Evolving Locomotion Controllers for Virtual Creatures

Michael John Sanders

A thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science
The University of Auckland
February 2000.

Abstract

This thesis considers the problem of automating the locomotion of virtual creatures. A virtual creature is a computer-simulated animal that exists in a simulated environment. The animal's body and environment are modelled according to physical laws, such as those of Newtonian mechanics. Our virtual creatures are modelled as mass-spring systems.

We investigate a controller-based approach to virtual creature animation. Our controllers are simple 'locomotion brains' that produce locomotion by instigating and sequencing contractions and expansions of virtual muscles in the creature's body. We allow controllers to observe the creature's local environment through sensors in the creature's body.

An evolutionary algorithm (EA) is used to synthesise locomotion controllers for a small set of virtual creatures. We demonstrate the behaviour of our controller-synthesis EA on several different creature bodies but focus the bulk of our investigation upon a worm-like creature. Two types of controller are evolved: those that make use of sensors, which we term *closed-loop controllers* and those that do not, which we term *open-loop controllers*. We show that the creature's body determines which of these controller types our EA will produce. Initial results from this work were published in [Sand_2000].

We investigate the benefits of applying a niching method to our evolutionary algorithm. We show that a niching method can improve the expected performance of our EA. Additionally, niching results in the synthesis of a range of different locomotion controllers for each evolution, usually including both open-loop and closed-loop controllers.

We show that by applying small random variations to the terrain over which a creature's locomotion controller is evolved we improve the expected quality of controller. Additionally, we show that the amount of variation in a creature's environment influences the style of locomotion.

Throughout our experiments we present 3D visualisations of evolutionary data. We discuss and demonstrate the benefits of 3D data visualisation for EAs as opposed to traditional textual output.

We present a simple, robust and highly effective distributed computation system for sharing the cost of creature simulation between many computers over a local area network.

Preface

The past year has been an enormously educational and entertaining experience. Through my supervisors and others I have learned a great deal about evolutionary algorithms, physically-based simulation and good research technique.

Observing creatures as they learn to move is fascinating and has provided countless hours of amusement. I hope that the images and animations in this thesis and on the enclosed CD-ROM will provide to my readers similar insight and entertainment.

It is my hope that this thesis will provide a good basis for future work on the automatic synthesis of animation controllers for realistic virtual creatures.

Acknowledgements

CPU time
I am greatly indebted to various people for donating CPU time for my distributed computation system over the course of this thesis. In chronological order: Chris Anderson, Richard Lobb, Dougal Cowan, Peter Dance and James Harper. Without their help this work would have been impossible.

Inspiration
Karl Sims, for his work on evolving virtual creatures. The controller representation described in this thesis is heavily based on Karl's.

Financial support
My parents Gordon and Val, and my sister Carolyn. Their timely monetary contributions over the past year have allowed me to avoid a diet of rice and pigeon.

Supervisors
Dr Richard Lobb and Dr Patricia Riddle. Thank you for your guidance, patience and good humour during the evolution of this thesis.

Others
The members of the graphics and visualisation group (GVG), 1999, for providing a supportive working environment and for their useful comments and suggestions. Dr Phillip Reiser, for suggestions and advice regarding evolutionary algorithms.

Michael John Sanders
February 2000

Table of Contents

Abstract i
Preface iii
Acknowledgements iii
Introduction 1
1.1 Automatic Controller Synthesis 1
1.2 Existing work 1
1.2.1 Karl Sims 1
1.2.2 Ngo and Marks 2
1.2.3 Van de Panne and Fiume 3
1.2.4 Terzopoulos, Grzeszczuk and Tu 4
1.2.5 Critique and Conclusions 4
1.3 Thesis Goals 5
Chapter 2 Virtual Creatures 7
2.1 Morphology Representation 8
2.1.1 Genotype Morphology 8
2.1.2 Phenotype Morphology 8
2.2 Controller Representation 12
2.2.1 Sensors 12
2.2.2 Neurons 12
2.2.3 Effectors 13
2.2.4 Constants 13
2.2.5 Connections 13
2.3 Nested Controllers 14
2.4 Controller Dynamics 16
2.5 Control Type 16
2.6 Summary 16
Chapter 3 Mass-Spring Systems 17
3.1 Definitions 17
3.1.1 Masses 17
3.1.2 Forces 17
3.1.3 Springs 17
3.1.4 External Forces 18
3.2 Environment Modelling 18
3.2.1 Collision Modelling 18
3.2.2 Surface Friction 19
3.3 Animating the Mass-Spring System 21
3.4 ODE Solvers 22
3.4.1 Euler 22
3.4.2 Midpoint 23
3.4.3 Runge-Kutta 4th order 23
3.4.4 Accuracy 24
3.4.5 Adaptive Stepsizing 24
3.4.6 Performance Comparison, Conclusions. 25
Chapter 4 Evolution of Controllers 27
4.1 Evolutionary Algorithm Introduction 27
4.2 Algorithm Overview 27
4.3 Selection Mechanisms 27
4.3.1 Fitness Proportionate Selection 28
4.3.2 Rank Selection 28
4.3.3 Properties 29
4.4 Reproduction Methods 29
4.4.1 Mutation 29
4.4.2 Crossover 31
4.4.3 Grafting 32
4.4.4 Cloning 33
4.4.5 Validity Enforcement and Garbage Collection 33
4.5 Distributed Fitness Evaluation 34
4.5.1 Motivation 34
4.5.2 Master/Slave Architecture 34
4.5.3 Performance Benefits and Conclusion 36
4.6 Summary 36
Chapter 5 Visualisation of Evolution 37

5.1 Introduction and Motivation 37
5.2 Views Available 37
5.3 Demonstration 38
5.4 Summary and Conclusions 38
Chapter 6 Models and Parameters 41
6.1 Test Creatures 41
6.1.1 Worm 41
6.1.2 'Ring' Creature 42
6.1.3 Octagonal Creature 42
6.1.4 Biped 43
6.2 Fitness Functions 44
6.3 Parameter Description 45
6.3.1 Simulation Parameters 45
6.3.2 Evolutionary Algorithm Parameters 45
Chapter 7 Initial Results 47
7.1 Worm Model 47
7.2 Other Creatures 50
7.2.1 Ring model 51
7.2.2 Octagonal model 51
7.2.3 Biped model 52
7.3 Quality of animation 52
7.4 Randomised Terrain: Fitness Evaluation in a Noisy World 53
7.4.1 Flat Terrain versus Varying Terrain 55
7.4.2 Animation Comparison 57
7.4.3 Conclusion 57
7.5 Summary 57
Chapter 8 Niching 59
8.1 Introduction to Niching Methods 59
8.1.1 Differencing Functions and the Niche Radius 61
8.1.2 Number of Niches 61
8.2 The 'Clearing' Niching Method 62
8.2.1 Algorithm Overview 62
8.2.2 A Differencing Function for Our Candidate Representation 63
8.3 An Experimental Comparison between Traditional Selection and Clearing-based Niching. 66
8.3.1 Variation of Controller Type and Locomotion Gait 70
8.3.2 Computational Cost. 71
8.4 Conclusions 71
8.5 Summary 72
Chapter 9 Discussion and Conclusions 73
9.1 Evolving Locomotion Controllers: Conclusions 73
9.2 Knowledge Obtained 73
9.3 Future Directions 74
Bibliography 75

Chapter 1 Introduction

Computer Generated Imagery (CGI) is rapidly replacing puppetry and stop-motion photography as the preferred method of animation in cinema and television. Films such as *Jurassic Park: The Lost World* and *The Phantom Menace* have shown that computer graphics can accomplish what previously would have required the construction of detailed models or puppets, or may have been all but impossible with pre-CGI techniques.

1.1 Automatic Controller Synthesis

Common techniques for animating realistic virtual creatures involve painstaking frame-by-frame control of a 3D model of the creature's body by highly skilled animators. As the demand for convincing, realistic virtual creatures increases, techniques for automating their animation must be developed. Creatures should ideally be transformed from virtual puppets into virtual actors or agents, with the ability to autonomously perform complex animation tasks. Virtual worlds populated by autonomous, realistic, physically and behaviourally plausible virtual creatures are already becoming feasible [Blum_95; Grze_95; Funge_99]

An interesting sub-problem is that of creature locomotion. A virtual creature capable of automatically moving itself over virtual terrain in a convincing and realistic fashion would be extremely useful to an animator. Using such a creature an animator could generate large quantities of high-quality animation with relatively little effort. This low-level locomotion control problem does not involve behavioural animation, trajectory planning or other high-level artificial intelligence techniques such as those investigated in [Tu_94; Yama_94; Kodj_98; Funge_99].

Our approach to producing creature locomotion is to consider the task to be a *control* problem; the creature's virtual muscles must be made to act in a way that propels the creature forwards. A *locomotion controller* is a device that instigates and sequences these muscle movements to produce locomotion. We present locomotion controllers as localised neural systems; they "exist" within the creature's body and may only obtain information about the creature's body or local environment that would be readily available to the brain of a real creature. We term controllers that make use of sensor information and thereby alter their behaviour in response to local stimuli to be *closed-loop*. Controllers that do not use sensor information and yield time-based cyclic output are termed *open-loop*.

The automatic synthesis of locomotion controllers for physically-based virtual creatures may have applications in robotics research. A locomotion controller for a real-world robot could be evolved using an accurate physically-based simulation of the robot's body and its environment. Learning in simulation will usually have advantages of speed, economy and safety.

1.2 Existing work

Controller-based animation techniques have received attention in recent years due to their success at animating a wide variety of virtual creatures. These creatures range from simple two-dimensional stick figures to realistic three-dimensional fish and articulated-rigid-body creatures[1].

1.2.1 Karl Sims

Karl's 1994 paper "Evolving Virtual Creatures" [Sims_94] introduced an evolutionary algorithm to synthesise three-dimensional 'block' creatures. His creatures concurrently evolved both morphology and controller for such tasks as swimming, walking, jumping, and following a light source. Creatures existed in a physically-based world with bodies constructed from articulated rigid blocks as illustrated in Figure 1-1.

[1] Diagrams reproduced with permission.

Figure 1-1. An evolved 'water snake'.

Creature morphology is encoded in a directed graph, termed a *genotype*. Each node of the genotype graph contains information specific to one class of block in the creature's body. Links between nodes describe the relative positions of blocks (including rotation and scaling) and joint-articulation information. The creature's morphology is constructed by performing a traversal of the genotype graph, originating from its fixed *root* node. Evolution of morphology involves making changes to the genotype graph in terms of topology or content. Adding a new node creates an additional block class. Adding a new link results in the creation of an additional instance of a particular block (or hierarchical block structure) in the creature's body. Alterations to the information held by nodes or links affect the size and shape of blocks and properties of the articulated joints connecting them.

Creature controllers are directed data-flow graphs similar to electrical circuits. These controller graphs consist of three types of node:

- *Sensors*, which supply the controller with information about the creature's body and its local environment
- *Neurons*, which perform various arithmetic, geometric and signal-processing operations upon input signals
- *Effectors*, which apply input controller signals as torques to the creature's articulated joints.

Links between controller nodes define the flow of data within the controller, from sensors through neurons to effectors. Evolution of a controller involves altering the topology and content of the data-flow graph.

Natural selection and reproduction are repeatedly applied to a population of block creatures. A creature's fitness is determined by evaluating its performance according to a *fitness function* during a period of simulation. Fitness functions are metrics of the form "*How far/fast did it swim?*", "*How high did it jump?*". Creatures with high fitness values are given preference in reproduction; they have the opportunity to propagate their morphological structure and controller information to the next generation. Reproduction also applies random changes to morphology and controller. Many of these changes are detrimental to fitness but some result in superior performance, leading to selection for reproduction.

Sims' evolutionary algorithm produced a diverse range of virtual creatures for both land and marine virtual environments. Some, such as the 'water snake' illustrated in Figure 1-1, were of familiar form. A few were unlikely contraptions of largely redundant blocks, yet still managed to provide effective locomotion.

1.2.2 Ngo and Marks

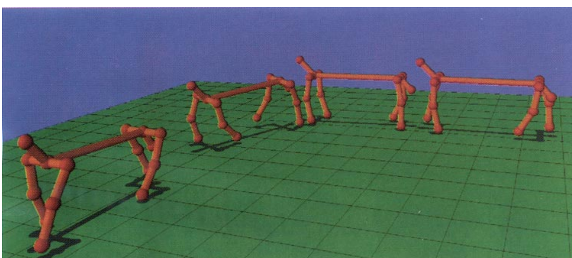


Figure 1-2. A stick-figure quadruped.

Ngo and Marks et al [Ngo_95] describe an evolutionary algorithm for the synthesis of animation controllers for two-dimensional and three-dimensional stick figures. They define a controller as a group of stimulus-response rules, which they term a Banked Stimulus Response (BSR) controller.

A BSR controller is a finite-state machine in which each state defines a deformation (or *pose*) for the creature's body to assume. Transitions can occur

between any two states according to sensor data or the world-time parameter, thus allowing for either open-loop or closed-loop control.

A BSR controller contains R rules, where each rule R_i contains vectors of stimuli parameters $\overrightarrow{S_i^{lo}}$ and $\overrightarrow{S_i^{hi}}$, and response parameters $\overrightarrow{\theta_i^{o}}$ and τ_i. A rule outputs a vector $\overrightarrow{\theta^{o}}(t)$ of target joint angles and a time constant, τ_i, which define a deformation of the stick figure and the rate at which the creature's geometry should assume the deformation from the current physical configuration. Only one rule can be active at a time. The active rule is determined by ranking each rule according to coverage of the instantaneous sensor vector $\vec{S}(t)$ by the hyper-rectangle formed by the upper and lower limits in the rule's $\overrightarrow{S_i^{hi}}$ and $\overrightarrow{S_i^{lo}}$ sensor vectors. Each element of a rule's output vector $\overrightarrow{\theta^{o}}(t)$ is static during simulation.

A BSR controller's set of rules are evolved by both global and local search. Global search replaces one of the rules with a new rule containing a randomly generated target deformation vector, time constant and sensor vectors. Local search occurs by the random perturbation of values belonging to an existing rule.

For 2D stick figures, of which a five-pronged star creature and a bipedal walker were described, the BSR synthesis algorithm is run for 40,000 physical simulations. For the 3D stick figures, of which a quadruped and a bipedal walker were presented, between 100,000 and 200,000 simulations were required to discover and optimise a locomotion controller.

Against their expectations, Ngo and Marks et al found that the majority of controllers discovered for 2D stick figures were time based and therefore using open-loop control. As expected, they were unable to generate robust open-loop controllers for the 3D bipedal walker due to its inherent instability, but they were successful in generating useful closed-loop sensor-based controllers for that model.

1.2.3 Van de Panne and Fiume

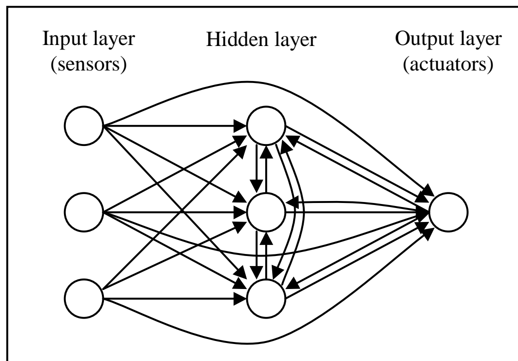


Figure 1-3. Topology of a simple SAN.

Van de Panne and Fiume [Van_93] describe a method of locomotion controller synthesis for small virtual creatures using sensor-actuator networks of a fixed topology. A sensor-actuator network (SAN) is very similar topologically to a three-layer feed-forward neural network in that each layer of units is strongly connected to the next layer. The layers are the input (sensors), the hidden layer and the output layer (actuators). Additionally, the input layer of a SAN is strongly connected to the output layer, the hidden layer is strongly connected within itself and the output layer is strongly connected to the hidden layer. Figure 1-3 illustrates a simple SAN topology.

Creatures are constructed from rigid links and are two-dimensional. Sensors in the creature's body supply a binary output signal. For example, a 'touch' sensor would return a value of 1.0 if its associated body part is in contact with something, or 0 if no contact was detected. Actuators are of two types - linear or angular - and act upon a joint between two links. Actuators and hidden-layer units calculate their output as a weighted sum of their inputs from other units, and include an output-damping mechanism to reduce "chattering" due to rapidly fluctuating sensor values.

Sensors and actuators are defined and placed in the creature's morphology by the human experimenter. One-to-one mappings exist between units in the input layer and sensors in the creature, and between output layer units and actuators in the creature. The number of units in the SAN's hidden layer is fixed, and is specified by the experimenter.

Controller synthesis involves setting the weights of connections in the SAN and adjusting the mutable parameters of sensors and actuators. In an initial phase, sets of weights and parameters are generated randomly until a set results in some form of locomotion. The algorithm then enters a period of either simulated annealing or stochastic gradient ascent in which the locomotion is optimised by fine-tuning the weights and parameters. As with the previously described approaches, controller fitness is determined by evaluating the creature's behaviour during a period of simulation.

The authors present a range of simple creatures for which locomotion controllers have been generated. Often, repeated controller synthesis trials upon a particular creature morphology would result in several different locomotion gaits.

1.2.4 Terzopoulos, Grzeszczuk and Tu

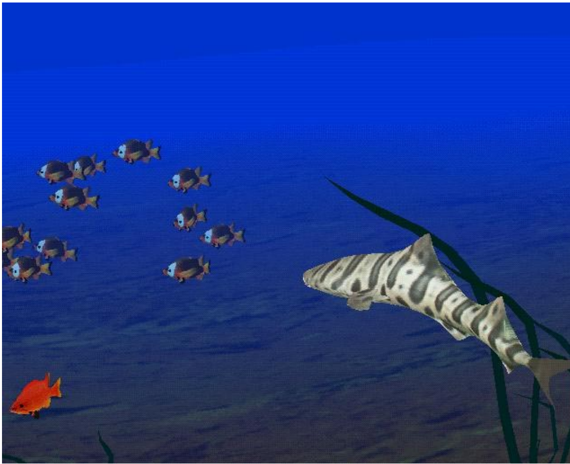


Figure 1-4. The fish of Terzopoulos, Grzeszczuk and Tu.

The virtual fish and rays of Terzopoulos, Grzeszczuk and Tu [Tu_94; Grze_95] exist in a physically-based virtual world of striking aesthetic and behavioural authenticity. Primitive animation controllers for actions such as "swimming forwards", "turning left" and "turning up" are automatically learned and then are sequenced by a behavioural animation controller.

The various piscine inhabitants of the aquatic environment are modelled as three-dimensional mass-spring systems. A controller may deform a fish's body by adjusting the rest lengths of springs along the fish's sides. Hydrodynamic forces are modelled to produce propulsion in response to model deformation.

Primitive locomotion controllers are sets of periodic wave functions, each of which acts upon an *effector*. An effector is an arbitrary grouping of springs in the model. Each spring in an effector has its rest length set as a linear function of the effector's single input. Wave functions are defined either *spatially* in terms of control points, or *spectrally* in terms of frequency components. Optimisation of a controller involves finding a set of control points or frequency components such that the model performs optimally in its environment according to a fitness function. Simulated annealing is used to converge upon an optimal controller.

A high-level behavioural animation controller switches between the periodic primitive controllers to accomplish an animation task. Interpolation between adjacent controllers' component wave functions allows a smooth transition of control.

1.2.5 Critique and Conclusions

The approaches to controller synthesis described above are diverse and novel. Each approach is highly specialised for one genre of creature morphology, be it articulated rigid body, stick figure or mass-spring structure. Approaches based on a purely random search of controller space such as the SANs of van de Panne and Fiume are unlikely to succeed in complex creatures; the probability of locating a meaningful set of weights by chance will surely become vanishingly small as creature complexity increases. The BSR controllers of Ngo and Marks place strict limits on the number of poses a creature may adopt, thus limiting their ability to adapt to complex environments. The modular locomotion controllers of Terzopoulos, Grzeszczuk and Tu are well suited to a homogenous aquatic environment but do not allow for the incorporation of sensorial data at the locomotion controller level. A creature moving over rough terrain would surely benefit from being able to adapt its locomotion to suit changing local conditions.

The dataflow-graph controllers of Sims are scalable in complexity to suit a wide range of virtual creatures. These controllers may develop either open-loop or closed-loop control as suits the task, and do not require any *a priori* information in the form of sensor or effector placement. We consider Sims' controller representation and evolutionary algorithm to be the most highly automated and general method of those investigated.

1.3 Thesis Goals

The original goal of this thesis was to investigate the automatic evolution of animation controllers for *Amoeba Man*, a physically-based soft object model developed by Daniel Nixon [Nixon_99]. Initial experiments in controller evolution made it clear that the very high computational cost of animating the Amoeba Man model would preclude any substantial investigation. In the interests of animation speed we chose an alternative creature modelling technique – mass-spring systems.

Our goal was therefore to investigate controller synthesis by an evolutionary algorithm for mass-spring virtual creatures. We aimed to develop a robust, reliable controller synthesis method capable of producing realistic locomotion for a variety of morphologies. Additionally, we aimed to investigate the automatic development of a range of qualitatively different locomotion styles for each creature.

Chapter 2 Virtual Creatures

We define a virtual creature to consist of a virtual body and a controller. The body must contain at least one *degree of freedom* that is alterable over a range of values by the controller. A degree of freedom is a one-dimensional activation level of a muscle or actuator in the creature's body, e.g. the rest length of a spring, a torque about a fixed axis, etc. The behaviour of a body with respect to its environment may be simulated using physically-based animation techniques, or may be non physically-based. Recent physically-based virtual creatures have been modelled using articulated rigid bodies [Sims_94], mass-spring systems [Mill_88; Tu_94; Grze_95] and stick figures [Van_93; Ngo_95; Laszlo_96].

A controller is a mechanism for animating the creature's body by acting upon the body's degrees of freedom. We define an e*ffector* to be an interface between the controller and a single degree of freedom in the creature's body. A controller may obtain information about the creature's body and its environment through *Sensors.* Figure 2-1 illustrates the flow of information between the controller and the creature's body.

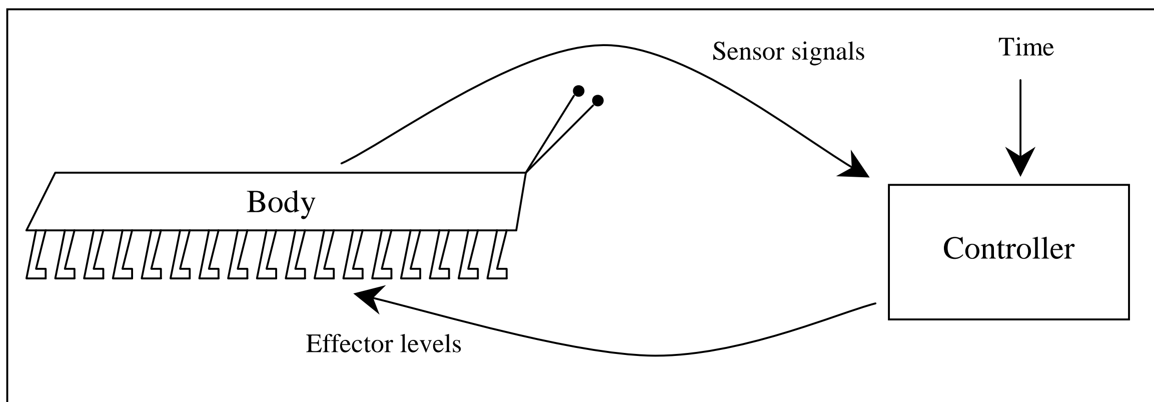


Figure 2-1. The controller's interaction with the creature's body.

A controller can be defined as a relation *R* between input sensor signals and output effector levels: $E_{1..N} = R(S_{0..M})$, where $E_{1..N}$ are the activation levels sent to the *N* effectors, $S_{1..M}$ are the activation levels of the *M* sensors, and S_0 is a special case in which the input is the time parameter. We seek to develop creatures as autonomous agents similar to real-world animals. Towards this goal we model the controller as part of the creature's body in a similar fashion to a brain and nervous system.

Our virtual creatures' bodies are structures of masses and springs. These structures are inherently flexible, which makes them suitable for modelling soft-bodied creatures such as worms, caterpillars, fish, amoeboid blobs and many other interesting lifeforms. Our creatures' mass-spring bodies have only one type of degree of freedom: the rest length of a spring. We will discuss mass-spring systems in detail in Chapter 3. Figure 2-2 shows a selection of simple 2D mass-spring creature morphologies.

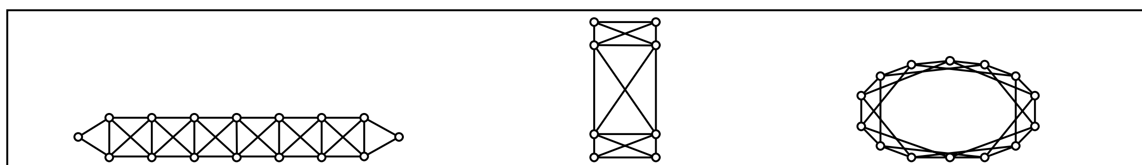


Figure 2-2. Mass-spring creatures: Worm-like, hopping and rolling morphologies.

2.1 Morphology Representation

Because we store the creature's controller with the creature's body, the way in which the creature's body, or *morphology*, is defined has a great effect upon the types of controller that may exist within it and the types of locomotion that are possible. Following the work of Sims [Sims_94] we use a genotype/phenotype representation for morphology. Our *genotype* is a plan of how to build the creature's body encoded as a directed graph. The *phenotype* is the body constructed by following the plan specified in the genotype. The genotype allows us to group similar features into a single substructure; For example, we may define a "leg" once in the genotype and instantiate it several times in the phenotype at different positions and orientations.

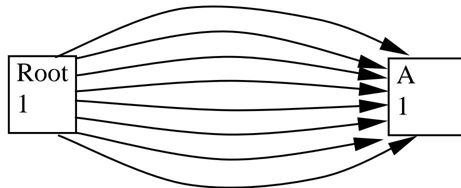


Figure 2-3. A grouped genotype graph.

Figure 2-3 demonstrates grouping in a simplified genotype graph. The corresponding phenotype of this graph will contain eight instantiations of node 'A'.

2.1.1 Genotype Morphology

Our genotype is a directed graph consisting of nodes and links. Each node represents one or more masses in the phenotype morphology. Links are directed connections that represent parent-to-child relationships between masses. The genotype graph has a root node, which defines the starting point for the construction of the phenotype mass-spring morphology.

Nodes specify information relevant to one type of mass in the phenotype and the construction of the creature's mass-spring body, including:

- The mass scalar of the phenotype mass
- Links to child nodes
- A recursive limit, which specifies the number of times the mass should be generated in a recursive cycle.
- A *virtual* flag, which is set if the mass is acting as an alias to another mass. This flag is set *false* by default.

Links contain information specific to the parent-child relationship:

- A reference to the child node.
- The minimum rest length, maximum rest length and strength of the spring connecting the parent mass to the child mass.
- The position and orientation of the child relative to the parent
- A *terminal-only* flag, which is set if the connection should only be applied at the end of a recursive chain of masses. This flag is set *false* by default.

2.1.2 Phenotype Morphology

The phenotype morphology is a tree structure created by performing a depth-first traversal of the genotype graph, starting from its root node. Each phenotype node represents a single mass and each phenotype connection represents a single spring. Figure 2-4 outlines the algorithm used to perform this traversal. Figure 2-5 illustrates the construction of a phenotype morphology graph.

```
generateMorphologyPhenotype()

begin
    expandGenotypeNode(rootNode, null, null)
end generateMorphologyPhenotype

expandGenotypeNode(node, connectionFromParent, parentNode)
begin
    Create a new phenotype node phenotypeNode from the genotype node node.
    if connectionFromParent = null then
        phenotypeNode.position = the origin (0, 0)
    else begin
        phenotypeNode.position = parentNode.position + connectionFromParent.position
        Create a new phenotype link from the genotype link connectionFromParent. {This
        phenotype link connects parentNode to phenotypeNode}
    end else

    for each connection to a child connectionToChild do begin
        if ((node.recursiveLimit > 0) and (not connectionToChild.terminalOnly)) or
          ((node.recursiveLimit = 0) and connectionToChild.terminalOnly) then begin
            Decrement(node.recursiveLimit)
            expandGenotypeNode(connectionToChild.childNode, connectionToChild,
                                phenotypeNode)
            Increment(node.recursiveLimit)
        end if
    end for
end expandGenotypeNode
```

Figure 2-4. The genotype morphology expansion algorithm.

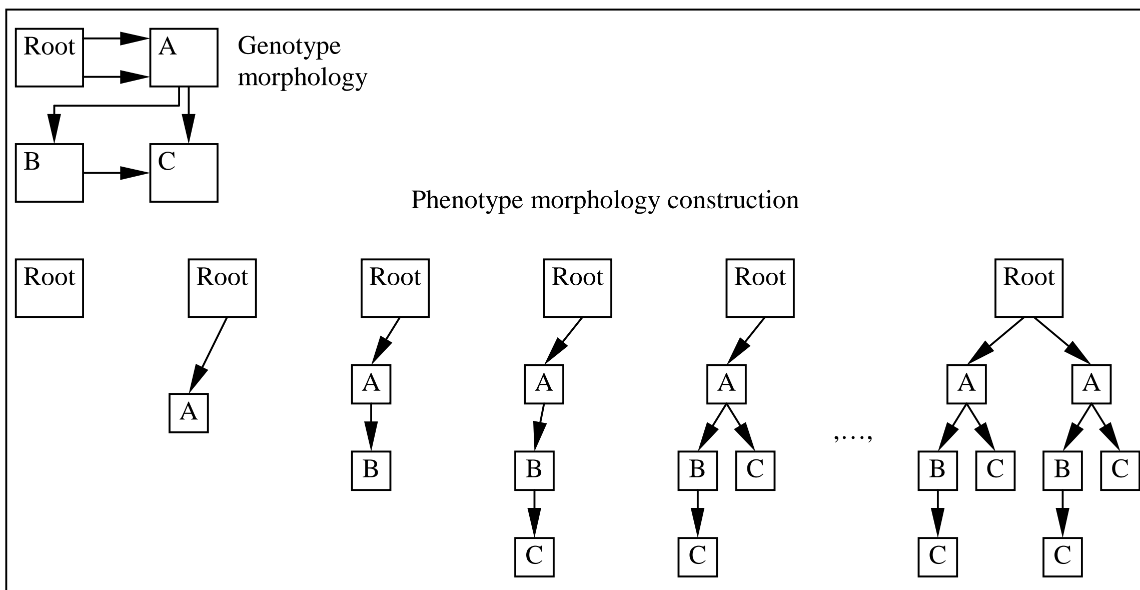


Figure 2-5. Constructing the phenotype morphology by traversing the genotype graph.

Unfortunately, mass-spring trees cannot form useful 2D or 3D bodies. Without structural cyclicity a mass-spring system cannot maintain its shape when subjected to forces. For example, if we place a mass-spring tree into a 2D environment consisting of a ground line and a gravity force, the tree will collapse and become parallel with the ground line. Figure 2-6 illustrates this collapse.

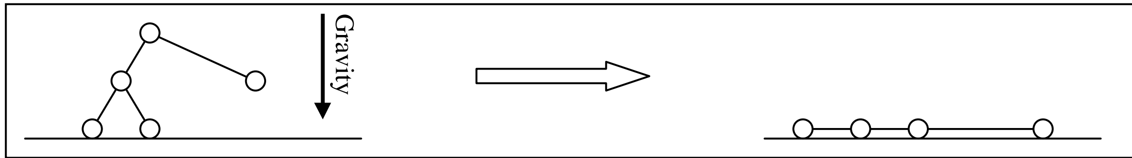


Figure 2-6. Structural collapse in a 2D mass-spring tree.

We implement two methods of creating cycles in the phenotype mass-spring system. Both of these methods may be used simultaneously in the same creature.

The first and simplest method is to specify a set of "extra" springs that are added to the phenotype mass-spring tree. This method is trivial to implement and allows us to connect any two masses, thus allowing the creation of arbitrarily cyclic structures, but has a major disadvantage in that these extra springs are not part of the mass-spring tree. Our controllers (described in section 2.3) are only capable of acting upon springs in the phenotype mass-spring tree, so cannot control "extra" springs.

Our second method allows for structural cycles by mapping multiple masses in the phenotype morphology tree onto a single mass in the mass-spring system. We declare certain genotype masses to be *virtual*. Upon generation of the mass-spring system each of these virtual masses must have its position exactly matching that of a non-virtual mass. Any springs connected to the virtual mass are redirected to the matching non-virtual mass. Figure 2-7 displays one possible genotype for a cyclic three-mass mass-spring system using a virtual mass.

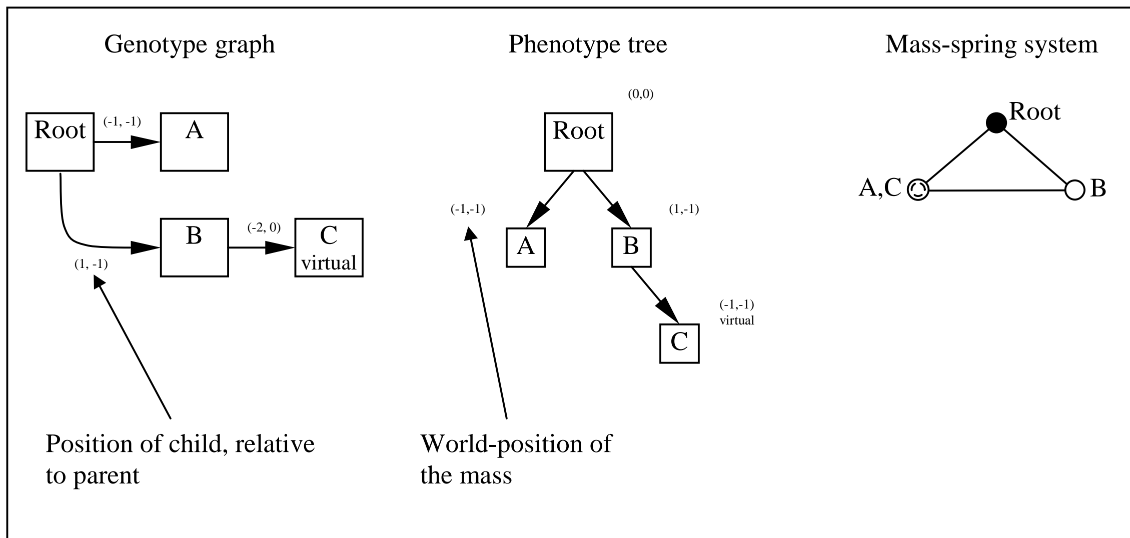


Figure 2-7. Virtual-mass example.

Note that the position of mass C exactly matches that of mass A. The spring connecting B and C is remapped to connect B and A, forming a cycle. This structure will resist collapse under the influences of gravity and other forces.

Figure 2-8 illustrates several simplified morphology genotypes and possible resulting phenotype mass-spring systems.

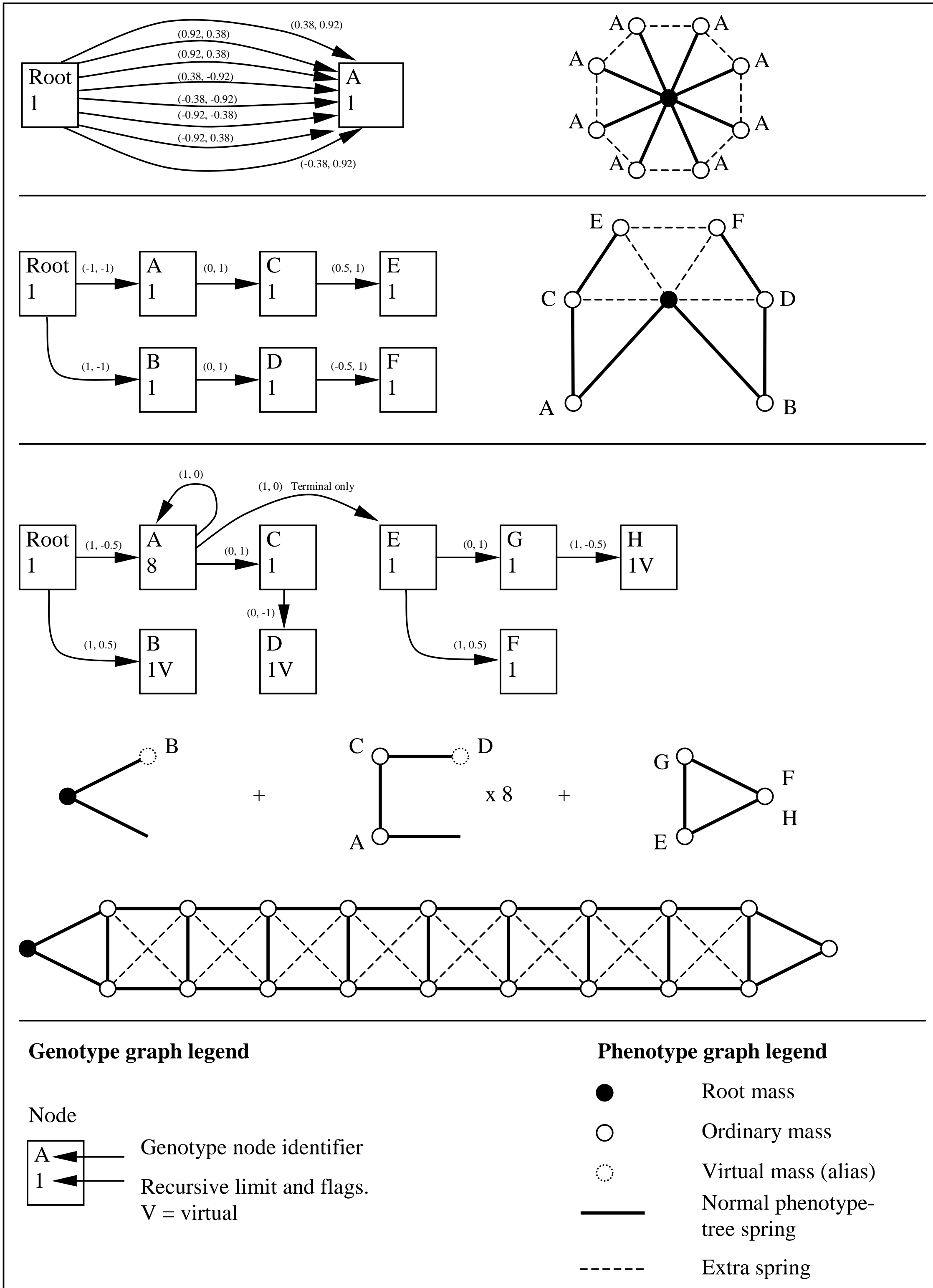


Figure 2-8. Morphology genotypes and their phenotype mass-spring systems.

2.2 Controller Representation

Our controller representation is based on the work of Sims [Sims_94], and is similar to a dataflow diagram. We represent a controller as a directed graph of data processing nodes and data connections. Data processing nodes are of four types: Sensors, Neurons, Effectors and Constants. Connections between nodes define the flow of data within the controller. A node can be abstracted as an input/output structure defining a number of inputs, zero or one outputs and an internal relation. All input and output values are real-valued numbers.

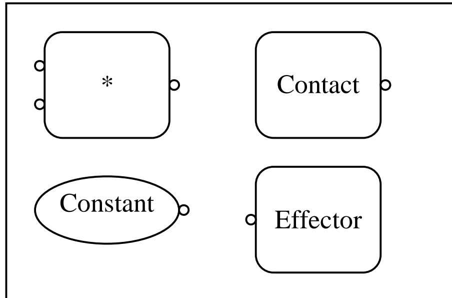
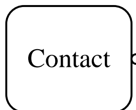


Figure 2-9. Four controller nodes.

Figure 2-9 contains examples of data processing nodes. Inputs are on the left side of each unit. Outputs are on the right.

- Top-left: a 'product' neuron. This unit's relation is the multiplication operator.
- Top-right: a 'contact' sensor.
- Bottom-left: A constant unit. This unit supplies a constant value.
- Bottom-right: A 'spring-length' effector.

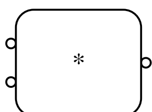
2.2.1 Sensors



Sensors are used to supply the controller with information about the creature's mass-spring system and the environment. Each sensor is associated with a single mass in the phenotype morphology and returns a property of that mass. Sensor nodes accept no input from other nodes and have a single output. A Sensor may be one of the following types:

- *Contact*: An output of +1.0 indicates that the sensor's host mass is in contact with a surface in the environment. An output of –1.0 indicates no contact.
- *Spring length*: The sensor returns the Euclidean distance between the host mass and the host's parent mass according to the phenotype morphology tree.
- *Altitude*: The vertical separation between the mass and the terrain, where "down" is defined by the gravity vector.
- *Vertical velocity*: A signed value describing the mass's vertical velocity.
- *Horizontal velocity*: A signed value describing the mass's horizontal velocity.
- *Orientation*: A scalar indicating the orientation of the parent mass relative to the child mass. For example, values of 0, $\pi/2$, π and $7\pi/4$ indicate that the parent mass is directly above, to the right, directly below, and to the upper-left of the child mass respectively.

2.2.2 Neurons



Neuron nodes perform a transformation upon input data to calculate an output value. They have a single output and at least one input. Most neurons' output is calculated directly from the instantaneous input data. Some neurons have an internal state that is used during calculation of output, allowing them to provide output based on previous input data as well as current input data. Neurons may use temporal context in their output calculation, allowing varying output in the face of unchanging input values. The set of neuron types is listed below:

- Arithmetic and logical neurons:
  Sum, *Product, Divide,*
  Sum-threshold, *Greater-than*, *Min*, *Max*,
  Sign-of, *Absolute-value*, *If-then-else, Interpolate*
- Trigonometric neurons:
  Sin, *Cos*, *Arctan*

- Wave generator neurons:
  Oscillate-wave, *Oscillate-saw*
- Miscellaneous neurons:
  Log, *Exp*, *Sigmoid*,
  Integrate, *Differentiate*,
  Smooth, *Memory*, *Delay*

2.2.3 Effectors

Effectors are the abstraction through which the controller acts upon the creature's body. Each effector in the controller is assigned to a unique spring in the mass-spring system. Input to the effector is used to assign the spring's rest length within the bounds of maximum and minimum rest length specified in the genotype morphology.

The rest-length of a spring is not allowed to change instantaneously; to do so would allow the instantaneous injection of energy into the system, which is not physically plausible. A damping measure is used to discourage rapid changes in rest-length:

$$r_{new} = r + \frac{t\left(r_t - r\right)}{k_r}, \qquad t \ll 1, \quad k_r \geq 1$$

where r_{new} is the new rest length,
r is the current rest length,
r_t is the target rest length,
k_r is the coefficient of rest-length damping, and t is the timestep size (see section 3.3).

2.2.4 Constants

Constant value nodes simply supply a fixed value as their output. These nodes do not accept input, and their output value does not change at any time.

2.2.5 Connections

Connections are responsible for carrying signals from the outputs of nodes to the inputs of other nodes. Each input of a data processing node is bound to one connection, which connects to the output of another node (or the output of the same node, in a recursive cycle). Each connection scales its signal by a mutable weight.

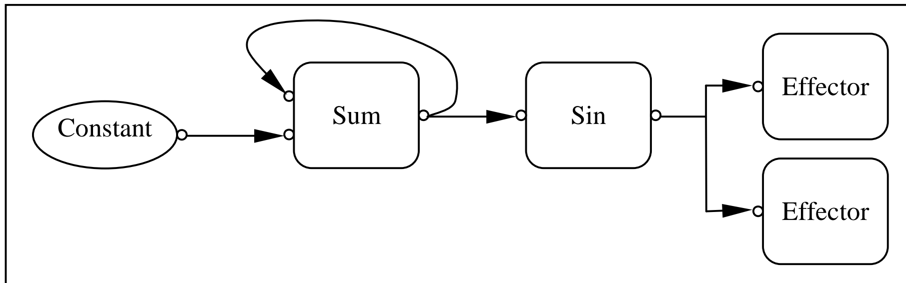


Figure 2-10. A simple directed-graph controller.

Figure 2-10 illustrates a simplified controller graph containing one constant value node, two neurons and two effectors. This controller generates a sine wave, which is used to oscillate the rest lengths of the effectors' springs. The frequency and amplitude of oscillation depend upon the constant node and the scaling weights of the effector units' connections. The *sum* neuron generates an ever-increasing output value, which is passed through a *sin* neuron to create a sine wave.

2.3 Nested Controllers

We store the controller elements with the creature's morphology. By defining the controller within the nodes of the genotype morphology graph we are able to take advantage of any grouping present. If a morphological substructure is instantiated multiple times in the phenotype morphology, we need only define one sub-controller for the substructure rather than define the same sub-controller multiple times. Grouping reduces the number of controllers that are possible for any particular creature - this advantage will become apparent in later chapters.

Each controller node is associated with a particular *part* of the creature's genotype morphology graph. The set of *parts* consists of all the morphology genotype nodes and a special node, the *unassociated area*. The unassociated area contains controller nodes not associated with a morphological genotype node.

To encourage the development of local sub-controllers, connectivity between controller nodes is restricted according to rules of adjacency. A node in part *x* may connect its input(s) to the output of a node in part *y* if one of the following is true:

- x = y
- A parent-child link exists between *x* and *y* in the genotype morphology graph.
- Either *x* or *y* is the unassociated part.

Because effectors act upon the connection between their associated part and the part's parent they cannot exist within the unassociated area or the root node. Likewise, sensors draw information from their associated part and cannot exist within the unassociated area. With the exception of the *spring length* and *orientation* sensors, sensors can exist within the root node. The unassociated area exists to allow for coordinated control of multiple sub-controllers throughout the creature's morphology. Figure 2-11 demonstrates a genotype morphology graph containing a simple wave-based controller.

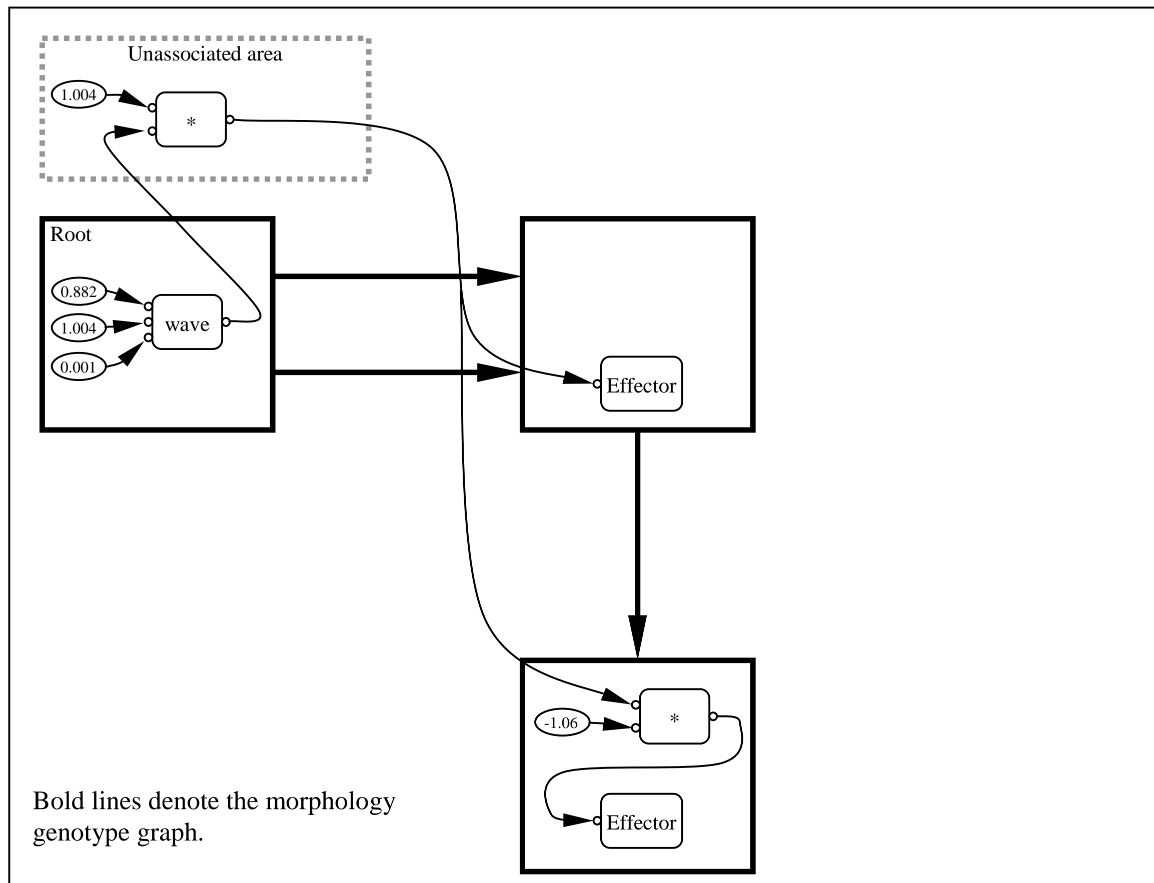


Figure 2-11. Nested controller genotype.

Controller nodes contained within parts of the creature's genotype morphology are replicated along with the genotype nodes when creating the phenotype morphology. Figure 2-12 illustrates the phenotype morphology tree and phenotype controller for the genotype presented in Figure 2-11.

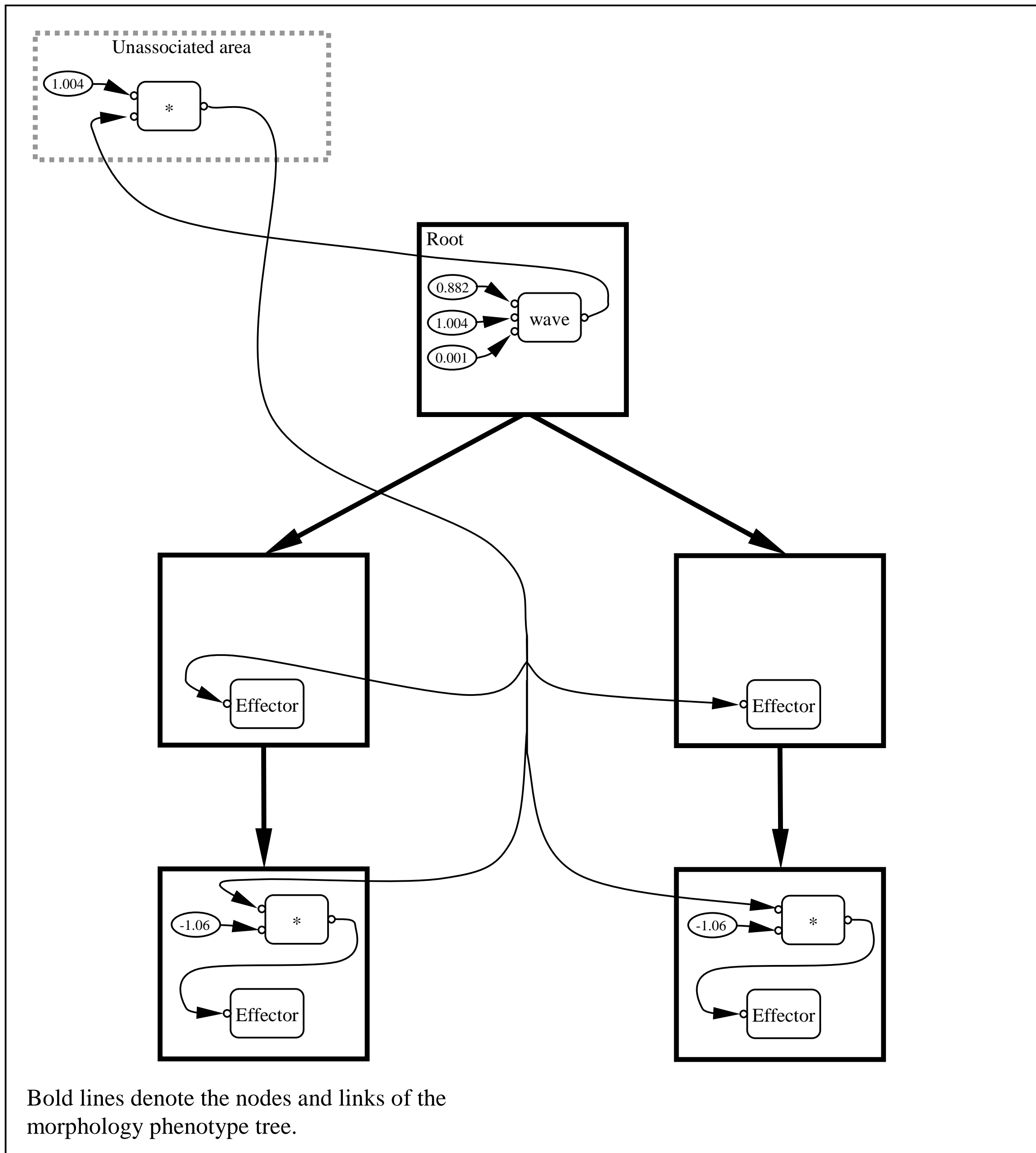


Figure 2-12. Nested controller phenotype.

2.4 Controller Dynamics

Effector values are calculated as a function of the controller's internal state and input data from sensors. We propagate signals through the controller in discrete steps by periodically evaluating the outputs of all nodes, which are calculated according to the nodes' inputs, states and internal relations. Over several controller updates, signals will propagate from sensors or wave generator neurons through to the controller's effector nodes. The order of node evaluation determines the rate of signal flow through the controller. For example, for the controller presented in Figure 2-12 signals from the wave generator may take anywhere from 1 to 3 controller updates to reach the middle-position effectors. Order of node update is determined by the order in which nodes are added to the controller graph.

The rate at which we update the controller is a trade-off between controller response time and computational cost. We must update the controller at a sufficient rate that signals propagate quickly, but not so often that the computational cost becomes significant**.** We find that an update rate of 50 times per simulation second provides adequately fast 'response-time', and is of negligible cost compared to the physical simulation.

2.5 Control Type

Controllers may or may not incorporate sensor units. In the absence of sensor data the controller cannot respond to environmental conditions, and is termed *open-loop*. If sensor data is available, feedback loops may form between the environment, the creature's morphology and its controller. We term sensor-based controllers *closed-loop*.

Locomotion control involves moving the creature's body along an animation path through world-state space. For our task these animation paths contain cyclic deformations of the creature's body, which are repeated as the creature advances. For example, a bipedal walking motion is a cycle in which the creature brings its left leg and centre of mass forward, places its left foot upon the ground, brings its right leg and centre of mass forward, places its right foot upon the ground, etc. We term a repeated cyclic motion a *limit cycle*. Our limit cycles are a result of the creature's interaction with its environment, which includes gravity, viscosity, surface-collisions and friction.

Many virtual creatures can use a range of limit cycles. For example, a human morphology could contain limit cycles for running, walking, hopping, crawling on all fours, wriggling on its belly etc. Some of these limit cycles such as those for crawling and wriggling may be termed *stable limit cycles*; any deviation from the cycle is passively corrected as a result of the creature's interaction with its environment. For example, a crawling or wriggling motion is stable because the centre of mass is low and the creature is not prone to overbalancing as a result of small changes in terrain or the application of small external forces, etc.

Other limit cycles such as those for walking, running and hopping do not have the benefit of passive deviation correction, and are termed *unstable limit cycles*. For example, a walking motion will be very vulnerable to small changes in terrain or the application of small external forces. Unless active measures are taken to keep the creature's animation close to the limit cycle, the creature will overbalance and fall.

Clearly, open-loop control can only succeed where the creature's morphology contains stable limit cycles. Unstable limit cycles require the application of closed-loop control.

2.6 Summary

We use a controller-based approach to the animation of virtual creatures. Our virtual creatures' bodies are modelled as mass-spring structures. A genotype/phenotype representation is used to define morphology, and our controllers are defined within the morphology in a similar fashion to a distributed nervous system.

Chapter 3 Mass-Spring Systems

A mass-spring system is a simple particle system based on the laws of Newtonian mechanics. The system is comprised of a set of point-masses and a set of springs that connect pairs of masses. Mass-spring systems have no rigid parts or constraints and have been successfully applied to the modelling of soft, deformable objects such as cloth [Provot_95], fish and rays [Grze_95] and elastic surfaces [Nixon_99]. The animation of mass-spring systems requires little computational effort when compared with rigid-body modelling or fluid-based modelling, thus making them attractive when simulation speed is of high priority. A good introduction to the theory and implementation of a physically-based particle system can be found in [Witkin_97].

3.1 Definitions

3.1.1 Masses

A mass is defined as an infinitely small particle; it has no volume, area or rotational velocity. A mass particle has the following properties:

- **Mass** (*m*), an inertia scalar measured in kilograms.
- **Position (p)**, a displacement vector measured in metres.
- **Velocity (v)**, a vector defining the direction and magnitude of distance travelled per unit time.

3.1.2 Forces

A force is defined as an acceleration vector applied to a mass. The mass accelerates in the direction of the force vector at a rate proportional to the magnitude of the total force vector and inversely proportional to the mass's mass scalar. Force vectors are measured in kilogram metres per second.

3.1.3 Springs

A spring represents a link between a pair of masses m_a and m_b. Springs apply equal and opposite forces to m_a and m_b, and have the following properties:

- **Rest length** (*r*), a scalar defining the length at which the spring will apply no restorative forces to m_a and m_b.
- **Compression coefficient** (k_s), a scalar representing the strength or stiffness of the spring.
- **Damping coefficient** (k_d), a scalar defining the degree of viscous damping associated with the spring.

Springs calculate their forces $\mathbf{f_{ma}}$ and $\mathbf{f_{mb}}$ as a function of the distance and relative velocity between m_a and m_b. The spring's forces are applied along the vector **L**, where $\mathbf{L} = m_a.\mathbf{p} - m_b.\mathbf{p}$.

$$\mathbf{f_{ma}} = -\left[k_s\left(|\mathbf{L}| - r\right) + k_d\frac{(\Delta\mathbf{v}\bullet\mathbf{L})}{|\mathbf{L}|}\right]\frac{\mathbf{L}}{|\mathbf{L}|}, \text{ where } \Delta\mathbf{v} = m_a.\mathbf{v} - m_b.\mathbf{v} \quad (3.1)$$

$$\mathbf{f_{mb}} = -\mathbf{f_{ma}} \quad (3.2)$$

The k_s term in equation 3.1 calculates a restorative force that is used to push or pull the masses towards the spring's rest length. These forces increase in magnitude as a linear function of the displacement from the rest length. If no other forces are applied to the masses, the distance between them will tend towards the rest length.

The k_d term in equation 3.1 applies a damping force, which resists the component of the mass's velocity in the direction of the spring, **L**. This term emulates the shock absorber of an automobile; rapid change in the

distance between the two endpoints is resisted. Suppressing high-frequency oscillation is beneficial to the stability of mass-spring systems.

3.1.4 External Forces

Forces applied by springs are balanced with respect to the momentum of the mass-spring system. Other forces which may be applied such as gravity and atmospheric viscosity are unbalanced and will usually change the system's total momentum. These forces can therefore be considered external to the mass-spring system, and are applied to each mass individually.

The force of gravity experienced by each mass m_a is dependant upon the mass's mass scalar and the gravity vector $\mathbf{k_g}$. The vector $\mathbf{k_g}$ defines the direction and magnitude of gravity throughout the system.

$$\mathbf{f_{ma}} = m_a.m\mathbf{k_g}$$

Atmospheric viscosity is implemented as a force resistive to motion in the system. The "air" is treated as static with respect to the coordinate system. Each mass m_a experiences a force proportional to the square of its velocity. The scalar k_v defines the coefficient of viscosity throughout the system.

$$\mathbf{f_{ma}} = -m_a.\mathbf{v}\left|m_a.\mathbf{v}\right|k_v$$

Other forces such as electromagnetic or gravitational interaction between particles, non-linear springs, friction etc can be implemented easily.

3.2 Environment Modelling

3.2.1 Collision Modelling

Many animation tasks require some form of collision detection between the mass-spring system and some objects. In this work, that object is the terrain over which the mass-spring creature must travel. For the sake of simplicity and speed, collisions between surfaces and masses are treated as elastic; masses are allowed to penetrate the surface slightly. A mass penetrating a surface will experience a repulsion force proportional to its degree of penetration and the firmness of the surface.

Our two-dimensional terrain is represented using a height-map. For each horizontal position x we can determine a height value $h(x)$ and a unit normal vector $n(x)$. Our height-map is stored as a piecewise linear function in which every section occupies a constant horizontal interval. Figure 3-1 illustrates a section of our terrain.

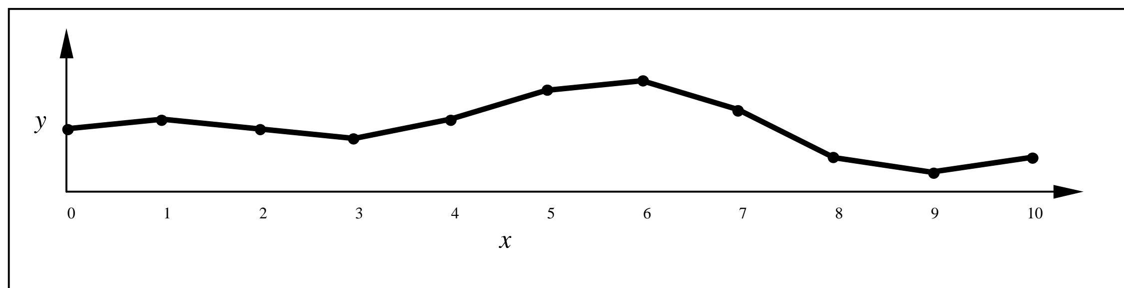


Figure 3-1. Terrain height-map.

To detect a collision between a mass and the terrain we just need to identify the relevant terrain segment and perform some elementary calculations. Assuming that our terrain segments begin at a horizontal value

of 0 and have a width of 1 unit, the terrain height at point $m.\mathbf{p}.x$ is denoted as $h(x)$, and is calculated in equation 3.3

$$h(x) = a.y + \lambda(b.y - a.y) \quad (3.3)$$

where a and b are the left and right endpoints of the terrain segment identified by $\lfloor x \rfloor$. λ is the parametric distance of point x along the terrain segment, and is given by $x - \lfloor x \rfloor$.

Once $h(x)$ has been obtained we calculate $d = m.y - h(x)$, the vertical displacement between mass m and the terrain. If d is negative the mass is penetrating the terrain segment and we must investigate further. Figure 3-2 illustrates.

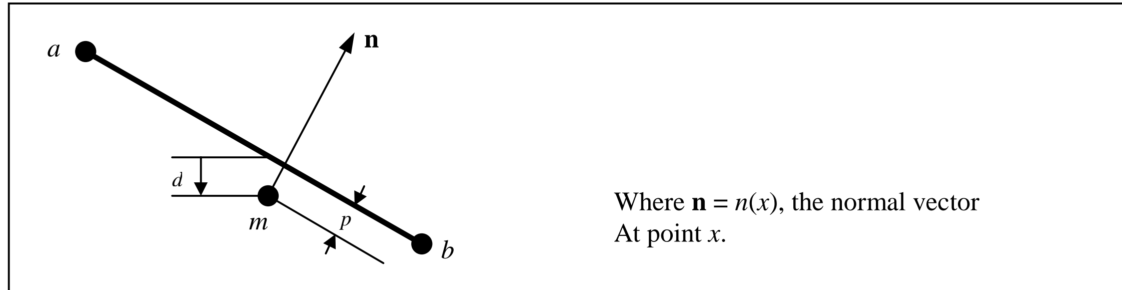


Figure 3-2. Terrain segment intersection.

In the case of terrain penetration, we must calculate p, the distance of penetration. p is given by $-d\mathbf{n}.y$.
We can now calculate a repulsion force to resist the mass's penetration of the surface. Equation 3.4 calculates this force.

$$\mathbf{f} = \mathbf{n} \times K_{surface} \times p \quad (3.4)$$

where p is degree of penetration as shown in Figure 3-2.
$K_{surface}$ is the spring coefficient of the surface.

In addition to resisting penetrations our terrain acts to absorb kinetic energy. We apply a damping force to oppose the mass's momentum while it is moving into the surface. Equation 3.5 obtains v_n, the component of the mass's velocity in the direction of the surface normal.

$$v_n = m.\mathbf{v} \bullet \mathbf{n} \quad (3.5)$$

If v_n is negative we calculate and apply the damping force specified in Equation 3.6

$$\mathbf{f} = -v_n \times K_{surfaceDamping} \times \mathbf{n} \quad (3.6)$$

3.2.2 Surface Friction

As well as providing repulsion forces a surface will usually be required to provide friction forces to resist motion of particles in contact with the surface. Two kinds of friction force exist; static and kinetic. Static friction forces exist between a particle and a surface when the particle is resting on the surface but is not moving with respect to the surface. Kinetic friction forces exist when the particle is moving across the surface. Static friction is always stronger than kinetic friction; a motion requires more force to start than it does to maintain. Assuming floating-point values are used to represent the particle state, the particle should be declared static relative to the surface if its tangential speed is beneath a very small non-zero limit.

If an unbalanced tangential force acts upon a particle at rest upon a surface, the particle will accelerate across the surface. Static friction seeks to exactly balance the unbalanced tangential force so that the particle has a net tangential force of zero and remains at rest. The maximum magnitude of a static friction

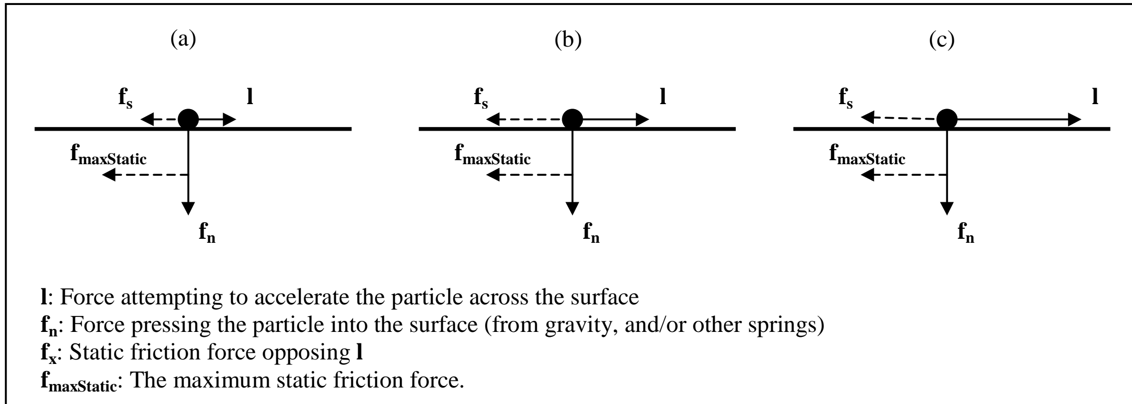


Figure 3-3. Static Friction.

force is the product of the size of the force pressing the particle into the surface and the surface's static friction coefficient, K_{static}. Equation 3.7 calculates this limit.

$$\mathbf{f_{maxStatic}} = |\mathbf{f_n}| \times K_{static} \tag{3.7}$$

Figure 3-3 demonstrates three instances of a static friction force. In (a) and (b) force **l** is smaller than $\mathbf{f_{maxStatic}}$, the maximum static friction force and can be exactly balanced by force $\mathbf{f_s}$. In case (c), force **l** is larger than $\mathbf{f_{maxStatic}}$, so cannot be balanced. In (c), the particle will "break" static friction and begin to accelerate. Force $\mathbf{f_s}$ is calculated in Equation 3.8

$$\mathbf{f_s} = \begin{Bmatrix} -\mathbf{l}, & |\mathbf{l}| \le \mathbf{f_{maxStatic}} \\ -\mathbf{l} \times \dfrac{\mathbf{f_{maxStatic}}}{|\mathbf{l}|} & otherwise \end{Bmatrix} \tag{3.8}$$

If the particle is in contact with the surface but is not at rest, we must apply kinetic friction rather than static friction. Kinetic friction acts to oppose velocity rather than unbalanced forces and is totally independent of the speed of the particle as long as the speed is non-zero. The kinetic friction coefficient $K_{kinetic}$ of a surface is always smaller than its static friction coefficient K_{static}. Equation 3.9 calculates the kinetic friction force $\mathbf{f_k}$.

$$\mathbf{f_k} = -\mathbf{v} \cdot \frac{|\mathbf{f_n}| \times K_{kinetic}}{|\mathbf{v}|} \tag{3.9}$$

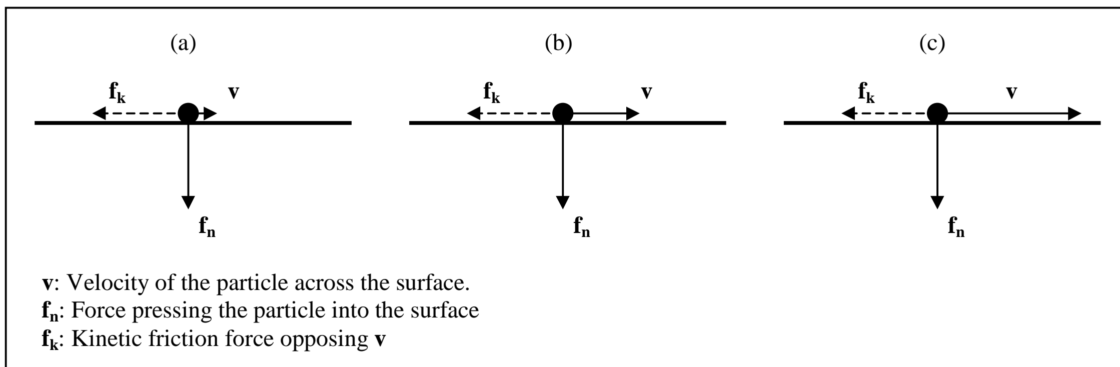


Figure 3-4. Kinetic Friction.

Figure 3-4 demonstrates three situations in which only the particle's velocity differs. In all cases the kinetic friction force applied is the same.

In Figures 3-3 and 3-4 the surface was always horizontal and extended to infinity in both directions. For our piecewise linear terrain we must be able to apply friction forces to particles in contact with surfaces of arbitrary gradient.

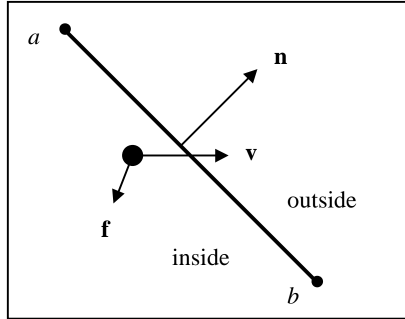


Figure 3-5. Friction calculation for our segmented terrain.

a,b: Endpoints of the surface segment
n: Unit normal for the terrain segment
f: Total force (excluding surface-repulsion) acting upon the particle.
v: Velocity of the particle.

Figure 3-5 illustrates a particle penetrating a segment of our terrain as described in section 3.2.1 above. We must determine which form of friction should be applied. Equation 3.10 calculates $\mathbf{v_t}$, the component of the particle's velocity in the direction of the surface.

$v_t = \mathbf{v} \bullet \mathbf{t}$, where **t** is a unit tangent for the surface. (3.10)

$$\mathbf{f_n} = -\mathbf{f} \bullet \mathbf{n} \quad (3.11)$$

$$\mathbf{l} = \mathbf{f} + \mathbf{f_n} \quad (3.12)$$

To calculate $\mathbf{f_n}$, the force pressing the mass into the surface we obtain the component of the total force acting on the mass in the direction of the surface normal (Equation 3.11). Depending on the value of v_t we apply either static or kinetic friction as shown in Figures 3-3 and 3-4 respectively. If v_t is "zero" we apply static friction, obtaining the unbalanced tangential force **l** by using Equation 3.12. Otherwise, we apply kinetic friction using our value of v_t obtained in Equation 3.10.

3.3 Animating the Mass-Spring System

In general, the dynamics of mass-spring systems are too complex to solve analytically. Instead, we use numerical methods to approximate the behaviour of the system over time. The system can be modelled as a set of Ordinary Differential Equations (ODEs) as shown in Equation 3.13

$$\mathbf{x}' = f(\mathbf{x}, t) \quad (3.13)$$

where **x** is the system state (typically a vector), *t* is the time parameter, $\mathbf{x}'$ is the state derivative vector and *f* is a function that can be evaluated for the required ranges of state and time. In the classic Initial Value Problem we know the state $\mathbf{x}_0$ at t_0, the starting time, and we need to calculate the state **x** as time *t* increases.

Figure 3-6 illustrates the change in state of a variable with respect to its derivative. It is important to note that the derivative field will usually change over time; it is usually defined in terms of other variables within the system.

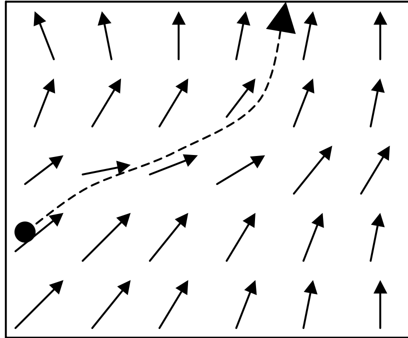


Figure 3-6. Variable in a Derivative Field

The variable moves through state-space over time. A derivative field is defined over the state space, influencing the variable's state.

Small arrows indicate derivative values at certain positions in state space. The dashed path tracks the variable's trajectory.

The derivative of **x** with respect to time, $\mathbf{x}'$, defines how **x** changes as time increases. An ODE solver uses $\mathbf{x}'$ and **x** to advance **x** in discrete timesteps. By taking sufficiently small timesteps we can approximate the true path of **x** to arbitrary accuracy. Section 3.5 describes three common ODE solvers and discusses their accuracy with respect to time-step size.

For our mass-spring system, our state variables are the positions and velocities of each mass. All derivatives are with respect to time. The derivative of position is velocity, and the derivative of velocity is acceleration. Acceleration is simply the total force acting upon a mass divided by its mass scalar as shown in Equation 3.14

$$a = f/m \qquad (3.14)$$

3.4 ODE Solvers

An ODE solver is a function or algorithm that calculates a subsequent state for a variable based on its current state, **x**, and its derivative, $\mathbf{x}' = dx/dt$. As well as those described below, there are many specialised forms of ODE solver such as predictor-correctors and Bulirsch-Stoer methods. Due to the wide variety of differential equations to which an ODE solver may be applied there is no single "best" solver; instead, there are usually several "good" solvers with varying degrees of speed and accuracy. It is often best to experiment with several solvers on the problem domain and choose the one that provides the best trade-off between speed and accuracy. In the subsections below we shall describe each solver and comment upon its accuracy with respect to stepsize. In all cases, the expected error of a solver decreases with stepsize and the number of higher derivatives evaluated. In all cases, stepsize $\ll 1$.

3.4.1 Euler

The Euler method is the simplest form of ODE solver and uses only the first derivative of the variable. Equation 3.15 specifies the Euler function.

$$\mathbf{x}_{0+h} = \mathbf{x}_0 + h\mathbf{x}_0', \text{ where } h \text{ is the step size.} \qquad (3.15)$$

This method only considers the variable's first derivative and has an expected error $e \approx O(h^2)$.

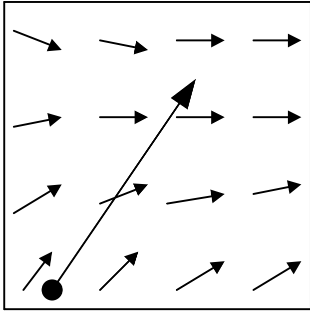


Figure 3-7. The Euler ODE solver

The variable's path to its new state after a single Euler step.

3.4.2 Midpoint

The midpoint method tries to obtain a more accurate approximation of the variable's path through state space by determining the variable's derivative at half an Euler step forwards in time. A full Euler step is then taken using this "midpoint" derivative. The midpoint method uses an approximation to the variable's second derivative, so error $e \approx O(h^3)$.

$$\mathbf{x}_{0+h} = \mathbf{x}_0 + h\mathbf{x}'_{0+h/2}$$

where $\mathbf{x}'_{0+h/2}$ is the derivative at point $\mathbf{x}_{0+h/2}$, $\mathbf{x}_{0+h/2} = \mathbf{x}_0 + \frac{h}{2}\mathbf{x}'_0$

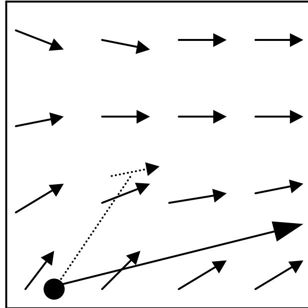


Figure 3-8. The Midpoint ODE solver

The derivative at half an Euler step.

The variable's path to its new state after a single Midpoint step.

3.4.3 Runge-Kutta 4th order

The Runge-Kutta 4th order (RK4) is so called because it approximates the fourth order derivatives of the variable. First, we take half an Euler step and calculate the derivative, $\mathbf{x}'_1$. We then return to the starting state and take another half Euler step using $\mathbf{x}'_1$. We obtain $\mathbf{x}'_2$, the derivative after the second Euler step. We now take a single full Euler step from the original point using derivative $\mathbf{x}'_2$ and obtain a last derivative, $\mathbf{x}'_3$, at this point. We then advance the state of $\mathbf{x}$ by using a blend our four derivatives as described in Equation 3.16. Figure 3-9 illustrates the RK4 algorithm in action. The RK4 algorithm has an expected error $e \approx O(h^5)$.

$$\begin{aligned}
\mathbf{x}'_1 &= f(\mathbf{x}_0 + \mathbf{x}'\frac{h}{2},\ t_0 + \frac{h}{2}) \\
\mathbf{x}'_2 &= f(\mathbf{x}_0 + \mathbf{x}'_1\frac{h}{2},\ t_0 + \frac{h}{2}) \\
\mathbf{x}'_3 &= f(\mathbf{x}_0 + \mathbf{x}'_2 h,\ t_0 + h) \\
\mathbf{x}_{0+h} &= \mathbf{x}_0 + \left(\frac{1}{6}\mathbf{x}' + \frac{1}{3}\mathbf{x}'_1 + \frac{1}{3}\mathbf{x}'_2 + \frac{1}{6}\mathbf{x}'_3\right)h
\end{aligned} \tag{3.16}$$

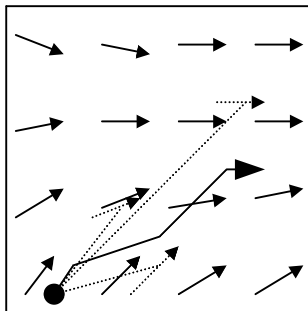


Figure 3-9. Runge Kutta 4th order ODE solver.

Intermediate derivatives

The variable's path to its new state after a single Runge-Kutta step.

3.4.4 Accuracy

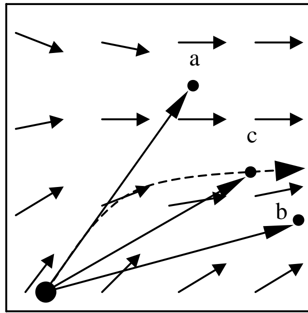


Figure 3-10. Accuracy Comparison

- - -► The true path of the variable during the timestep.

The variable's state according to the:

- a Euler method.
- b Midpoint method.
- c Runge-Kutta 4th order method.

Figure 3-10 contains the state of the variable after a single step according to the three ODE solvers and the true state of the variable. For our simple example domain the three methods are ranked *Euler<Midpoint<RK4* according to accuracy. All three methods are capable of approximating the true path through state space to within an arbitrary small tolerance, but for the Euler method this may require an extremely small step size. For many computational tasks the RK4 method is considered to be a good compromise between speed, accuracy and ease of implementation.

3.4.5 Adaptive Stepsizing

Choosing an appropriate stepsize for the domain is important. Ideally, we want a stepsize that is sufficiently small to allow accurate approximation of rapid changes in state space, but large enough that we can simulate the system at a reasonable speed. Adaptive Stepsizing refers to the automatic adjustment of the step size according to local topology of the state space. By adjusting the stepsize we can take small steps through rapidly changing areas and large leaps through smoother more predictable space. Figure 3-11 outlines the adaptive stepsize algorithm. Note that instead of specifying a step size h, we specify a maximum acceptable error e_{max} and an appropriate value for h is chosen automatically.

begin

  calculate $\mathbf{x_a}$, the state after a single step of size h

  calculate $\mathbf{x_b}$, the state after two steps of size $\frac{h}{2}$

  calculate e, an estimate of the error:

    $e = |\mathbf{x_a} - \mathbf{x_b}|$

  calculate h_{new}, the appropriate stepsize given the current error e and the maximum acceptable error e_{max}.

  $h_{new} = h\left(\frac{e_{max}}{e}\right)^{\frac{1}{l}}$, where l is the order of the ODE solver.

  if $h_{new} < h$ **then**

    calculate $\mathbf{x}$, the state after a single step of size h_{new}.

  else

    $\mathbf{x}$ = xb

  $h = h_{new}$

end

Figure 3-11. The Adaptive Stepsize algorithm.

Adaptive stepsizing can be computationally expensive. Note that for each timestep we advance we must take two or three ODE solver steps compared with only one for a fixed stepsize. If we are to benefit from adaptive stepsizing we must be able to reduce the total number of steps taken during the course of our simulation. This can only be achieved by taking larger steps, which will only be possible in a relatively slow-changing, smooth region of derivative space. The archetypal function for which adaptive stepsizing is

suited contains large tracts of smooth derivative space interspersed with occasional areas of rapid change. Such a function allows the stepsize to alter over a significant range; at least an order of magnitude. If our function is very homogenous or allows only a narrow range of stepsizes, a fixed stepsize may yield faster computation with no significant change in error.

3.4.6 Performance Comparison, Conclusions.

For mass-spring systems the dominant computational cost is calculating the state derivative (function *f* in Equation 3.14). Using this operation as a performance measure we see that the Euler method requires only one evaluation, the midpoint method requires two and the RK4 method requires four. When comparing the different solvers we have two concerns:

- *Accuracy and numerical stability*. The solver must approximate the dynamics of the system to within a certain tolerance. For mass-spring virtual creatures this tolerance level is rather vague; We are usually more concerned with whether the animation *appears* physically accurate rather than *is* physically accurate. In any case, the system must not become numerically unstable because of accumulated error due to inaccurate simulation.
- *Speed of simulation*. We obviously wish to simulate the system as rapidly as possible for any fixed accuracy criteria.

Table 3-1 presents the results of applying a visual stability metric to the three ODE solvers.

ODE solver	Euler	Midpoint	RK4
Maximum stable step size	0.0075	0.045	0.08
Time per simulation run at maximum step size (seconds)	29	10.2	12.1

Table 3-1

Of the three ODE solvers the midpoint method appears to offer the best combination of performance and accuracy. The RK4 method is only slightly slower; it's ability to take larger steps is more than offset by the increased computational cost per step. The Euler method is totally outclassed.

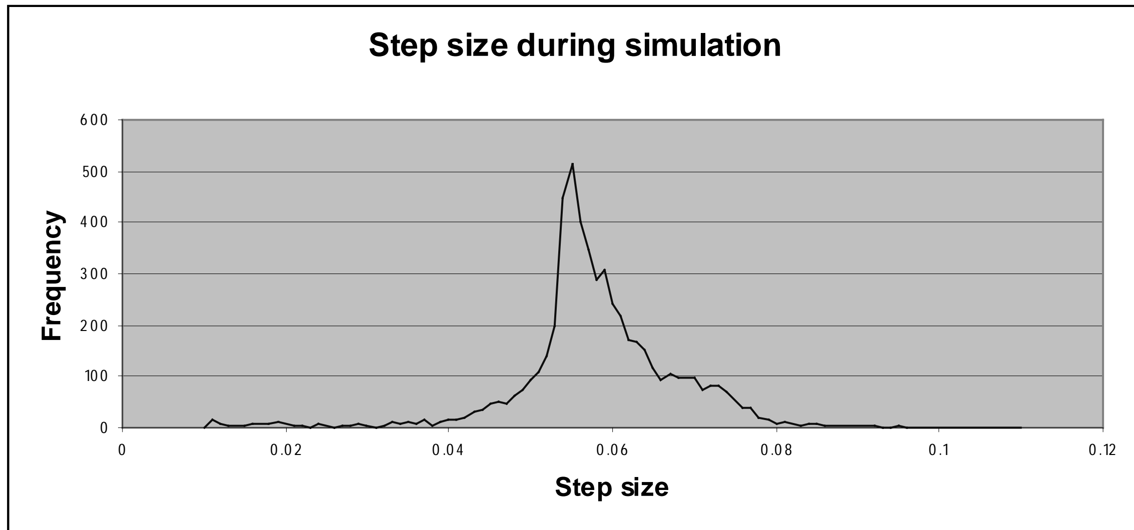


Figure 3-12. Adaptive step size during a period of simulation.

Adaptive stepsizing was applied to the mass-spring systems introduced in this chapter. Figure 3-12 illustrates the range of stepsizes observed during a typical period of simulation (approximately 30 seconds simulation time). A maximum error of 0.01 was specified and the ODE solver was the midpoint method. The stepsize values observed lie within the range [0.01, 0.12] and more than 97% of stepsize values lie

within the range [0.04, 0.08]. A range of creature simulations[2] were run using both adaptive stepsizing and a fixed stepsize of 0.02, which appears to offer at least the same degree of accuracy as adaptive stepsizing. Table 3-2 compares the running times of these two sets of simulations.

	Adaptive step size (seconds)	Fixed step size (seconds)
Simulation 1	37	21
Simulation 2	17	21
Simulation 3	32	21
Simulation 4	15	21
Simulation 5	29	21
Average	26	21

Table 3-2

Table 3-2 clearly shows that adaptive stepsizing does not offer a great increase in simulation speed for mass-spring systems. In some cases such as simulations 2 and 4 a small advantage to the adaptive method was noted. However, on average the fixed step size method appears to offer a moderate increase in simulation speed without a significant loss of accuracy.

We conclude that a combination of the midpoint ODE solver and a fixed step size will yield the best speed/accuracy trade-off in the simulation of mass-spring systems. We use this combination for all further simulation in the evolution of virtual creatures.

[2] The mass-spring creature used and world parameters for these trials are specified in sections 6.1.1 and 6.3

Chapter 4 Evolution of Controllers

Evolutionary Algorithms (EAs) are search techniques based on the process of evolution via natural selection. EAs have proven to be robust, general methods capable of efficiently searching vast high-dimensional spaces and finding optimal or near-optimal solutions [Gold_89]. Given our subject matter it seems only appropriate that we should use an evolutionary technique to synthesise our controllers.

4.1 Evolutionary Algorithm Introduction

An EA uses a generate-and-test methodology. A set of solution candidates (called a *population*) is maintained by the algorithm, and the optimality of candidates in the population increases over time. Candidates are evaluated according to some optimality measure (called a *fitness function*), and those with the highest optimality are investigated further. The optimality of the population is improved over time by generating new candidates from old ones via *reproduction* methods modelled on natural processes. EAs are often referred to as parallel beam search techniques.

If the population size remains constant and reproduction takes place in small increments the algorithm is said to be *steady state*. The other common reproduction strategy is to generate a new population in one large operation after all candidates have had their optimality evaluated. In this case the algorithm is said to be *generational*. A comparison of the properties of generational and steady state EAs can be found in [Noever_92]

4.2 Algorithm Overview

Our algorithm is a traditional generational EA based on that presented by Sims [Sims_94]. Our population consists of controller candidates for a creature of static morphology. Our fitness functions are defined in terms of the phenotype's mass-spring system and its interaction with the simulated environment during a short period of simulation. Figure 4-1 contains a high-level outline of our algorithm. Our initial population is a set of small, randomly generated controllers.

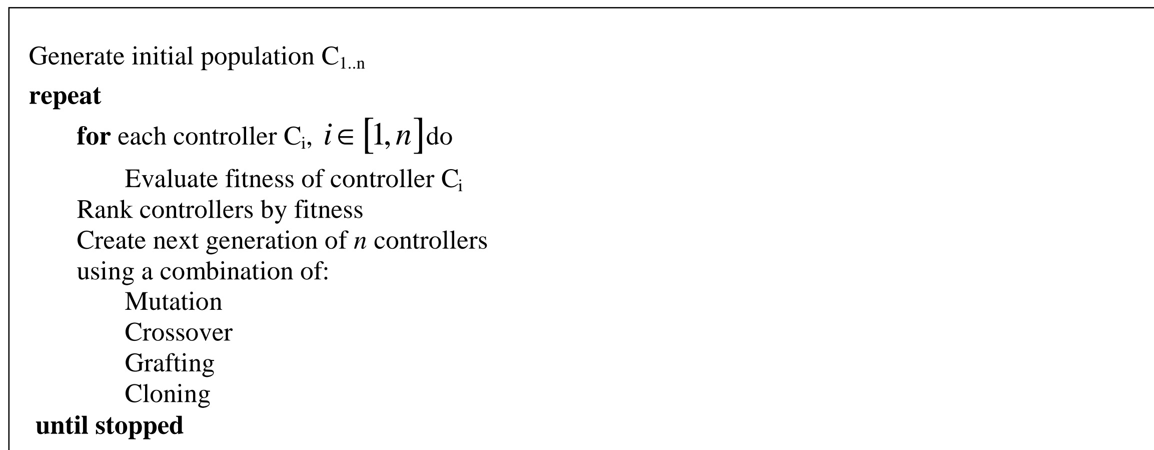

Generate initial population $C_{1..n}$
repeat
    for each controller C_i, $i \in [1, n]$ do
        Evaluate fitness of controller C_i
    Rank controllers by fitness
    Create next generation of *n* controllers
    using a combination of:
        Mutation
        Crossover
        Grafting
        Cloning
until stopped

Figure 4-1. Evolutionary algorithm overview.

4.3 Selection Mechanisms

The process of natural selection, or "survival of the fittest" is taken from nature, and provides a mechanism by which we can investigate promising solution candidates at the expense of less promising ones. For generational EAs this selection operation takes place when creating the next generation of candidates, and "survival" refers to the candidate's genetic content, i.e. the number of offspring created by the candidate.

```
for i=1 to populationSize do begin

    parent_a = selectParent(controllers);
    parent_b = selectParent(controllers);

    choose a reproduction method, method
    if method is Mutation then
        newControllers[i]= mutate(parent_a);
    else if method is Cloning then
        newControllers[i]= clone(parent_a);
    else if method is Crossover then
        newControllers[i]= crossover(parent_a, parent_b);
    else {Grafting}
        newControllers[i]= graft(parent_a, parent_b);

end for

Data structures:
newControllers:  An array [1..populationSize] for storing the new generation of candidates.
controllers:     An array [1..populationSize] containing the current generation, sorted
                 into non-increasing order of fitness.
```

Figure 4-2. The generational reproduction algorithm.

Figure 4-2 illustrates the selection mechanism in the context of the algorithm used to create the subsequent generation. The selectParent(*controllers*) method implements a probabilistic algorithm that chooses a controller candidate from the given population in such a way that fit candidates are more likely to be selected than unfit candidates. There are two common implementations of this method; *fitness proportionate selection* and *rank selection*.

4.3.1 Fitness Proportionate Selection

In fitness-proportionate selection the probability of a candidate being selected for reproduction, $p_{candidate}$, is directly proportional to its fitness.

$$p_{candidate} = \frac{F_{candidate}}{F_{total}},\ where\ F_{total} = \sum_{i=1}^{populationSize} controllers[i].fitness$$

4.3.2 Rank Selection

Rank selection algorithms set the selection probability for a candidate based on the candidate's rank according to fitness in the population, rather than the candidate's raw fitness value. This function may be linear or non-linear.

$$p_{candidate} = \mathrm{f}\left(R_{candidate}\right),\ where\ \left(\sum_{x=1}^{populationSize} \mathrm{f}\left(x\right)\right) = 1,$$

$$\mathrm{f}\left(x+1\right) < \mathrm{f}\left(x\right),\ x \in \left[1,\ populationSize-1\right],\ and\ R_{candidate}\ is\ the\ candidate's\ rank$$

Our rank selection algorithm is non-linear, and is specified in Equation 4.1

$$selectedParent = populationSize \times q^{bias}, \tag{4.1}$$

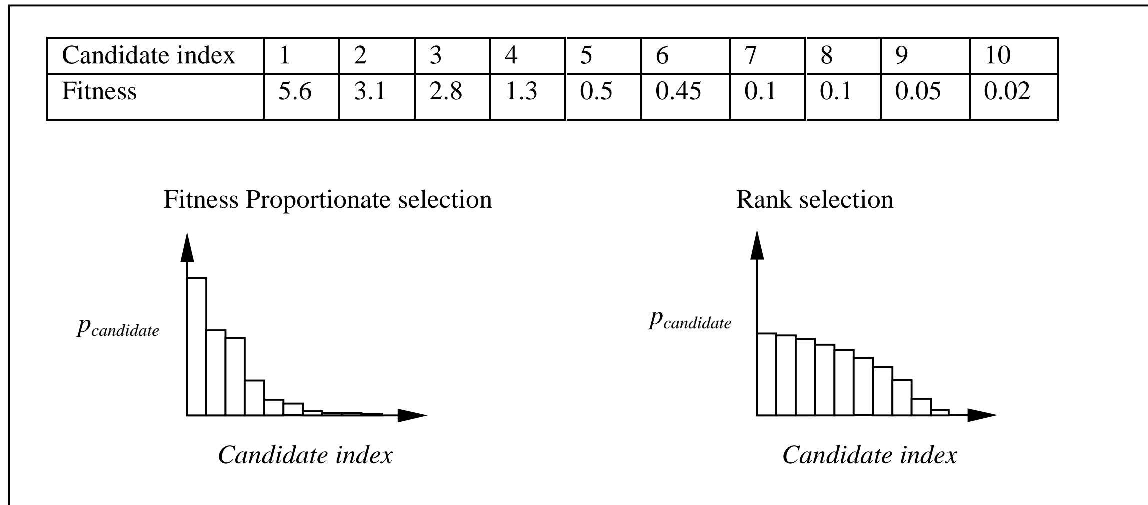
where q is a random real-valued number[3] in the range [0, 1), and $bias \geq 1$

[3] Note: a new random value for q is generated each time the selection algorithm is evaluated.

4.3.3 Properties

We define *selection pressure* to be the degree of bias of a selection mechanism towards choosing fit candidates over less-fit candidates. High selection pressure means that fit candidates are likely to produce many offspring, and unfit candidates are likely to produce few or none.

Fitness proportionate selection is the most widely used selection mechanism in EAs, but may apply too much selection pressure for some domains. In search spaces containing many local maxima, high selection pressure may lead to premature convergence upon a local peak. On the other hand, in predominantly convex search spaces high selection pressure should yield fast convergence to the global maximum. Rank selection provides a less intense selection pressure. Figure 4-3 illustrates the selection preferences expressed by the two methods discussed in 4.3.1 and 0 over a small population.



Candidate index	1	2	3	4	5	6	7	8	9	10
Fitness	5.6	3.1	2.8	1.3	0.5	0.45	0.1	0.1	0.05	0.02

Figure 4-3. Selection probability under Fitness Proportionate and Rank Selection methods.

It is often desirable to exclude unfit candidates from selection by only selecting from a subset of the population. This subset is typically a top portion of the entire population and is obtained by applying a *survival ratio*. Following Sims [Sims_94] our EA uses a survival ratio of 0.2.

4.4 Reproduction Methods

Except in the initial population, all solution candidates are created by reproduction. A reproduction method creates a new candidate by copying and modifying one or more ‘parent’ candidates. Modification of the parent(s) genetic information is a random process, and may result in major or minor differences between the parent(s) and the child. Although most random modifications will be detrimental to the child’s fitness, a few may be beneficial. Coupled with a selection mechanism as described in section 4.3, reproduction allows the fitness of the population to increase over successive generations. We implement four reproduction methods for our directed graph genotype. A theoretical and experimental comparison between mutation and crossover can be found in [Spears_92].

4.4.1 Mutation

Mutation is the primary method by which new controller nodes are added to a candidate. A single parent is chosen, cloned, then the clone is subjected to random alterations in both content and structure. Mutation involves two distinct phases:

- Adding nodes to the graph

  The number of new nodes to add, *newNodeCount*, is a randomly generated integer in the range [0, *MaxNewNodes*], where *MaxNewNodes* is an adjustable parameter of the EA. For each new node we select a random location within the creature’s genotype morphology (including the Unassociated Area). The choice of location may restrict the types of node that we can add, as mentioned in section 2.1. Each

new node is generated randomly from the available types; its internal parameters and the initial weights of its input connections are randomised to lie in the range [-1, 1]. The node's input connections are initially connected to Constant nodes, and the addition of the new node to the chosen location has no immediate functional effect on the controller. Figure 4-4 outlines the algorithm for the addition of new controller nodes.

for *i*=1 to *newNodeCount* do **begin**

location = randomLocation()
if *location* is the Unassociated Area **then**
generate *newNode*, a random neuron node
else if *location* is the Root Node **then**
generate *newNode*, a random sensor or neuron node.
else {The location is in one of the non-root genotype nodes}
generate *newNode*, a random sensor, neuron or effector node.

add *newNode* to the given *location*
end for

where
randomLocation() returns a particular part of the creature's genotype morphology as defined in section 2.1

Figure 4-4. Adding new nodes to the controller graph.

- Altering weights and reassigning connections.

The number of mutations (alterations) to inflict, *mutationCount*, is an integer in the range [1, *MaxMutations*], where *MaxMutations* is an adjustable parameter of the EA. *mutationCount* is generated according to a gaussian distribution about *MaxMutations*/2. For each mutation we randomly select a controller node, then randomly select a particular weight, parameter or connection to adjust. Figure 4-5 outlines this algorithm.

The mutateValue(*value*) method adjusts the given value by adding a random number in the range [-1, 1], of gaussian distribution about zero. Additionally, the magnitude of the change applied to *value* is scaled by the size of *value*; small values are altered less than large ones. This mechanism allows small numbers to be fine-tuned and large numbers to be altered greatly.

for *i*=1 to *mutationCount* do **begin**

node = selectNode()
location = the genotype morphology 'part' enclosing *node*.
component = selectComponent(*node*)

if *component* is a weight or parameter **then**
mutateValue(*component*)
else {component is a connection}
mutateConnection(*component*)

end for

where
selectNode() selects a controller node at random from the controller graph.
selectComponent(*node*) randomly chooses a weight, parameter or connection belonging to *node*.

Figure 4-5. Selecting a weight, parameter or connection to adjust.

The mutateConnection(*connection*) method is responsible for adjusting the given input connection to draw its source data from the output of a randomly chosen node. Because of the rules of adjacency specified in section 2.1, our connection will usually be restricted to choosing from a small subset of the total nodes in the controller. Figure 4-6 illustrates the result of a mutation upon the controller presented in Figure 2-11.

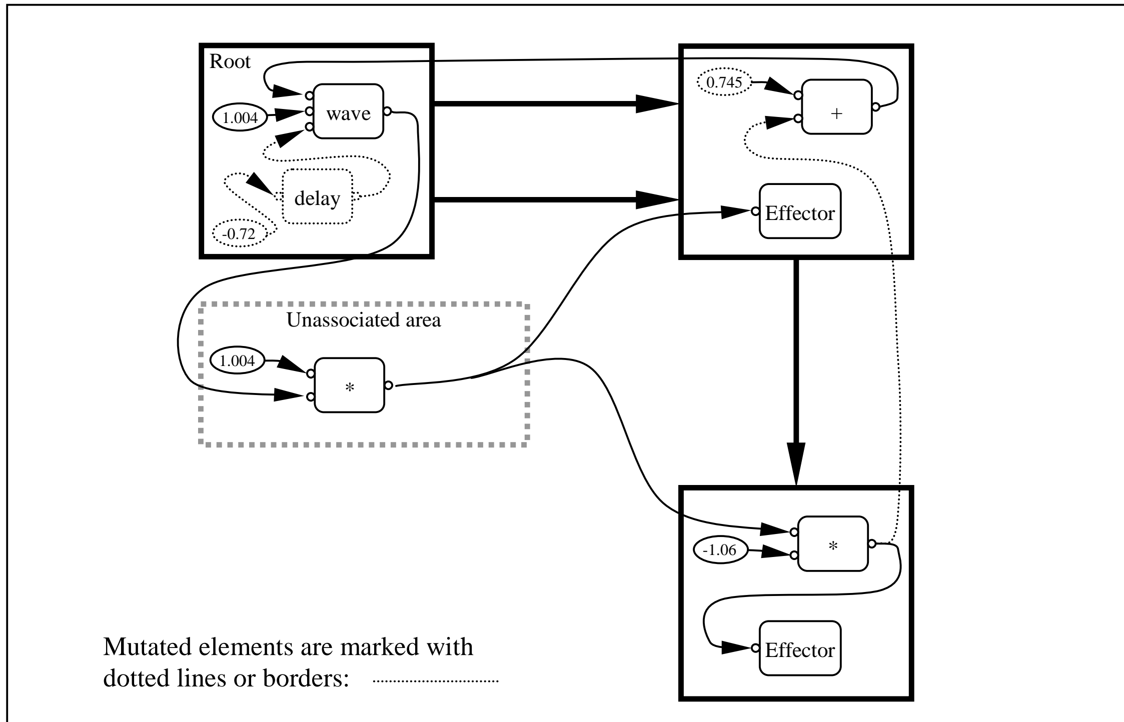


Figure 4-6. Mutation applied to a small controller graph.

4.4.2 Crossover

The phenomenon of hybrid vigour within natural systems is well documented, if not well understood. Mixing genetic information from two parents can allow offspring to combine “good” genes from each, and become fitter than either of their parents. In terms of a search algorithm, crossover can allow two solution candidates to be combined into a more optimal solution containing elements of both.

Our crossover operation involves arranging the parents’ genetic information in two parallel lines and copying information from them alternately. A randomly chosen number of *crossover points* are placed at random locations upon the lines, at which the copying swaps to the other parent’s genotype. Figure 4-7 illustrates the crossover operation applied to a simple fixed-length array genotype. Note: no ‘horizontal’ migration of genes occurs during crossover.

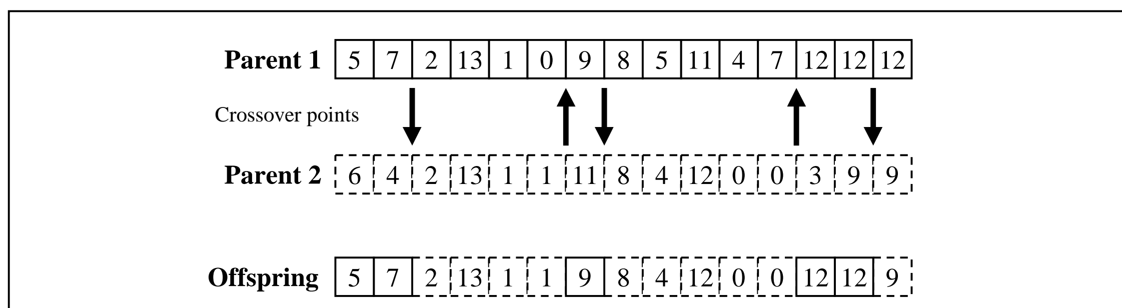


Figure 4-7. Crossover on a simple genotype.

To implement crossover between two directed graph controllers we make use of the fact that they reside within the genotype morphology graph. Crossover points are chosen randomly while copying according to an instantaneous probability *CrossoverProbability*. Figure 4-8 outlines our crossover algorithm.

```
parent = parent1
for l=1 to |locations| do begin
    n=1
    while n ≤ parent.locations[l].numberOfNodes do begin
        node = parent.locations[l].nodes[n]
        add node to offspring.locations[l]

        r = random in range [0, 1)
        if r < CrossoverProbability then
            if parent = parent1 then
                parent = parent2
            else
                parent = parent1
        n = n + 1
    end while
end for

where
    parent1, parent2 are the two parent candidates.
    offspring is a copy of the creature's genotype morphology initially devoid of controller elements.
    locations is the set of all 'parts' of the genotype morphology (The genotype nodes plus the
              unassociated area)
    CrossoverProbability is typically 0.1
```

Figure 4-8. Crossover algorithm for our genotype representation.

4.4.3 Grafting

Grafting is a two-parent reproduction method that allows 'horizontal' migration and multiple duplication of genetic information to occur. This method allows useful local sub-controllers or controller elements to be rapidly propagated throughout the creature's morphology. For simplicity, we implement grafting on a morphology 'part' basis, i.e. we copy all nodes in a source part to one or more target parts.

```
sourceLocation = choosePart(parent1)
targetCount = random integer in the range [1, |locations| ]

for i=1 to targetCount do begin
    targetLocation = choosePart(offspring)

    remove all controller nodes from offSpring.locations[targetLocation]
    copy all nodes in parent1.locations[sourceLocation] to offspring.locations[targetLocation]

end for

where
    parent1, parent2 are the two parent candidates
    offspring is initially a clone of parent2
    locations is the set of all 'parts' of the genotype morphology (The genotype nodes plus the
              unassociated area)
    choosePart(candidate) chooses a random 'part' from the set of locations in candidate.
```

Figure 4-9. The grafting algorithm.

4.4.4 Cloning

Cloning is a single-parent reproduction method that exactly duplicates the parent. This method allows controller candidates to pass unaltered into the next generation, which may be beneficial if other reproduction methods have a high probability of making detrimental changes. The presence of a cloning method may allow higher mutation rates than is otherwise the case, leading to a wider investigation of the search space.

4.4.5 Validity Enforcement and Garbage Collection

The mutation, crossover and grafting methods will often result in the offspring breaking the rules of controller validity. In section 2.1 we stated that controller nodes are only permitted to connect to nodes in the same part or nodes in immediately 'adjacent' parts. Additionally, effectors and some types of sensor are not allowed to exist in some parts of the creature's morphology. With the exception of the cloning method, we must enforce the rules of adjacent connection and node placement in every offspring.

Firstly, we check for and remove any redundant effectors or sensors. These nodes are considered redundant if there already exists in the same part a node of the same type. Secondly, we verify or reassign each connection of every controller node. If a connection is found to be invalid it is reassigned to another available node. Figure 4-10 contains a flow chart illustrating the connection verification process.

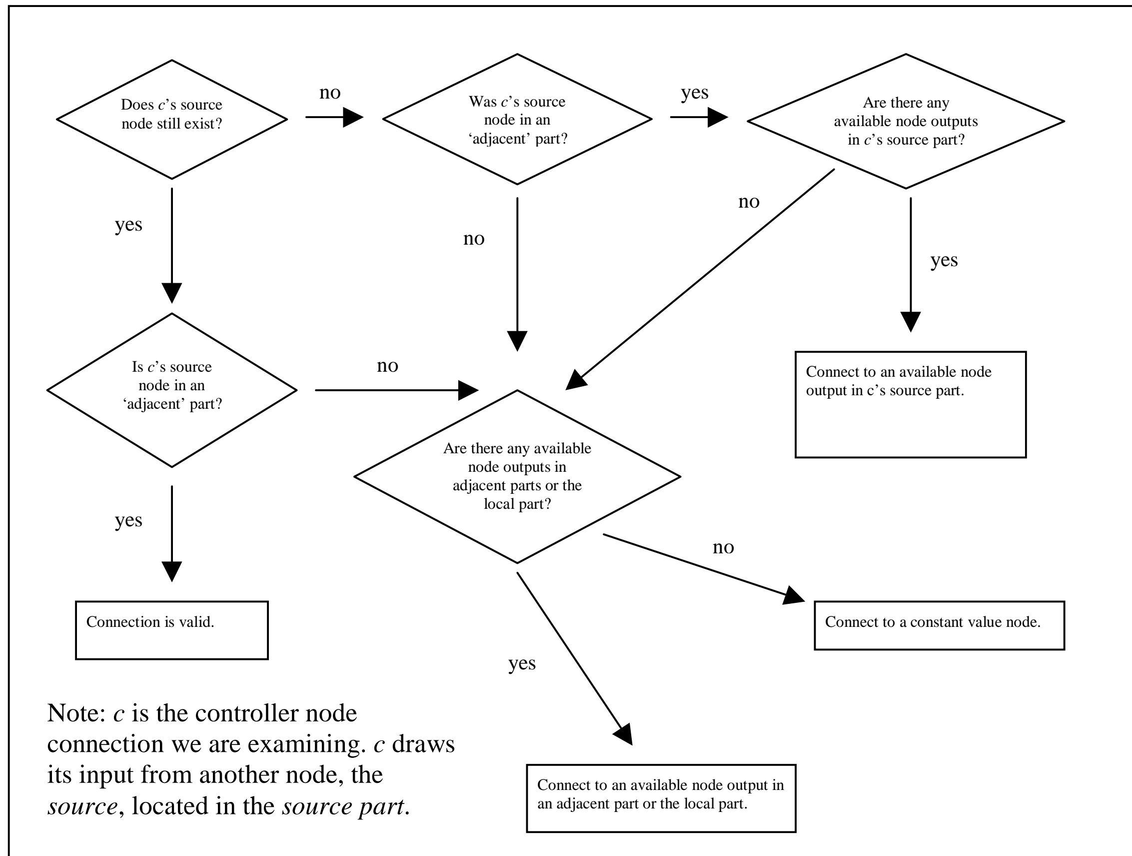


Figure 4-10. Verifying or correcting a controller connection.

Following duplication-removal and connection verification, nodes that cannot contribute to effector output are garbage collected. Garbage collection helps to keep the size of controllers to a minimum, which in turn keeps the controller evaluation time minimal (see section 2.4). The garbage collection algorithm is presented in Figure 4-11.

```
begin garbageCollect(candidate)
    mark all nodes 'unvisited'
    for each node n in the controller do
        if n is an effector then begin
            source = the source node of n's connection.
            visitNode(source)
        end if
        end for
    remove all nodes still marked 'unvisited'
end garbageCollect

begin visitNode(node)
    if node is flagged as 'unvisited' then begin
        mark node as visited
        for each input connection c of node do begin
                source = the source node of connection c
            visitNode(source)
        end for
    end if
end visitNode
```

Figure 4-11. Garbage collection algorithm.

4.5 Distributed Fitness Evaluation

Distributed fitness evaluation refers to the application of parallel processing to the task of evaluating the fitness for a population of controller candidates. In our implementation this involves sharing the cost of physical simulation across many computers on a local area network (LAN).

4.5.1 Motivation

Our fitness functions are all based around the performance of the controller when applied to the creature's body in a period of physical simulation. Physical simulation is enormously computationally expensive when compared to the EA's other tasks such as selection and reproduction. When evaluating a generation of controllers, the order of evaluation is irrelevant and there is no interaction between different controllers. All the fitness evaluations are totally independent. Making the assumption that our population size will always be greater than or equal to the number of processors available, we can easily and efficiently harness multiple processors to the task.

4.5.2 Master/Slave Architecture

Our distributed system uses a master/slave architecture. We have a single master process that implements the EA *sans* physical simulation, and multiple slave processes that implement physical simulation only.

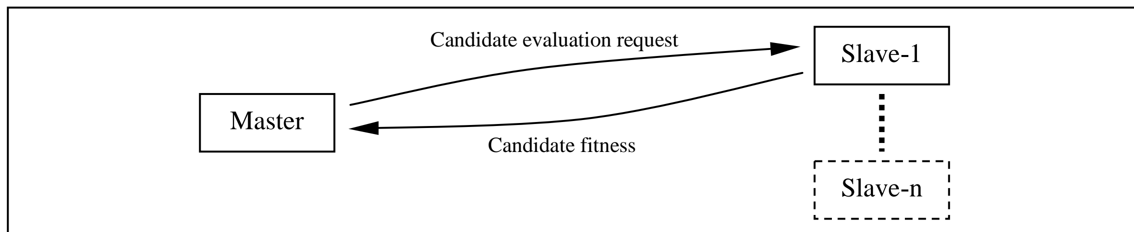


Figure 4-12. Master/Slave message passing.

To make efficient use of a fluctuating number of available processors in the face of differing processor speed, network unreliability, etc requires a flexible, fault-tolerant approach. Pseudo-code for the master's population-evaluation loop is presented in Figure 4-13

```
evaluatePopulation()
    mark all candidates as 'unevaluated' and 'unassigned'

    repeat
        candidate = getFirstUnassigned()
        if candidate = null then
            candidate = getUnevaluated()

        if candidate <> null then begin
            slave = getAvailableSlave()
            send candidate to slave for fitness evaluation
            mark slave as 'unavailable'
            mark candidate as 'assigned'
        end if
    until (getUnevaluated() = null)

    send cancellation messages to any 'unavailable' slaves
end evaluatePopulation

receiveFitnessEvaluation(candidate, slave)
    record candidate's fitness value
    mark candidate as 'evaluated'
    mark slave as available
end receiveFitnessEvaluation


where
    getFirstUnassigned()  returns a candidate that has yet to be assigned to a slave for fitness
                          evaluation, or 'null' if all candidates have been assigned.
    getUnevaluated()      returns a candidate for which a fitness value has not yet been received
                          from a slave, or 'null' if all candidates have been evaluated. If more than
                          one candidate is 'unevaluated' one of them is chosen at random .
    getAvailableSlave()   returns a slave that has not been assigned a candidate. This method will
                          not return until a slave becomes available.

Note: The receiveFitnessEvaluation(candidate, slave) method is asynchronous to the
evaluatePopulation method, and executes whenever the master receives a candidate's fitness evaluation
from a slave.
```

Figure 4-13. Distributed fitness evaluation algorithm.

Our algorithm allows for different processor speeds and the possibility of slave process termination mid-evaluation. Each slave process is given one candidate at a time, and processes candidates at its own pace. If a candidate is assigned to a slave that then terminates prior to finishing its evaluation, the candidate will subsequently be reassigned to another slave.

4.5.3 Performance Benefits and Conclusion

Performance is linear with processor power and the number of slaves available up to the size of the population. Our distributed system allows for efficient utilisation of multiple processors across a LAN, and is tolerant of the addition and removal of slave processes during evolution. Network traffic is sparse and small. Implementation of our algorithm is straightforward and is highly recommended for any evolutionary computation task with costly fitness evaluation. We cannot think of a better use for a computer laboratory during a university holiday ☺.

4.6 Summary

An evolutionary algorithm involving a generate-and-test approach to fitness evaluation is used to generate controllers. Our EA implements mutation, crossover and grafting reproduction operations for our directed-graph genotype. We have presented a simple master/slave distributed computation architecture to spread the cost of fitness evaluation over many computers on a LAN.

Chapter 5 Visualisation of Evolution

The progression of an evolutionary algorithm through a search space may be very unintuitive. A good understanding of the effects of the EA's various control parameters is required if we are to make intelligent parameter choices rather than use a blind try-it-and-see approach. Additionally, evolutionary algorithms are notoriously difficult to debug; the implementation may contain subtle bugs causing the EA to behave sub-optimally or erratically. Some form of data processing to allow human understanding and investigation of the EA's operation is required. An overview of visualisation techniques applied to evolutionary algorithms can be found in [Collins_97].

5.1 Introduction and Motivation

Humans are very good at interpreting three-dimensional visual information. Our eyes provide a very high-bandwidth input mechanism to our brain, and our brain has evolved specialised processing facilities to deal with three-dimensional data. Data-visualisation technologies use the human visual system as an intuitive and natural interface between information and intellect. To present the appearance of three-dimensional data on a two-dimensional computer screen we use animation; we allow the user to translate and rotate the viewpoint over time.

We present a simple visualisation utility designed specifically for our controller-synthesis evolutionary algorithm. Data is collected from controller candidates during evolution and is imported into the *Visualiser* utility for offline investigation. Knowledge obtained from investigating this data can be used to adjust EA parameters or may lead to the discovery of algorithmic or implementation problems.

Our Visualiser displays a point cloud within a cube in 3-space. Each point represents a single controller in the evolution sequence. The Cartesian coordinates of points within the view volume represent various metrics selectable by the user. Lines between points represent parent-child relationships. The number of points in the volume is determined by a user-specified range of generations. Point and line colours denote each controller's generation and reproduction method.

5.2 Views Available

The quantity represented by each axis can be selected from a set of data fields that are stored for each controller. Our fields are:

- Fitness.
- The number of neurons
- The number of sensors
- The reproduction method used to create the child
- The child's niche[4]

The user specifies a range of generations by selecting *start* and *stop* generations, where *start* < *stop*. The colour of a point denotes the controller's position in the generation range; controllers in the *start* generation are shown as dark blue points, ranging through to white points for controllers in the *stop* generation.

Parent-child line colour represents the reproduction method used to create the child controller:

White: Mutation
Red: Crossover
Yellow: Grafting
Blue: Cloning

[4] This field only exists if niching is used; niching is introduced in Chapter 8 .

In the case of crossover and grafting, two parent-child lines exist. Colour gradient is used to apply directionality to the line; the child always lies at the brighter end of the line. The user may select to display only parent links for which the child controller is fitter than the parent, or hide all parent links to obtain an unobstructed view of the point cloud.

Two data scaling methods are available. *Local scaling* scales data within the selected generation range to fit the view volume, and is useful for obtaining a close-up view of the selected range. *Global scaling* scales the entire evolution's data to fit the view volume, and is useful for comparing differences of scale between generations.

A simple animation facility is available to help visualise the EA's dynamics over subsequent generations. A two-generation window slowly advances through the selected range of generations. The user may rotate the viewpoint using a standard trackball interface.

5.3 Demonstration

Figure 5-1 contains a view of the first two generations of an evolution. The x, y and z axes, shown in blue, light grey and green respectively, represent the number of neurons (dataflow units), fitness and number of sensors. Note the explosion of mutated offspring from a fit controller in the first generation. Many of the mutations have resulted in superior controllers; some spectacularly so. A variety of controllers produced by grafting are extending along the z axis towards us; they contain large numbers of sensors, few neurons and have very low fitness values.

Figure 5-2 follows the same evolution for another two generations. In the fourth generation a controller created by grafting has proven to be vastly fitter than either of its parents. Other less spectacularly improved controllers have been created by crossover and grafting operations. Note: vertical scaling has changed as a result of the peak fitness increasing by a factor of 20.

5.4 Summary and Conclusions

Visualisation techniques can help provide a useful and intuitive understanding of an evolutionary algorithm's operation. Our Visualiser application simply presents raw evolutionary data in 3-space, but aids understanding considerably due to its ease of use and rapidity of data access. In particular, visualiser views are superior to textual data for examining groups of controllers or determining the distance between controllers in the search space.

Data visualisation increases the likelihood of the discovery of algorithmic or implementation bugs because patterns or occasional aberrations in a 3D data set are usually much more obvious than in tabulated textual data. Our visualisation tool has contributed to the discovery of many bugs during the course of this thesis.

Our current Visualiser has some limitations. We find that the number of parent-lines must be limited for clarity, typically to at most a thousand. The viewpoint is currently restricted to a sphere around the visual volume; the ability to move the viewpoint and centre of rotation to arbitrary locations within the volume would allow detailed investigation of sub-groups of controllers. A more flexible data-selection method than our generation range would enable the user to remove superfluous information.

In later chapters we will include visualisation views to demonstrate aspects of the evolution process. Such views will usually contain two generations (a *parent* generation and a *child* generation). Unless otherwise specified our three axes will be:

- Green: Number of sensor nodes.
- White: Fitness.
- Blue: Number of neuron nodes.

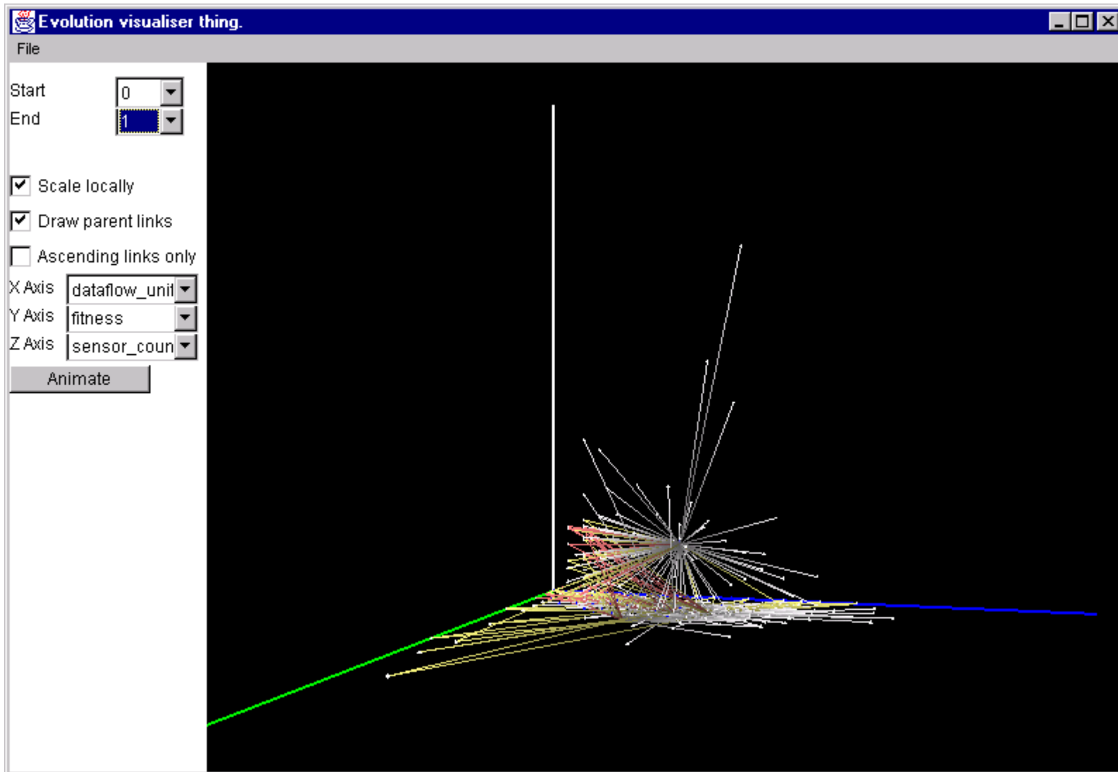


Figure 5-1. Mutation explosion.

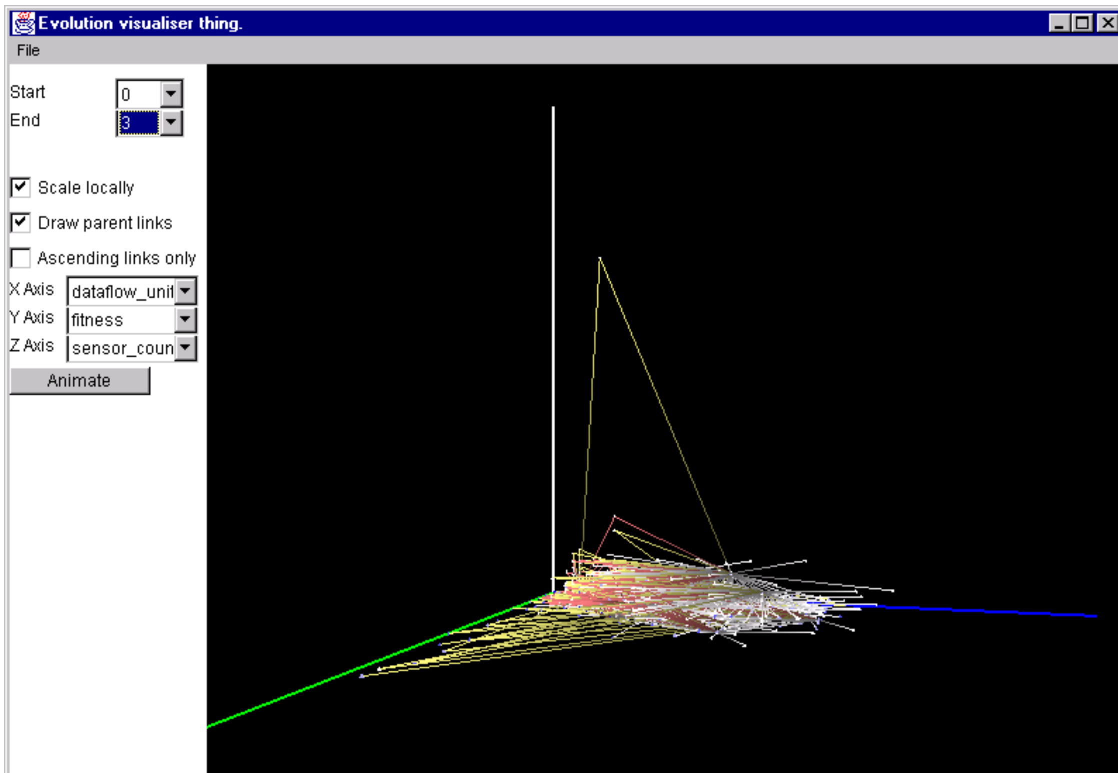


Figure 5-2. Grafting produces an improvement.

Chapter 6 Models and Parameters

Our controller synthesis experiments involve a range of creature morphologies and animation goals. We now describe these morphologies and fitness functions in detail.

6.1 Test Creatures

The set of test creatures contains worm-like, ring-like, octagonal and bipedal morphologies. A creature model contains only the minimum number of masses and springs necessary to approximate the target morphology shape; simulation speed is of higher priority to us than aesthetic issues.

6.1.1 Worm

Our primary test creature is a long, thin worm-like structure. This morphology's locomotion space was envisaged to contain stable limit cycles, so could be animated by either open-loop or closed-loop control. The genotype for this creature does not use grouping; there is a one-to-one mapping between nodes in the genotype graph and masses in the mass-spring system.

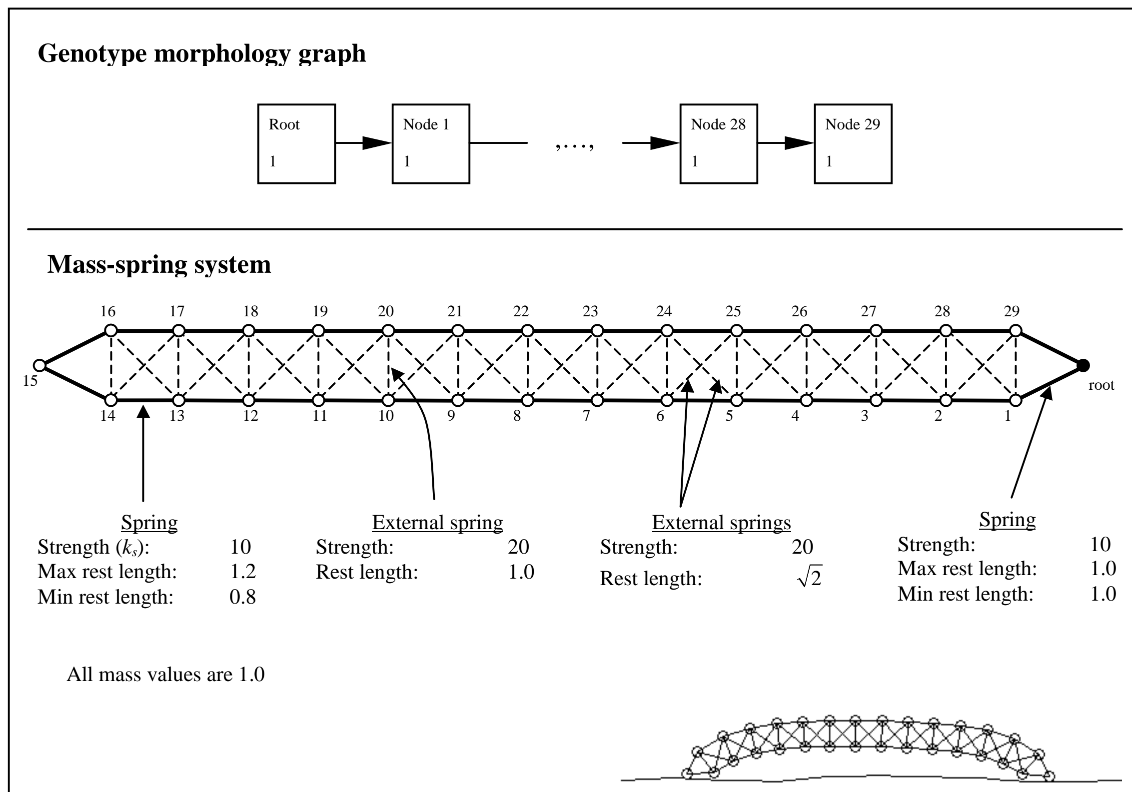


Figure 6-1. Worm model.

The creature may adjust the rest lengths of the lateral springs along its top and bottom. Extra springs are used to provide vertical and diagonal bracing for each quadrilateral cell. For our normal world conditions this creature has sufficient muscular power to form an arch, touching the terrain only at its two ends.

6.1.2 'Ring' Creature

This creature's morphology is a 16-mass ring designed for rolling. The controller may alter the rest lengths of the circumferential springs and thereby move masses along the circumference. Extra springs internally reinforce the muscular ring, resulting in a strong but flexible morphology. It was envisaged that a locomotion controller for this creature would require sensor information, i.e. be closed-loop.

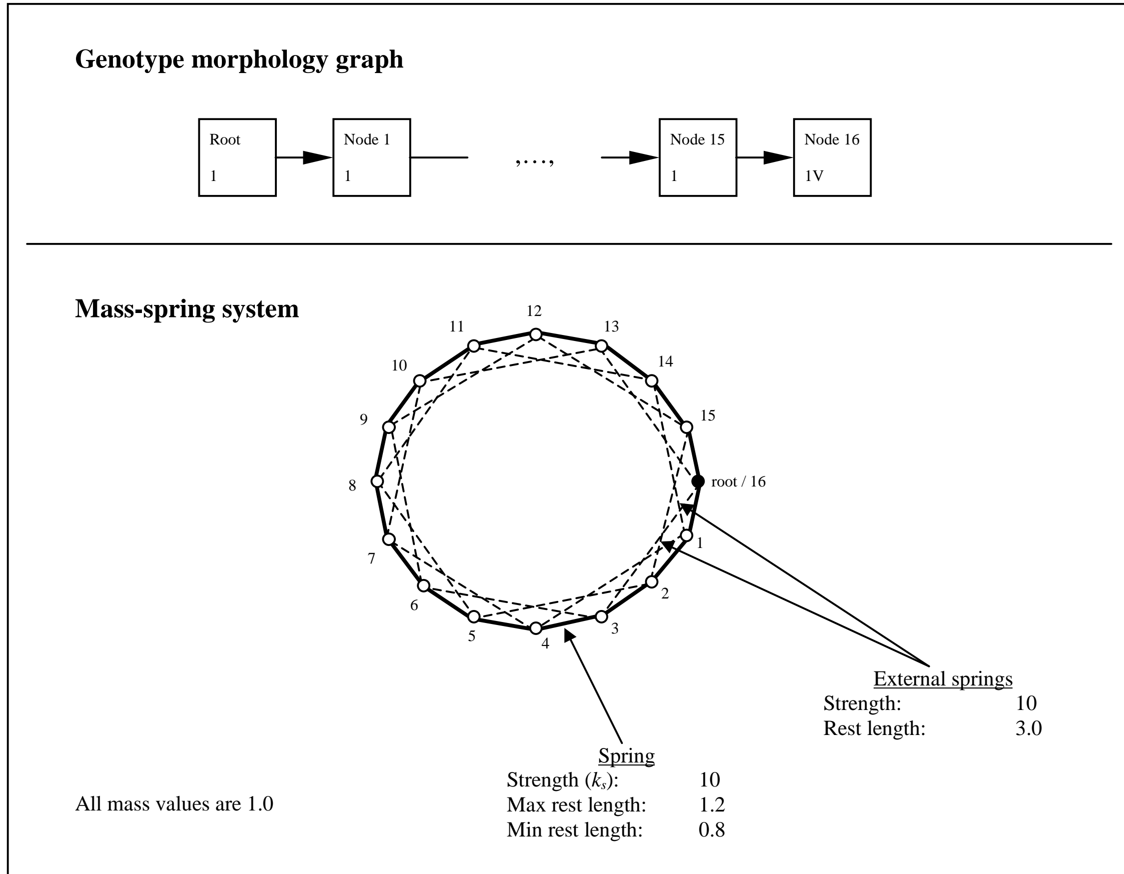


Figure 6-2. Ring model.

6.1.3 Octagonal Creature

Our octagonal creature is designed to roll. We specify the creature in two different genotype morphology graphs; with and without using grouping. Figure 6-3 illustrates the mass-spring system for our octagonal creature. Figure 6-4 illustrates its two genotype morphology graphs.

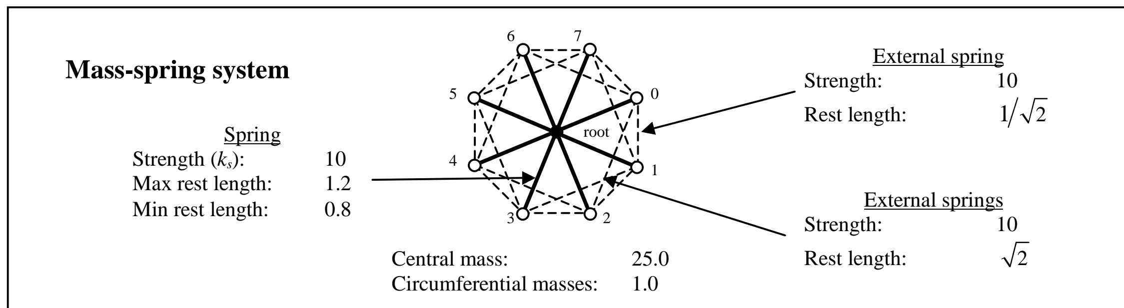


Figure 6-3. Octagonal creature mass-spring system.

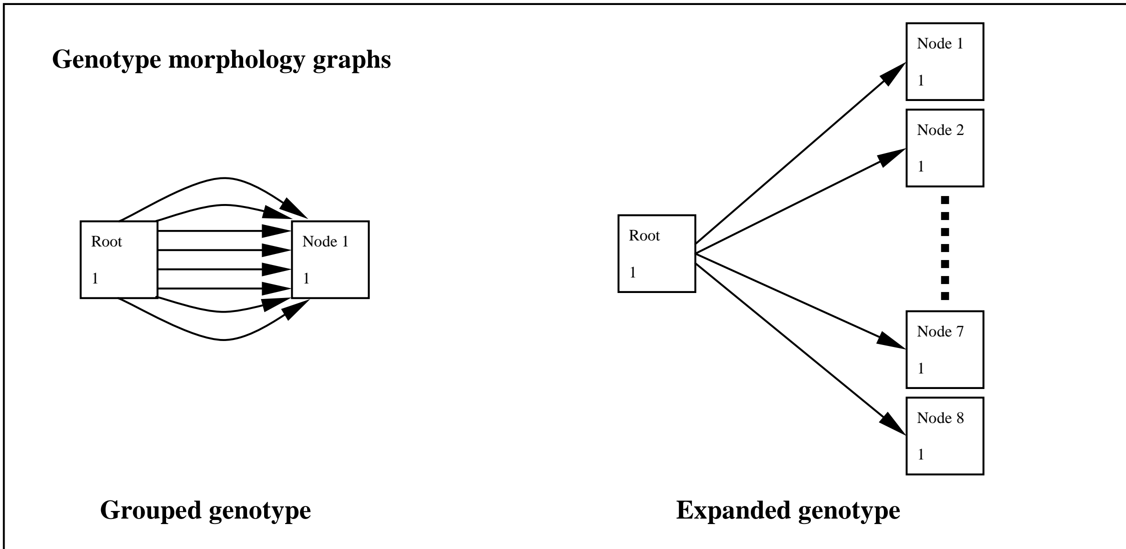


Figure 6-4. Octagonal creature genotype morphology graphs.

The grouped morphology graph forces the local controllers in different circumferential masses to be identical. Because of this enforced symmetry local controllers can only apply differing effector output if sensor units are used, thus making only closed-loop control viable. The expanded genotype allows different local controllers in different circumferential masses, so may allow either open-loop or closed-loop control.

6.1.4 Biped

We implement a simple bipedal morphology to investigate 'walking' behaviours. This creature uses an expanded genotype morphology graph, and is expected to contain both stable and unstable limit cycles.

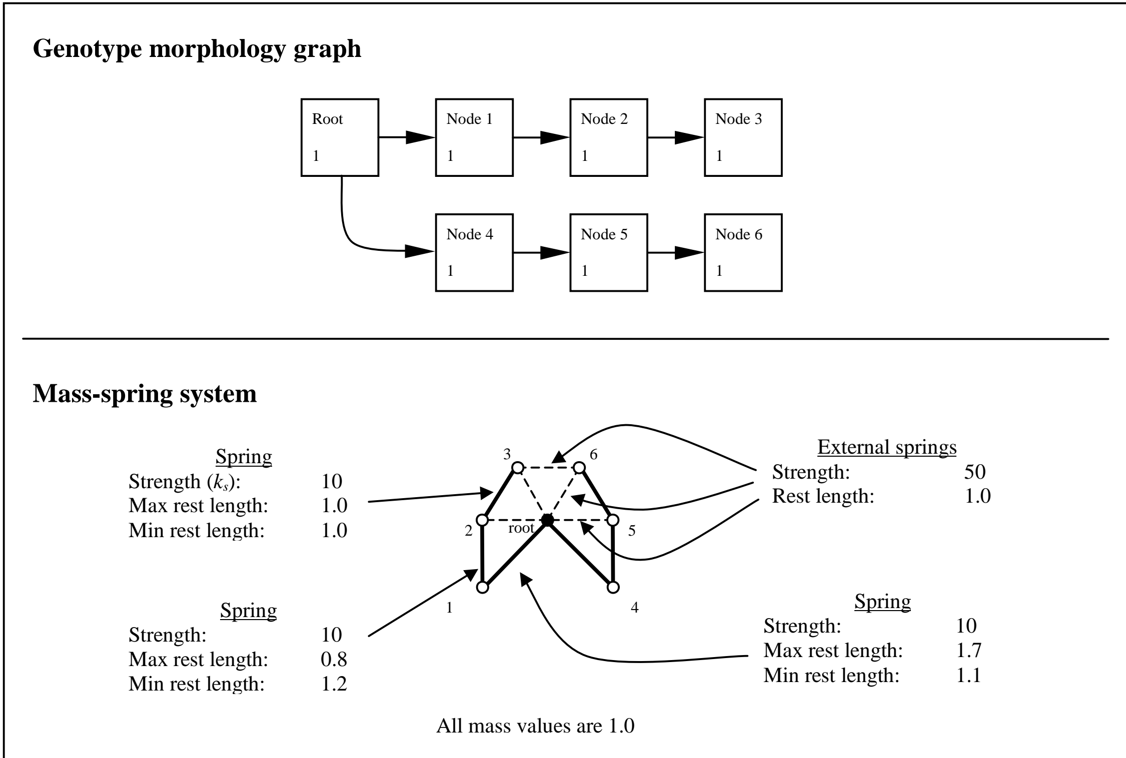


Figure 6-5. Biped model.

6.2 Fitness Functions

Our fitness functions are real-valued metrics defined entirely in terms of the creature's interaction with its simulated world. Following the work of Grzeszczuk and Terzopoulos [Grze_95] a term α evaluates the motion and a term β evaluates the controller's energy efficiency. Fitness functions are of the form $\alpha + K_{efficiency}\alpha\beta$, where $K_{efficiency}$ specifies the weighting for the efficiency term.

Our primary term α is calculated according to one of the following:

- Absolute average velocity of the centre of mass.

$$\alpha = \left| \int_0^T \left[\sum_{i=1}^{M} m_i.mass * m_i.\mathbf{velocity}.x \right] dt \right|$$

- Maximum horizontal displacement of the centre of mass

$$\alpha = \left\lceil \left| \sum_{i=1}^{M} m_i.mass * m_i.\mathbf{position}.x \right| \right\rceil \textit{ during simulation}$$

- Maximum vertical separation between the centre of mass and the terrain.

$$\alpha = \left\lceil \sum_{i=1}^{M} m_i.mass * a(m_i) \right\rceil \textit{ during simulation}$$

where T is the length of simulation, M denotes the number of masses in the creature's morphology and m_i denotes the ith mass in the creature's mass-spring system. Function $a(m_i)$ returns mass m_i's altitude, i.e. the vertical displacement between m_i and the terrain.

The energy-efficiency term β is given by:

$$\beta = \frac{1}{1 + \int_0^T c(t)\,dt}, \textit{ where } c(t) = \sum_{i=1}^{S} \left(s_i.r_{t-1} - s_i.r_t\right)^2 \Big/ S$$

where T is the length of simulation, S denotes the number of springs in the creature's phenotype morphology tree and s_i denotes the ith spring. $s_i.r_t$ denotes the rest length of spring s_i at time t, and $s_i.r_{t-1}$ denotes the rest length of spring s_i at the previous simulator timestep.

Our rationale for an efficiency term is that real-world creatures exhibit remarkably energy-efficient locomotion [Light_70]. Energy efficiency may therefore be a useful performance measure in the search for realistic locomotion. Additionally, we believe that the promotion of energy-efficient controllers may result in more graceful, aesthetically pleasing locomotion.

6.3 Parameter Description

Parameters for our experiments can be separated into two groups, simulation parameters and EA parameters.

6.3.1 Simulation Parameters

Gravity vector:	(x, y) = (0, -0.05).
Atmospheric viscosity:	0.001
Static friction coefficient, K_{static}:	1.0
Kinetic friction coefficient, $K_{kinetic}$:	0.9
Spring damping (global), K_d:	0.3
Terrain hardness:	100
Terrain damping:	0.05
Terrain height:	2.0
Terrain steepness:	0.5
ODE solver:	Midpoint
Step size:	0.02

6.3.2 Evolutionary Algorithm Parameters

Population size:	300 unless otherwise stated.
Simulation time:	600
Fitness function:	Absolute average velocity of the centre of mass.
Efficiency weighting, $K_{efficiency}$:	0.1
Survival proportion:	0.2
Selection method:	Rank selection
Rank selection bias:	4.0
Reproduction method probabilities	
Mutation:	0.4
Maximum mutations:	20
Maximum new units:	50
Crossover:	0.3
Per-unit crossover probability:	0.1
Grafting:	0.25
Cloning:	0.05

Note that the mutation probability is very high compared with other EAs; another popular form of EA, the *Genetic Algorithm* represents solution candidates as bit-strings and typically has a bit-flip mutation probability of 0.005 or less. Our mutation probability is taken from Karl Sims' work [Sims_94].

Chapter 7 Initial Results

Controller-synthesis trials were conducted using the worm, octagonal, ring and biped creature models and the parameters specified in chapter 6. For the worm and biped morphologies open-loop control was quickly discovered and optimised. For the octagonal and ring creatures closed-loop control proved dominant. The bulk of our investigation is focused upon controllers for the worm model.

Sets of trials were run for each creature. For each trial an initial random number seed was chosen based on the least significant bits of the system real-time clock. In most cases the EA was stopped after 100 generations. Some trials were allowed to continue for up to 300 generations to illustrate a particular aspect of the EA's behaviour. Due to the high computational cost of a 100-generation evolution run, the number of trials per experiment is at most 10. Parts of this work have previously been published in [Sand_2000].

7.1 Worm Model

The worm model is very stable, both statically and dynamically. We envisaged that both open-loop and closed-loop controllers could be synthesised, but that closed-loop control should have a fitness advantage. Figure 7-1 displays a few selected frames of closed-loop worm locomotion; the worm is travelling to the right using a stretch-and-anchor gait.

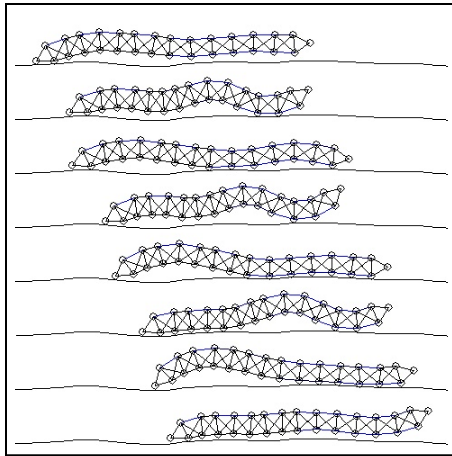


Figure 7-1. Worm locomotion

The EA synthesised a range of open-loop locomotion controllers over repeated trials with identical parameters (except the random seed). Figure 7-2 contains a graph of fitness against generation to illustrate the variation in controller optimality over different trials. Note that with the exception of three trials the controllers are of similar peak fitness and have similar fitness histories.

In three trials, the algorithm's rate of ascent during the first 10 generations is very low, and the population has relatively low peak fitness after 100 generations. Figure 7-3 traces one of these trial's fitness for more than 300 generations and suggests that the EA has become trapped in a local maximum.

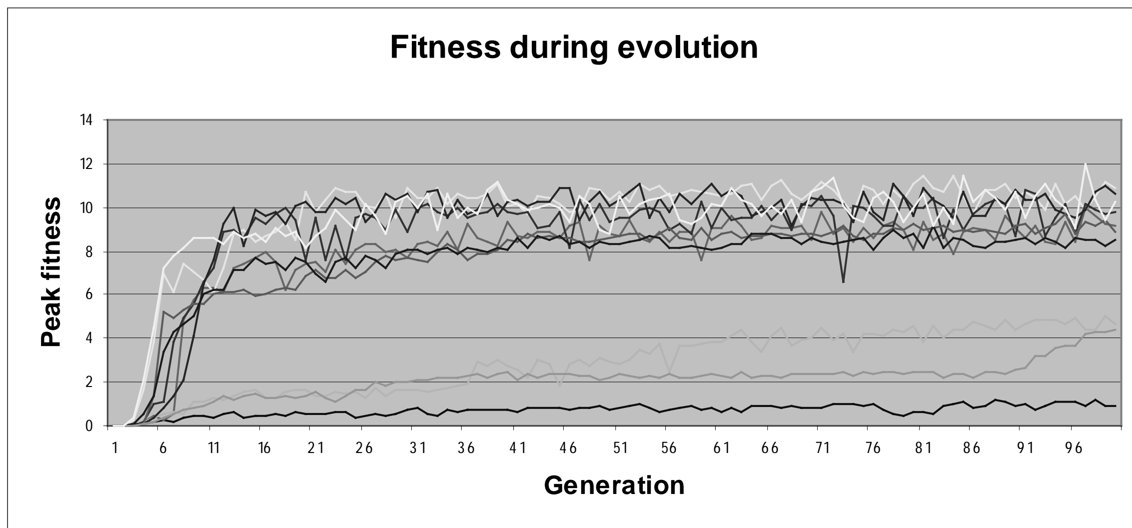


Figure 7-2. Evolution of worm locomotion controllers.

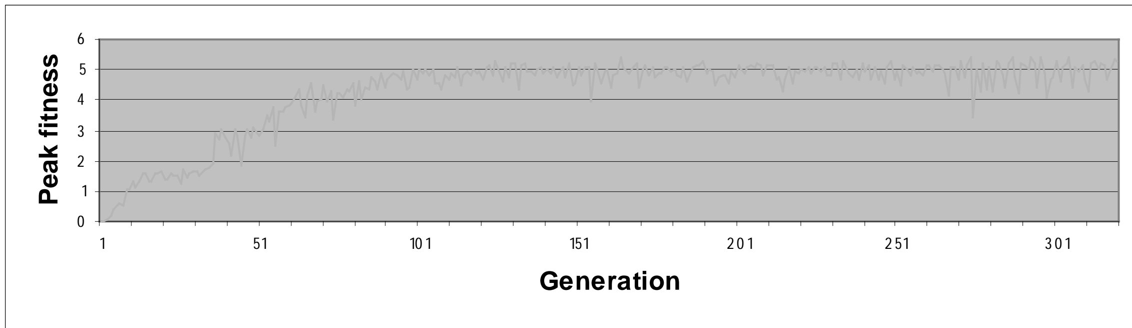


Figure 7-3. A local maximum in worm-locomotion-controller space.

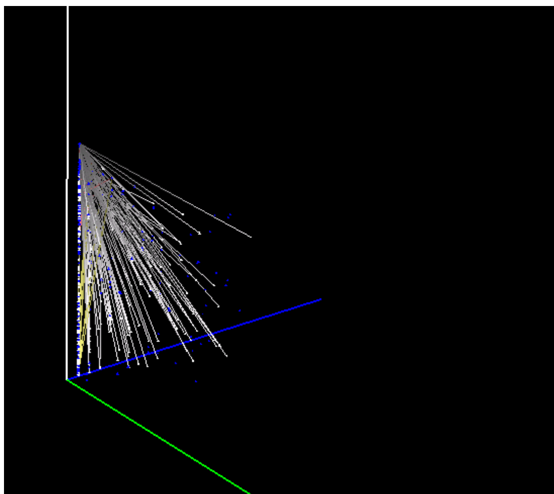


Figure 7-4. Visualisation of a local maximum.

Figure 7-4 contains a visualisation view of the trial in Figure 7-3 at generations 320-321. This view illustrates the loss of genetic diversity in the population and confirms that the EA has indeed become trapped in a local maximum. Observe that all parent-child lines appear to originate from a single point near the upper-left corner; The population has become dominated by clones (or near-clones) of a single genotype. The EA is exploring a small region of controller-space local to the dominant genotype but is unable to advance; the dominant genotype has been optimised as much as it can, and further improvement may require a different controller architecture. The probability of spontaneously generating a fit controller of a different architecture is extremely small, so the EA is unlikely to escape the local maximum in a reasonable number of generations.

We found that our EA always produced an open-loop locomotion controller based upon one or more wave generator nodes. Closed-loop control was only discovered once wave generator nodes had been disabled. This is interesting because closed-loop (sensor-based) controllers were of similar fitness to open-loop controllers and were evolved after a similar number of generations. This evidence implies that open-loop

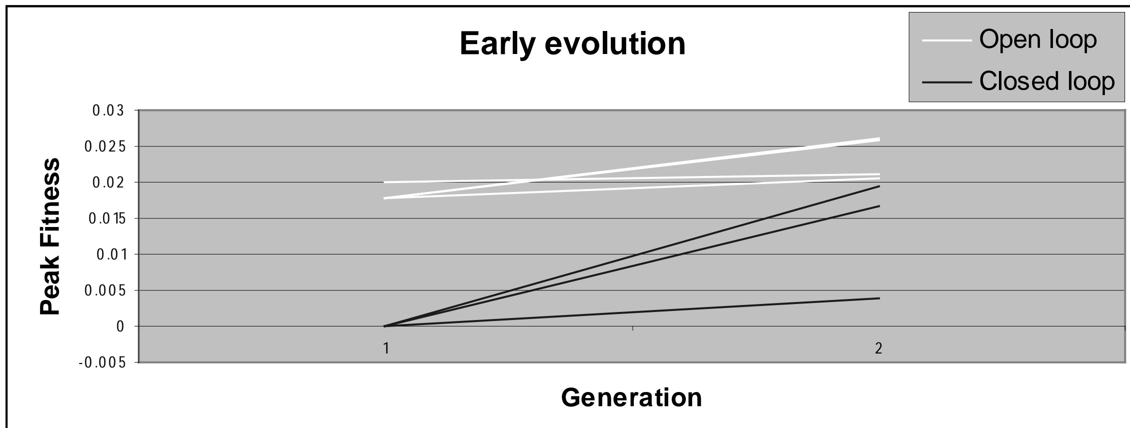


Figure 7-5. Open-loop control has an early fitness advantage.

control has an advantage early in evolution, and consequently that for this model the EA is biased towards open-loop control. Although the worm model is dynamically stable and does not require closed-loop control to reach high locomotion speeds, we are still interested in closed-loop control from an animation perspective. Figure 7-5 illustrates the fitness advantage of open-loop control in the first two generations of evolution. The trials plotted in Figure 7-5 are of two different types; In open-loop trials we have disabled sensor nodes, and in closed-loop trials we have disabled wave generator nodes. Note that closed-loop trials have a first-generation peak fitness close to zero, and that open-loop trials have a greater first-generation peak fitness of at least 0.017. Because of this early fitness advantage to open-loop control and our small survival proportion of 0.2, the EA is biased away from investigating closed-loop control and subsequently generates an open-loop locomotion controller.

We believe that the early advantage of open-loop control stems from the fact that a cyclic open-loop motion is much more likely to be produced than a cyclic closed-loop motion. Simple cyclic motions based upon one spring/muscle are the first beginnings of a locomotion controller. The periodic extension and retraction of a spring in the worm's body will usually result in an asymmetric contraction/expansion of the body as a whole, resulting in a motion bias towards the left or the right. The type of controller instigating the first such cyclic motion greatly influences the resulting course of controller evolution.

To produce a cyclic open-loop motion the EA need only connect the input of an effector node to the output of a wave generator node. The production of a cyclic closed-loop motion is more complex, and requires interaction with the creature's local environment. The most common form of evolved closed-loop controller uses contact sensors; a contact sensor outputs a value of +1.0 if its associated mass is in contact with a surface, otherwise a value of –1.0. A contact-sensor based motion cycle is formed by connecting the input of an effector node to the output of a contact sensor in such a way that the effector's spring lifts the

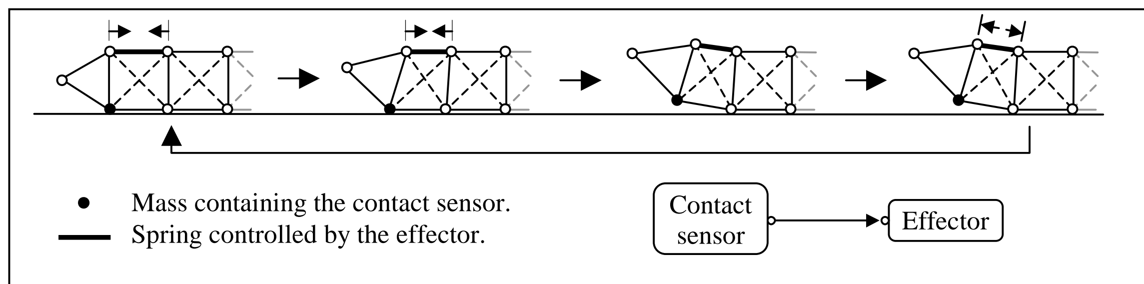


Figure 7-6. A simple sensor-based cyclic motion.

sensor's mass off the surface when contact is detected. Figure 7-6 illustrates this form of sensor-based cyclic motion.

The illustrated sensor-based cyclic motion is very sensitive to local conditions in the creature's body and environment. If the fully contracted spring does not lift the sensor's mass off the surface the cycle will be broken and no further motion will be produced. Because of this sensitivity, such closed-loop cyclic motions are less likely to be generated than open-loop cyclic motions. In our EA the probability of generating a closed-loop cyclic motion appears to be very much less than the probability of generating a wave-node based open-loop cyclic motion, thus explaining the lack of closed-loop control in the presence of wave generator nodes. By disabling wave generator nodes we reduce the probability of discovering an open-loop cyclic motion and thereby promote the discovery of closed-loop cyclic motions. Even without wave-generator nodes open-loop control may still be discovered (as in Figure 2-10), but such an event appears to be extremely improbable.

7.2 Other Creatures

Our controller synthesis EA has been applied to three other creature models in addition to our primary creature, the worm. This section briefly discusses the results of trials using each additional creature model. Model specifications can be found in section 6.1.

7.2.1 Ring model

Our 'ring' creature was designed for rolling. It uses an 'expanded' morphology genotype graph, i.e. one node in the genotype graph for each mass in the creature's mass-spring system. This creature approximates a 2D incarnation of the *Amoeba Man*, a soft-object model mentioned in section 1.3. We believed that this model would require closed-loop control, and that rolling would be the archetypal locomotion method.

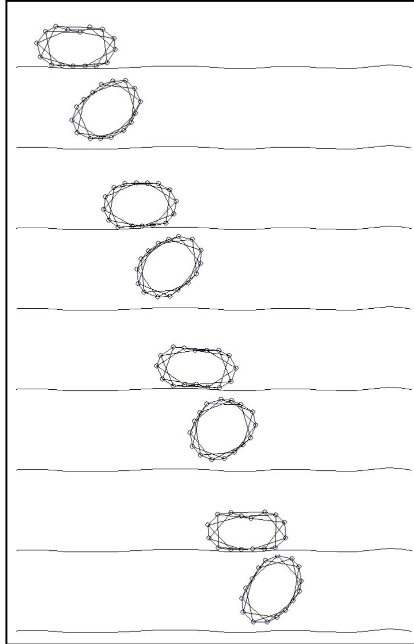


Figure 7-7. 'Ring' locomotion.

Several forms of closed-loop controller were discovered by the EA. A controller was always based around one type of sensor, including contact sensors, orientation sensors and velocity sensors.

Figure 7-7 illustrates the locomotion produced by a velocity-sensor based controller. This gait is a combination of rolling and hopping, and enables the creature to jump up sudden steps in the terrain. Controllers based on orientation sensors or contact sensors produced a faster, more fluid rolling gait.

Interestingly, over the small number of trials run the EA was never observed to become trapped in a very low-fitness local maximum. We believe that this was due to open-loop control being totally infeasible for the model, and thus removing the vast majority of local maxima from the controller search space.

7.2.2 Octagonal model

This creature's morphology was specified in two different ways, both 'grouped' and 'expanded' genotype graphs as described in section 6.1.3. The dual morphology graphs of this creature allow us to demonstrate the benefits of grouping. It was envisaged that this creature would be capable of locomotion by rolling. A rolling controller would push or pull the heavy central mass towards one side of the creature's body, resulting in a change to the creature's centre of mass and subsequent rotation. This form of locomotion would almost certainly require sensor information, as the speed of rotation would vary depending on terrain and linear velocity. Figure 7-8 displays a few frames of evolved high-speed rolling locomotion.

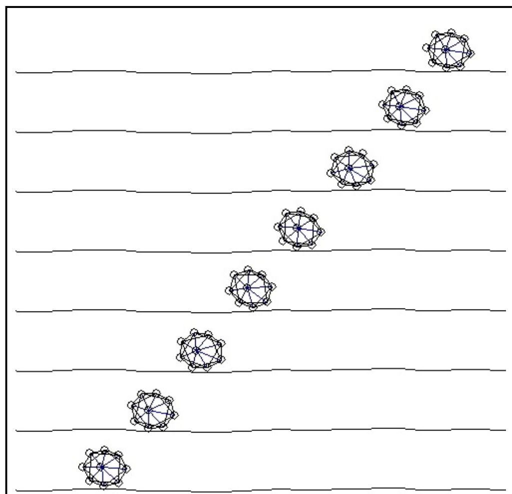


Figure 7-8. 'Octagon' creature locomotion.

Rolling controllers such as that illustrated in Figure 7-9 were only discovered in the grouped genotype.

Evolution using the expanded genotype produced a range of open-loop wave-node based controllers, none of which produced a rolling motion. These low-quality controllers moved the creature by random hopping and thrashing, which usually resulted in the creature travelling some small distance from the starting point. Such controllers correspond to local maxima in the search space.

By using the grouped genotype we force all circumferential parts to contain identical controller elements. Open-loop control can therefore only produce a symmetric contraction/expansion of the radial springs; it cannot adjust one radial spring without adjusting all of them. Conversely, closed-loop control may adjust radial springs based on sensor data from the circumferential masses, allowing for individual adjustments based on different local conditions in each mass's environment. Our grouped genotype provides a controller search space in which open-loop control is infeasible, thus removing a vast number of local maxima.

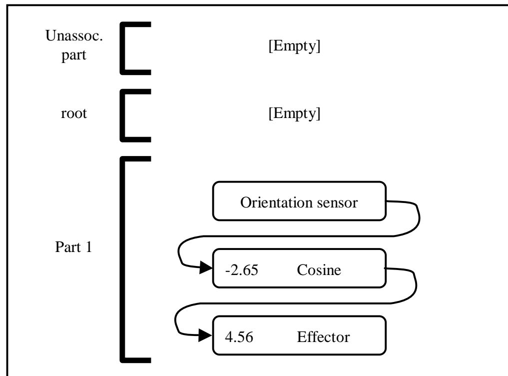


Figure 7-9. Sensor-based rolling controller for the octagonal creature's grouped morphology graph.

Evolution using the grouped genotype has produced two different rolling controllers based on two varieties of sensors. Figure 7-9 illustrates an orientation-sensor based rolling controller. The controller elements in "Part 1" are replicated for each circumferential mass in the creature's body. The other closed-loop rolling controller was of similar simplicity and was based on contact sensors. We believe that the grouped genotype has allowed us to find the maximally fit locomotion controller for this creature.

7.2.3 Biped model

The biped model uses an expanded genotype, and moves by adjusting the rest lengths of its four 'leg' springs. We believed that this model was stable enough to be open-loop controllable, but that closed-loop control might allow higher-speed locomotion. It was envisaged that the fastest gait for this model would be rapid side-stepping.

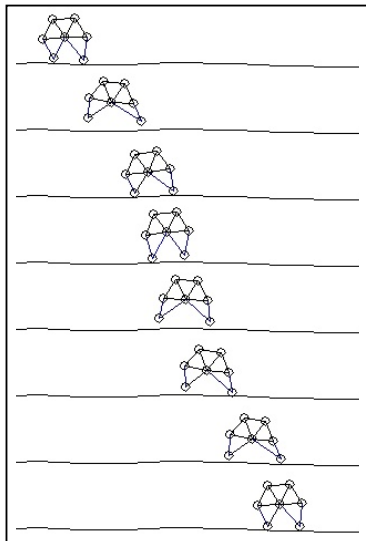


Figure 7-10 Biped locomotion.

As with previous models, several different locomotion styles were discovered by the EA. Open-loop controllers based around wave generator nodes were common, as were closed-loop controllers based around contact sensors.

Although the fittest evolved locomotion style was side-stepping, against our expectations side-stepping controllers were open-loop. The EA was observed in several trials to become trapped in a local maximum with the creature falling onto its side, where controller evolution was focused on learning to move while lying sideways. 'Walking' controllers were only discovered in trials where falling-sideways did not occur in early evolution. Modifying the fitness function to penalise overbalancing and falling would help the EA to avoid these local maxima, and should be beneficial when evolving controllers for highly unstable creatures.

Figure 7-10 illustrates an interesting open-loop hopping gait. This gait provides rapid locomotion and is robust against considerable variations in terrain. Very similar sensor-based gaits were also evolved.

7.3 Quality of animation

Animations produced by our evolved locomotion controllers are of varying aesthetic plausibility. Some gaits are immediately recognisable and believable, others are novel, and a few are hilarious. The physically-based nature of our mass-spring systems is obvious to an observer: things fall, bounce, roll and flail in a believable manner. The most significant plausibility-limiting factor is the inability of a 2D mass-spring system to conserve its enclosed area. However, this limitation is often subtle and may not be noticed

unless explicitly drawn to an observer's attention. Non-conservation of area in the worm model can be observed in the left half of Figure 7-16. The vertical thinning in this model as a response to horizontal stretching is a consequence of the internal diagonal bracing springs.

In order to produce higher-quality animations of our worm model we have applied texture maps to the worm and its environment. We implement this texture wrapping in an OpenGL-enabled version of our slave application, frames from which are illustrated in Figure 7-11.

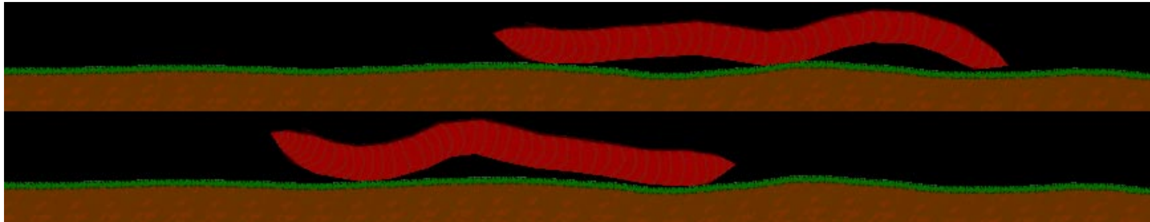


Figure 7-11. Texture-mapped worm model and terrain.

7.4 Randomised Terrain: Fitness Evaluation in a Noisy World

It has been demonstrated that including a small amount of random noise in the fitness evaluation of an EA may improve the robustness of solutions [Reyn_94]. By introducing random noise we help weed out 'fragile' solutions and promote more robust, noise-tolerant solutions.

There are many ways we could introduce random noise into our creature's fitness evaluation, such as slightly perturbing spring strength coefficients, spring rest length bounds, mass values, the gravity vector, etc. We choose to apply our random noise in the form of varying terrain. A random terrain is constructed for each generation of controller candidates so that each generation experiences a slightly different local environment. Ideally, fragile controller types will soon encounter terrain outside their operational parameters and will be surpassed by more robust controllers.

The random nature of our terrain generator results in some landscapes that are 'easier' and some that are 'harder'. This fluctuation does not affect the convergence properties of the EA as selection pressure only exists between candidates in the same generation. The 'difficulty' of a particular terrain must be taken into account when comparing candidates from different generations, such as in parent-to-child comparisons. The performance of cloned candidates from the parent generation provides a useful benchmark to determine the relative difficulty of the child's terrain to the parent's.

Our terrain generation algorithm creates a regular-spaced height-map according to a set of three input parameters, K_{height}, $K_{steepness}$ and *terrainSeed*. K_{height} and $K_{steepness}$ specify the maximum height/depth of the terrain, and the maximum gradient that a terrain segment may assume. Our *terrainSeed* parameter supplies

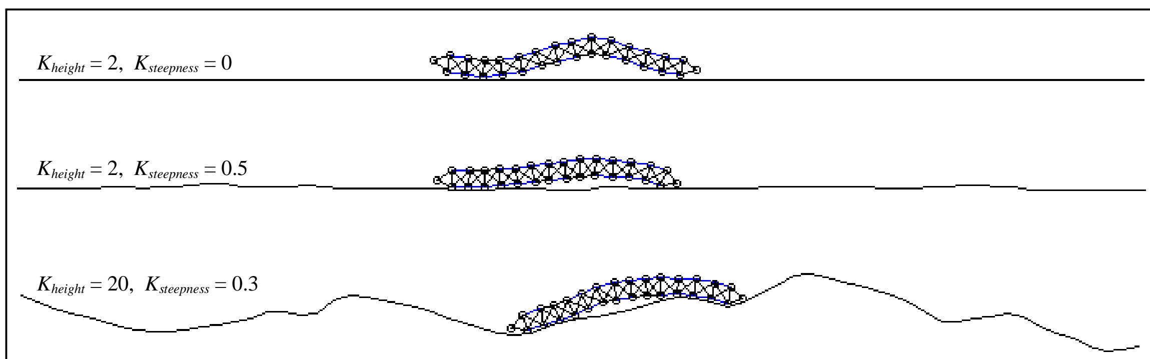


Figure 7-12. 'flat', 'normal' and 'extreme' terrain types.

an initial seed for a random number generator specific to our terrain generator. By defining a terrain as a function of these three parameters we minimise the network traffic to our distributed slave processes. Figure 7-12 contains sections of three randomly generated terrains of differing difficulty.

In this set of experiments we evolve locomotion controllers for the worm model over both 'flat' and 'normal' terrain types. We compare the performance of our EA over the two problem domains, and draw comparisons between our two sets of evolved controllers.

7.4.1 Flat Terrain versus Varying Terrain

Figure 7-13 graphs the fitness curves of trials run using flat terrain. Figure 7-14 graphs the trials run using 'normal' varying terrain.

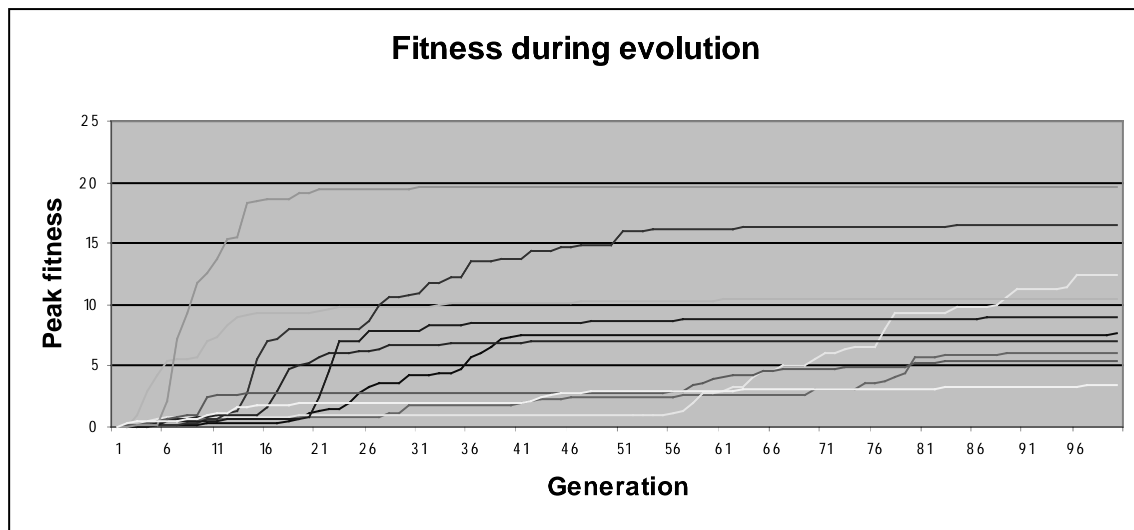


Figure 7-13. Controllers evolved over 'flat' terrain.

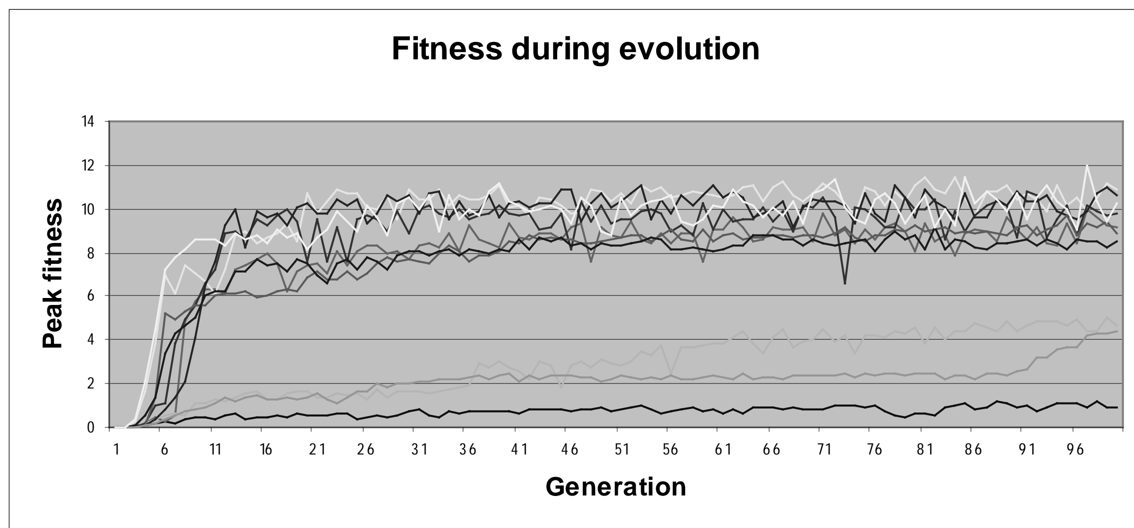


Figure 7-14. Controllers evolved over 'normal' type terrain.

It can be seen that trials run using flat terrain may attain higher fitness, but exhibit much more sensitivity to initial conditions than trials run using varying terrain. The 'flat' trials have produced a very wide range of fitness curves, no two of which are very similar. By comparison, trials run using varying terrain can be grouped into two sets: the majority quickly ascend to a near-plateau in the range [8..11], and the others ascend very slowly and attain much lower peak fitness values.

Evolved over flat terrain						Evolved over varying terrain					
Fitness over varying terrain			Fitness over flat terrain			Fitness over varying terrain			Fitness over flat terrain		
Mean	Median	Std. dev	Mean	Median	Std. dev	Mean	Median	Std. dev	Mean	Median	Std. dev
4.348	4.24	2.177	9.741	8.28	5.14	7.799	9.007	3.31	8.318	9.698	3.392

Table 7-1. Flat/varying terrain fitness summary.

Table 7-1 contains a brief summary of peak fitness values after 100 generations. We include both flat-terrain and varying-terrain fitness values. Figure 7-15 contains scatter diagrams of the same source data.

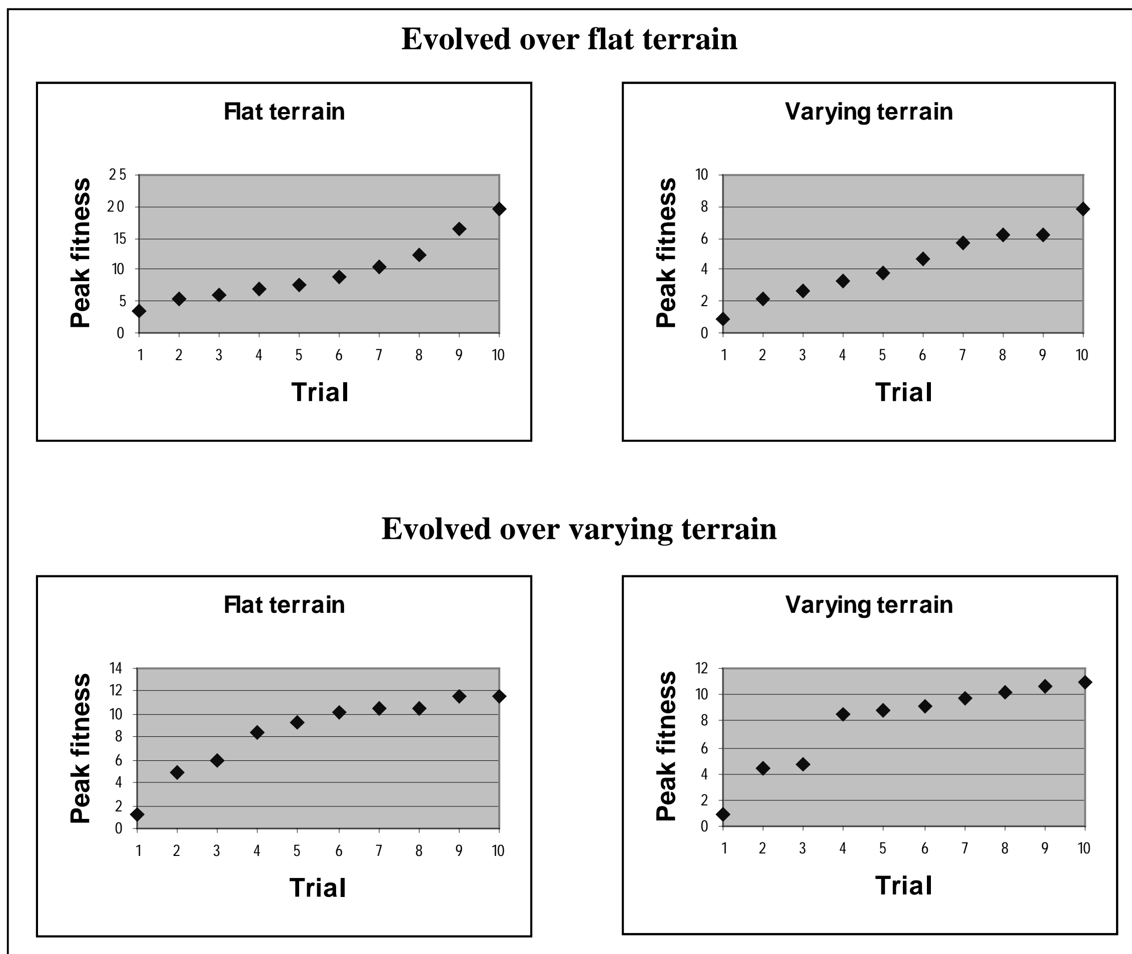


Figure 7-15. Peak fitness distributions for trials evolved over flat terrain / varying terrain.

Given that the set of trials evolved over flat terrain appears to contain an outlying trial, we believe a median-value comparison to be more appropriate than a mean-value comparison. Making this assumption, the application of noise to the creatures' environment has resulted in a higher expected peak fitness value.

We hypothesise that randomised terrain has resulted in increased median fitness by significantly reducing the probability of a trial becoming trapped in a local maximum. Controllers of low fitness usually rely upon only a few springs to provide locomotion, and are vulnerable to terrain variation around those springs. Over

several generations it becomes likely that such a controller will encounter terrain it cannot traverse early in its fitness evaluation and therefore increase the probability of alternative controllers being selected for reproduction.

7.4.2 Animation Comparison

Evolution over flat terrain results in a qualitatively different locomotion style to evolution over randomised terrain. Controllers evolved over flat terrain typically evolve a 'skating' gait in which the body is held very low, or lies entirely upon the surface. Randomised terrain yields gaits in which the body is held higher, usually in one or more arches. The effect of terrain variation on locomotion style may be useful to an animator, who may be seeking a particular type of gait. Figure 7-16 illustrates these differences in gait.

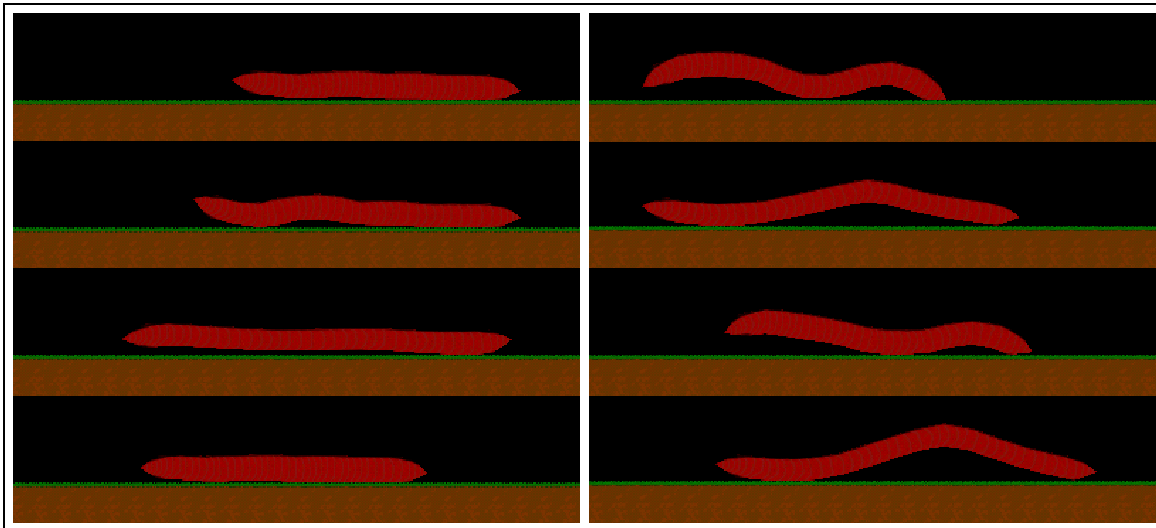


Figure 7-16. Gait comparison: Evolved over flat terrain / evolved over varying terrain.

7.4.3 Conclusion

Controllers evolved over varying terrain appear to be of higher median fitness when evaluated over both varying terrain and flat terrain. Our trials evolved over flat terrain have produced a very wide range of fitness values, so we believe that more trials would reinforce this conclusion. However, Figures 7-13 and 7-14 clearly show that randomised terrain has resulted in increased predictability of peak controller fitness. Lastly, the degree of variation in the terrain over which a controller is evolved has a noticeable qualitative effect on the controller's style of locomotion.

7.5 Summary

Our EA is successful in generating locomotion controllers for a small range of mass-spring creatures but in some cases is prone to becoming trapped in a local maximum. A creature's genotype morphology graph defines the controller search space, and may bias the EA towards either open-loop or closed-loop control. In our worm model we were not able to synthesis closed-loop controllers until wave generator nodes had been disabled. Including small, random variations in the creature's environment results in more consistent EA behaviour, and appears to reduce the expected number of generations required to synthesise a locomotion controller.

Chapter 8 Niching

In a Simple Evolutionary Algorithm (SEA) selection pressure exists over the entire population of solution candidates; each candidate is in direct competition with every other candidate for the right to produce offspring. This selection pressure is necessary to improve the optimality of the population, but has some undesirable effects; the population tends to lose genetic diversity and the search algorithm may easily become trapped in a local maximum.

Genetic diversity is lost as a consequence of the search algorithm finding a maximum in the search space and a fit genotype repeatedly producing the bulk of the offspring. Some of the fit genotype's offspring will be very similar to their parent and will have similar fitness. Unless the local maximum lies near a region of higher fitness in the search space, offspring that differ significantly from the candidate at the maximum are highly unlikely to be fitter. On the other hand, offspring that are similar to the genotype at the maximum are very likely to be of similar fitness, and so quickly dominate the upper end of the population. If this occurs, the search algorithm has virtually ceased to *explore*, and is focusing all its energies upon *exploiting* the local maximum.

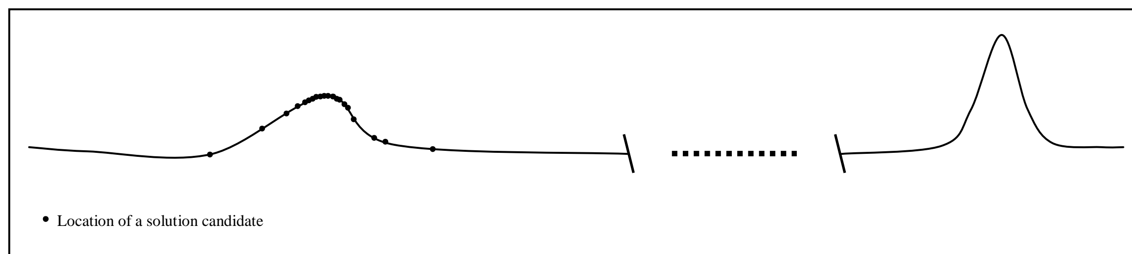


Figure 8-1. Stuck in a local maxima.

Figure 8-1 illustrates a SEA that had become trapped in a local maximum of a simple one-dimensional search space. The peak of the local maximum (left) is saturated with nearly identical solution candidates, and very few candidates lie outside the slopes of the local maximum. For the algorithm to find the higher maximum (right) it must instantaneously generate a candidate that falls within the region of the higher maximum and is at least as fit as candidates upon the peak of the leftmost local maximum. If the distance between the two maxima is high, this is statistically very unlikely to occur.

Gradient ascent is crippled once a local maximum has been found. It is possible or even likely that a candidate may be generated that lies on the lower slopes of the rightmost maximum. Because selection pressure ignores candidates' location in the search space, this new *explorer* candidate will be in direct competition with the many *exploiter* candidates on the peak of leftmost maximum. Even if the new candidate's descendants would be afforded a smooth and fast ascent to the higher peak it will probably not be given the opportunity to reproduce. The new candidate's potential will be lost, and the SEA will remain in the local maximum.

8.1 Introduction to Niching Methods

The success or failure of a SEA is dependent on the topology of the search space. If the search space is globally convex, a SEA should obviously find the global maximum. Real-world domains often contain many local maxima, and the global maximum may be very localised in the search space. Domains containing many local or global maxima are termed *multimodal*. It has been demonstrated that SEAs cannot reliably solve such domains, and in practice perform poorly over them [Mahf_95].

Niching methods enable evolutionary algorithms to efficiently and reliably solve multimodal optimisation problems by ensuring genetic diversity in the population and thereby encouraging *exploration*. The

population is divided into multiple *niches* in the search space. A niche is a group of candidates that all share a common characteristic, e.g. they all exist within a local region of the search space. Selection pressure exists within each niche, but may not directly exist between candidates in different niches.

Borrowing a parallel from nature, niches can be thought of as populations of a single species existing in different regions of space. Populations may or may not be in direct competition with each other for resources. If two species do not compete for the same resource then for the purposes of survival and reproduction the prowess of one species does not affect the other. If resource competition exists between two species, selection pressure may exist between the two. In addition, candidates in different niches are usually allowed to interbreed.

All niching methods use a *differencing function*: a metric used to determine the degree of dissimilarity between two solution candidates. Differencing functions are based solely on the genetic composition of the candidates and do not include lineage information. By applying the differencing function to a population of candidates we can divide the population into a number of niches, where every candidate in niche n_i is within a certain distance r_i of the fittest candidate in n_i. We term r_i the niche radius.

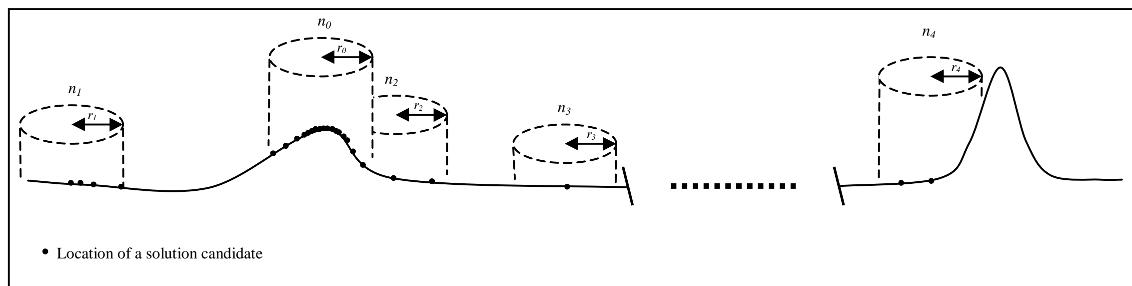


Figure 8-2. Niches in a one-dimensional search space.

Figure 8-2 illustrates the effect of applying a niching method to the domain presented in Figure 8-1. Note that in the case of overlapping niche-spaces a candidate in the overlap region belongs to the niche containing the fittest candidate.

In this example the local maximum (left) has been found by the EA, and is being occupied by many candidates. A niche, n_0 is defined around this peak and contains all candidates within radius r_0 of n_0's fittest candidate according to our differencing function. Controller candidates have been dispersed throughout the search space by reproduction methods such as mutation. Many of these candidates have difference values greater than r_0 with respect to n_0, so are said to be outside $n_{0.}$ Thus multiple niches exist within the search space.

Some candidates have landed upon the slopes of the rightmost maximum. Assuming that selection pressure between niches is low, the candidates in this rightmost niche will probably generate enough offspring to afford good gradient ascent properties. It is highly likely that the rightmost niche will migrate over subsequent generations to the peak of the maximum.

Choosing an appropriate selection pressure between niches is a trade-off between speed of convergence and ability to escape local maxima (exploitation versus exploration). High inter-niche selection pressure results in fit niches creating the most offspring and less fit niches creating fewer. This bias towards choosing parents from fit niches may speed the convergence of fit niches to their maxima, but might cause the less fit niches to lose their convergence properties and subsequently be destroyed. Recent work in niching methods [Petro_97] has shown that in highly multimodal domains, direct selection pressure between niches is inferior to unbiased selection in which all niches have the same expected number of offspring regardless of niche fitness. By giving each niche the same expected number of offspring we strike a good balance between exploration and exploitation.

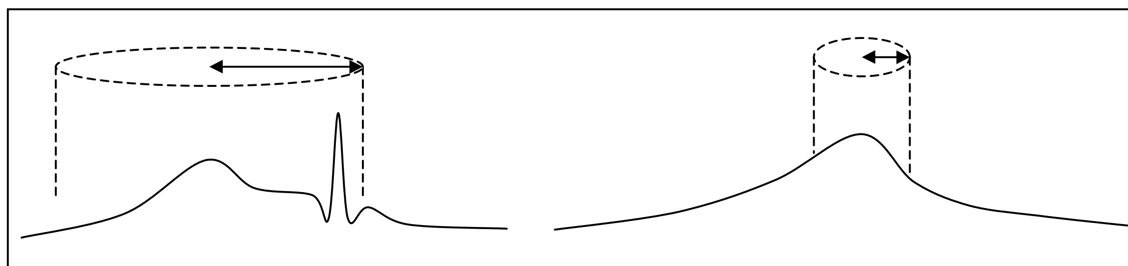


Figure 8-3. Problems with inappropriate niche radius size.

8.1.1 Differencing Functions and the Niche Radius

The combination of a differencing function and a niche radius define the region of the search space occupied by a particular niche. Ideally, this region should encompass a single maximum and any ascending slopes that would guide a gradient ascent search method to that maximum. In other words, a niche's region should define the area that we need to look *outside* if we are to locate other maxima.

The choice of differencing function depends on the candidate representation. Genetic algorithms usually represent candidates as bit-strings, so may choose to use the Hamming[5] distance. Candidates represented as n-tuples of real-valued variables might use the Euclidean distance in n-space. The size of the niche radius will depend on the topology of the problem domain, and may be difficult to choose.

Figure 8-3 illustrates problems associated with a too-large or too-small niching radius. A too-large radius encompasses more than one local peak. In this case, the niche is centred upon the lesser of the two peaks and the higher peak can only be discovered by pure random search rather than gradient ascent. A too-small radius results in inefficiency: slopes outside the niche's region may direct the search straight back to the niche.

Ideally, each niche's radius should be chosen separately from all others and should reflect the topology of the niche's local region of the search space. We should make surveys of the search space surrounding the niche's fittest member and choose our radius accordingly. However, this survey process is computationally infeasible for domains such as ours where the evaluation of a candidate may take 20 to 30 seconds CPU-time (on a Pentium-II-400MHz machine).

8.1.2 Number of Niches

The number of niches required to solve a particular domain reflects the domain's degree of multimodality. Each local or global maximum can attract a single niche, so if all maxima must be concurrently located we must maintain at least as many niches. Assuming a fixed population size, we are limited in the number of niches we support in any particular generation. Each niche must create enough offspring to afford good convergence properties. If a niche creates only a small number of offspring it becomes very possible that all offspring will be less fit than their parents. Over successive generations, instead of converging to a maximum the niche may actually become less fit. The number of offspring required to maintain a niche and provide good convergence properties will be specific to the problem domain and the reproduction methods used.

If after niche determination our population contains more niches than we are able to support, we keep only the best niches. In this way there is an indirect selection pressure between niches - only the fitter niches survive. If we know nothing about the topology of the search space we will usually try to support as many niches as is computationally feasible.

---

[5] The Hamming distance between two bit-strings is the number of single-bit bit-flips required to transform one bit-string to the other.

Surprisingly, locating global maxima in the presence of large numbers of local maxima may only require a relatively small number of niches. Petrowski [Petro_97] was successful in reliably and concurrently locating all 32 global maxima of a massively multimodal function containing several million local maxima. Global maxima were of value 5.0 and local maxima were in the range [3.203, 4.641]. A population size of 800 was used, implying that the number of niches present at any one time must be considerably less than 800.

Because we allow candidates from different niches to interbreed though multi-parent reproduction operations, each niche has access to a diverse range of genetic material. This genetic diversity helps bias the search towards exploration.

In summary, niching methods for evolutionary algorithms modify selection pressure in such a way that exploration is encouraged, but exploitation of existing maxima is maintained. Niching methods can dramatically improve the performance of a SEA over multimodal problem domains.

8.2 The 'Clearing' Niching Method

The Clearing Based Selection niching method [Petro_97] applies extreme selection pressure within each niche. Only the single fittest candidate within each niche, the niche's 'winner', is allowed to reproduce - all other candidates are discarded. Selection pressure does not exist between different niches, so all niches have the same expected number of offspring.

8.2.1 Algorithm Overview

Clearing Based Selection draws its name from the 'clearing' operation it performs each generation. After we have evaluated the fitness of all candidates in the generation we use a differencing function to divide the population into a number of niches. Within each niche there exists a single 'winner' whose fitness is greater than or equal to that of all other candidates in its niche. We remove all non-winner candidates from the population, and create the next generation from the set of winners. Figure 8-4 outlines the clearing algorithm.

```
applyClearing(candidates)
    sort(candidates)        {Sorts into non-ascending order by fitness}

    for i=0 to PopulationSize-1 do begin
        if  (candidates[i].fitness > 0) then begin
            for j=i+1 to PopulationSize-1 do begin
                if (candidates[j].fitness > 0) and
                (difference(candidates[i], candidates[j]) < ClearingRadius) then
                    candidates[j].fitness = 0
            end for
        end if
    end for

    sort(candidates)
end applyClearing

where:
    PopulationSize denotes the number of candidates in a generation.
    ClearingRadius is a real-valued constant defining the niche radius in controller-space.
    candidates is an array[0..PopulationSize-1] containing the current generation of candidates.

    sort(candidates) sorts the array of candidates into non-increasing order of fitness.
    difference(candidate1, candidate2) is our differencing function.
```

Figure 8-4. The Clearing algorithm.

After applying clearing, the only candidates with non-zero fitness are our 'winners'. We then set our survival proportion to include only the winners, or a subset of the winners. If there are more winners (and therefore niches) than some specified maximum, the survival proportion includes only the fittest *N* winners. Figure 8-5 outlines the post-clearing calculation of the survival proportion.

```
.
.
applyClearing(candidates)
i = 0
while (i < min(PopulationSize, MaxNiches)) and (candidates[i].fitness > 0) do
    i = i + 1
survivalProportion = i/PopulationSize
.
.

where:
    MaxNiches defines the maximum number of niches to maintain.
```

Figure 8-5. Calculating the survival proportion.

Selection for reproduction is unbiased. Each winner in the survival proportion has an equal probability of being selected as any other.

8.2.2 A Differencing Function for Our Candidate Representation.

Our directed-graph controller candidate representation does not provide an obvious differencing measure. Controller graphs may contain both topological differences and parameter/weight differences. We choose to ignore changes in parameters/weights and define our differencing function in terms of controller graph topology and node distribution throughout the creature's genotype morphology graph.

Our differencing function is quite simplistic and makes some assumptions about the number and distribution of controller nodes within the creature's genotype morphology graph:

- It is rare for two or more controller nodes of exactly the same type (with the exception of constant value nodes) to exist within the same part of the creature's body.
- Effectors that are connected directly to a constant value node should not be considered to be part of the controller, and should be ignored for the purpose of determining difference.
- Constant value nodes should be treated as less important than sensors, neurons or effectors.

The differencing function calculates its value for two controller candidates C_1 and C_2 by stepping through each part of the creature's body. For each body part P_i, C_1's controller nodes in P_i are compared to C_2's nodes in P_i. Figures 8-6 and 8-7 outline the outer and inner loops of the differencing function respectively.

```
difference(candidate1, candidate2)

    d = 0
    for each body part i in candidate1 do begin
        p1 = the ith body part of candidate1
        p2 = the ith body part of candidate2
        d = d + differenceBetweenParts(p1, p2)
    end for

        difference = d
end difference
```

Figure 8-6. The outer loop of the differencing function.

```
differenceBetweenParts(part1, part2)

    d = 0                    {Differences found}
    t = 0                    {Total number of differences possible}
    if part1 contains more controller nodes than part2 then
        largerPart = part1; smallerPart = part2
    else
        largerPart = part2; smallerPart = part1

    for each unit u1 in largerPart do begin
        if (u1 is not a constant value node) and (u1 is not an effector node connected to a constant
        value node) then begin
            u2 = findSimilarUnit(u1, smallerPart)
            |C| = u1's input arity (the number of input connections of u1)
            t = t + 1 + |C|

            if (u2 = null) then
                d = d + 1 + |C|
            else
                for each input connection C_i of u1 do begin
                    D_i = the i^th input connection of u2
                    source1 = The controller node from which C_i draws its data.
                    source2 = The controller node from which D_i draws its data.

                    if not sameType(source1, source2) then
                        d = d + 1
                end for
        end if
        end for

    differenceBetweenParts = d/t
end differenceBetweenParts
```

where:

findSimilarUnit(*unit*, *part*) returns a controller unit from the given *part* that is of the same type as *unit*, or returns null if no such controller unit exists in *part*. The sameType(*unit1*, *unit2*) method defined below is used in this method to determine similarity.

sameType(*unit1*, *unit2*) returns *true* if *unit1* and *unit2* are of matching classes and sub-types. eg: if *unit1* is a 'product' neuron and *unit2* is a 'sum' neuron, sameType(*unit1*, *unit2*) will return *false*.

Figure 8-7. The inner loop of the differencing function.

Figure 8-8 demonstrates the calculation of the difference value for a pair of simple controller graphs. The first controller is based around a wave generator unit that lies in the 'unassociated' part of the creature's genotype morphology graph. The second controller is based around a contact sensor in the first non-root part of the creature's genotype morphology graph.

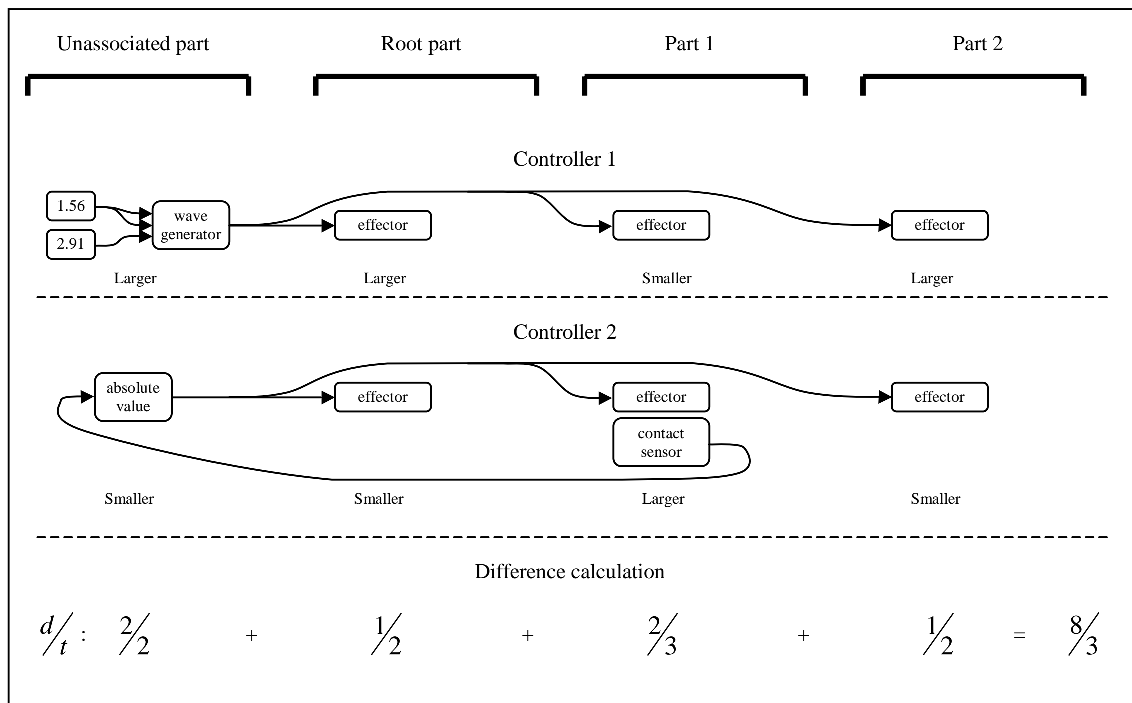


Figure 8-8. Demonstration of the differencing function.

The performance of a niching method is directly tied to the suitability of its differencing function. We have introduced a simple differencing function highly specific to our controller representation that relies on certain assumptions such as the sparseness of evolved controller graphs[6]. Our differencing function will fail if these assumptions are false. A more elaborate differencing function that calculates the influence of each controller node on effector output might provide a superior distance metric.

An alternative type of differencing function might be defined at the phenotype level; comparing gaits rather than controller graphs. Quantifying differences in gait might involve spectral analysis of the controllers' effector outputs coupled with a Euclidean distance calculation.

[6] See Figures 8-13 and 8-14 for examples of typical evolved controllers. Note the relatively few controller nodes in each part of the genotype morphology graph.

8.3 An Experimental Comparison between Traditional Selection and Clearing-based Niching.

In this experiment we compare the performance of traditional selection to our clearing-based niching method using our worm model. One trial parameter differs from those specified in chapter 6 – we use a population size of 1000. For the niching trials we use a clearing radius of 15.0 and set our maximum number of subpopulations (niches) to be 10. Our increased population size results in 100 expected offspring for each niche's winner, and thus should provide good convergence properties. Trials were stopped after 100 generations unless otherwise stated. We ran 10 trials for each selection method.

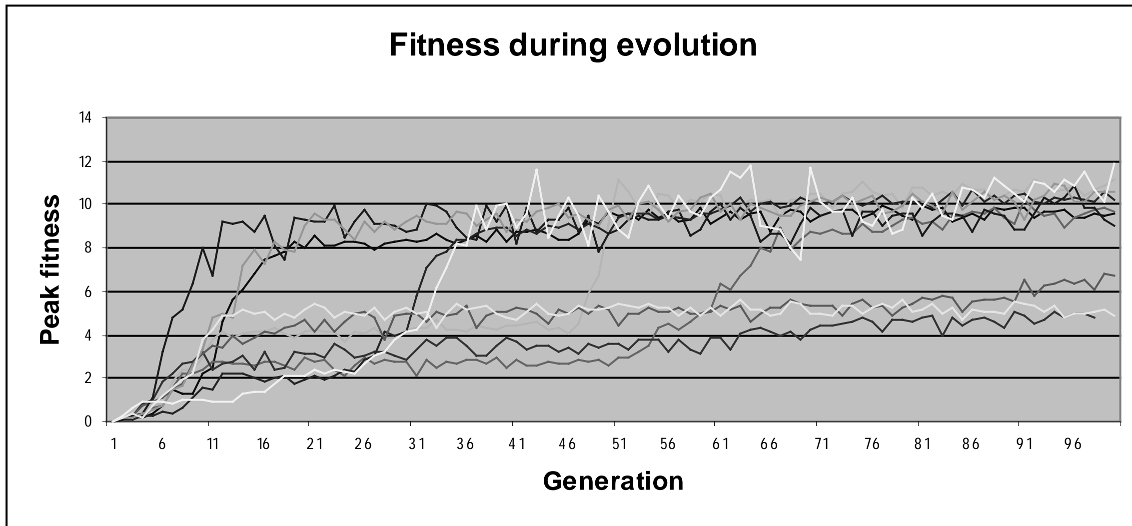


Figure 8-9. Evolution using traditional selection.

Figure 8-9 graphs the results of trials run using traditional selection. This set of trials has mean peak fitness after 100 generations of 8.87 and a standard deviation of 2.48. In all cases all the evolved locomotion controllers were open-loop, based upon wave-generator nodes. In seven trials the EA has discovered a controller with peak fitness in the range [9, 11.5]. Of these seven trials, the EA has ascended to a plateau/maximum at a variety of rates; the first three trials appear to have reached a plateau after just 25 generations, whereas the other four trials ascended at sometime between 35 and 80 generations.

In three trials the EA has encountered a local maximum or near-plateau in the search space, and has only obtained a peak value in the range [5, 7] after 100 generations.

Figure 8-10 illustrates a typical non-niched population after 100 generations. Again, the population has become dominated by a single genotype.

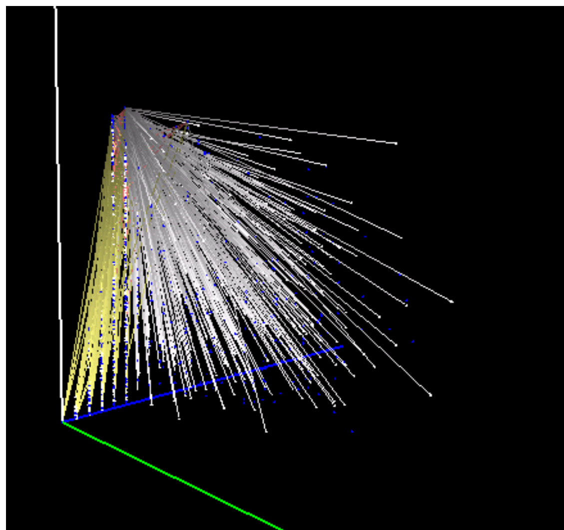


Figure 8-10. Diversity after 100 generations.

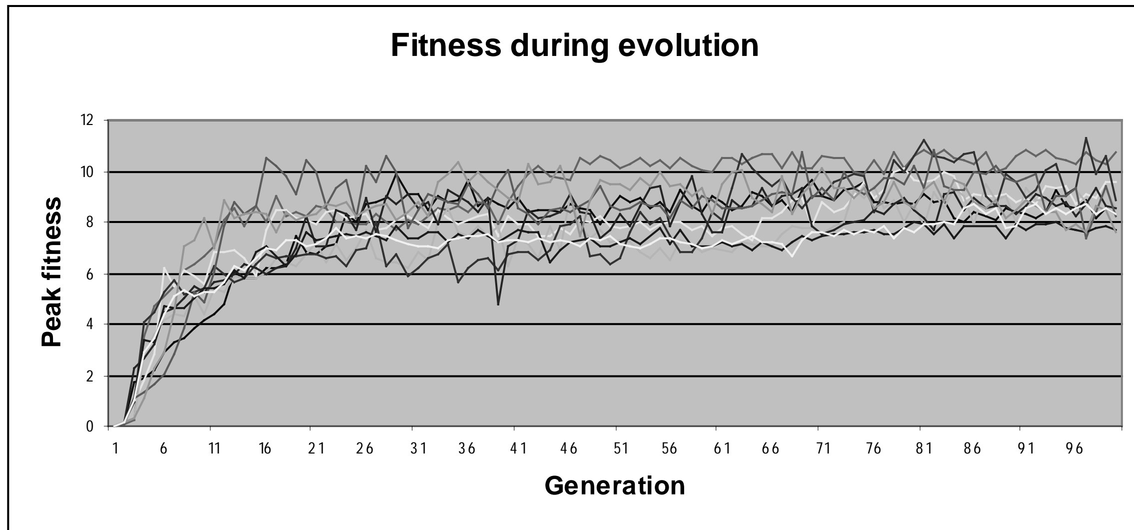


Figure 8-11. Evolution using clearing-based selection.

Figure 8-11 graphs the results of trials run using clearing-based selection. This set of trials has mean peak fitness after 100 generations of 8.63 and a standard deviation of 0.92. Peak fitness values for the entire set of trials fall in the approximate range [8, 11]. The EA is clearly able to escape the local maxima observed in Figure 8-9, and thereby provide a more consistent peak fitness value after 100 generations. Note also that the fitness curves are very similar over the hundred generations; The population experiences a period of rapid improvement during the first 20 generations, usually followed by a more sedate improvement or fitness plateau over the remaining generations.

Although the behaviour of the EA under clearing-based selection is more predictable and reproducible than under traditional selection, some variation in performance is still apparent. In terms of search, this variation equates to the discovery of different maxima over different trials; even with niching operating it appears unlikely that our EA will discover exactly the same maximum on two separate trials. This behaviour may be due to an insufficient number of niches, an inappropriate differencing function or a highly complex search space.

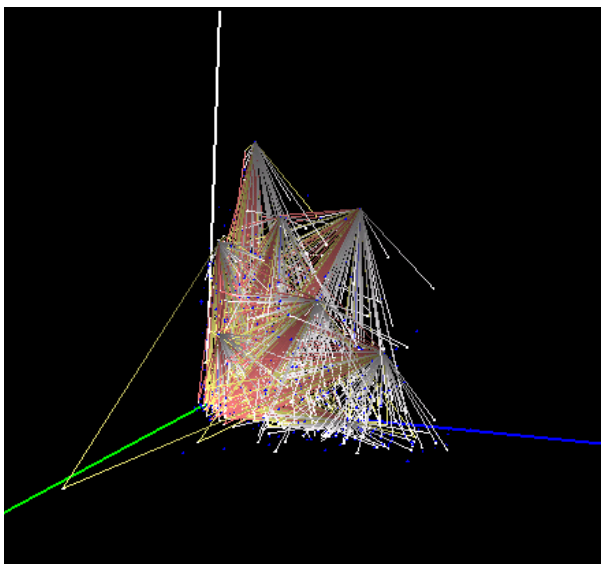


Figure 8-12. Genetic diversity abounds.

Figure 8-12 illustrates the genetic diversity present in a niched population after 100 generations. Note the obvious 'winners' made visible by explosions of outgoing child-links.

In 9 of our 10 trials the fittest evolved controller was open-loop, based upon one or more wave generator nodes. Figure 8-13 presents an interesting wave-generator based controller; the main wave node's frequency slowly oscillates under the influence of a secondary wave node, allowing the worm to overcome a wide range of terrain.

In the remaining trial the fittest controller was closed-loop, based around a vertical velocity sensor. This closed-loop controller is displayed in Figure 8-14.

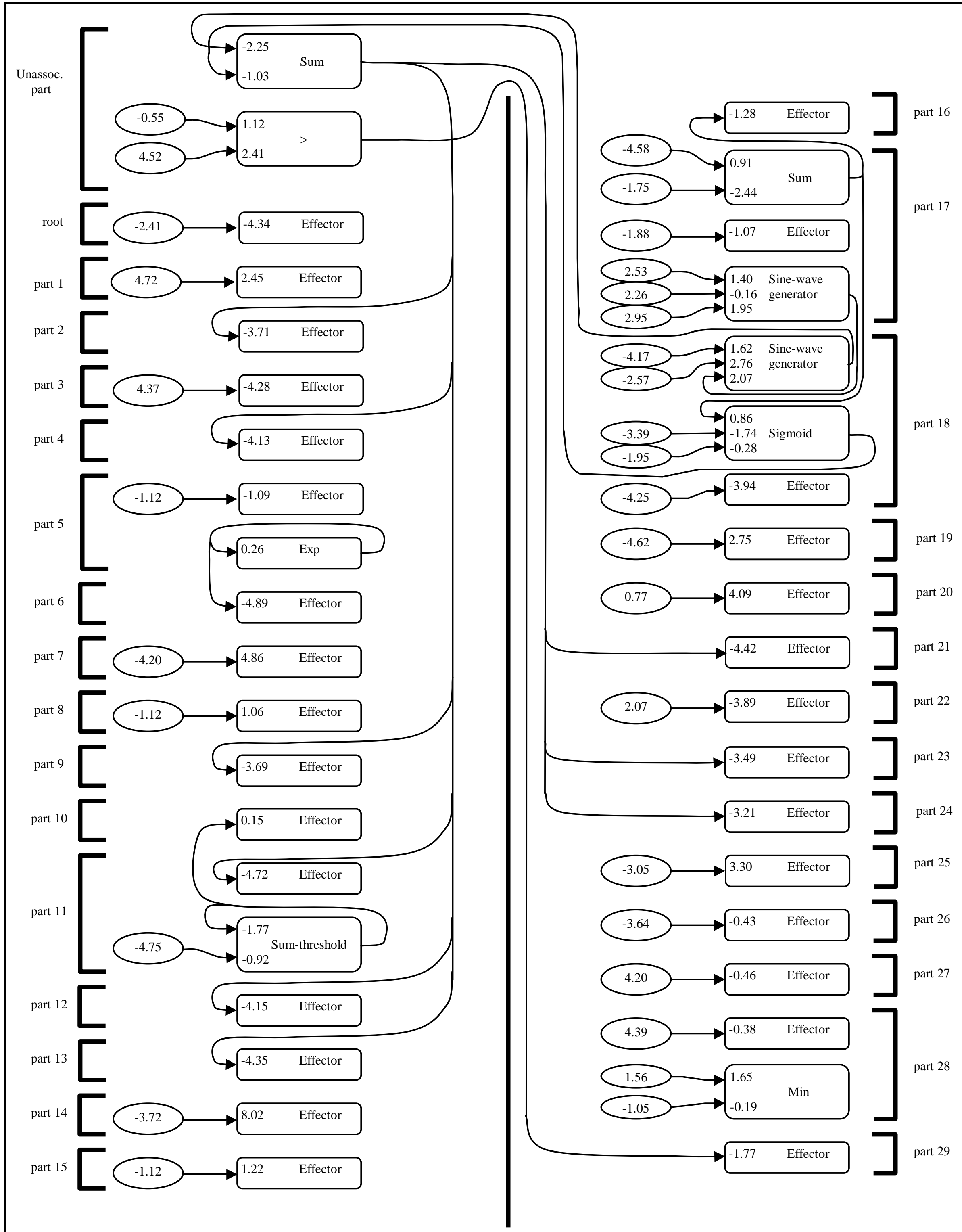


Figure 8-13. Evolved open-loop worm locomotion controller.

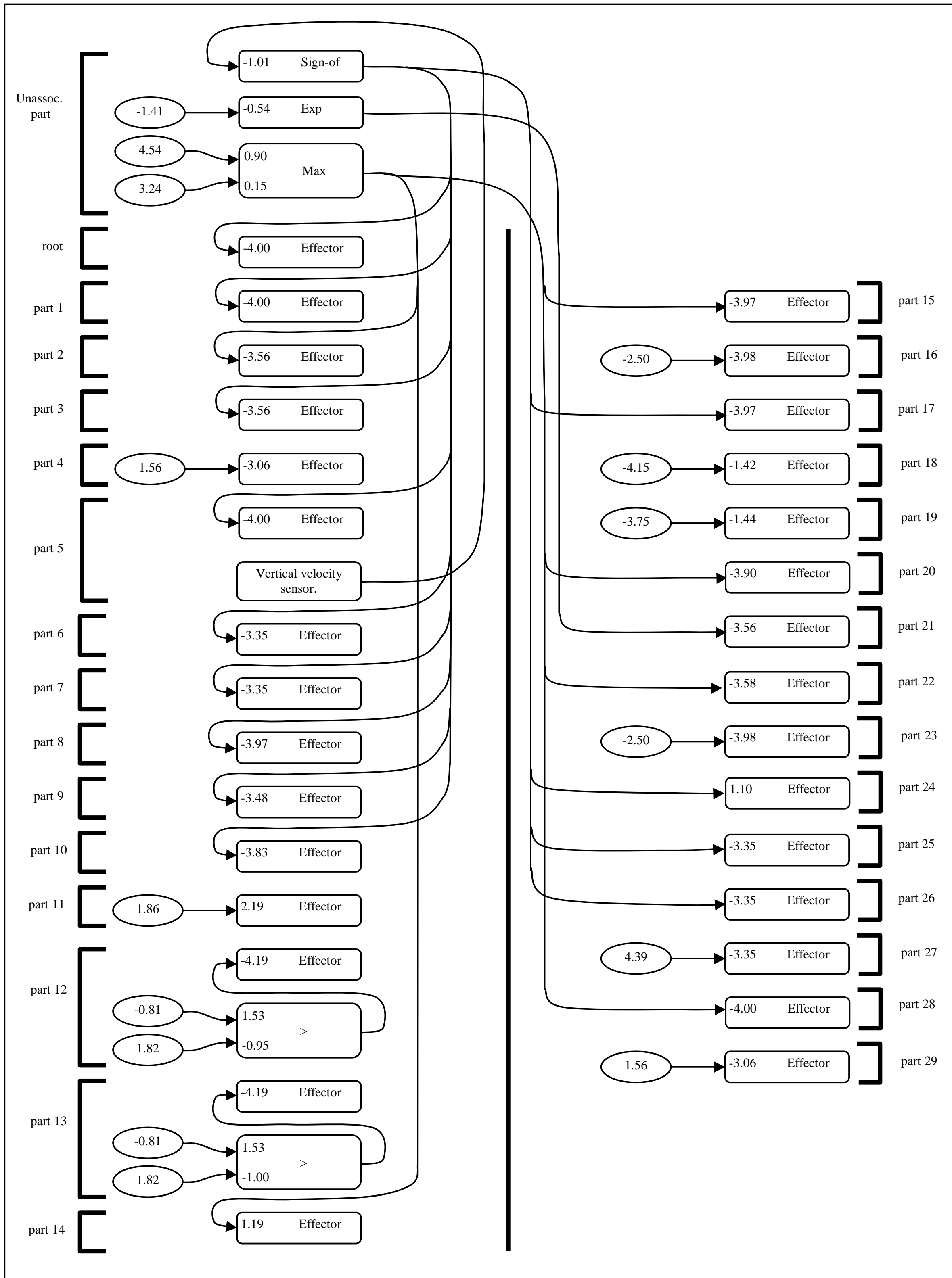


Figure 8-14. Evolved closed-loop worm locomotion controller.

8.3.1 Variation of Controller Type and Locomotion Gait

When using traditional selection the EA generates only one type of locomotion controller per trial. Unless we disable wave generator nodes, this controller has always been open-loop.

The application of our niching method results in each evolution producing a range of locomotion controllers; one for each niche. This behaviour is extremely desirable from an animation perspective – We can obtain a palette of distinct and different locomotion controllers from a single controller-synthesis process. This palette of controllers usually includes both open-loop and closed-loop controllers covering a range of gaits and locomotion speeds. An animator might browse such a palette to select an appropriate controller for a particular animation task.

Figure 8-15 contains a visualisation view of a niched population after 100 generations. The x-axis (blue) has been set to indicate the candidate's niche. Our candidates are arranged in 10 niche-planes in y-z space. Note the very high degree of connectivity between niche planes. Inter-niche parent-child relationships have a variety of causes:

- Two-parent reproduction operations may choose parents from different niches. Recall that our selection method is unbiased with respect to niche fitness; we would expect 9/10 two-parent reproduction operations to choose parents from different niches.
- A significant change in genotype between the child and the parent. If the child candidate is vastly different from the parent it may fall outside the parent's niche.
- Niche migration. Niches are sorted into non-increasing order of fitness. A niche, or genotype, may migrate up the order by using gradient ascent to improve itself beyond the fitness of other niches. Because we do not retrospectively reassign the parent's niche, a child appearing to be in a different niche to its parent may in fact be of very similar genotype to its parent.

Figure 8-16 displays the same view as Figure 8-15 with the exclusion of parent-child lines. This view makes clear the relative fitness of the 10 niches.

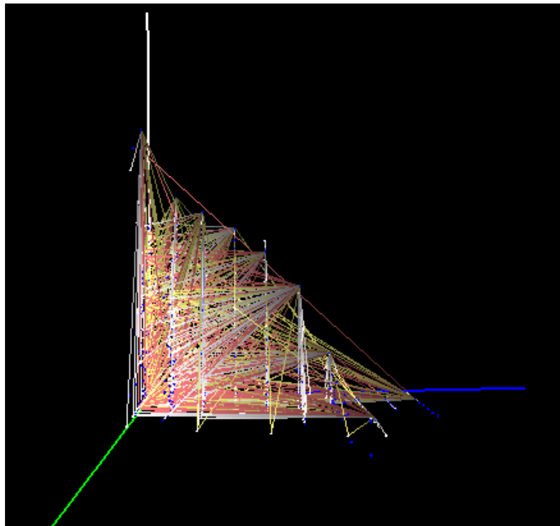


Figure 8-15. Interbreeding and migration between niches.

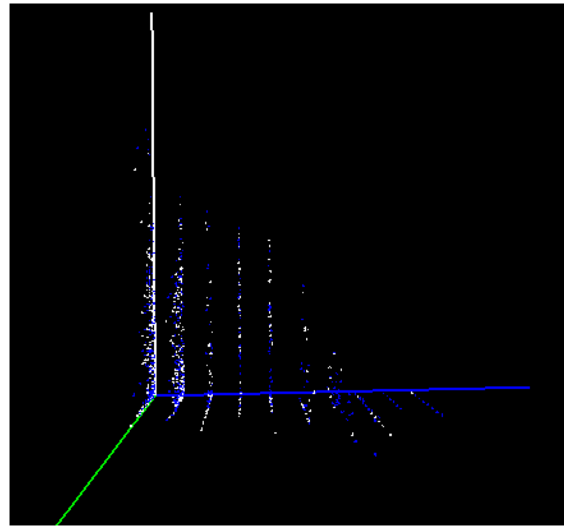


Figure 8-16. Comparison of niche fitness.

Such a range of controllers may be very difficult to obtain with a SEA. As evidenced in the trials run with traditional selection, the synthesis of closed-loop sensor-based controllers is highly unlikely in the presence of wave nodes. We believe that this is symptomatic of the SEA's inability to locate many such maxima in controller-space, implying that a even very large number of SEA trials may not yield the same diversity of controllers that a single niched trial may generate.

8.3.2 Computational Cost.

In our trials using the worm model a 'good' controller would typically lie in the fitness range [8, 11]. Using traditional selection such a controller was often discovered and optimised in as little as 20-30 generations. However, in three trials the SEA failed to locate a 'good' controller after 100 generations. With clearing-based selection a 'good' controller typically required 40 to 50 generations to be discovered and optimised, and the EA always produced such a controller after 100 generations.

It is intuitive that our EA is capable of converging more rapidly under traditional selection than under niching. If the EA is fortunate enough to start its search on the slopes of a 'good' maximum, the heavy bias towards exploitation will cause rapid ascent to the maximum. The best-case performance of our niching method appears to be equal to that of traditional selection.

The worst-case performance of traditional selection is very bad indeed. In three trials the EA fails to synthesise a 'good' controller after 100 generations. By comparison, the worst-case performance of the niching method results in a 'good' controller after approximately 75 generations.

An average-case analysis must allow for restarting the EA if it becomes trapped in a local maximum under traditional selection. Figure 8-9 implies that we should not restart the EA before at least 65 generations have been evaluated. Reading from Figure 8-9, we obtain:

P_{good} $\approx$ 0.6, the probability that any particular trial will produce a 'good' locomotion controller.

G_{good} $\approx$ 31, the average number of generations required to synthesise a 'good' controller, given that the trial produces such a controller within 65 generations.

G_{bad} = 65, the number of generations after which we restart the trial.

The expected number of generations required to synthesise a 'good' controller under traditional selection is given by Equation 8.1:

$$G_{expected} = \sum_{i=0}^{\infty} \left(1 - P_{good}\right)^i \left(P_{good} \times G_{good} + (1\text{-}P_{good}) \times G_{bad}\right) \tag{8.1}$$

$$\approx 47 + 18.8 + 7.52 + 3.00 + 1.20 + 0.48 + \ldots$$
$$\approx 78$$

By comparison, trials run under clearing-based selection require an average of $\approx$ 45 generations to synthesise a 'good' locomotion controller.

8.4 Conclusions

Clearing-based selection clearly outperforms traditional selection in our controller synthesis EA. Although niching has not resulted in the discovery of significantly fitter locomotion controllers, it has decreased the expected number of generations required to synthesis a fit controller by improving the EA's ability to escape local maxima. Importantly, the subpopulations maintained by a niching method provide a range of controller genotypes and locomotion styles. This incidental property of niching is extremely useful to an animator; a single evolution may produce a palette of locomotion controllers.

We believe that our EA is not yet capable of reliably finding global maxima in the controller-space of the worm model. Trials run using our niching method exhibit sensitivity to initial conditions and appear to locate different approximately equal maxima each time the algorithm is run. We believe that a superior differencing function and a larger number of niches would result in more consistent behaviour. We have reason to believe that the search space is highly complex, containing hundreds or thousands of local maxima. We hypothesise that higher maxima in this search space remain undiscovered.

8.5 Summary

Simple evolutionary algorithms (SEAs) perform poorly over multimodal search spaces. By modifying with the selection operator in such a way that candidates in different areas of the search space do not directly compete for reproductive rights, we greatly enhance the EA's ability to escape local maxima.

A *niching method* for an evolutionary algorithm maintains multiple subpopulations of candidates in the search space. By lowering the selection pressure between candidates in different subpopulations, or *niches*, we promote the existence of genetic diversity in the population and thereby increase our coverage of the search space.

The extension of our EA by the inclusion of a niching method significantly improves its performance over the locomotion-controller domain of our worm model. The most significant benefit of niching in our problem domain is that it produces a palette of different locomotion controllers in each evolution.

Chapter 9 Discussion and Conclusions

9.1 Evolving Locomotion Controllers: Conclusions

- Our evolutionary approach to controller-based animation has succeeded beyond our expectations. Our niching EA is capable of automatically and reliably generating locomotion controllers for a variety of mass-spring creatures. The main focus of this thesis has been on evolving locomotion controllers for a worm model, for which we were able to synthesise both open-loop and closed-loop controllers. The quality of our animations appears to be limited by the physical accuracy of the creature model, rather than the controller.

- Applying small random variations to the environment in which a locomotion controller is being evaluated results in improved EA performance by increasing its ability to escape local maxima. We have applied these variations in the form of randomly generated terrain, which can be scaled smoothly from totally flat to extremely rough. Rough terrain is a natural and attractive way of introducing variation to the creature's environment.

- Our master/slave distributed fitness evaluation architecture is robust, easy to implement and provides a linear performance increase with the number of slave processes up to the population size. Given the high computational cost of controller fitness evaluation, such an approach will probably be required for any further non-trivial controller synthesis.

- Implementation of a niching method has decreased the expected running time of our EA and provided an efficient way of obtaining a range of qualitatively different gaits for each creature model. Although our differencing function performs adequately, we believe that better functions exist and would provide superior EA performance. Our niching method has not exhibited any disadvantages with respect to traditional selection in our problem domains, and we believe that any future evolutionary approach to controller synthesis should include a niching method.

9.2 Knowledge Obtained

- A Simple Evolutionary Algorithm (SEA) can succeed in generating locomotion controllers for physically-based virtual creatures. However, the controller search space is often complex and a SEA may perform unpredictably over repeated trials.

- Most virtual creatures have multimodal locomotion controller search spaces; i.e. the creature's body is capable of many different forms of locomotion. The extension of an EA by a niching method can considerably improve performance over such multimodal domains. Niching methods are particularly suited to locomotion controller synthesis because they allow efficient location of multiple maxima, each of which results in a different type of controller or locomotion style.

- Visualisation techniques can be applied to the evolution process and can provide an intuitive, flexible and user-friendly method of data investigation. Benefits from visualisation include the encouragement of data exploration, leading to increased understanding of the EA's operation and increased probability of bug discovery.

9.3 Future Directions

- An obvious and straightforward enhancement would be to extend our system to evolve controllers for 3D mass-spring creatures. A 3D environment would provide a much more exciting and challenging control problem.

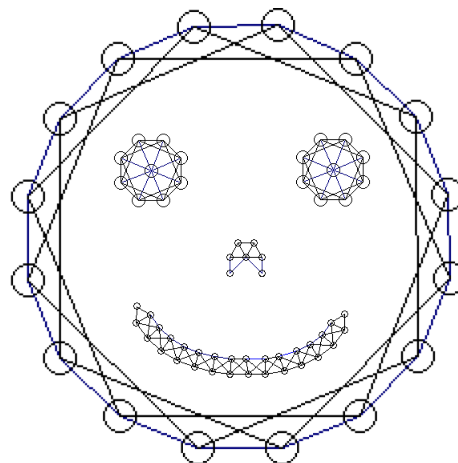
- Grouping in our genotype morphology graph allows us to create creatures of easily adjustable size. For example, we could add two more links to our octagonal creature's grouped genotype to create a decagonal creature. Could controllers evolved for an octagonal creature be successfully applied to creatures of fewer or greater sides? As another example, we could define a worm creature using a recursive genotype. By adjusting the recursive limit we could generate worms of arbitrary length. Could a controller evolved for a small worm be successful in controlling a larger worm? If so, significant savings in simulation computation could be made.

- Given that our controller representation has its origins in the articulated rigid-body virtual creatures of Sims [Sims_94], we believe that our EA would be very successful in generating controllers for rigid-body models. Skeletal animation by rigid-body modelling could be used in virtual creatures of high aesthetic quality.

- A high-level animation controller could take advantage of our evolved low-level controllers to autonomously perform complex animation tasks. Virtual creatures could then be used as virtual actors for animation tasks, or be used to populate interactive virtual worlds.

- An improved differencing function for our niching method will provide a more accurate metric for the degree of difference between two solution candidates. Would a better differencing function improve the performance of the niching method and should lessen its observed sensitivity to initial conditions?

- Extensions to our visualisation tool could increase its utility in investigating the evolution process. We suggest:
  - A facility to trace a candidate's ancestry.
  - More flexible control of data-selection.



Bibliography

[Blum_95] Blumberg, B. and T. Galyean (1995). Multi-level Direction of Autonomous Creatures for Real-Time Virtual Environments. ACM SIGGRAPH.

[Collins_97] Collins, T. D. (1997). Using Software Visualization Technology to Help Evolutionary Algorithm Users Validate Their Solutions. The Seventh International Conference on Genetic Algorithms (ICGA'97), Michigan, MI.

[Funge_99] Funge, T., X. Tu, et al. (1999). Cognitive Modelling: Knowledge, Reasoning and Planning for Intelligent Characters. ACM SIGGRAPH.

[Gold_89] Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimisation & Machine Learning, Addison Wesley.

[Grze_95] Grzeszczuk, R. and D. Terzopoulos (1995). Automated Learning of Muscle-Actuated Locomotion through Control Abstraction. ACM SIGGRAPH.

[Kodj_98] Kodjabachian, J. and J. A. Meyer (1998). "Evolution and Development of Neural Networks Controlling Locomotion, Gradient-Following, and Obstacle Avoidance in Artificial Insects." IEEE Transactions on Neural Networks. 9: 796-812.

[Laszlo_96] Laszlo, J., M. van de Panne, et al. (1996). Limit Cycle Control and its Application to the animation of Balancing and Walking. ACM SIGGRAPH.

[Light_70] Lighthill, M. J. (1970). "Aquatic animal propulsion of high hydromechanical efficiency." Journal of Fluid Mechanics 44: 265-301.

[Mahf_95] Mahfoud, S. W. (1995). Niching Methods for Genetic Algorithms. Illinois Genetic Algorithms Laboratory (IlliGAL), Dept of General Enginerring. Urbana, IL, University of Illinois

[Mill_88] Miller, G. S. P. (1988). The Motion Dynamics of Snakes and Worms. ACM SIGGRAPH.

[Ngo_95] Ngo, J. T., J. Marks, et al. (1995). "Further Experience with Controller-Based Automatic Motion Synthesis for Articulated Figures." ACM Transactions on Graphics 14, No. 4: 311-336.

[Nixon_99] Nixon, D. (1999). A Fluid Based Soft Object Model. Dept. of Computer Science. Auckland, New Zealand, University of Auckland

[Noever_92] Noever, D. A. and Baskaran (1992). " Steady-State vs. Generational Genetic Algorithms: A Comparison of Time Complexity and Convergence Properties." Santa Fe Institute Working Papers No. 92-07-032: 1-33.

[Petro_97] Petrowski, A. (1997). "A New Selection Operator Dedicated to Speciation." Proceedings of the Seventh International Conference on Genetic Algorithms.

[Provot_95] Provot, X. (1995). Deformation Constraints in a Mass-Spring Model to Describe Rigide Cloth Behaviour. Graphics Interface, Quebec City, Canada.

[Reyn_94] Reynolds (1994). Evolution of Corridor Following Behavior in a Noisy World. From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB94), Cambridge, Massachusetts.

[Sand_2000] Sanders, M. J. (2000). Locomotion Controllers for Mass-Spring Creatures. Advances in Intelligent Systems, Theory and Applications (AISTA2000).

[Sims_94] Sims, K. (1994). Evolving Virtual Creatures. ACM SIGGRAPH.

[Spears_92] Spears, W. M. (1992). Crossover or Mutation? Foundations of Genetic Algorithms Workshop, Vail, Colorado.

[Tu_94] Tu, X. and D. Terzopoulos (1994). Artificial Fishes: Physics, Locomotion, Perception, Behavior. ACM SIGGRAPH.

[Van_93] van de Panne, M. and E. Fiume (1993). Sensor-Actuator Networks. ACM SIGGRAPH.

[Witkin_97] Witkin, A. and D. Baraff (1997). Physically Based Modeling: Principles and Practice (Online Siggraph'97 Course notes) http://www.cs.cmu.edu/afs/cs.cmu.edu/user/baraff/www/sigcourse/index.html.

[Yama_94] Yamauchi, B. and R. Beer (1994). Integrating Reactive, Sequential and Learning Behavior Using Dynamical Neural Networks. the Third International Conference on Simulation of Adaptive Behavior.