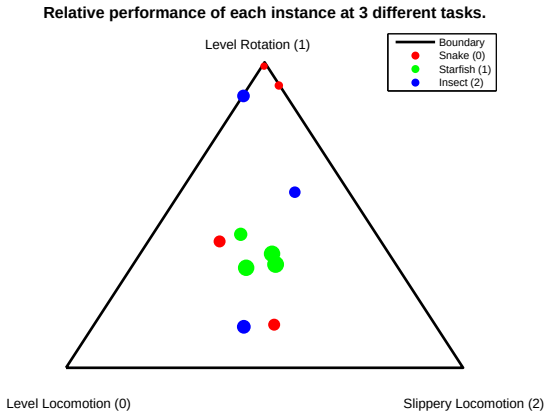

Figure 4.8: Relative performance of each instance at tasks 0, 1 and 2.

task, as can be seen by their location at the top of the Simplex. No particular structure can be seen in the placement of the the type 2 (insect) instances in this Simplex.

Figure 4.9 shows the relative performance of the instances at tasks 2 (slippery locomotion), 4 (unlevel locomotion) and 5 (rough locomotion). The type 0 (snake) and type 2 (insect) instances have clustered somewhere between the unlevel locomotion and rough locomotion corners, which shows that they had performed very poorly at the slippery locomotion task. On the other hand, one type 2 (starfish) instance, has performed extremely well at the slippery locomotion task without doing well at the other two. The large radius of the instance's projection on the Simplex also shows that it was one of the best performers at that task. This data suggests that in addition to generalist instances, some instances are only good at

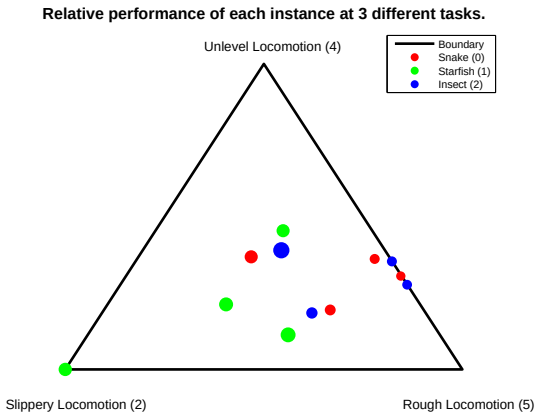

Figure 4.9: Relative performance of each instance at tasks 2, 4 and 5.

either slippery locomotion, or rough and unlevel locomotion.

These results further help to validate the first hypothesis presented in §3.1 (that there exists a correlation between the set of specified test tasks and the morphologies of the animats that excel at those tasks).

4.6 Chapter Summary

In this chapter, the results of the experiments described in the Chapter 3 were presented. A subset of these results was discussed in detail and analysed. Additionally, the hypotheses put forward in Chapter 3 were re-examined in light of the experimentation results.

5. Conclusion

This project has successfully provided a framework which has allowed the three hypotheses presented in §3.1 to be evaluated. The experimentation results have validated the hypothesis that correlations exist between the set of specified test tasks and the morphologies of the animats that excel at those tasks. In addition, it has been shown that some morphology types tend to specialise in a small subset of the tasks, whereas others have good performances over the entire range of tasks.

The third hypothesis, that a correlation exists between the features in the animats' morphologies and the tasks they specialise in, was invalidated. Our results did not find any such correlation, however by adding more tests and more animats the correlations may become more apparent. It may also have been beneficial to choose a more biologically representative set of features for this analysis.

The simulations have shown that starfish morphologies are consistently the strongest across the range of tasks. However, very few research programs currently employ starfish-like morphologies for their robots. The starfish has several properties (rotational symmetry being the most prominent) which makes it a highly versatile choice for many lines of research. It is hoped that this project will spark more investigation on the suitability of starfish morphologies for robotics.

The goals set out for the software implementation have also been met. The application has proven to be a reliable and efficient framework for robotics simulations, and the algorithms have allowed control strategies to be quickly found for a large range of morphologies.

The main reason for this success was the use of pattern generators (the spine controllers) to drive the animats' motion. Not only did this approach dramatically reduce the amount of time that was required to train the animats, it also ensured that only realistic gaits were found. These results echo recent findings in biologically-inspired robotics and confirm that these pattern generators may very well be the key to complex locomotion in the animal kingdom.

However, the need for more high-level, hierarchical pattern generators also became apparent. The animats with more complex morphologies (such as the insect-like morphologies with large numbers of legs) did not manage to make use of all of their limbs effectively, and only used their central spines to drive their motion. It is unlikely that such animats would make the move from sidewinding to the typical insect gait unless they were given a way to experiment with motions on all their limbs simultaneously. These results are seemingly at odds with the idea that adhoc control structures are sufficient for complex locomotion (so long as

they are given enough time to be optimised).

5.1 Implementation Issues

In hindsight, there are several areas in the project's design in which more structured decisions should have been made. These decisions could expand the project's experimentation scope and help produce more significant results.

Throughout the project's implementation, a large amount of time was spent on resolving issues related to the multi-threaded aspect of the project. The current implementation uses threads to separate the genetic algorithms from the routines which run and support the physical simulations. However, since the need for a multi-threaded setup was not identified from the beginning, a disproportionate amount of time was spent on resolving threading-related bugs. The fact that these family of bugs would almost always result in failures in the PhysX library (which cannot be easily debugged), only made matters worse.

Additionally, several issues with the physics engine were never completely resolved. In particular, the PhysX library consistently crashes when an animat with very few limbs is instantiated. Again, due to the fact that the PhysX routines are not accessible (they are closed source), debugging and fixing these issues has proven to be extremely difficult.

5.2 Future Work

There are a number of places in which extensions could be made to the project in the future. More advanced operators could be chosen for the genetic algorithms, for instance. Crossover and mutation operators which are aware of the underlying chromosome structure (the fact that the genes control the spines as groups of 4) could increase the rate at which good control strategies are found.

Due to time constraints, the number of generations used for the experiments was set to a relatively low number. Increasing this value is expected to result in better control strategy solutions for the animats, whilst making the differences between their abilities more visible. Additional measures could be added to decrease the probability of gene mutation as the generation count increases, as the mutation operator was found to be too disruptive towards the later generations.

The genetic algorithms can be extended to allow for adversarial chromosome evaluation, in which two different phenotypes are pitted against each other in a

single test. It would be of interest to see, for example, how the control strategies evolve as they compete with other chromosomes in the population.

The graphical models used to describe the animat morphologies could be extended. These additions could allow morphologies to be created which are not seen in nature. Examples of such morphologies include spherical, polyhedral or tree-shaped animats.

In order to accommodate for more unusual morphologies, additions would have to be made to allow the morphologies to incorporate a wider range of parts. The morphology specification language could be extended to allow for prismatic joints and linear motors. It would also be of interest to examine morphologies with loops of parts and with unactuated joints.

The list of tasks which the animats are trained to complete could also be expanded. It remains to be seen if the spinal control strategies used in this project can be used to learn gaits for flight and swimming. Whilst these types of simulations currently cannot be performed in real-time, particle based physics simulations can be used instead as approximations.

One possible addition which was attempted for this project, but eventually abandoned due to time constraints, is the use of a secondary GA to find the fittest morphologies for a given task. This would involve representing the graphical models as chromosomes, defining crossover and mutation operators on these chromosomes and using the score of the best control strategy for each model as its fitness. This technique could be used to confirm if properties such as symmetry or simplicity are due to the selection pressure on the animats.

Many of these additions require more computational resources and more time. For this reason, it would be highly desirable if the physics simulations could be accelerated with dedicated hardware. The physics SDK used for this project, PhysX, can be used with the Ageia PhysX card to do just that [4].

Appendix A. Results

Performances of type 0 (Snake) instances			
Instance Index (j)	Task Index (k)	Score (F_{0jk})	Normalised Score (F'_{0jk})
0	0	0.47172	0.0017288
	1	75.6918	0.31649
	2	0.50885	0.00095507
	3	0.47168	0.012986
	4	-1.44	0.20254
	5	0.35126	0.45596
1	0	0.45643	0
	1	77.8536	0.41671
	2	0.90013	0.032364
	3	0.45643	0.011828
	4	-1.0963	0.29931
	5	0.52379	0.49246
2	0	5.364	0.55513
	1	81.1466	0.56938
	2	3.5525	0.24528
	3	3.5407	0.24597
	4	-1.4334	0.2044
	5	1.0058	0.59443
3	0	5.1608	0.53198
	1	72.8827	0.18626
	2	7.9135	0.59534
	3	0.45553	0.011761
	4	0.09736	0.63534
	5	0.4936	0.48608

Table A.1: The scores of each instance from type 0 (Snakes) at each task.

Performances of type 1 (Starfish) instances			
Instance Index (j)	Task Index (k)	Score (F_{1jk})	Normalised Score (F'_{1jk})
0	0	5.5033	0.57072
	1	84.7318	0.73559
	2	5.1007	0.36955
	3	5.7947	0.41709
	4	0.52014	0.75436
	5	0.70343	0.53046
1	0	7.9285	0.84497
	1	89.36	0.95038
	2	12.9547	1
	3	7.7802	0.56782
	4	-2.1595	0
	5	-1.8039	0
2	0	9.2994	1
	1	87.4452	0.86138
	2	9.9343	0.75755
	3	13.4732	1
	4	-1.3585	0.22548
	5	2.9228	1
3	0	7.3797	0.78292
	1	90.4353	1
	2	11.5001	0.88324
	3	6.9399	0.50402
	4	-0.78492	0.38696
	5	0.74633	0.53954

Table A.2: The scores of each instance from type 1 (Starfish) at each task.

Performances of type 2 (Insects) instances			
Instance Index (j)	Task Index (k)	Score (F_{2jk})	Normalised Score (F'_{2jk})
0	0	1.4077	0.10758
	1	88.4142	0.9063
	2	0.49695	0
	3	0.6663	0.027761
	4	-1.1618	0.28085
	5	0.60596	0.50984
1	0	7.9324	0.84541
	1	73.9355	0.23507
	2	8.7624	0.66348
	3	4.5838	0.32516
	4	1.3927	1
	5	2.3981	0.889
2	0	0.46803	0.0013119
	1	68.86	0
	2	0.49728	2.6971×10^{-5}
	3	0.30062	0
	4	-1.4521	0.19913
	5	0.63579	0.51615
3	0	1.8519	0.1578
	1	83.411	0.67435
	2	4.6793	0.33572
	3	0.57402	0.020756
	4	-1.3849	0.21806
	5	1.1254	0.61974

Table A.3: The scores of each instance from type 2 (Insects) at each task.

Appendix B. Animat Features

Number of parts for each instance		
Type Index	Instance Index	Number of parts
0 (Snakes)	0	2
	1	3
	2	6
	3	9
1 (Starfish)	0	7
	1	9
	2	11
	3	13
2 (Insects)	0	15
	1	30
	2	27
	3	38

Table B.1: The number of parts each morphology instance has.

Number of spines for each instance		
Type Index	Instance Index	Number of parts
0 (Snakes)	0	1
	1	1
	2	1
	3	1
1 (Starfish)	0	3
	1	4
	2	5
	3	3
2 (Insects)	0	7
	1	13
	2	11
	3	9

Table B.2: The number of spines each morphology instance has.

Length for each instance		
Type Index	Instance Index	Length
0 (Snakes)	0	2
	1	3
	2	6
	3	9
1 (Starfish)	0	3
	1	3
	2	3
	3	5
2 (Insects)	0	3
	1	6
	2	5
	3	5

Table B.3: The length of each morphology instance.

Appendix C. Plots and Graphs

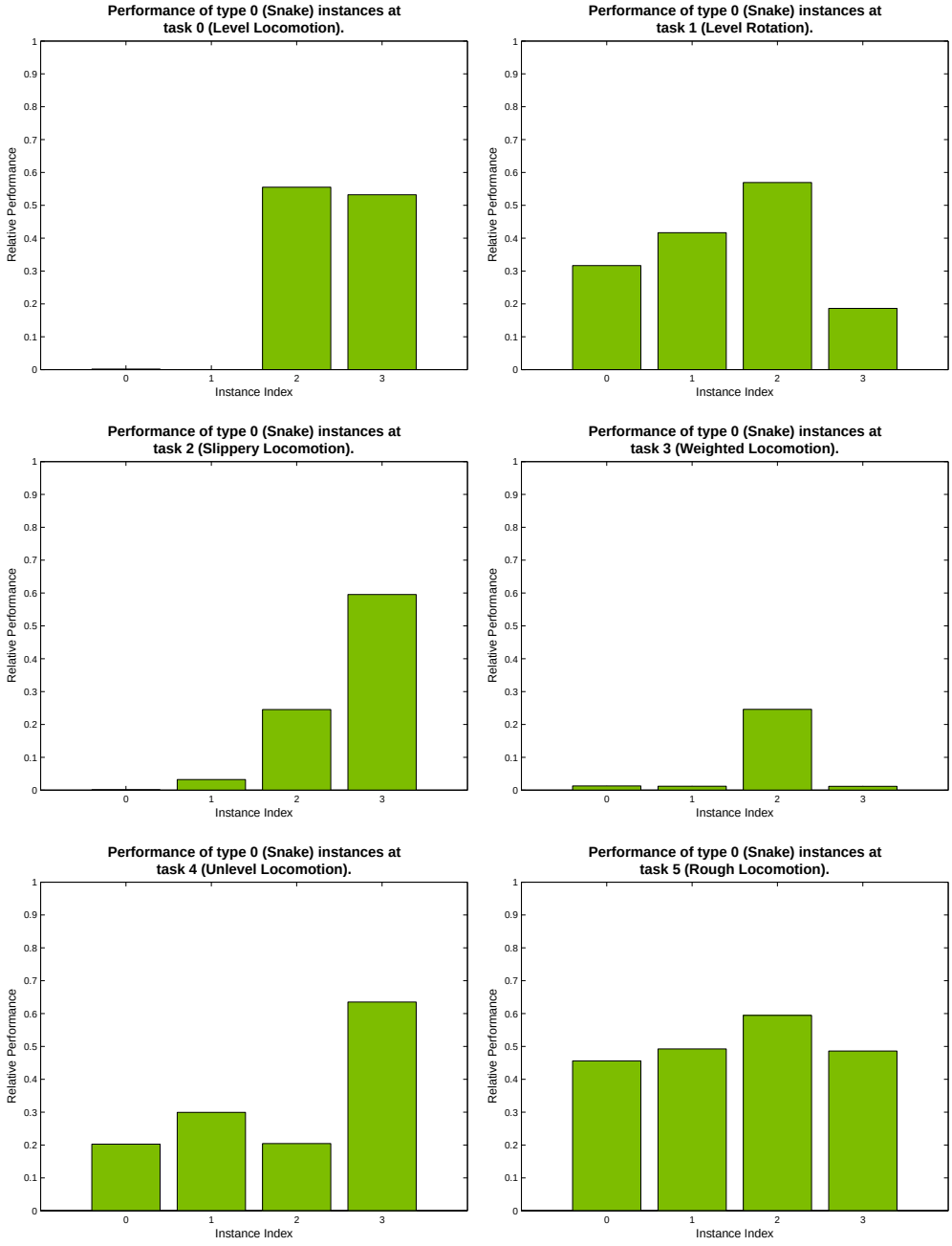

Figure C.1: Performances of type 0 (Snake) instances at various tasks.

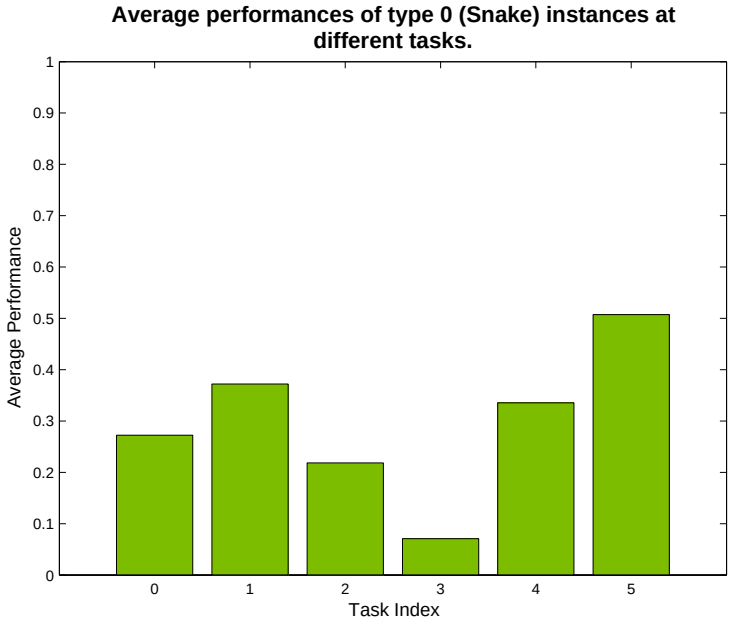

Figure C.2: Average Performance of type 0 (Snake) instances at various tasks.

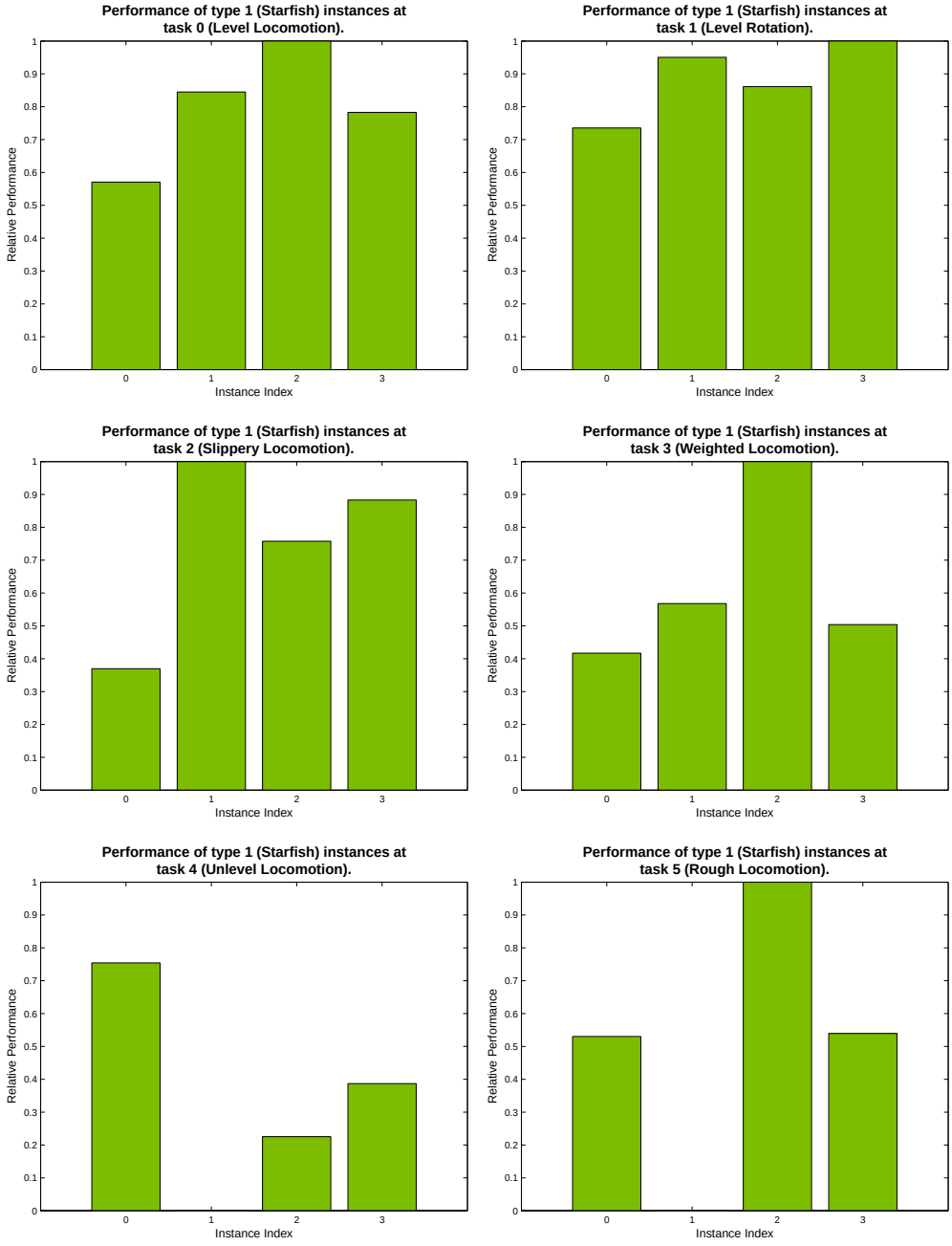

Figure C.3: Performances of type 1 (Starfish) instances at various tasks.

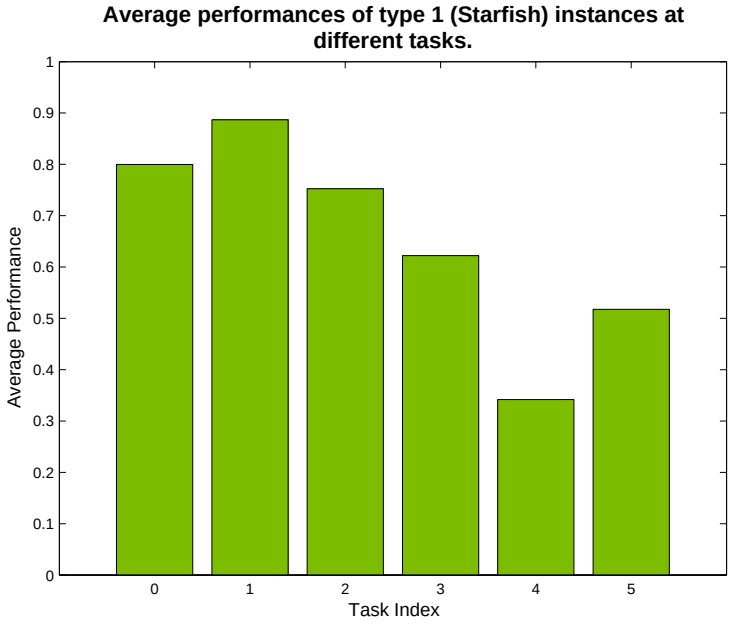

Figure C.4: Average Performance of type 1 (Starfish) instances at various tasks.

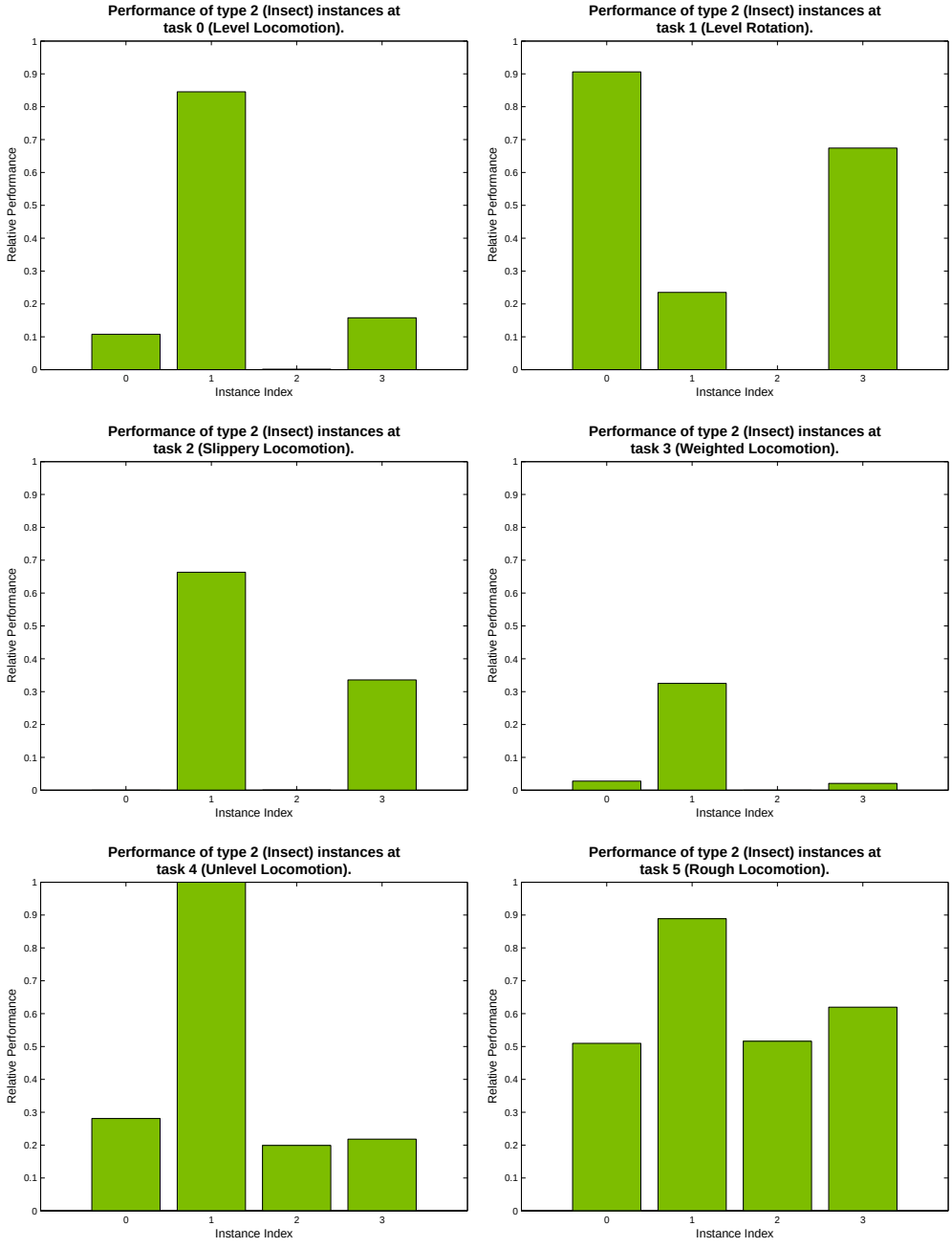

Figure C.5: Performances of type 2 (Insect) instances at various tasks.

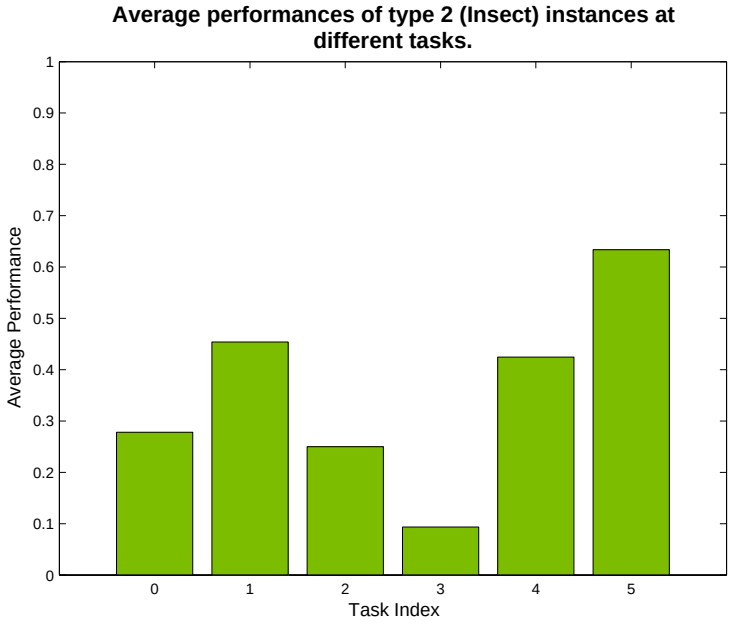

Figure C.6: Average Performance of type 2 (Insect) instances at various tasks.

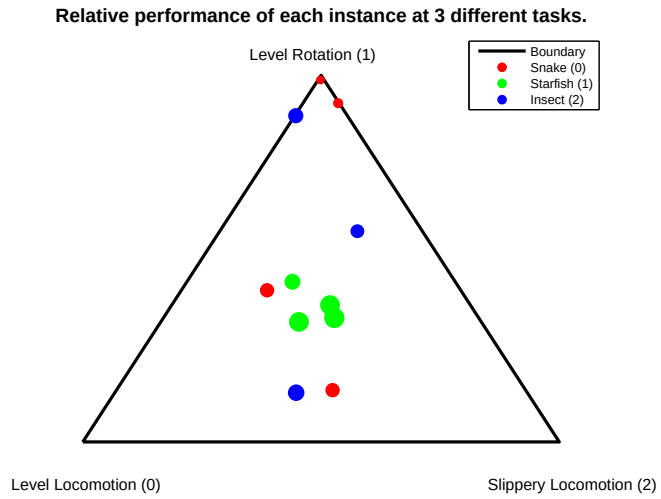

Figure C.7: Relative performance of each instance at tasks 0, 1 and 2.

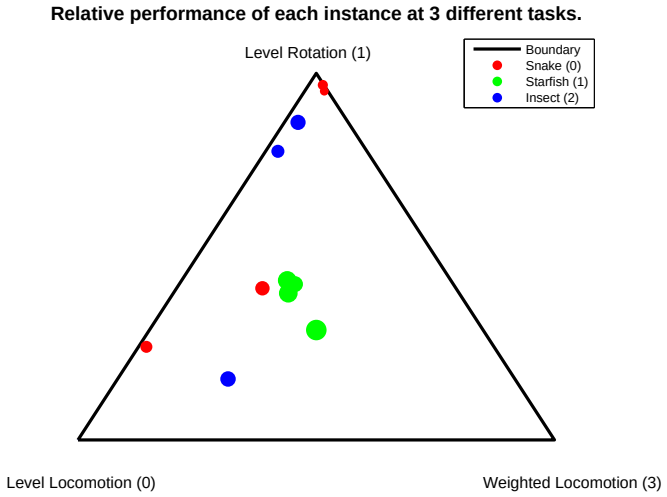

Figure C.8: Relative performance of each instance at tasks 0, 1 and 3.

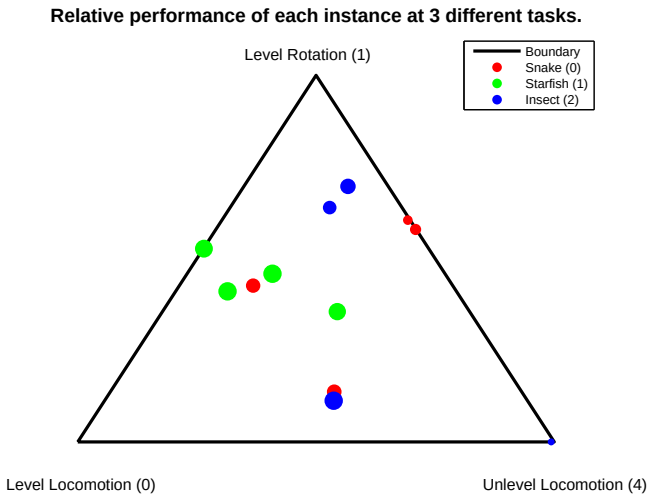

Figure C.9: Relative performance of each instance at tasks 0, 1 and 4.

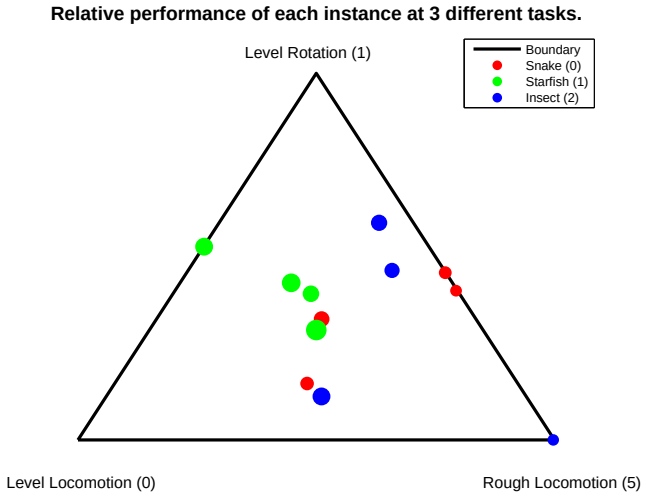

Figure C.10: Relative performance of each instance at tasks 0, 1 and 5.

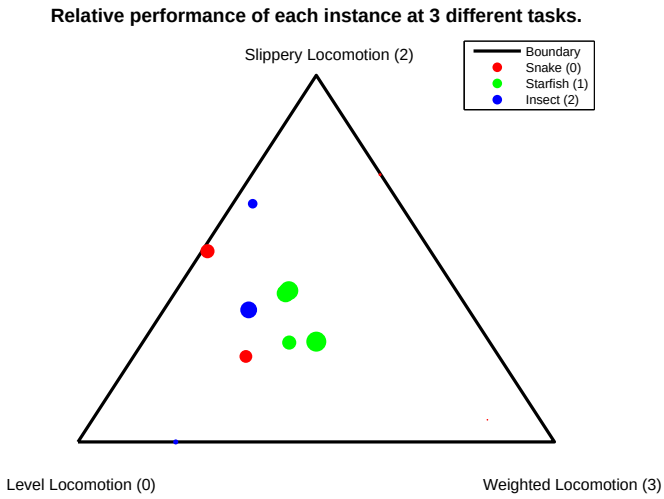

Figure C.11: Relative performance of each instance at tasks 0, 2 and 3.

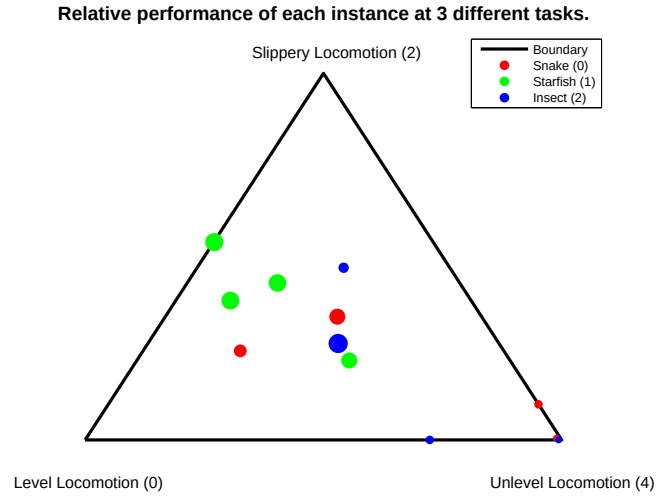

Figure C.12: Relative performance of each instance at tasks 0, 2 and 4.

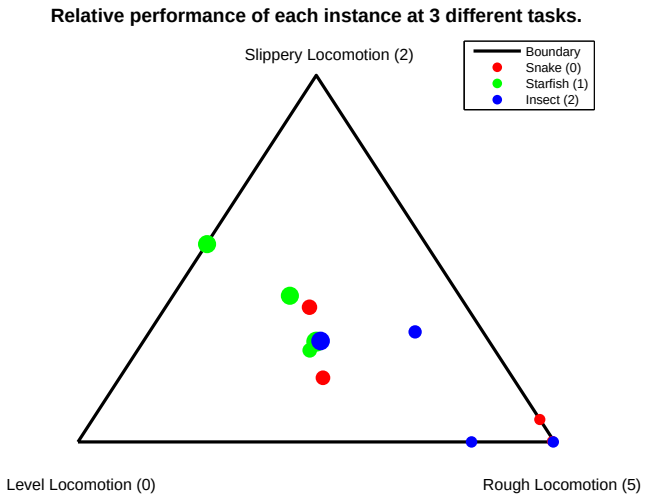

Figure C.13: Relative performance of each instance at tasks 0, 2 and 5.

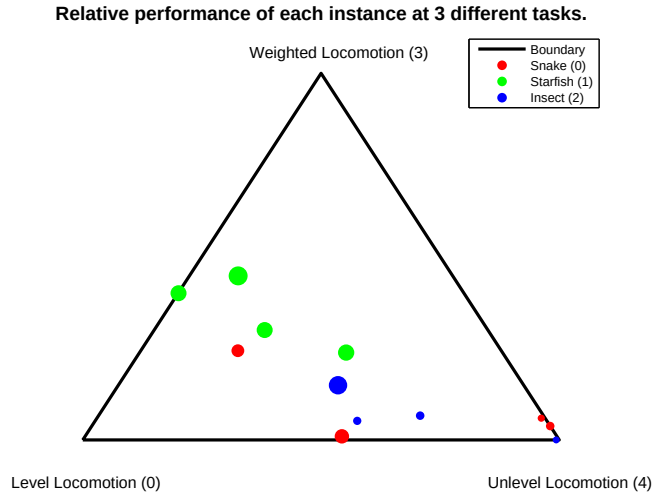

Figure C.14: Relative performance of each instance at tasks 0, 3 and 4.

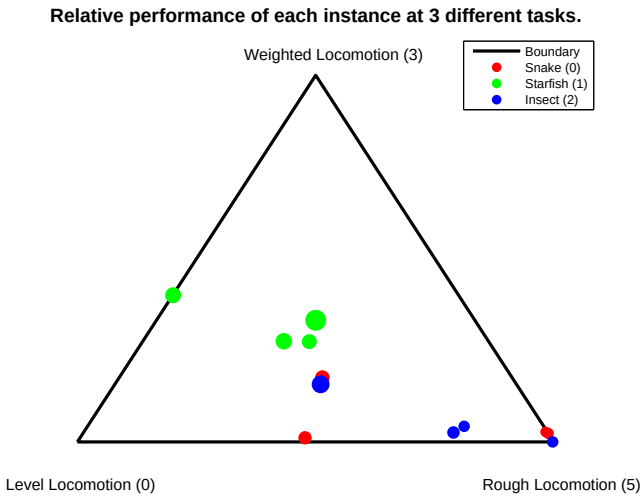

Figure C.15: Relative performance of each instance at tasks 0, 3 and 5.

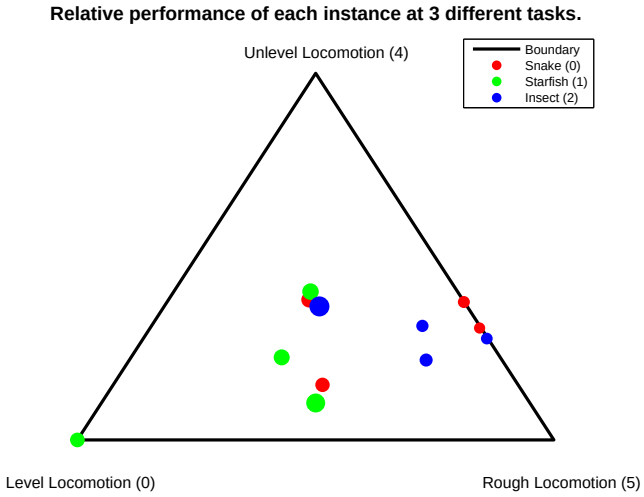

Figure C.16: Relative performance of each instance at tasks 0, 4 and 5.

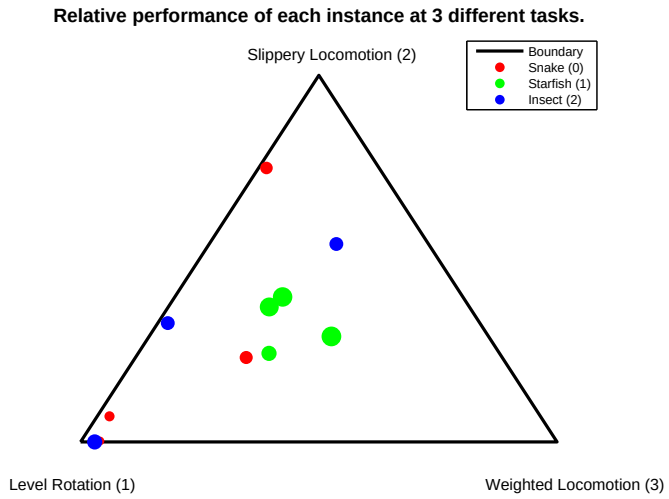

Figure C.17: Relative performance of each instance at tasks 1, 2 and 3.

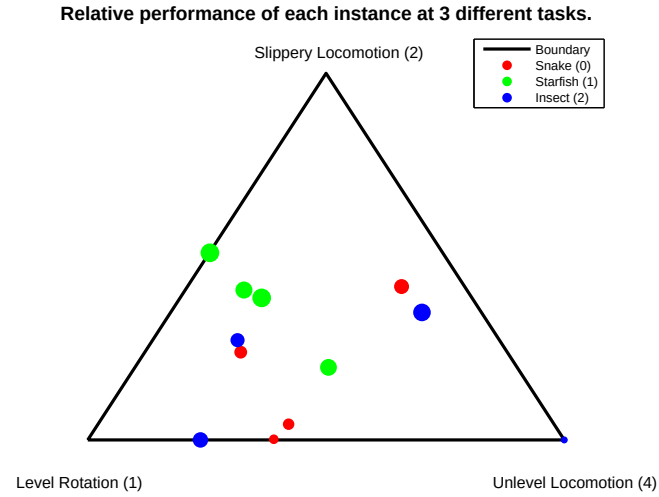

Figure C.18: Relative performance of each instance at tasks 1, 2 and 4.

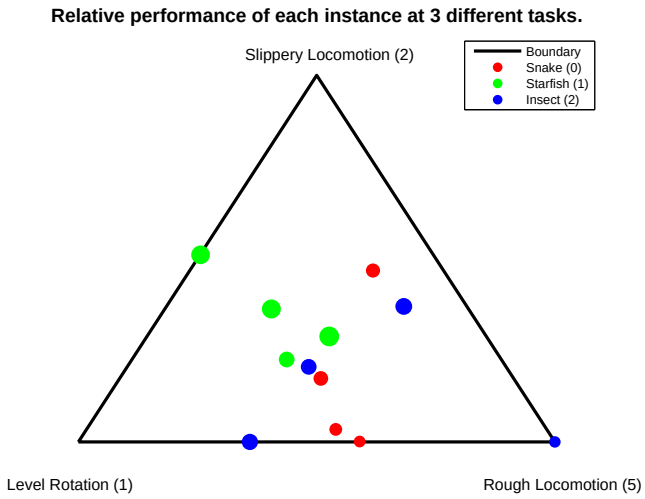

Figure C.19: Relative performance of each instance at tasks 1, 2 and 5.

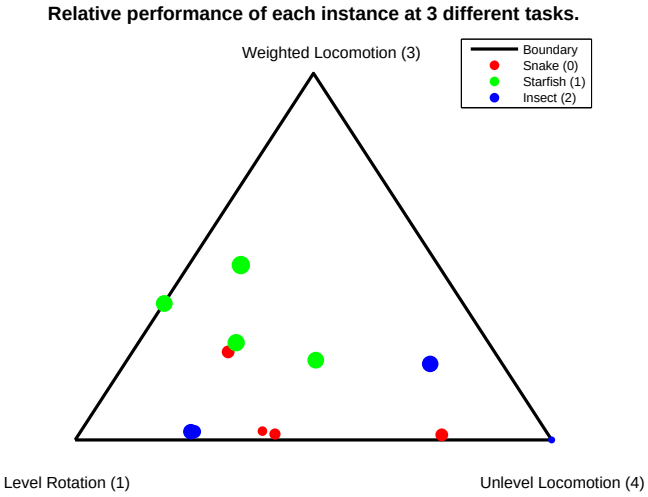

Figure C.20: Relative performance of each instance at tasks 1, 3 and 4.

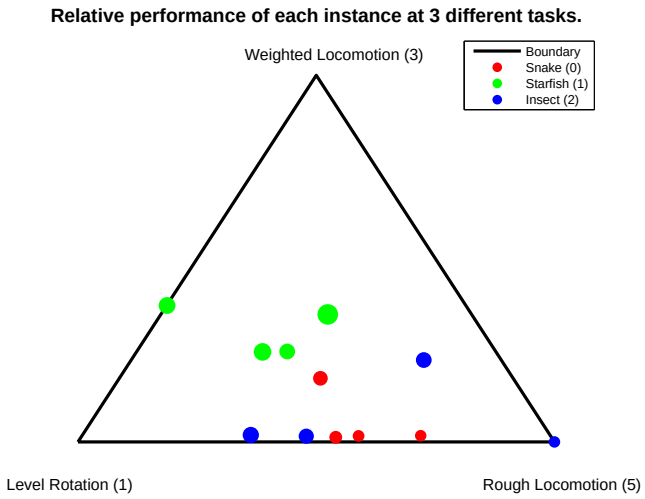

Figure C.21: Relative performance of each instance at tasks 1, 3 and 5.

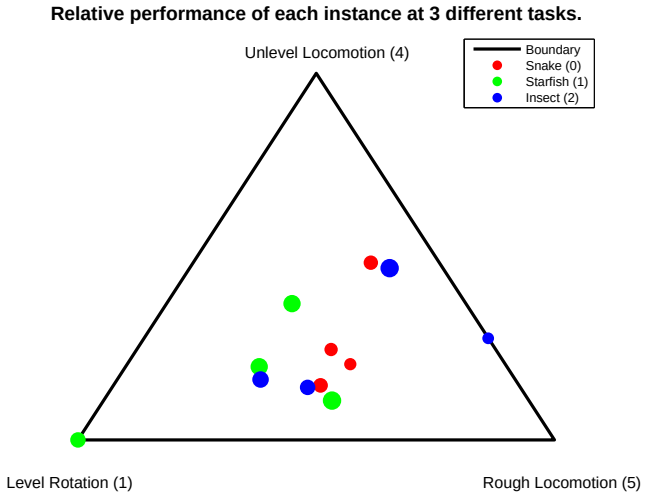

Figure C.22: Relative performance of each instance at tasks 1, 4 and 5.

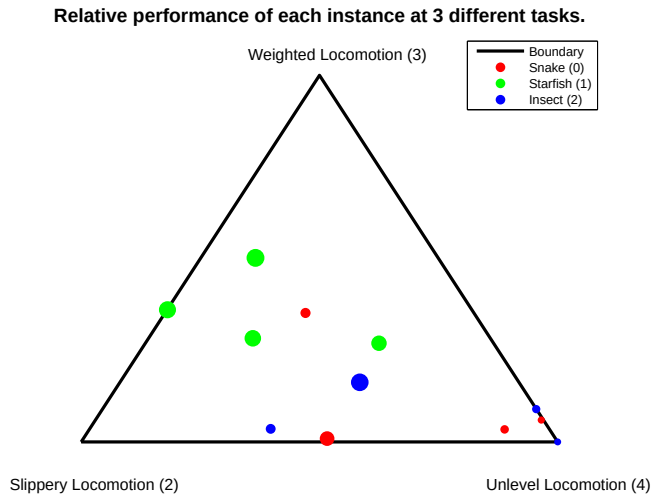

Figure C.23: Relative performance of each instance at tasks 2, 3 and 4.

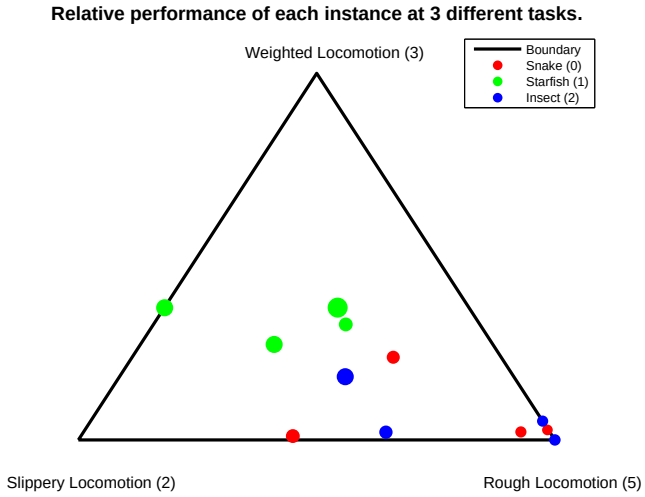

Figure C.24: Relative performance of each instance at tasks 2, 3 and 5.

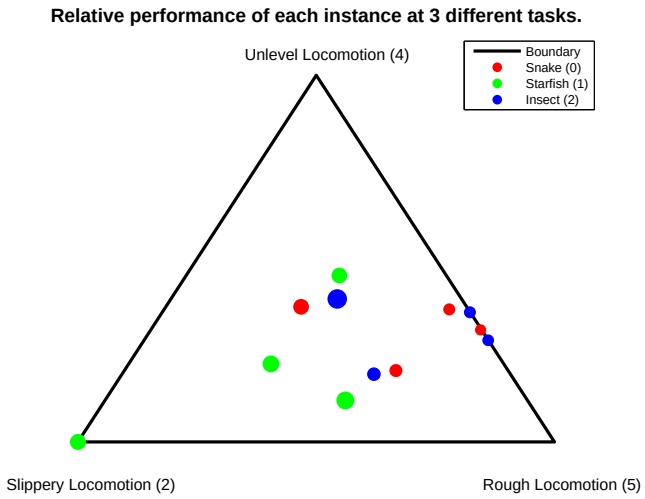

Figure C.25: Relative performance of each instance at tasks 2, 4 and 5.

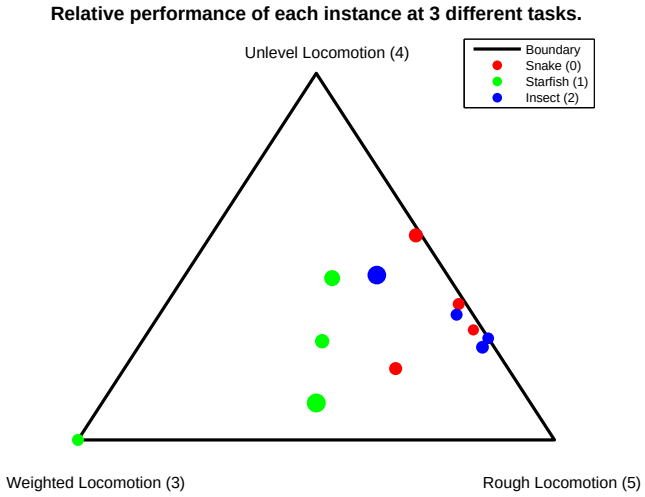

Figure C.26: Relative performance of each instance at tasks 3, 4 and 5.

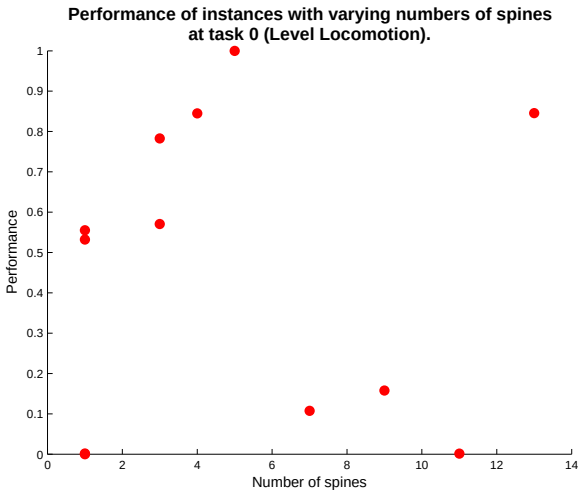

Figure C.27: Performance of instances of varying size at task 0.

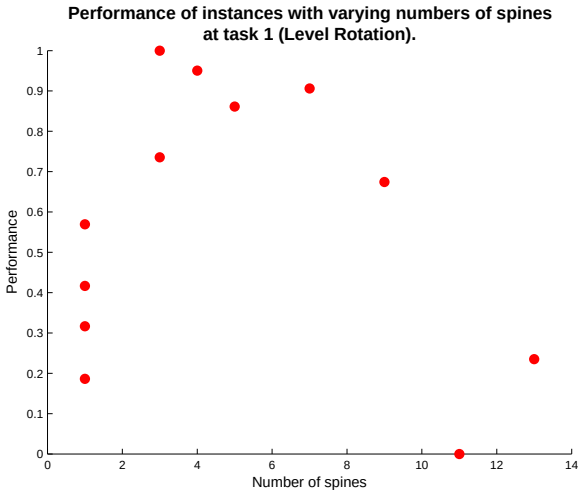

Figure C.28: Performance of instances of varying size at task 1.

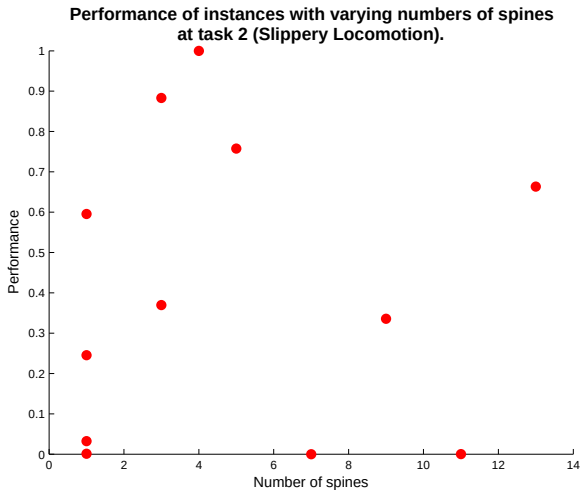

Figure C.29: Performance of instances of varying size at task 2.

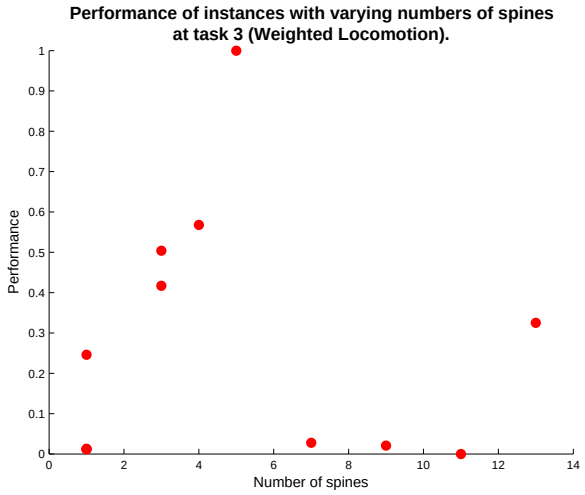

Figure C.30: Performance of instances of varying size at task 3.

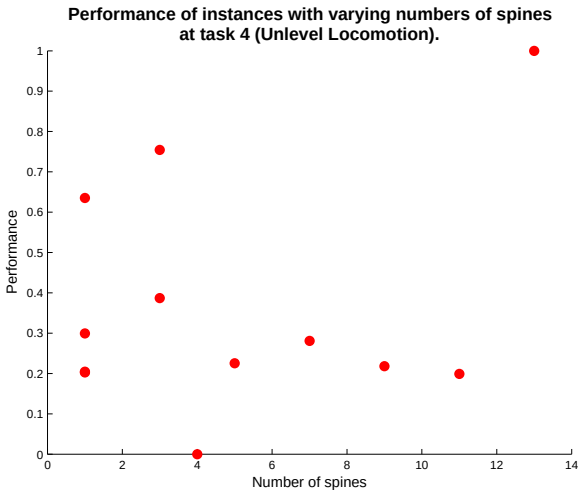

Figure C.31: Performance of instances of varying size at task 4.

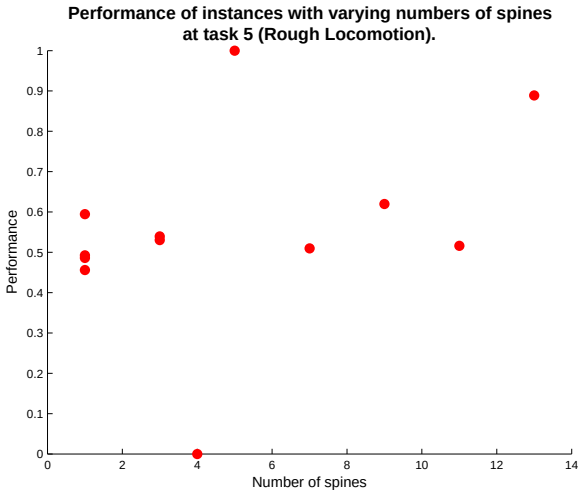

Figure C.32: Performance of instances of varying size at task 5.

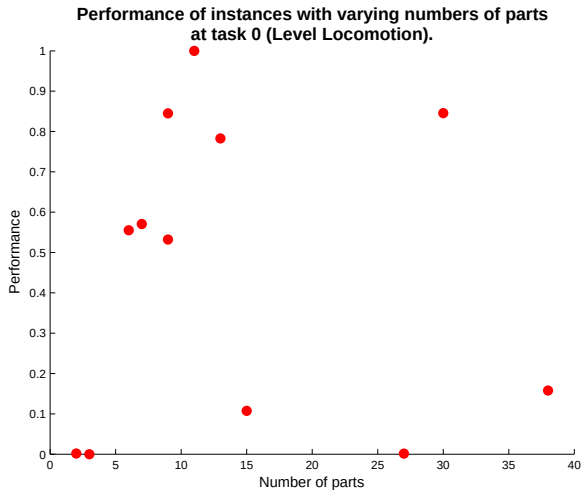

Figure C.33: Performance of instances of varying size at task 0.

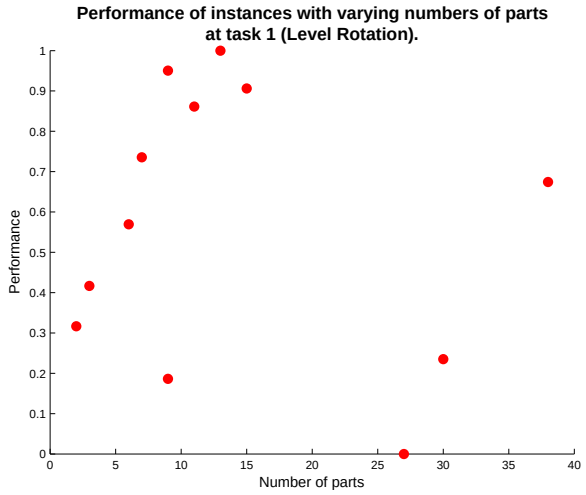

Figure C.34: Performance of instances of varying size at task 1.

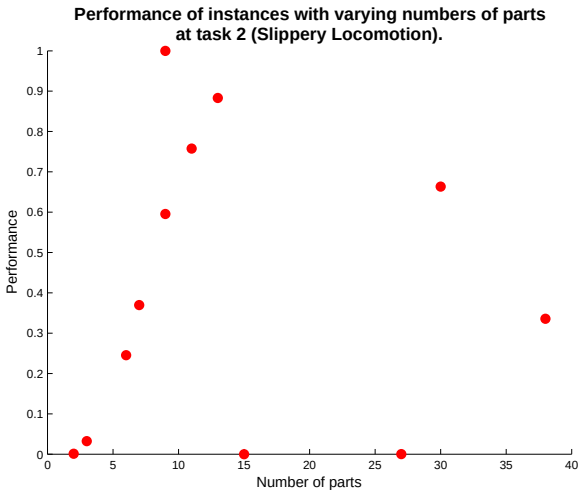

Figure C.35: Performance of instances of varying size at task 2.

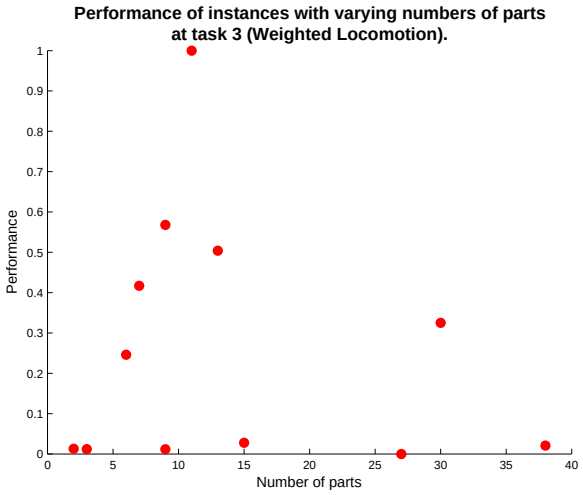

Figure C.36: Performance of instances of varying size at task 3.

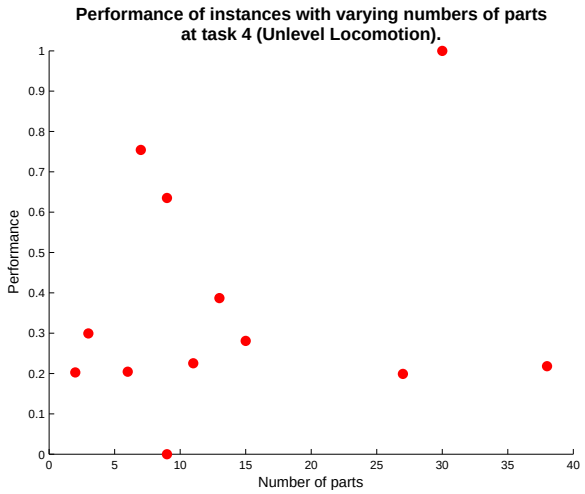

Figure C.37: Performance of instances of varying size at task 4.

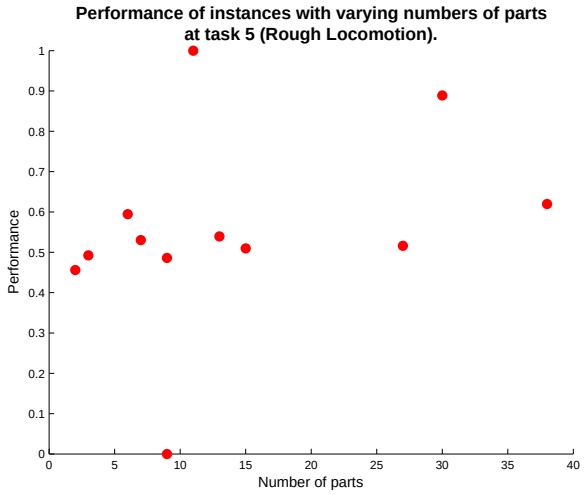

Figure C.38: Performance of instances of varying size at task 5.

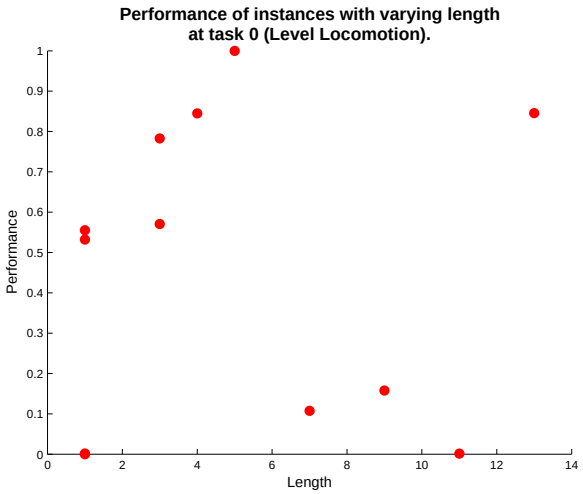

Figure C.39: Performance of instances of varying size at task 0.

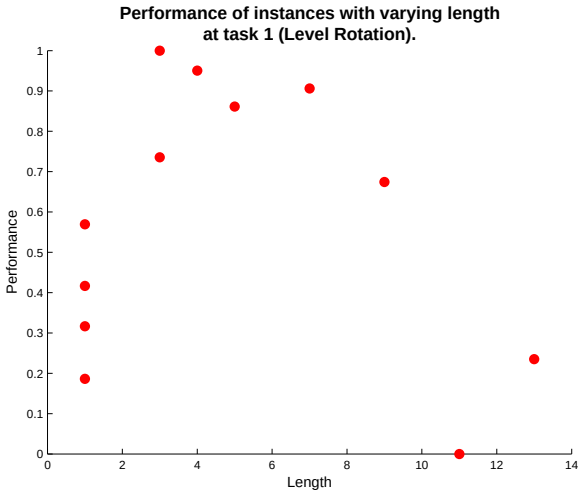

Figure C.40: Performance of instances of varying size at task 1.

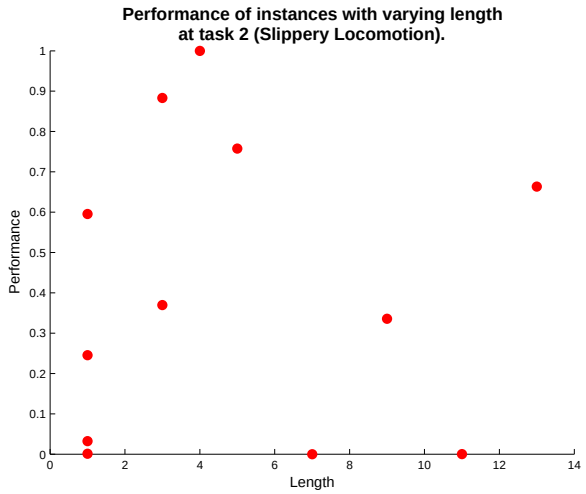

Figure C.41: Performance of instances of varying size at task 2.

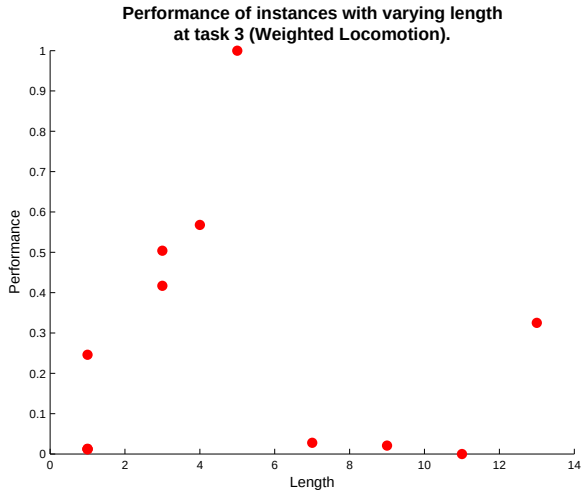

Figure C.42: Performance of instances of varying size at task 3.

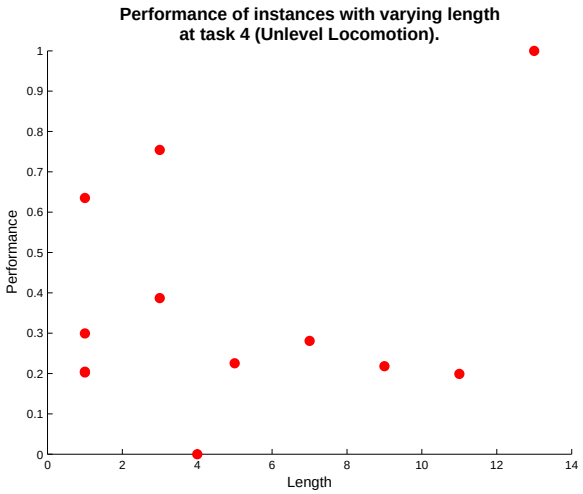

Figure C.43: Performance of instances of varying size at task 4.

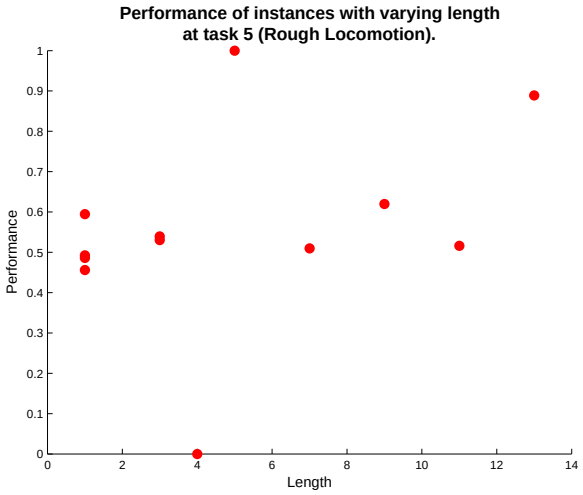

Figure C.44: Performance of instances of varying size at task 5.

Appendix D. Code

Filename: `settings.txt`
Location: `/settings/`
Lines: 9-17, 89-93, 118-121, 137
Description: Segments of the settings file, which is loaded as the application initialises. The settings are made globally available by the `Settings` static class.

```
version<3>                    15 01 2009
fullscreen<1>                 0
screenWidth<1>                640
screenHeight<1>               480
screenReportWidth<1>          1024
screenReportHeight<1>         768
...
GAPopulationSize<1>           50
GAIterations<1>               10
GASelectionPower<1>           5
GAElitism<1>                  20
GAPhenotypeTimeUnit<1>        3
...
spineRaiseAmount<1>           50
spineSwingAmount<1>           30
spineSpring<1>                20
spineMaxFrequency<1>          1
...
// Type    Instance  Task
// 0 Snakes
// 1 Starfish
// 2 Insects
// 3 Test
//           -1 Viewing
//            0 Level Locomotion
//            1 Rotation
//            2 Slippery Locomotion
//            3 Weighted Locomotion
//            4 Unlevel Locomotion
//            5 Rough Locomotion
expIndex<3>                   1 0 -1
```

Filename: `run.py`
Location: `/`
Lines: 44-92
Description: The python script used to run all the experiments. The script is designed to automatically detect and handle potential crashes.

```
# Copy the latest build to this directory
shutil.copy( '../release/Simulation.exe', 'Simulation.exe' )

# For each morphology type
for i in range(0,numMorphologyTypes):

  # For each morphology instance
  for j in range(0,numMorphologyInstances):

    # For each tasks
    for k in range(0,numTasks):

      count = k + j * numTasks + i * numTasks * numMorphologyInstances
      if count < int(startExperiment)-1:
        continue

      # Flag to capture crashes
      succeeded = False

      # Repeat until we've completed this experiment
      while not succeeded:

        # Print some useful information to the output stream
        percentage = count / totalExperiments * 100
        print('Experiment index: %d %d %d (%f percent)' % (i, j, k, percentage))

        # Start the experiment and wait for its termination
        command = 'Simulation.exe %d %d %d' % (i, j, k)

        # Copy the solutions to the folder too
        try:
          for l in range(0, numGenerations):
            sourcename = 'storage/%s/%d.chrom' % (taskForIndex(k),l)
            targetname = 'storage/results/%d%d%d/%d.chrom' % (i, j, k, l)
            shutil.move(sourcename, targetname)
          succeeded = True
        except IOError:
          succeeded = False
```

Filename: animats_modularanimat_controlstrategies_spinal_spine.cpp
Location: /animats/animats_modularanimat/
Lines: 62-105
Description: The spine controller's `tick()` method. The spine's motors are actuated based on their position in the spine and the spine's input signals.

```
void Spine::tick
( D6Motor* motors,
  float    frequency,
  float    phaseShift,
  float    raiseOrSwing,
  float    propogationDelay )
{
  float k = 1;
  float raise;
  float swing;
  float force;

  if ( frequency == 0 )
  {
    k = 0;
  }
  else if ( frequency == -1 )
  {
    counter = NxPi / 2;
    frequency = phaseShift = raiseOrSwing = propogationDelay = 0;
  }

  counter += ( ( frequency ) * mMaxFrequency / mNumMotors ) / NxPi;

  for ( int i=0; i<mNumMotors; i++ )
  {
    raise = mRaiseAmount * raiseOrSwing * sin( counter
      + i * propogationDelay * NxPi / 2 );
    swing = mSwingAmount * ( 1 - raiseOrSwing ) *
      cos( counter + i * propogationDelay * NxPi / 2 + phaseShift * 2 * NxPi );
    force = mSpring * k;

    motors[ mMotorIndices[ i ] ].startRaise( raise, force );
    motors[ mMotorIndices[ i ] ].startSwing( swing, force );

    mEnergyUsed += frequency / 100 / mNumMotors;
  }
}
```

Filename: animats_modularanimat_morphologies_clique.cpp
Location: /animats/animats_modularanimat/
Lines: 159-192, 340-425
Description: The start() method on a CliqueMorphology (a graphical model) creates an instance of that model in the physics scene.

```
void CliqueMorphology::start
( NxScene*   scene,
  NxActor**  limbs,
  NxJoint**  joints,
  D6Motor*   motors )
{
  // Set the counters
  limbCount  = 0;
  jointCount = 0;

  // Start the depth first add
  depthFirstStart( scene, limbs, joints, motors, mNodes[0],
    NxVec3( 0.0f, 2.0f, 0 ), NxVec3( -1.0f, 2.0f, 0 ), 0, true );

  // Staple the joints together to prevent collisions
  for ( int i=0; i<numLimbs(); i++ )
  {
    for ( int j=i+1; j<numLimbs(); j++ )
    {
      Scene::createFreeD6Joint(
        scene,
        limbs[ j ],
        limbs[ i ] );
    }
  }

  // Create the motors
  for ( int i=0; i<numJoints(); i++ )
  {
    motors[i].setJoint( (NxD6Joint*) joints[i] );
  }
}

int CliqueMorphology::depthFirstStart
( NxScene*    scene,
  NxActor**   limbs,
  NxJoint**   joints,
  D6Motor*    motors,
```

```
  CliqueNode* node,
  NxVec3      cursor1,
  NxVec3      cursor2,
  int         depth,
  bool        recurse )
{
  NxVec3      newCursor;
  CliqueEdge* potentialChildEdge;
  int         currentLimbCount;
  int         childLimbCount;

  // Store the current limb count
  currentLimbCount = limbCount;

  // Create the limb itself
  limbs[ limbCount ] = Scene::createBox(
    scene,
    cursor1,
    cursor2,
    thicknessForEdge( depth ),
    thicknessForEdge( depth ),
    1,
    0 );

  // Set its colour
  if ( limbCount < mMaxData )
  {
    mUserData[ limbCount ]->renderData             = mRenderData[ limbCount ];
    mRenderData[ limbCount ]->primaryColourIndex   =
      (int) Settings::get( "animatPrimaryColourIndex" )[0];
    mRenderData[ limbCount ]->secondaryColourIndex =
      (int) Settings::get( "animatSecondaryColourIndex" )[0];
    mRenderData[ limbCount ]->colourAdjustment     =
      ( (float) depth ) / ( (float) mDepth );
    limbs[ limbCount ]->userData                   = mUserData[ limbCount ];
  }
  else
  {
    limbs[ limbCount ]->userData            = mUserData[ mMaxData - 1 ];
  }

  // Increase the limb counter
  limbCount += 1;

  // Consider all the children
```

```
  for ( int i=0; i<node->numEdges(); i++ )
  {
    potentialChildEdge = mEdges[ node->getEdgeIndex( i ) ];

    if ( ( recurse || potentialChildEdge->getTargetIndex() != node->getIndex() )
      && depth < 2 * mDepth )
    {
      // Update the cursor's position
      newCursor = newCursorPosition( i, depth, node, cursor1, cursor2 );

      // Recurse
      childLimbCount = depthFirstStart(
        scene,
        limbs,
        joints,
        motors,
        mNodes[ potentialChildEdge->getTargetIndex() ],
        cursor2,
        newCursor,
        depth + 1,
        ( depth + 1 < mDepth ) ? true : false );

      // Join the two nodes together
      joints[ jointCount ] = Scene::createD6Joint(
        scene,
        limbs[ currentLimbCount ],
        limbs[ childLimbCount ],
        cursor2,
        newCursor - cursor2,
        true, mTwistLimitDesc,
        true, mRaiseLimitDesc,
        true, mSwingLimitDesc );

      // Create the corresponding motor
      motors[ jointCount ].setJoint( (NxD6Joint*) joints[ jointCount ] );

      // Increase the joint counter
      jointCount++;
    }
  }

  return currentLimbCount;
}
```

Bibliography

[1] `http://www.nvidia.com/object/nvidia_physx.html`. NVIDIA PhysX. Last retrieved: March 27, 2009.

[2] `http://www.opengl.org/`. OpenGL - The Industry Standard for High Performance Graphics. Last retrieved: March 27, 2009.

[3] `http://www.libsdl.org/`. Simple DirectMedia Layer. Last retrieved: March 27, 2009.

[4] `http://www.nvidia.com/object/physx_accelerator.html`. AGEIA PhysX Accelerator. Last retrieved: March 27, 2009.

[5] Randall D. Beer, Roger D. Quinn, Hillel J. Chiel, and Roy E. Ritzmann. Biologically inspired approaches to robotics: what can we learn from insects? *Commun. ACM*, 40(3):30–38, 1997.

[6] Josh Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121, November 2006.

[7] M. Buehler, U. Saranli, and D. Papadopoulos. Dynamic locomotion with four and six-legged robots, 2000.

[8] Nicolas Chaumont, Richard Egli, and Christoph Adami. Evolving virtual creatures and catapults. *Artif. Life*, 13(2):139–157, 2007.

[9] Kevin J. Dowling. *Limbless locomotion: learning to crawl with a snake robot*. PhD thesis, Pittsburgh, PA, USA, 1997.

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

[11] Radek Grzeszczuk. *Neuroanimator: fast neural network emulation and control of physics-based models*. PhD thesis, Toronto, Ont., Canada, Canada, 1998.

[12] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, April 1992.

[13] A. J. Ijspeert, A. Crespi, D. Ryczko, and J. M. Cabelguen. From swimming to walking with a salamander robot driven by a spinal cord model. *Science*, 315(5817):1416–1420, March 2007.

[14] Haruhisa Kurokawa, Kohji Tomita, Akiya Kamimura, Shigeru Kokaji, Takashi Hasuo, and Satoshi Murata. Distributed self-reconfiguration of m-

tran iii modular robotic system. *Int. J. Rob. Res.*, 27(3-4):373–386, March 2008.

[15] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978, August 2000.

[16] Thomas Miconi. *The Road to Everywhere: Evolution, Complexity and Progress in Natural and Artificial Systems.* PhD thesis, Birmingham, United Kingdom, 2007.

[17] Melanie Mitchell. *An Introduction to Genetic Algorithms (Complex Adaptive Systems).* The MIT Press, February 1998.

[18] R. Reeve and J. Hallam. An analysis of neural models for walking control. *Neural Networks, IEEE Transactions on*, 16(3):733–742, 2005.

[19] Karl Sims. Evolving virtual creatures. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 1994. ACM Press.

[20] M. Yim, Wei-Min Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *Robotics & Automation Magazine, IEEE*, 14(1):43–52, 2007.

[21] Victor Zykov, Efstathios Mytilinaios, Bryant Adams, and Hod Lipson. Self-reproducing machines. *Nature*, 435(7039):163–164, May 2005.