The Evolution of Autonomous Agents Using Concurrent Genetic Programming

Adrian Trenaman

Submitted to the National University of Ireland,

Maynooth, for the Degree of Ph.D.

October 1999

Department of Computer Science

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:

Abstract

This thesis addresses the issue of how computational agents interact with and represent their environment in order to effect goal-achieving behaviour. It argues that the internal representations used by the agent to describe objects in the world should be based on how the agent perceives these objects and not necessarily on the representations a human designer might impose. A bottom-up methodology is proposed for the automatic design of distributed algorithms and internal representations to control autonomous agents. In particular, this thesis proposes and evaluates a new mechanism for the evolution of agents: "concurrent genetic programming". In this encoding scheme an agent is controlled by a set of evolved programs that are executed concurrently to yield an emergent control algorithm for the agent. This encoding forms a natural interpretation of the emergent principles of the discipline of artificial life in an evolutionary context, and so elucidates the ability of evolutionary computation to create such emergent systems. The performance of the approach is investigated as a function of several parameters. These are: using different numbers of programs in the agents, explicit memory, distributed memory architectures, deterministic and non-deterministic scheduling strategies, different levels of granularity of concurrency, and the evolution of scheduling strategy. These issues are investigated through the application of concurrent genetic programming to the standard Tartarus and Dozer virtual-robotics benchmarks. It is shown that concurrent genetic programming produces better agents for these environments than a conventional genetic programming approach. It does this by employing an implicit form of state that supports the development of cyclical behaviour strategies. Implicit representations of the environment are acquired at an evolutionary level rather than at the level of the agent's experience. Although this form of internal representation leads to fit agents, it does not exhibit the formation of explicit models of the agent's environment. Instead, it allows the development of a form of internal state appropriate to achieving good fitness.

Acknowledgments

A Ph.D. is a long and arduous task, in the course of which many people play a part. In particular, thanks are due to my supervisor, **David Vernon**, for his guidance and clear mind. You gave me much freedom with this work, as much rope as I needed, and yet were always there to untangle the knots and loops I invariably made.

Throughout this work travel grants were made to me by the American Association for Artificial Intelligence and EvoNet to attend genetic programming conferences. These conferences provided a great forum for discussion and debate, and certainly helped mold and shape this work. Thanks are due to **William B. Langdon**, **Dan Ashlock**, **Forrest Bennett III** and **Una-May O'Reilly** for their insight and words of encouragement at these events. Thanks also to the Irish GP crowd, **Conor Ryan**, **Michael O'Neill** and **Laur Ivan**.

The staff of NUI Maynooth have been most supportive throughout: thanks are due to **James Cotter**, **Peter Boran** and **Mark Frain** for helping me use laboratories of machines at a time. **Diarmuid O'Donohue** always lent an ear to my ideas on artificial intelligence, providing a much needed sounding-board and great source of friendship. Thanks also to **Aman Kohli** for his insightful comments at the proof-reading stage.

Thanks to my family for their love and support over the last four years: you were great! Finally, thanks to **Steve**, **Dave**, **Pierce**, **Emmet**, **Sue**, **Mark**, **Fearghal** and **Niav** for sticking with me through thick and thin. You are the stuff of true friends. And for anyone I've left out, don't worry, I'll put you in next time ...

Contents

Chapter 1

Introduction

1.1 Introduction

This thesis addresses the issue of how computational agents interact with and represent their environment in order to effect goal-achieving behaviour. At the heart of this problem is how an agent should perceive and represent the environment about it.

Artificial intelligence has traditionally assumed an anthropomorphic view of reality: that the way a human perceives the external world gives a true and absolute representation of reality. It is assumed that when a person perceives an object, say a chair, there is a corresponding concept of this particular chair in the person's mind. The chair is described using some representational scheme, and intelligent activity involves using these descriptions of the objects in the real world to effect behaviour, for example, going to the chair and sitting down. The descriptions of the objects in the world consitute a model of the world, which is kept consistent with reality through on-going perception of the outside world. This approach, although common-sense, is problematic. First, maintaining an isomorphic mapping between the objects in the real world and those in the world model has proven to be an elusive goal. Second, even if an accuracte world model was available, we are left with the problem of finding the "intelligent" algorithms that can give the desired behaviour.

An antithetical approach to the problem of maintaining and using world models comes from the argument that we *don't* use representations of the real world. Our intelligent behaviour arises through out interaction with reality, not through internal representations of it. Adopting this stance, the problem of representation and manipulation of world-models is removed, and replaced with the developement of mechanisms that give an appropriate response to input from the environment. This approach has been effective in producing agents that interact with their environment; however, it will only work when there is always enough information readily available to the agent for it to decide the best action to take.

I propose a methodology to unite these opposing approaches, acknowledging the importance of interaction with the environment and supporting the development of internal representation in environments where it is necessary for successful behaviour. Essentially, I argue not that representation is wrong, but that the way reality has been represented is wrong. A person's perception of reality is based intimately on his experience of it through the senses of sight, touch, smell, sound and taste. In the same way, an agent's perception of reality should be based fundamentally on its sensory experience, building internal representations that are appropriate to its sensory interface with the environment.

In this methodology a bottom-up approach to agent-design is advocated where the bias towards a "human" world-model is removed and the agent's representations and algorithms are found through an adaptive evolutionary process. The evolutionary approach allows the specification of the computational building blocks that are combined to give an algorithm and representation that give the desired behaviour. The evolution of agents involves the specification the computational building blocks, the way these blocks are put together, and the way they are encoded at the evolutionary level.

This thesis introduces "concurrent genetic programming", a new encoding for evolutionary computation. Agents are controlled through the interleaved execution of a number of evolved programs, the resulting behaviour an emergent effect of the interaction between the programs. The incorporation of building blocks allowing the development of internal representations is examined, along with issues relating to the granulariy of concurrency and the number of programs controlling the agent. Concurrent genetic programming can produce better agents than conventional genetic programming, and builds internal representations through an implicit form of state due to the interleaved execution of the programs.

1.2 Contributions of this Thesis

The key contributions of this thesis are the investigation of concurrency in the context of genetic programming, and the use of explicit state mechanisms to develop internal state for agents.

The conventional application of genetic programming to agent-design is to evolve a single program that decides the best action for an agent to take based on input information from the environment. This thesis addresses an emergent approach, where the agent is controlled by a set of evolved programs executing in parallel. This architecture, based on emergent principles from the field of artificial life, does not offer anything beyond the space of Turing machines already explored by genetic programming. What it offers is a different representation of the program space that may be more amenable to evolutionary search. Key elements of this encoding are investigated. First, the number of programs controlling the agent: can better agents be found when agents contain more programs? Results show that while there is a clear benefit in using two concurrent programs in the agent, there is no further benefit in using a greater number of programs. Second, the effect of granularity of concurrency: should the programs in an agent be evaluated in pseudo-parallel using fine-grained concurrency or should coarse-grained concurrency be adopted? Alternatively, should such issues be left to the evolutionary process? The most successful agents evolved where those that evolved their own scheduling strategies. Third, should the interleaving of program execution be achieved in a non-deterministic or deterministic fashion? Deterministic and non-deterministic schedulers were examined, and while both gave better results than single program agents, the latter provided more robust solutions. Fourth, if explicit state mechanisms are to be employed, should they be distributed, using local or shared memory modules? Results show that regardless of different memory architectures, the concurrent approach can instead exploit an alternative implicit mechanism for internal state. In examining these issues this thesis investigates the efficacy of an emergent approach to behaviour in an evolutionary context and adds another tool to the toolbox of evolutionary computation.

The second contribution of this thesis is to further the investigation of the evolution of internal state. To act autonomously an agent must interact with and represent facets of its environment. Representation is a hard subject, rooted in the fundamental philosophical issues of how we perceive reality. Rather than remove the problem of representation and concentrate on purely reactive behaviour, the discovery of appropriate representations is left to an evolutionary process. Genetic programming has previously employed implicit and explicit state mechanisms to allow representations to emerge. While explicit state mechanisms were incorporated into concurrent genetic programming, it was found that an implicit form of state due to the interleaved execution of programs was exploited by the evolutionary process. Ultimately, the agents developed suitable internal state enabling them to effect goal-achieving behaviour through the repetition of action sequences. This state reflects knowledge of the agent's environment in a novel manner. The contribution is twofold: concurrent genetic programming provides an implicit form of state, but it also suggests that the benchmarks used may not promote the use of explicit state mechanisms as was previously thought.

1.3 Organisation of this Thesis

This thesis is organised as follows. Chapter 2 defines the term "autonomous agent", and argues that autonomous agents should have the ability to create their own representations of their environment. The representations that the agent-designer employs are not necessarily the most appropriate for an agent, which perceives the environment using different sensory apparatus. It is proposed that the algorithms and representations used to exhibit intelligent behaviour can be arrived at through an evolutionary approach, and an overview of evolutionary computation is given in in Chapter 3. Chapter 4 provides a survey of how evolutionary computation has been applied to the design of agents.

Chapter 5 introduces *concurrent genetic programming*, a natural implementation of the emergent ideas of artificial life in the context of evolutionary computation. Studies in artificial life show that structure and behaviour can emerge through the interaction of entities controlled by simple rules; however, they are lacking in that they do not offer a method to find these simple rules from a high-level description of the desired behaviour. In concurrent genetic programming, agents are controlled through the interactions of a number of evolved programs, who co-operate to produce an emergent behaviour. Two different implementations of concurrent genetic programming are described, one using a non-deterministic scheduler, the other deterministic. The latter implementation was built using an open object-oriented framework developed by the author, described in Appendix A. The issues raised by concurrent genetic programming, namely the number of programs controlling an agent, granularity of concurrency, and the use of explicit memory as a tool for the development of internal representations of the environment are examined in Chapter 6. Concurrent genetic programming is applied to the evolution of agents in a number of virtual robotics worlds, and produces fitter agents then those produced using the conventional genetic-programming approach. The ability of concurrent genetic programming to allow the formation of an implicit form of state is discussed.

Conclusions are presented in Chapter 7, along with a critical appraisal and directions for future work.

Chapter 2

Autonomous Agents

2.1 Introduction

This chapter begins with definitions of some key concepts, namely intelligence, agent, and autonomy, with the aim of finding a constructive starting point for the design of autonomous agents. Essentially, designing agents involves finding the correct algorithm and representation that control the agent within its environment, so that it achieves goals set by the agent-designer. An important issue in agent design is how its environment is represented: should the agent maintain an internal representation of the outside world, and if so, what does this internal representations look like? The answer to the first question depends on the environment in which the agent acts: there are some environments when internal representation is needed, and others where it is not. The second question is harder to answer: this chapter argues that the internal representations an agent uses must be suited to the sensory and motor abilities of the agent, and that these representations may be quite different from those considered by a human designer. By examining techniques whereby the correct algorithm and representation can be acquired with the minimum of intervention from the agent-designer, better agents can be created.

2.2 Artificial Intelligence and Autonomous Agents

The goal of artificial intelligence is arguably to create machines that behave as intelligently as humans. To achieve this goal one must be clear on what 'human intelligence' means; however, this term eludes definition. Acknowledging this difficulty, Turing instead proposed a test for human intelligence, where a machine would be termed as intelligent as a human if it could respond under interrogation in such a manner that a human operator could not tell if it was a machine or a person responding [73]. The Turing test has been the source of much debate [65], nonetheless it set the target for artificial intelligence research. In concentrating on human intelligence, the field of aritificial intelligence became entrenched in the debate as to whether it is possible for computing machines to embody human intelligence, and in ways to make machines do the kind of things humans do.

An alternative approach to artificial intelligence is to remove the aspiration to human intelligence and concentrate instead on more fundamental aspects of intelligent behaviour. This is a pragmatic point of departure for artificial intelligence research, focussing on how to create machines that perform complex tasks. Fogel offers the following description of intelligent behaviour:

> "...intelligence can be viewed as the ability of any decision-making entity to achieve a degree of success in seeking a wide variety of goals under a wide range of environments." [18]

This definition distills the essence of intelligent behaviour, offering a constructive starting point for artificial intelligence research. The ascription of intelligence to behaviour is subjective: the only way to measure intelligence is to observe the behaviour of the agent in the environment. As Searle has pointed out, this behavioural assessment can mislead, as the ability of a machine to mimic intelligence does not imply that the machine is intrinsically intelligent. However, in the absence of a non-behavioural test to show if a machine is intelligent or not, the behavioural stance will have to suffice. The issue of whether a machine is intrinsically intelligent or not is set aside, and instead the mechanisms by which intelligent behaviour can be effected are investigated.

Fogel's definition of intelligence above does not refer to intelligence as a solely human property. The movement of artificial intelligence towards this abstract view of intelligence from the traditional anthropocentric standpoint is epitomised by Brooks:

> "I wish to build completely autonomous agents that coexist in the world with humans, and are seen by those humans as intelligent beings in their own right." [7]

The term "agent" is often used to describe a machine that acts intelligently in its environment. Essentially, an agent is "one that acts or exerts power," however the term has become hackneyed and worn with use. Wooldridge proposes a weak and strong definition of agency [83]. Weak agency refers to computer systems that exhibit autonomy, social ability (communication), reactivity, and proactiveness. Strong agency incorporates all the elements of weak agency but tends to be conceptualised in terms commonly applied to humans, for example, knowledge and intention. By distinguishing between weak and strong agency, research can focus on the more fundamental aspects of weak agency without getting side-tracked by more anthropmorphic aspects of strong agency.

An "autonomous" agent is "one that acts without external control." External control can manifest itself in two forms: through the intervention of an operator or through input from the environment. A motorised toy car roaming a room and avoiding obstacles is not autonomous if an external operator makes its control decisions over a remote control link. When the operator ceases to play, the car cannot operate "on its own." Agents exhibit autonomy when they can operate without outside intervention. Of course, such an agent is still subject to external control, as its decisions are made based on the input from the environment, an external force. This external force is inescapable: interaction with and response to the environment are fundamental aspects of agency. However agents can reduce the extent to which their actions are dependent on the environment by maintaining internal state and making control decisions based on a combination of internal state and external input.

State means a "mode or a condition of being": for example, a person may be described as being in a state of readiness, or in a higly nervous state. For the purpose this thesis, "state" is that information that describes the internal structure of an agent at an instant in time. An internal representation of an external object corresponds to a particular state within in the agent. At any moment of time an agent can be said to be in one of a set of possible states. Over time the state of an agent changes, and an agent can make different choices in the face of the same environment input due to its internal state. Letting E denote a finite set of possible inputs from the environment, A denote a finite set of possible actions the agent make execute, and S denote a finite set of internal states, then the agent's behaviour is determined by the mapping: [1]

$$F : E \times S \rightarrow A \times S \tag{2.1}$$

This abstract model of an agent is shown in Figure 2.1. Designing autonomous agents, at its most abstract, involves finding the mapping F from the set of environmental perturbations E and internal state S, to actions A and new state S.

[1] This notation will be used throughout the thesis to describe mappings (functions). The notation $F : A \rightarrow B$ says "F is a mapping from the elements of set A (the domain) to the elements of set B (the range)". The symbol "$\times$" is a binary operator used to denote the Cartesian product of two sets.

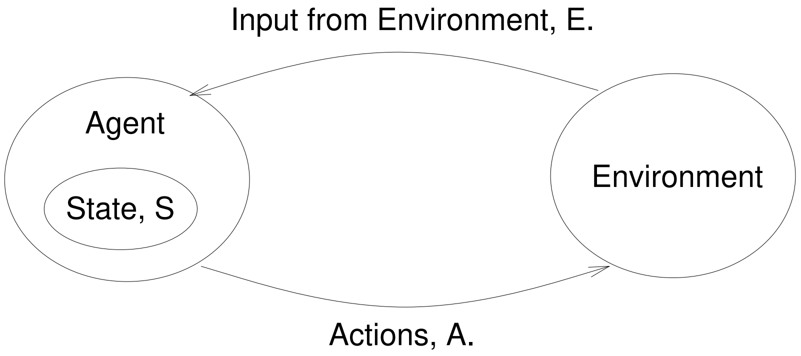

Figure 2.1: An agent exists with an environment, and interacts with the environment by receiving input from the environment and producing actions based on the input and the internal state.

2.3 Agents as Information Processing Systems

The agent model incorporating state shown in Figure 2.1 is essentially an information processing system: the agent receives information from the environment, processes this information, and uses the result to choose an action. Marr maintains that to design and understand complex information processing system, they must be addressed at three levels: computational theory, algorithm and representation, and implementation [50].

The computational theory describes at an abstract level what is being computed, and why this computation is appropriate to solve the problem at hand. Given a computational theory that addresses a problem, a representation and algorithm can be devised that implements the computational theory. A "representation" is a formal system for making explicit certain entities or types of information, together with a specification of how the system does this. When an entity is described using a particular representation, we have a "description" of the entity. The representation is used to formulate the input and output of the computational process; the algorithm then performs a transformation through which input information is converted to output information. The third level in Marr's framework is the implementation level, where the algorithm and representation are physically implemented. Marr relates this level to hardware; however, we will loosen this here and let it apply to software also: the implementation is the mechanism by which the algorithm and representation are realised.

Relating Marr's framework to the functional description of agents in Equation 2.1, the representation characterises the input from the environment E, the internal state S and the output actions A, and the algorithm corresponds to the mapping function F. The clarity offered by Marr's framework is that in order to design an information-processing system, in our case the quadruple (F, S, E, A), one must begin with a computational model. Marr argues that traditional artificial intelligence has come at the problem from the wrong direction, concentrating at the level of algorithm and representation rather than on a computational theory of intelligence:

> "For far too long, a heuristic program for carrying out some task was held to be a theory of the task, and the distinction between what a program did and how it did it was not taken seriously." [50, page 28]

The neglect of computational theory in aritificial intelligence is captured by Winograd and Flores [81] in their four-stage description of the "traditional" approach to artificial intelligance. First, characterise the task environment, identifying the different states of the environment, the actions that can change this state, and the goals that must be attained. Second, form an internal representation to describe the environment within a computer. Third, determine the outcome of different possible courses of action. Fourth, choose one of these actions that leads to the attainment of one or more goals. This approach focusses directly on Marr's second level, that of representation and algorithm, without any mention of a computational model. In doing so, traditional artificial intelligence has produced a wealth of techniques to tackle such tasks as chess playing and searching without addressing the central problem of what is intelligent behaviour.

2.4 Representational and Anti-Representational Approaches

The shying away from a computational theory may be rooted in what Winograd and Flores call the rationalist tradition of western philosophy [81]. The rationalist approach begins with the assumption that we inhabit a real world, filled with objects with certain properties, and our actions take place in this world. Then, there are objective facts about this real world that are independent of our subjective interpretation. These facts about the real world are registered in our thoughts by the process of perception. Finally, thoughts and intentions about actions can cause physical motion of our bodies in this real world, and so instantiate actions. This view of reality is fundamentally representational; that is, alongside the objective real world the individual maintains an internal model of the world, and it is this internal representation that is the basis for the individual's action and experience. This common-sense notion of internal knowledge, separate from the domain of real things, has dominated western thinking and is responsible for the strong representational aspect of artificial intelligence [77].

This mode of thought became adopted as a *de facto* computational theory of intelligence: intelligence was seen a process of perception, representation and action, and artificial intelligence focussed on providing the algorithms and representations to implement this process. Brooks is highly critical of this representational approach [7, 8]. He argues that the decomposition of intelligence into seperate processes of perception, representation, and action has resulted in separate fields of inquiry with limited success in each domain but no general theory of how to place these components together as one unified intelligent system. This is particularly evident in robotics, where the problems of perception and representation of internal models of the world has hindered real progress. For even simple block-world environments, errors easily creep in to the internal model due to model inaccuracy, sensor noise and motor drag, and the model ceases to accurately represent the outside world. Any reliance of control algorithms on incorrect descriptions of external objects thus leads to poor performance.

Brooks concluded that representation is a stumbling block, and so moves to design autonomous agents *without* internal representation. Using his "subsumption architecture", where an agent's behaviour emerges through the combination of simpler behavioural modules, he created autonomous agents that roam in the real world, avoid obstacles, and even identify power-points to allow "refuelling" when an agent's energy was low. These successes indicated that internal state is not necessary for autonomous behaviour, that is, that autonomous behaviour can be produced by a mapping F from environment input E to action A [2]:

$$F : E \to A \tag{2.2}$$

This abstract agent model differs from that adopted previously, in that there is no mention of internal state. It is sometimes referred to as behavioural, in that it is concerned primarily with mapping from input to action, without reference to the underlying internal mechanisms that effect this mapping. The success of this state-less approach in producing autonomous agents is misleading; Minksy is highly critical of the rejection of representation:

> "To give you one idea of some of the dumb things that have diverted the field, you need only consider situated action theory. This is an incred-

[2] This abstract model of behaviour is similar that of Equation 2.1, with the internal state, S, removed. This can be thought of as letting $S = \emptyset$

> ibly stupid idea that somehow swept the world of AI. The basic notion in situated action theory is that you don't use internal representations of knowledge." [67, Chapter 2, page 17]

Minsky's response is both acerbic and potent: knowledge representation is such an obvious part of intelligence that it is hard to justify an approach that ignores it completely. Nonetheless, we must still acknowledge the successes of the non-representational approach. The two extremes can be reconciled by referring to the complexity of the environment: whether state is required depends on the agent's environment and the behaviour required. As Reik explains [61], when there is enough information in the environment input to allow the agent to choose the optimal action (where by optimal we mean an action that brings the agent closer to its goal) then the environment is a Markov environment. For example, if light-seeking behaviour is required in a robot, and the robot's sensors indicate the direction of highest light intensity, then no internal state is required as there is enough information in the input to decide the best action to take. When the next state of the environment cannot be deduced from the current state, the environment is non-Markov: internal state must be employed to model the hidden-states in the environment that can be used to determine the next state. If a light-seeking agent roams momentarily into a shaded area, it should "remember" the direction it was going, and continue in that direction in the absence of immediate stimulus.

Wilson pursues a broader classification of environments than just "Markov" or "non-Markov" in his sensory state machine (SSM) formalism [80]. An environment is "Class 0" if for every sensory stimulus there exists at least one action that is known to lead to positive reinforcement, that is, bring the agent closer to achieving a goal. A "Class 1" environment is one for which only some sensory stimuli there exists an action that gives positive reinforcement: there are some stimuli for which no "optimal action" can be chosen. Finally, "Class 2" environments are those for which the environment can viewed as a non-deterministic state machine, where it is not possible to predict on the basis of current stimulus alone what the next state will be. For these environments, it should be possible to predict the next state by maintaining a history of previous stimuli. These environments can then be thought of as "Class $2.k$", where k describes how much previous stimuli must be remembered in order to perform well in the environment.

2.5 An Alternative Ontology

The need for internal state in achieving intelligent behaviour depends on the environment in which the agent must act: some environments will require internal state and others will not. When internal state is required, does this mean adopting the representational approach that Brooks has found to be so problematic? Vernon argues that an alternative ontology (theory of existence) might offer different perspective on reality than that of the rationalist tradition [77]. Reality and the mind's perception of it has proved a troublesome issue for philosophers: Berkeley's assault on the existence of matter (described succintly in [38]) is indicative of the difficulty of this area. Kant's notion of *noumena* and *phenomena*, respectively "things as the are" and "things as they seem", is more intuitive, assuming the existence of an absolute objective reality, but acknowledging that it is only made known to us through perception. As individuals in a world, we can only know of objects through our limited perception of them, and so we build our own phenomenological interpretation of reality based on the objective world.

Heidegger develops Kant's experiential notion of existence and reality further [81]. He maintains that Philosophy has stepped back from the primacy of experience and understanding that comes without reflection. Rather than take the of viewpoint of the contemplative philosopher who ponders his existence, it is better to consider the real-life everyday experience that we are part of and act in *without thinking*. Heidegger's work evoke's mixed reaction: Scruton, in [38], says of Heidegger's book *Being and Time* "... a formidably difficult book, unless it is utter nonsense, in which case it is laughably easy." Heidegger's ideas are useful here, because they offer a radically different view of "being" and "reality", based on the dynamic interaction between an entity and its environment rather than on the representation of reality through perception.

For Heidegger, practical understanding is more fundamental than theoretical understanding, that is, detached contemplation about intelligence and being is far less important than actually "being there", a concept he terms "thrownness". Thrownness describes those everyday situations where we act without thinking, without internal representations, without reflection on the effect of our actions. We are literally thrown into a situation, where even choosing inaction is an action in itself.

Thrownness can be demonstrated using a simple everyday scenario. A typicial breakfast might involve someone fetching a bowl, filling it with breakfast cereal, pouring milk and eating. There is no internal representation of the objects at hand, namely, the bowl, the milk or the spoon: as a result of their readiness-to-hand they become inconsequential. The person doesn't "think" of these things at all, in the same way that the person doesn't think of the arm holding the spoon, his mouth or his digestive system. Such objects only come into play as entities in their own right through the process of breaking-down, in which they become "present-at-hand". A spoon only becomes "represented" when it is not ready-to-hand, and so must contemplate on what a spoon is and where to find one.

Heidegger believes that reality and existence are not defined by some objective all-knowing observer, nor by the individual subjective observer, but rather through the dynamics of interaction between the individual and the environment, through involved activity with the potential for breaking-down. As a consequence of thrownness and breaking-down, we do not relate to things primarily through having internal representations of them: rather, we relate to things through our interaction with them. That is not to say there is no such thing as internal representation: clearly, to think, we must represent abstract concepts in some way. The point is that such representation is secondary to experience with the real world. Brooks [7] is cautious not to associate himself with this "German Philosophy", but there are clear links to be seen between his non-representational approach and the philosophy of Heidegger.

Adopting Heidegger's perspective, instant reaction to environmental perturbations without representation and conscious thinking becomes the prime facet of the intelligent creature. However, this does not mean adopting an extreme behavioural stance, as Brooks has done, where there is no representation at all. Heidegger did not believe that abstract thought does not exist: such reflective thought most certainly does exist, but as a secondary process. Adopting this view, the debate over the reactive versus representational aspects of intelligence is resolved by acknowledging that both have a part to play in the generation of behaviour, and autonomous agents should incorporate both. It is a matter of finding the right representation for the task at hand.

2.6 Why Representation is Hard

When internal representations come into play, they introduce what Heidegger terms "blindness". Blindness describes the way that once an abstract representation is employed, a new closed universe of discourse is created for the problem at hand, shutting out certain details as irrelevant. Blindness is an inescapable artifact of representation.

Blindness has two implications. First, we are forced to assume that the intelligent behaviour we are looking for can be achieved using the representation adopted.

No algorithm can succeed if the underlying representation of the environment obfuscates those aspects of the environment that are important for successful behaviour. Intelligence is sometimes viewed as the ability to create new representations in which solutions to problems can be found: if a machine can exploit a number of alternate representations, or alternatively can adapt representations to the problem at hand, the limitations of blindness can be alleviated. The second problem posed by blindness is that even if the representation is appropriate (that is, it does not hide salient details of the environment), the resulting descriptions must be kept consistent with the outside world. This problem, of keeping an internal model consistent with the external reality, played a key role in Brooks's rejection of internal representations.

Blindness can be used to propose the following argument against machine intelligence: "if machines must use internal representations for intelligent behaviour, then they will as a result be blind, that is, limited to a predefined domain of discourse and not intelligent". However, this argument applies equally to humans. If we believe that humans, our best example of intelligent behavioural entities, use internal representation, then the above argument asserts that they suffer the same limitations as machines. Blindness does *not* imply that a machine (or human) is not intelligent. Rather, blindness simply captures why it is hard to make machines intelligent: because the representations we impose on the machine are blinding.

Consider a mobile robot with a set of proximity and light sensors, roaming a room, avoiding obstacles, and seeking light. What is an appropriate representation for this task? A human designer might propose a description of the objects in the room, their size, shape and velocity. This is a natural way for a human to describe a room, based on the information rich stimuli available to the human. However, the robot perceives the room through a small set of sensors that return a set of numbers, indicating for example the light intensity on sensor or the proximity of external objects. The alien model of the world imposed on the robot simply doesn't work with the robot's limited perceptual capabilities. If a model is to be employed, it must be based first and foremost on the perceptual capabilities of the agent, not on those of the agent-designer.

For the designer of autonomous agents, the problem then becomes an epistemological issue: how can knowledge of the world be represented inside the agent in a manner that is not biased towards our own representation of the world and is appropriate to the sensory and motor capabilities of the agent? The next chapter proposes an approach to this problem that acknowledges the success of behavioural architectures while incorporating mechanisms for the creation of internal representations to allow more sophisticated behaviour. The approach focuses on the development of appropriate algorithm and representation (Marr's second level), rather than on the development of a computational theory of intelligence. In doing so, the implicit assumption is made that a computational theory does exist: if there is no computational theory then any attempt to find an algorithm or representation is pointless. When the computational theory is not known we can still proceed if we ensure that the mechanisms are put in place by which an adaptive process can find the algorithm and representation to implement it.

2.7 Summary

The arguments of this chapter can be summarised as follows. First, an agent acts in an environment, effecting a behaviour that leads to the achievement of goals that are specified by an agent-designer. Second, the agent experiences its environment through sensors, which give it representations of the reality external to the agent. In some environments, agents may need to maintain internal representations of the external reality to achieve the goals laid down by the designer. Third, the representation of external reality is a hard problem. The human "world-view" is rooted in the sensory inputs provided through sight, smell, taste, touch and sound, conjuring a world in space and time that may not be inappropriate to the sensory capability of an agent. Fourth, rather than implicitly imposing a human-based interpretation of reality on an agent through the design of its algorithm and representation, as artificial intelligence has often done, an alternative approach is to investigate mechanisms by which an agent can develop its own interpretation of reality better suited to the agent's sensory ability. Evolutionary computation offers a methodology to suit this, and is presented in the next chapter.

Chapter 3

Evolutionary Computation

The previous chapter discussed intelligence and autonomy and suggested that an evolutionary approach could be used to find suitable representations and algorithms for agents. This chapter describes the field, showing how it can be applied to different problem domains using variations of the same evolutionary algorithm with different genotype structures.

3.1 Evolutionary Computation

Evolutionary computation was pioneered by Fogel as early as 1966 [18], formalised by Holland in 1975 [27], and more recently has enjoyed a revival and flourishing through the works of Koza [40], Goldberg [21] and Fogel [17]. The evolutionary approach is straightforward: a population of individuals is randomly created, and the fittest of this population are kept while the poorer individuals are eliminated. The fittest individuals are used to create offspring, combining the traits from two or more parents to provide a new and better individual. Over time the fitness of the population as a whole rises, hopefully above the fitness that could be achieved through standard analytic design techniques. Although evolutionary computation is attractive in that it seems to mimic nature's way of producing highly fit individuals, it is fundamentally an adaptive search technique, and we will treat it as such here. The allusions to natural evolution must be taken as metaphorical, drawing analogies that may help to explain how the process works.

Evolution guides species through a space of possible forms, eliminating those forms that are below fitness and accentuating those forms that perform better in the environment. Central to natural evolution is the fundamental distinction between the *genotype* (the coded description of an individual) and the *phenotype* (the physical instance of the individual). Through a process of development, the genotype becomes the *phenotype*, that is, the physical instance of the genotype. The phenotype then lives a short lifetime in which it tries to survive and if possible reproduce to make new progeny. In the mating or reproductive process, genes can be copied directly, mutated slightly to introduce variation, or combined to produce new genotypes with combined attributes of the parents. Only those phenotypes capable of reaching maturity, that is, only fit individuals, succeed in propagating their genetic code: in this way, the species evolves over time, towards some peak of fitness.

This form of evolution, *Darwinian* evolution, allows attributes to be passed from parent to offspring only through the genetic code, that is, the genotype. On the other hand, *Lamarckian* evolution allows those attributes acquired throughout the lifetime of a parent to be passed down to the child. In evolutionary computation, Darwinian evolution is the most prevalent form used, although attempts at augmenting the evolution with non-genetic inheritance of attributes has been investigated [46].

In genetic *parlance*, the genotype of an individual is its collection of chromosomes. Each chromosome consists of a number of genes, where each gene determines a particular facet of the entity. For example, certain strains of garden pea have a gene to determine blossom colour. Each gene can hold a number of possible values, known as alleles: the gene to determine blossom colour may hold the allele for pink or the allele for white. Let $allele_i$ be the set of possible values for $gene_i$, that is,

$$gene_i \in allele_i$$

The chromosome is a list of the n genes, and so is an element of the cartesian product of the Allele sets:

$$chromosome \in allele_1 \times allele_2 \times ... \times allele_n$$

Here, the symbol $\times$ is used to denote Cartesian product of sets. The cardinality of the chromosome, that is, the number of different chromosomes that can be created, is:

$$|chromosome| = |allele_1| \times |allele_2| \times ... \times |allele_n|$$

A typical vertebrate has approximately 10,000 genes. Conservatively taking two alleles for each gene, the number of possible chromosomes is 2^{10000}. Any instance of a chromosome is just 1 in a space of some 10^{2000} possibilities. Darwinian evolution provides a method to progressively navigate this space of possibilities towards some chromosome corresponding to a highly fit structure. The result of the evolutionary process should not be considered to be necessarily the *optimum* structure in the space: in most cases, evolution will simply lead to a *very good* solution.

3.2 Adaptive Plans & Genetic Operators

After Holland [27], an adaptive plan is that which guides a population of entities through a space of possible structures, in an attempt to maximise some fitness function. Using α to describe the space of of possible structures, then the adaptive plan τ is a function of the form

$$\tau : I \times \alpha \to \alpha$$

That is, the adaptive plan takes information from the environment, I, and some structure from the structure space α, and produces a new structure for the next generation. The idea of an adaptive plan is suitably abstract and removed from natural evolution. By beginning with the abstract notion of an adaptive plan, we can see how evolutionary inspired plans are a subset of a wide range of possible approaches.

Holland identified three classes of adaptive plan, $\mathcal{R}^1$, $\mathcal{R}^d$ and $\mathcal{R}$. The most narrow class of plans is called $\mathcal{R}^1$. Algorithms for plans from $\mathcal{R}^1$ work as follows. First, an initial random population of individuals is created, $\{A_1(t), A_2(t), ..., A_m(t)\}$, where $A_i(t)$ means "individual i at time t". The performance of each individual, $\mu(A_h(t))$, is observed and recorded. A probability distribution is then cast over the individuals of the population, assigning to each a "selection" probability equal to the ratio of its fitness to the total population fitness.:

$$P(A_h(t)) = \frac{\mu(A_h(t))}{\sum_i \mu(A_i(t))}$$

This distribution will favour those individuals with higher performance. One or more individuals are probabilistically selected using the above fitness-proportionate selection mechanism and a structure-altering operator is applied to the individual. The resulting new structure is placed back into the population. This form of evolution is known more commonly as "steady-state" evolution.

This fitness-proportionate selection (sometimes known as roulette-wheel selection) is not the only kind of selection mechanism. An alternative selection strategy is tournament selection, where a number of individuals are chosen randomly and ranked according to their fitness, the fittest individual then being selected. Ryan's [64] *Pigmies and Civil Servants* selection algorithm allows multimodal selection to proceed according to an individual's fitness and parsimony.

Plans from the class $\mathcal{R}^d$ differ from those in $\mathcal{R}^1$ in that at each iteration a second tentative population is required to hold the offspring from the original population.

The proportional fitness of each individual determines how many offspring each individual will create: fitter individuals are thus allowed more offspring. This form of evolution is known as "generational", as at each stage in the evolutionary process a new generation is created.

Holland amalgamates the two classes $\mathcal{R}^1$ and $\mathcal{R}^d$ to give the $\mathcal{R}$ class of plans. This is the adaptive plan that has become most widely accepted as the Evolutionary Algorithm, easily summed up as:

1. Randomly create an initial population of individuals.

2. Evaluate each individual in the population, giving each a fitness-score.

3. Select individuals according to their fitness and use these to create new individuals for the next generation.

4. Repeat steps 2 and 3 until a suitably fit structure is arrived at.

An alternative classification of adaptive strategies, after Rechenberg and Schwefel, is described by Fogel [17, pages 85–89]. The $(1+1)$ strategy involves just one parent in the population, from which one offspring is made through some mutation process. Parent and child then compete, and the fitter becomes the parent in the next generation. This strategy is then extended to incorporate multiple parents and multiple offspring, through the $(\mu + \lambda)$ and (μ, λ) strategies. In the former, (characterised by the "plus" notation) μ parents create λ offspring, which then compete with the parents to be the μ fittest, which become the parents for the next generation. In the latter, (characterised by the "comma" notation) only the λ offspring compete, the original population being completely replaced by the offspring.

The essence of an adaptive strategy then is: from an existing population, new structures are created through modification of the existing structures, which then compete to replace the existing structures in the next generation. Adaptive *operators* are used by adaptive plans to create new offspring from one or more existing individuals. Holland took inspiration from nature for his adaptive operators, and so the operators chosen are known as the *genetic* operators: *crossover*, *mutation*, *inversion* and *reproduction*.

Crossover Crossover involves selecting two individuals from the population, according to some distribution usually based on fitness. We call these two structures $A = a_1 a_2 ... a_l$ and $A' = a'_1 a'_2 ... a'_l$. A number x between 1 and l is randomly picked, and two new individuals are created using the alleles $1 \ldots x$ and $(x+1) \ldots l$. Finally, one of these individuals is discarded at random. The process is shown in Figure 3.1.

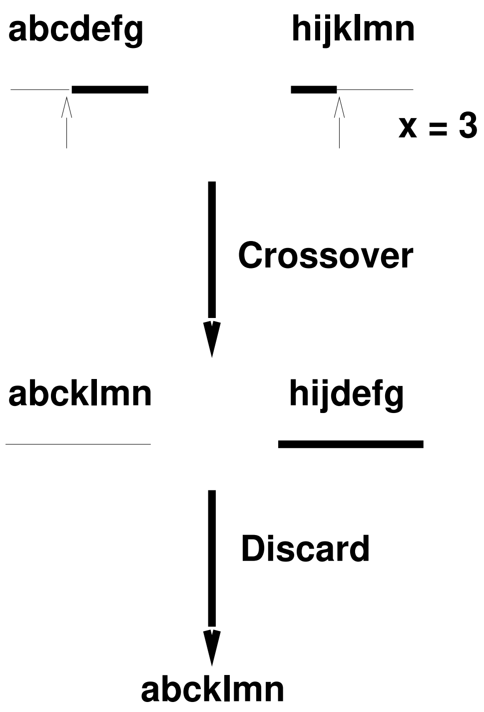

Figure 3.1: The genetic operator *crossover*. Two individuals are cut at a specified point and spliced together.

Mutation Mutation involves randomly mutating one of the alleles of an individual. An individual is selected, a number x in the range $0 \ldots l$ is chosen at random, and the corresponding allele is changed in some manner. The process is shown in Figure 3.2.

Figure 3.2: Mutation causes the random change of one or more of the alleles in the individual.

The original work of Fogel used mutation alone to generate new individuals for the adaptive plan [18]. Using mutation allows one to effect an unordered enumerative search of the space: if mutation leads to a fitter individual, then that individual replaces the original in the population.

Inversion Inversion is used rarely in Evolutionary Computation. Inversion works on one individual: two random numbers x_1 and x_2 in the range $1 \ldots l$ are selected and list of alleles between them are inverted, as shown in Figure 3.3.

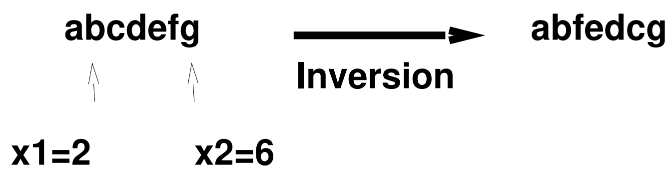

Figure 3.3: The inversion operator, in which sub-lists of alleles are reversed.

Reproduction Reproduction is easily misunderstood to mean sexual reproduction, involving one or two parents. In Holland's terms reproduction should be considered in the sense of Xerox reproduction: a new copy of the individual is placed in the next generation. Reproduction allows individuals of good fitness to remain in the population until their progeny succeed their fitness.

3.2.1 Holland's Schema Theorem

As part of his investigation into adaptive search, Holland developed his *Schema Theorem*, explaining how an adaptive search using the generalised genetic operators on fixed-length chromosomes would eventually converge on highly fit individuals. The approach is based on the notion of a schema: a pattern of alleles associated with high fitness. A schema can be thought of a chromosome with alleles fixed in certain positions and "wild-cards" in the remaining positions. For example, the schema $ab=====$ describes all the chromosomes with alleles a and b in the first two positions. The "=" symbol is wild-card, used to signify any valid entry at that position. The crossover and mutation operators then work to preserve those schema (patterns) responsible for high fitness, while introducing new schema to broaden the search.

The crossover operator is beneficial to the adaptive process for two reasons: firstly, crossover generates new instances of schemata already in the pool of individuals; and secondly, crossover introduces new schemata into the pool to be tried for fitness. For example, the crossover in Figure 3.1 created a new instance of the schema $ab=====$ and $=====mn$, thus preserving the two schema $ab=====$ and $=====mn$. Further, it introduced a new instance of the schema $==ck===$, a pattern not previously in the population.

Mutation does not preserve schemata in any way, in fact, it destroys some potentially beneficial alleles. Why then should it be used? Holland declares that mutation should be considered as a background process, allowing new schemata to be introduced into the population to reduce the convergence on local maxima [27].

The schema theorem goes some way to explaining how the evolutionary algorithm works on fixed-length strings. However, its validity is questionable in many variants of the evolutionary algorithm, particularly those variants where the underlying chromosome representation is variable in length. The next section describes how this evolutionary algorithm has been developed into a number of distinct areas of inquiry.

3.3 Evolution *in Silico*

Much work on evolutionary computation has emerged since Holland's work in 1975. Evolutionary Computation has become a catch-all term embracing all aspects of evolution as adaptive search, including such fields as genetic algorithms, genetic programming, evolutionary strategies, evolutionary programming and evolutionary robotics. At the core of each of these fields is the use of evolutionary principles in searching a hyperspace of solutions. The fields differ in their encoding of structures at the genotype level and the operators used to drive the evolutionary process.

3.3.1 Genetic Algorithms

In genetic algorithms, individuals of a population can be encoded as fixed or variable length bit strings, to which operators such as crossover, reproduction and mutation are applied, as shown in Figure 3.4. For variable length encodings, other operators can be introduced to allow the strings to increase their size. Crossover is performed in a straightforward manner: two individuals are selected on the basis of their fitness and split at a random point. A new individual is created by splicing together alternate parts from each of the two parents. Mutation is performed by randomly flipping a bit in the individual. Phenotypes are constructed from the bit-string genotypes in a consistent manner and these are then tested for fitness.

Fixed length encodings are preferred, as for such encodings the search space, though large, is limited, and the schema theorem (as described by Goldberg in [21]) give a mathematical foundation of this approach. Harvey, in his SAGA work [24], proposes that the use of variable length encodings allows for the structures evolved

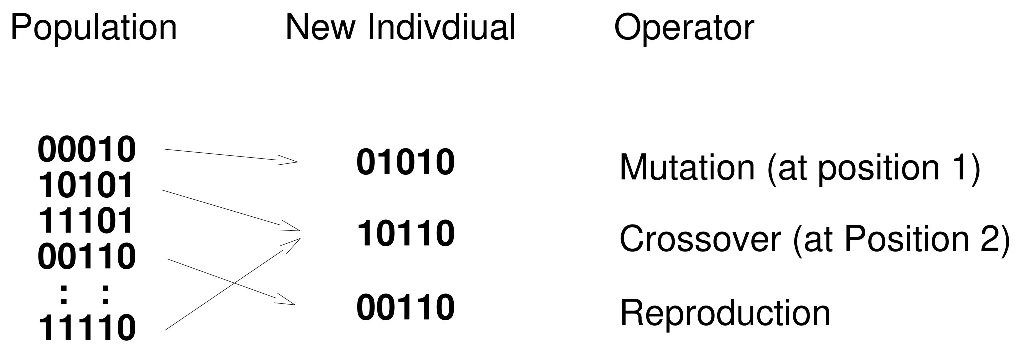

Figure 3.4: Genetic algorithms treat individuals as fixed length strings of 1's and 0's. Operators used are crossover, reproduction and mutation.

to be truly unlimited, but acknowledges that there is (as yet) no equivalent schema theorem for such structures. However, he observed that with his variable-length encoding, length increase happened rarely, and when it did, it happened consistently right across the population. So, as the population made evolutionary leaps in complexity (where we consider length of genotype as a rough measure of complexity), the intermediate periods where there was no change in length can be thought of as obeying the schema theorem.

The question of whether fixed or variable length encodings should be used depends very much on the problem at hand: if the solution belongs into a clearly identifiable limited search space, then the space should be encoded as a fixed length string. If however the search is intended to be open-ended, then variable length encodings should be used.

3.3.2 Genetic Programming

Genetic programming involves the evolution of tree structures that represent computer programs, and was introduced by John Koza [40]. Programs are in the form of S-expressions in the LISP programming language and so lend themselves naturally to a tree representation. A population of programs is created at random, each of which is then used to solve a certain problem. This is done by specifying a *function-set* and *terminal-set*: the trees are then generated randomly using elements of the function-set as nodes and elements of the terminal-set as leaves. Programs are then allocated fitness according to how well they solve the problem at hand.

As with genetic algorithms, crossover, reproduction and mutation are applied; however, mutation is used rarely and often not at all. Crossover and mutation are re-interpreted in genetic programming to resemble pruning of program's sub-trees. The crossover technique is illustrated in Figure 3.5. Mutation is performed by randomly picking a sub-tree in a single individual and replacing it with a randomly generated sub-tree.

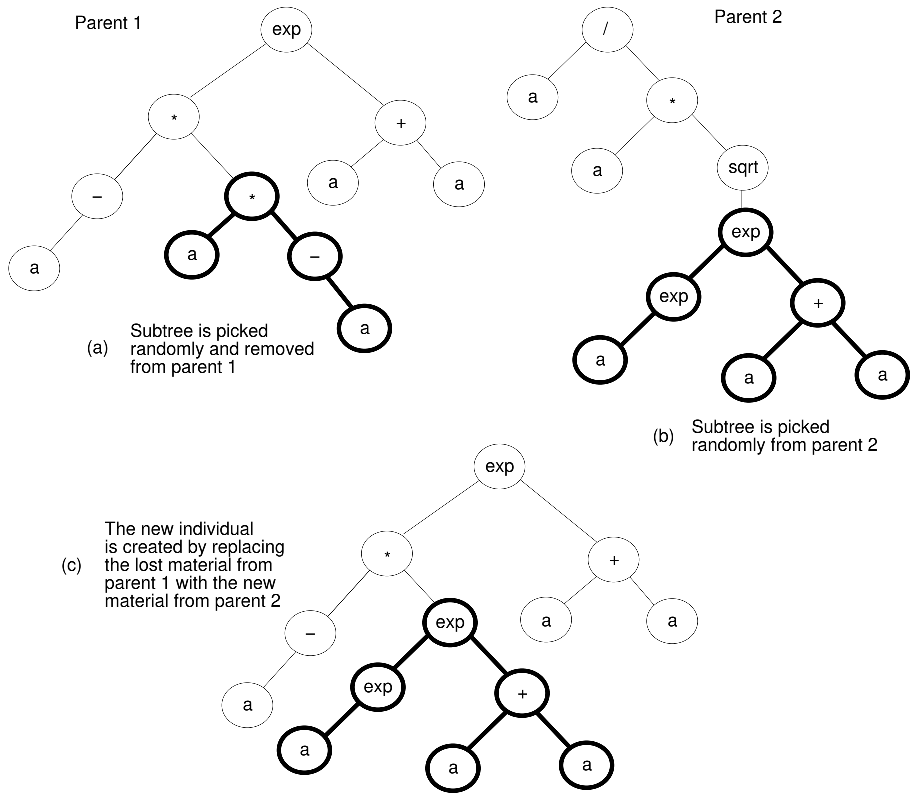

Figure 3.5: Crossover in genetic programming: sub-trees of two individuals are swapped, and one of the resulting trees is used as the new individual.

Schemes to integrate recursion and iteration in this approach are described in

Teller [69]. The use of data-structures with genetic programming has received much attention [68, 41, 6, 9, 1]. This is highly relevant for the evolution of agents with internal state, and so we return to this work later in Section 4.3. The evolution of modular programs, that is, with evolving functions and procedures used by an evolved main-line, has also been investigated in the form of automatically defined functions (ADFs).

As with variable length encodings, genetic programming also lacks a formal schema theorem explaining how it works. A good overview of the schema debate for genetic programming is presented by Poli and Langdon in [57], where they propose a schema theorem based on a restricted version of the crossover operator. The key argument of any schema theorem for genetic programming will centre on the ability of the operators to preserve building blocks, that is, segments of code that positively contribute to fitness. It had originally been thought that crossover preserved such building-blocks, and this is why it was used extensively and at the cost of mutation. However, the validity of crossover as a building-block preserver is questioned by Angeline [2] who showed that crossover between a fit individual and randomly generated individual (as opposed to individuals selected from the population according to high fitness) gave the same results if not better in comparison to conventional crossover. The results prompted him to view crossover as a form of large-scale *macro*-mutation. A large-scale empirical analysis of the performance of crossover over mutation by Luke and Spector in [48, 49] revealed that there is no statistically significant benefit in using crossover over mutation. They remark that their results show the effectiveness of the different operators to be highly dependent on the problem being solved, revealing such general statements as "crossover is better than mutation" to be unfounded.

An alternative encoding for genetic programming, used by Banzhaf in [4] involves the direct manipulation of machine-code instructions, where the program is encoded as a single array of bytes treated as a function in memory. This direct manipulation of machine code rather than high-level interpreted LISP programs is reported to give an evaluation speedup of a factor of 2000.

While there is clearly much to be learnt about the dynamics of genetic programming, there is much empirical evidence that shows its ability to find novel programming solutions in large program-spaces. Genetic programming has yielded an area of intense research in recent years and has been applied to emergent group behaviour, planning, mini-max strategies for games, optimal control, machine learning of functions, random number generation, symbolic regression and the design and training of neural networks.

3.3.3 Evolutionary Strategies

In the field of evolutionary strategies, chromosomes are fixed length vectors of real numbers. Applications for evolutionary strategies are wide, including network & routing problems, biochemistry, optics, engineering design and magnetics. One reason for their popularity is that the structures evolved are very intuitive. For example, a function to optimise the performance of a propeller (the phenotype) might evolve a sequence of real-valued numbers describing such parameters as blade-angle, diameter, and thickness.

Original work with evolutionary strategies used the following simple algorithm:

1. The problem is defined as an optimisation problem: there is a cost function $f : R^n \rightarrow R$, which given an n-dimensional vector returns a single real number, the fitness. The goal is to find an n-dimensional vector for which the the fitness is maximised.

2. A population of p vectors, labelled x_p is randomly created.

3. An offspring vector x'_p is created from each vector x_p, by adding a Gaussian

error variable with zero mean and a pre-selected standard deviation to each component of x_p.

4. Fitness of parents and offspring is evaluated by finding $f(x_p)$ and $f(x'_p)$.

5. A new population is created using the fittest individuals, and the process continues until a good solution is found.

Although early work in evolutionary strategies used only mutation; crossover is also possible with such an encoding. Crossover occurs with *discrete*, *intermediate*, or *random intermediate* recombination. Discrete recombination involves the use of a randomly determined bit-mask with an equal number of 1's and 0's, each corresponding to one of the real numbers in the individual's genotype. This bit-mask then determines from which parent each of the child's real-number alleles will come from. Intermediate recombination takes the average of corresponding alleles in the parents to produce an offspring. Finally, random intermediate recombination uses a probability distribution to calculate the effect each parent has on contributing to the child's alleles.

The discovery of optimal real-value parameters is not limited to the field of evolutionary strategies: Price and Storn have shown how real values can be encoded as short binary sequences, where the small number of bits is mapped to a progressively more accurate real-valued range, a technique known as differential parameter encoding [58].

3.3.4 Evolutionary Programming

Fogel's evolutionary programming technique was one of the earliest experiments in computational evolution, and was motivated towards the evolution of artificial intelligence. The approach is characterised by variable length encodings, which are then interpreted and developed into a program. Evolutionary programming differs from genetic programming in two respects: first, the representation is not tree-based (as in the genetic programming school), and second, new individuals are created typically using mutation alone. In Fogel's own words:

> "This search may be viewed as a selective random walk, a hill-climbing procedure in a hyper-space defined to include the finite state machines and an additional co-ordinate on which is measured the average cost of the error score." [18]

With this in mind, Fogel set out to evolve prediction machines, that is, finite state machines that accept a stream of input characters, at each stage predicting what the next symbol will be. Finite state machines are described by the number of states, the input & output alphabet, and a state transition table. The state-transition table is used to determine the next state and output of the finite state machine, given the current state and input. An example machine is shown in Figure 3.6: each node of the diagram represents a state of the machine, while the directed arcs represent transitions between states. The notation x/y implies that on input symbol x, the symbol y will be output and the next state will be that state to which the arc points. The state table for the machine from Figure 3.6 is shown below in Table 3.1.

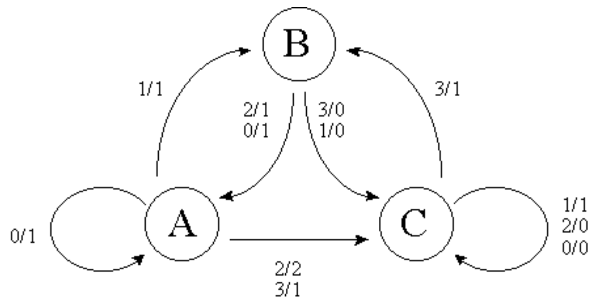

Figure 3.6: An evolved finite state machine.

Present State	A	A	A	A	B	B	B	B	C	C	C	C
Input Symbol	0	1	2	3	0	1	2	3	0	1	2	3
Next State	A	B	C	C	A	C	A	C	C	C	C	B
Output Symbol	1	1	2	1	1	0	1	0	0	1	0	1

Table 3.1: State Table for Finite State Machine from Figure 3.6

Input Stream	2	3	1	0	1	3	3	0	3	0	1
Output Stream	1	2	1	0	1	1	0	0	1	1	

Table 3.2: Example run for Fogel's evolved FSM

Given a state table such as that in Table 3.1, it is easy to select an entry of the table at random to adjust the behaviour of the FSM. This kind of random mutation was used by Fogel to evolve finite state machines. A randomly generated initial machine M_0 was fed with a sequence of input symbols, i.e. an environment. The output symbols were recorded and a measure of fitness was calculated according to the number of the output symbols successfully predicted the next input. For every incorrect symbol prediction, an error-counter was incremented by some amount. The final error was calculated be dividing the sum of all the errors by the number of symbols processed. An example run is shown in Table 3.2: for 10 symbols there are 8 errors and only two correct predictions, thus, weighting each error with an error cost of 1, we get a total error cost of 0.8.

Having evaluated machine M_0's ability to predict its environment, Fogel mutated M_0 to create M_1, a new machine very like its parent M_0. It would be tested against the environment in the same way, and its fitness (or rather, error cost) evaluated. Whichever machine proved to have the least error cost would become the parent in the next stage of evolution.

Fogel's techniques proved successful at creating machines that could adapt to cyclical environments, environments based on first-order Markov chains, and even environments of prime numbers (one machine successfully evolved the power to describe all the prime numbers less than a thousand, bar five).

3.3.5 Evolutionary Robotics

Evolutionary robotics describes the application of evolutionary techniques to the design of robot control and morphology for real-world environments. A review of the field is described by Meyer and Husbands [51]. Controllers are evolved for robots to exhibit autonomous behaviour such as attraction to light, obstacle avoidance, and wall following.

The encodings used are typically binary encodings, which are developed into neural network controllers. The neural network approach is favoured for a number of reasons. First, it is believed that the resulting search space is smooth and thus more suitable to navigation by an evolutionary search. Further, neural networks are viewed as an open-ended representation. Finally, the low level primitives at work in a neural network are so fundamental that they remove the possibility of designer-influence on the network architecture.

The neural networks evolved are unrestricted, dynamic, real-time networks [30]. Conventional neural networks are typically restricted in terms of their symmetric connectivity, feed-forward evaluation, noise-free processing, uniformity of neuron units, and synchronised timing. These restrictions are put in place for ease of design, understanding, and validation of the network. By surrendering the design of the network to an evolutionary process, these restrictions can be relaxed and solution spaces not normally considered by a human designer can be explored. For example, neurons in a dynamical network can change their activation function over time, for example, according to the close proximity of other firing neurons in Husbands' gas diffusion networks [31].

The challenges to the evolutionary robotics community lie in the exploration of new encodings, and the application of these to more complex real-world behaviours beyond mere reactive behaviour. One example of this is Jacobi's T-Maze environment, where a robot must evolve to turn left or right at a T-junction based on a light encountered on a corridor some time before [37]. Only by "remembering" this information using internal state can the robot perform well. An important technical challenge in evolutionary robotics has been how to evaluate fitness: fitness evaluation in the real world takes much time and typically involves a human supervisor to monitor performance. Evaluation in simulations may be faster, but can over-fit the resulting controller to the simulator rather than the real world. That is, the controller learns to perform well in the training environment, at the expense of more robust strategies that perform well in the real world. Rather than go to the extreme of making more accurate simulations, Jacobi's minimal-simulation technique [36], where only the very basic elements of the world are simulated with very high sensor noise, provides an effective and computationally low-cost alternative to this.

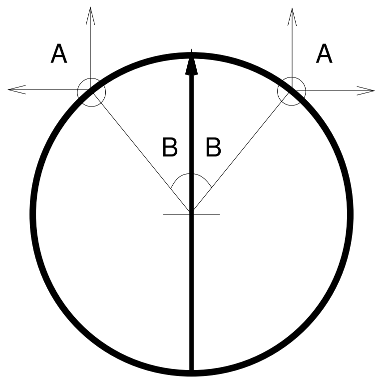

Figure 3.7: The physical structure of the agent, in terms of the angle of view A and angle of eccentricity B of its sensors is evolved alongside the controller

As well as evolving controllers for autonomous robots, evolutionary robotics also allows the evolution of the robot's sensor morphology. This directly addresses the issue of blindness described in Chapter 2, where by fixing the structure of an individual and we implicitly limit its view of the world. For example, Beer and Gallagher in [5] have investigated the evolution of adaptive behaviour using a genetic algorithm that evolves the configuration of the robot's sensors as well as a continuous real-valued network controller. The agent uses two visual sensors, each of which are located at an angle of eccentricity B from the orientation of the agent, and each of which have an angle of view A, as shown in Figure 3.7. These angles are described by real values, and differential parameter encoding is used to present these values as a binary string. This allows the structure of the agent to develop to suit the problem at hand. In a pursuer-evader scenario, predators evolved with eyes close together at the front of the agent, while evaders evolved with eyes further apart to get a wider field of view [12]. The evolution of sensors in *virtual* robotics environments has also been treated by Honovar and Balakrishnan in [74].

3.4 Summary

This section has introduced evolutionary computation in the general context of adaptive strategies, and has shown how the field has diverged using different encodings and adaptive operators. The next chapter presents a review of approaches how evolutionary techniques have been applied to the design of autonomous agents.

Chapter 4

Evolving Behaviour

4.1 Introduction

Evolutionary computation has been applied previously to the evolution of autonomous agents: this section describes some of these applications, the encodings used and the kind of behaviour achieved.

The approaches can be classified according to the kind of behaviour evolved, either Markovian or non-Markovian, and whether the focus is on individual rather than group behaviour. The distinction of environments in Chapter 2 as either Markovian or non-Markovian invites a classification based on whether the agents utilise internal state mechanisms or not. This state can be either explicit, implemented through the use of explicit state-altering operations, or implicit, arising naturally from the representation used.

Section 4.2 describes reactive agents, that is agents without state capable of only Markovian behaviour. Non-Markovian agents exploiting internal state mechanisms are described in Sections 4.3 and 4.4 where the former applies to agents evolved using *implicit* state mechanisms and the latter describes agents evolved using *explicit* state mechanisms. Finally, Section 4.5 describes how evolution has been applied to the design of *groups* of agents who co-operate to achieve group behaviour.

4.2 Reactive Agents

The agents in this section are characterised by their purely reactive behaviour: at a time t their output is a function of the environment variables alone.

4.2.1 Genetic Algorithm Rules

Wilson [79] used a genetic algorithm to evolve rules to control an agent (animat) in a two-dimensional tessellated world filled with food and obstacles. The agents were equipped with 8 sensors sensing the 8 immediate neighbouring cells. The sensors report a length-2 binary string: 11 for food, 01 for obstacle, 00 for open spaces. An agents input at a given time is a string of the form "11 00 01 01 11 01 00 00 11". There are 2^{16} possible input combinations: the aim is to associate with each possible combination an action that moves in such a way as to maximise the amount of food eaten (an agent eats food by moving over a cell with food in it). The eight possible directions of movement are labelled as binary strings 000 to 111, and so the behaviour function is a mapping from a 16-dimensional binary space to a 3-dimensional binary space. 2^{19} distinct behaviour functions exist: Wilson used an evolutionary approach to explore this space. The population contains a list of taxons. A *taxon* is a template for an input pattern, mapping a string of fixed input symbols and don't-care symbols to a move, for example:

$$
\begin{array}{rcl}
1\#\ 00\ 11\ 1\#\ 00\ 01\ \#0\ \#1 & \rightarrow & 100 \\
00\ 01\ 0\#\ 11\ \#1\ 00\ 11\ 00 & \rightarrow & 010 \\
\vdots & \vdots & \vdots \\
01\ 11\ 01\ \#0\ 1\#\ 01\ \#\#\ 00 & \rightarrow & 110
\end{array}
$$

The taxons describe how an entity will behave, giving a behaviour mapping very similar to the transition tables used to describe cellular automata. The animat is allowed roam the world, matching input to the taxons in the population to choose its next move. Fitness is measured by how much food is acquired: if a particular taxon is beneficial in acquiring food, then its fitness is increased. Since not all taxons will yield food, but may still be responsible for just part of a process by which food is discovered, the success of a particular taxon is shared with the taxon or taxons that immediately preceded it. This sharing of fitness is known as the "bucket brigade algorithm". The animat is subjected to genetic adaptation: the population in this case consists of the taxons that make up the individual's control mechanism. Periodically the taxons are sampled according to their fitness and are subject to mutation and recombination.

4.2.2 Evolved Neural Networks

Genetic algorithms have been used to generate network architectures and weights for neural networks. For example, in the work of Todd, Miller and Hegde [52], the networks are encoded as a connectivity matrix. For every element of the matrix, an L indicates that there is a link between the two indexing nodes, and a 0 indicates no link. This connectivity is easily encoded into a binary string where 1 represents an L, 0 represents 0 and each of the rows are placed one after the other, as shown in Figure 4.1.

Network weights need not be binary: the problem can be viewed as a matrix of real-valued connections. Then, the real valued weights can be encoded using a technique such as differential parameter encoding. Other network parameters, such as thresholds and time-delays can be incorporated into the encoding. However, the above direct encoding of the network connectivity matrix does not scale well, leading

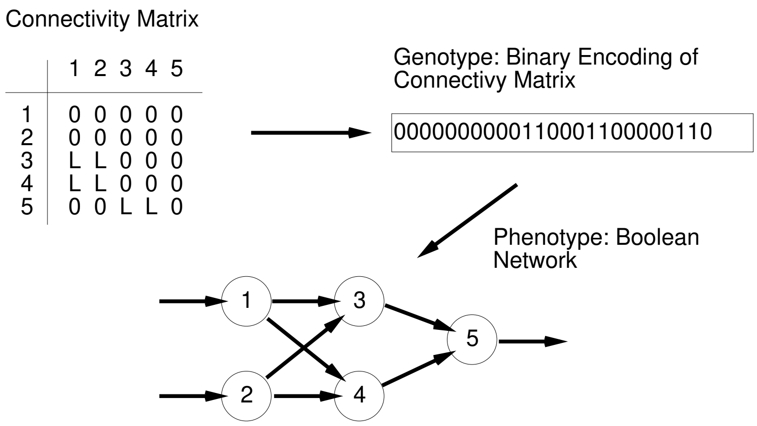

Figure 4.1: Direct binary encoding of a Boolean network.

to a polynomial increase in the size of the chromosome with respect to the number of neurons in the network. Also, genetic operators such as crossover and mutation will have large impact on the resulting phenotypes that may not lead to a smooth search of the problem space.

Cellular encoding uses the genetic programming representation to evolve programs trees with commands that construct a feed-forward neural network. Gruau describes this as *grammatical* rather than direct encoding [23]. The genotype is the genetic program, which is developed into the phenotype, the network. The development process is quite complex, but is shown by Gruau to be complete, compact, expressive and modular. The weights of the resulting network are bipolar (either 1 or -1), and these weights are the fixed for the life of the network. The technique has been applied to the parity problem for inputs of up to 21 units, and for decoders with 7 inputs. Cellular encoding has an advantage over direct encoding in that small genotypes can yield large phenotypes, thus overcoming the restrictions of direct encoding techniques. Through the use of automatically defined functions, modular structures can be repeated throughout the network. By evolving recursive trees (with leaves pointing to the root) recursive development of network structure can be achieved [22]. A further extension of Gruau's approach is *developmental* *learning* where the development of trees from genotype to phenotype is affected by the environment, the resulting network being a process of both ontogenetic and phyllogenetic learning.

4.2.3 Genetic Programming

Stateless agents have been evolved with genetic programming techniques. The agent's control function is evolved through a form of symbolic regression, as demonstrated by Koza's cart-centring example [40]. The inputs to the behaviour function are sensor signals from the environment, and the output of the function is wrapped to give the required behaviour. Reynolds employs a similar approach to the evolution of corridor-following behaviour for a robot in a world with noisy sensor readings [63]. In earlier work he had attempted to let the robot evolve roving sensors, however he found that genetic programming performed better when the direction of the sensors was fixed *a-priori*. Dracopoulos used genetic programming to evolve a stable control law for the non-linear attitude control problem for a satellite [15]. Howley evolved control systems to achieve near minimum-time manoeuvres for a space-craft in [29].

4.3 Agents with Implicit State

4.3.1 Genetic Programming

Implicit state is created in genetic programming trees when the functions and terminals produce actions. As the tree is evaluated, a sequence of actions will be performed, and so the behaviour of the agent is not based on environment alone, but also on the execution state of the program, that is, the position of the program counter. This differs from reactive genetic programming agents, who produce an action only when a tree has been evaluated in its entirety.

Koza's evolved solutions to the Santa-Fe trail demonstrate the sequencing of actions. Incorporated into the function set as a mechanisms for explicit action-sequencing is the `progn` command, which takes n arguments and executes each of its arguments in turn. Using a combination of `progn` and conditional functions, along with terminals such as `move`, `left` and `right`, agents can evolve strategies to find all of the food in a two-dimensional tessellated world. The use of embedded action in program trees is prolific: Moore in [55, 54] uses embedded action statements to evolve missile avoidance controllers for aircraft in both the extended two and three-dimensional pursuer / evader problem with counter measures. Deakin in [14] evolved agents for a variant of the pursuer-evader problem, where an evolved agent (the intruder) had to bypass a human designed agent (the guard). Agents were evolved by Dwormen *et al.* in [16] in a co-evolutionary set-up to develop bargaining strategies. The agents had to co-operate to form a coalition and profit-sharing scheme so as to maximise individual profit.

4.3.2 Evolved Dynamic Neural Networks

As described in Section 3.3.5, evolutionary robotics exploits the characteristics of dynamic neural networks, characterised by feed-back loops, lack of synchronisation, non-uniform neuron units, and noisy processing. Although the weights and structure of such a network may remain fixed, values in the neuron units of the internal layers constitute an implicit internal state.

A typical network encoding (as used by Cliff in [11]) is shown in Figure 4.2. The encoding scheme is variable length. The string is divided into nodes, each with a well-defined marker. Marker X denotes an input node, Y an intermediate (inner-layer) node, and Z denotes an output node. Each node then contains threshold information for the associated network unit and linkage information. Each link can be forward or backward, and addresses another node in the individual with either relative or absolute addressing. From this variable-length genotype a dynamic neural

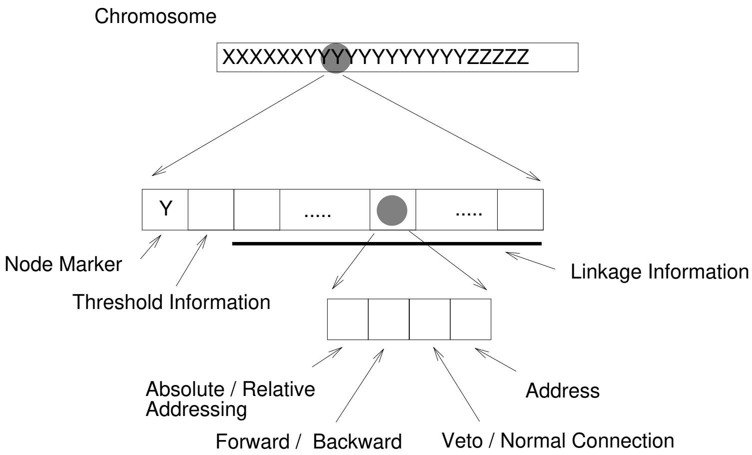

Figure 4.2: Encoding scheme for a dynamic neural network.

network is constructed. Appropriate adaptive operators are used in the evolutionary process. Although this encoding has given good results, the authors have their own criticisms. In particular, this encoding cannot easily capture repeated modular structures or symmetries, as that of Gruau (above).

4.4 Agents with Explicit State

So far, we have seen that state arises in genetic programs in an implicit manner, though the use of action functions embedded in the functions and terminals of the program trees. An alternative approach is to use explicit state mechanisms, such as scalar variables, linear indexed memory, multi-dimensional indexed memory and associative memory (Figure 4.3). Then, by incorporating appropriate `read` and `write` functions into the function set, programs can be evolved to use memory to effect internal state. Teller examined the use of indexed memory to develop internal state for agents solving a block-moving problem in [68], and discusses the theoretical ramifications of this extension in [69].

The use of indexed memory for internal state was also studied by Paterson in his work on evolving caching algorithms [56]. He found that allowing the caching algorithms access to indexed memory greatly sped up convergence of the genetic algorithm, but greatly slowed down the evolutionary process without leading to any improvement in the evolved algorithms. For his caching problems, the algorithms with access to indexed memory performed poorer than those without access to indexed memory, when tested over a range of cache sizes and different trace-files.

Andre uses a two-dimensional indexed memory isomorphic to the agent's world to evolve map-making capabilities in [1]. Andre's agent works in two phases - in the first, the agent can roam the world and record the locations of gold. Then, in the second phase, the agent is blindfolded and must use the internal map to locate and dig for gold. Since high fitness in Andre's and Teller's world can only be achieved with non-Markovian behaviour, the efficacy of explicit state mechanisms is demonstrated. Brave in [6] uses the same gold-digging scenario to show the effective use of directed graphs as associated memory, where the number of nodes is predefined but the connections between the nodes are made by the evolved agent. In [34], Bennett evolved an agent to exhibit four distinctively different behaviours (lawn mowing, gold-digging, wall-following and mine-sweeping) according to the colour of the room it was in, using only two scalar variables for internal state.

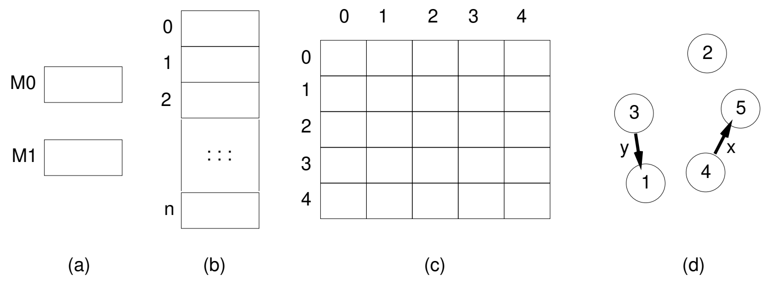

Figure 4.3: Different forms of explicit memory: (a) scalar variables, (b) indexed memory, (c) multi-dimensional indexed memory, and (d) directed graphs

Brave [6] evolved object-oriented programs as a number of genetically programmed trees using a single indexed memory to provide the state for the object. Langdon in [42] showed that genetic programs could successfully evolve data structures to implement a queue and a stack, and then showed in [41] that genetic programs could successfully use such high-level structures to recognise strings of balanced matching parentheses and evaluate reverse-polish-notation arithmetic expressions.

4.5 Multi-Agent Systems

Evolutionary approaches have been applied to the evolution of multiagent systems for both group behaviour *and* individual behaviour. In the former, a number of agents are evolved to co-operate to solve a particular problem, while in the latter a number of processes are evolved to co-ordinate the action of a single agent.

Koza evolved robot controllers for wall-following based on Brook's subsumption architecture in [39]. The leaves of the program tree corresponded to behaviours, and the nodes were conditional statements. The hierarchical network of conditional statements thus acted to select the appropriate behaviour given the environmental conditions. A similar multiagent approach was adopted by Bennett in [33], where the agent was evolved to solve a food-foraging problem.

Banzhaf evolved populations of machine-code genetic programs for real robots in [4]. The robot was controlled by each program in the population in turn, with fitness awarded to the those individuals who positively contributed to fitness. The resulting behaviour is thus the emergent effect of the entire population's co-operation. A distributed controller for a starfish robot is presented by Hondo *et al.* in [28]. Starfish have no central brain, instead having a distributed nervous system spread out amongst their five legs. A starfish robot is thus a natural candidate for a distributed controller. They evolved a genetic program to control each leg of the robot, and realised successful attraction to a stationary light source. However, they were unable to evolve attraction to a moving light source. They achieved concurrency through the use of a multithreaded programming technique, similar to that developed independently and presented in Chapter 5 of this work.

The evolution of event-driven programs has been studied by Ryan [64] and Crosbie [13]. In the event-driven paradigm, functions are assigned to handle particular events and are executed as the events occur. The agent thus senses the environment through interrupts (the arrival of new events) rather than by polling sensor values. In Ryan's work, populations of "call-back" functions were evolved along with a population of "main-programs" that assigned members of the call-back populations to particular events. The resulting event-driven agents acted in a competitive environment called "GPRobots", where agents had to shoot other agents and avoid getting shot at. Concise event-driven programs were evolved displaying a number of behaviours from "Fire at other agent and run away" to "Spin on the spot and keep firing." Crosbie [13] proposes an alternative approach to the evolution of event-driven programs based on coloured Petri nets. Here, events travel through a directed graph as *tokens*: each transition in the graph contains a guard, determining which tokens can travel along the transition. When an token enters a node, an action function in the node is executed. Crosbie proposes then to evolve the transitions, guards and action functions for a Petri net, towards the evolution intruder detection systems for computer security.

The evolution of group behaviour, where teams of agents are evolved to solve a task collectively, has also been treated by genetic programming. A notable success was Luke's soccer team evolved for RoboCup'97, where an evolved team of soccer players beat two teams with human-designed player strategies [45]. General issues relating to the evolution of teams, such as the use of homogenous or heterogeneous agents and different breeding strategies are investigated by Luke in [47]. Agents were evolved in a predator-prey scenario, where a gazelle was stalked by four lions that had to co-ordinate to trap the prey. Different co-ordination schemes were assessed and it was found that name-based sensing, where a lion had access to the position of each other lion by name, worked better than simple proximity sensing. Agent co-ordination has also been investigated by Iba in [32], who evolved teams of agents that co-operate to reach their destinations through narrow "one-robot-at-a-time" passages.

Inter-agent communication is examined by Qureshi in [60], where agents are evolved to rendezvous at a particular position. The communication is achieved through dedicated channels through which an agent can send and receive integer data from the other agent. The agents thus evolve to decide what to communicate and when to communicate it.

Multi-agent systems with "group-knowledge" have been investigated by Haynes in [25], applied to the clique detection problem. Useful knowledge is extracted from "search" agents and stored in data-repository, where it is acted upon by "process" agents. Luke and Spector in [46] have extended genetic programming to include the concept of culture, a persistent global indexed memory unit that agents can write to and read from. This cultural transmission of information between agents and across generations was shown to offer some improvement in the evolution of agents for symbolic-regression, lawn-mowing, and an agent problem involving gold-digging and pursuer avoidance.

4.6 Summary

This chapter has presented a review of approaches to the evolution of autonomous agents in Markovian and non-Markovian environments towards both individual and group behaviour. The next chapter introduces *concurrent genetic programming*, a new encoding for evolutionary computation that allows the evolution of concurrent algorithms and representation yielding emergent autonomous behaviour.

Chapter 5

Concurrent Genetic Programming

5.1 Introduction

This chapter introduces one of the central contributions of this thesis, "concurrent genetic programming", an extension of the genetic programming paradigm. Concurrent genetic programming is a natural application of evolutionary computing to the principles of emergent behaviour advocated by the field of artificial life, described in Section 5.2. In concurrent genetic programming an agent consists of a number of genetically evolved programs that are executed concurrently to effect an emergent behaviour. The programs can develop distributed internal state through the use of explicit state mechanisms. This flat distributed agent architecture is described in Section 5.3. The genotype encoding, operators and other details of concurrent genetic programming are presented in Section 5.4. Two different implementations of concurrent genetic programming are presented, a non-determinstic implementation built on the Java multi-threading system, and a fully-deterministic implementation based on a round-robin scheduler. These are described in Section 5.5. Finally, Section 5.6 outlines the issues created by this concurrent approach.

5.2 Artificial Life

The field of artificial life views intelligent behaviour as an emergent phenomenon, and sets out to examine the way that complex organisations give rise to life-like behaviour. Langton gives the following description of the essence of artificial life:

> "Artificial life involves the *realisation* of lifelike behaviour on the part of man-made systems consisting of *populations* of semiautonomous entities whose local *interactions* with one another are governed by a set of *simple rules.* Such systems contain no rules for the behaviour of the population at the global level, and the often complex high-level dynamics and structures observed are *emergent* properties, which develop over time from out of all of the local interactions among low-level primitives by a process highly reminiscent of *embryological development*, in which *local hierarchies* of higher-order structures develop and compete with one another for support among the low-level entities. These emergent structures play a vital role in organising the behaviour of the lowest-level entities by establishing the context within which those entities invoke their local rules and, as a consequence, these structures may *evolve* in time." [44, page XXII] (original emphasis)

Artificial life promotes a bottom-up approach to the development of autonomous behaviour. Complex behaviour is achieved through the interaction of many simple agents, instead of through the action of one sophisticated agent. The challenge in artificial life is to find the "simple rules" that control the agents and lead to sophisticated emergent behaviour. This is in contrast to Marr's approach, which begins with a computational theory and then leads to the development of algorithms and representations that implement this theory. The emphasis in artificial life is on the mechanisms that can create an emergent algorithm and representation for life-

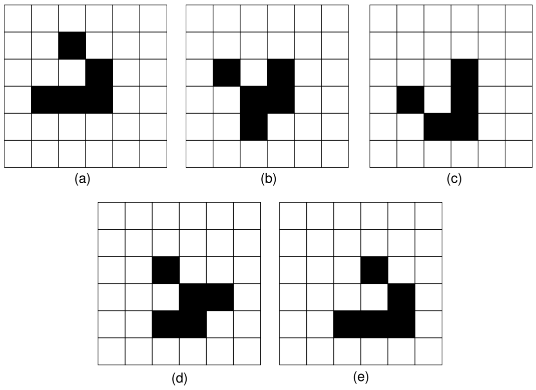

Figure 5.1: Emergence of glider configuration in Conway's Game of Life cellular automata.

like behaviour.

5.2.1 Emergence of behaviour and structure

The emergence of structure and behaviour through the interaction of simple rules is demonstrated in Conway's "Game of Life" [20]. Cells on a two-dimensional lattice update their state (either "alive" or "dead") depending on the number of alive and dead cells in their immediate neighbourhood. Using this simple update rule, spatiotemporal patterns such as the glider in Figure 5.1 emerge whose structure and behaviour is not coded explicitly in the update rule. More complex self-reproducing patterns can also be encoded in cellular automata, and Von Neumann showed such systems to be universal computing machines in the Turing sense [78].

Cellular automata show how complex organisations can emerge from the local interaction of simple rules. Varela [75] takes this idea one step further to simulate how cognitive living systems can emerge in the real world. An autonomous system emerges within a physical system through autopoiesis. Autopoiesis stems from two Greek words, $\alpha\upsilon\tau o\varsigma$ and $\pi o\iota\varepsilon\iota\upsilon$, meaning "self" and "to produce" respectively. An "autopoietic system" is defined as:

> "An autopoietic system is organised (defined as a unity) as a network of process of production (transformation and destruction) of components that produces the components that: (1) through their interactions and transformations continuously regenerate and realise the network of processes (relations) that produced them; and (2) constitute it (the machine) as a concrete unity in the space in which they exist by specifying the topological domain of its realisation as such a network." [75] (original parentheses)

Varela proposes that an autonomous system defines itself through the autopoietic processes of self-organisation and self-maintenance. The maintenance of the system's identity within the larger system (its environment) leads to activity, that is, behaviour. This is demonstrated in the tessellation example of autopoiesis [75, 84], where simple molecules combine to form a closed chain that is responsive to its changing environment and robust in the face of failure of link elements. This higher-level behaviour is encoded implicitly through simple rules, for example, if a piece of substrate is touched by two link elements that are linked to only one other element on the chain, and it has another piece of substrate adjacent to it, then form a new link element from the two substrates and link this new element to the open links.

In [76], Varela discusses how such autonomous systems can create the notion of a "selfless-self": the system consists of many components, all of which come together without any global supervision, to behave as a single unity. Such systems are "operationally closed", that is, they do not take input or output from the environment in the way an information processing system does. Instead "input" corresponds to change in the medium of which the agent is made up. The structure of the entity must be resilient to change if it is to maintain its unity, and so the structure is modified as the network of autopoietic processes struggle to maintain coherence. The system as a whole alters its structure and so effects behaviour: an observer may interpret this activity as meaningful intelligent behaviour. Varela calls this ascription a "surplus of significance", citing the example of a bacterium "swimming" up a sucrose gradient. The behaviour can be thought of as "intent" or "desire", but the underlying truth is that autopoietic mechanisms lead to an autonomous system that is perceived as a single behavioural entity.

Minsky develops the idea of emergent behaviour with his *Society of Mind* [53], where he views intelligence as the emergent effect of many simple computational processes, which he calls "mind-agents." As with artificial life, he posits a trade-off, where instead of a having a single sophisticated agent, many simple agents exhibit complex behaviour through the emergent effect of their interactions. An overview of multiagent systems and distributed artificial intelligence research can be found in [66].

5.2.2 Behaviour as a Multilevel Phenomenon

Langton proposes that behaviour is a multilevel phenomenon. He says "the most salient characteristic of living systems, from the behaviour generation point of view, is the *genotype/phenotype* distinction" [43] (original emphasis). This distinction separates the machinery (the genotype) from the behaviour (the phenotype). In nature, the genotype is a complete set of instructions that make up an organism's DNA. Through the developmental process of morphogenesis, the genotype becomes the phenotype, that is, the organism itself. The phenotype is a multilevel phenomenon, as shown in Figure 5.2. Through a developmental process, the genotype produces a level one phenotype, an "individual": this individual is an inanimate structure. When the behaviour of the individual in its environment is observed, this behaviour constitutes the level two phenotype. Finally, when the behaviour of many individuals interacting within the environment is observed, the emergent group behaviour gives a level three phenotype.

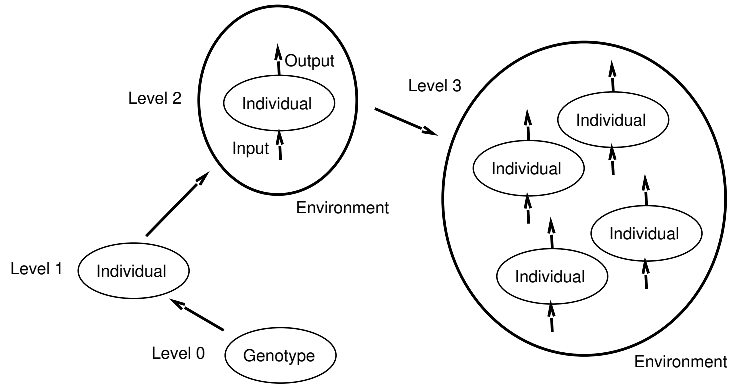

Figure 5.2: The phenotype as a multilevel phenomenon.

Different approaches in artificial life and artificial intelligence attempt to create behaviour at these different levels. For example, Lindenmayer *et al* develop the individual as structure (the level one phenotype) in their *graftal* work, where models of plants are produced through the application of grammatical rules to strings [59]. Investigation of autonomous agents concentrates on the behaviour of the individual with respect to its environment, and so treats the level two phenotype. Group behaviour, such as Reynolds's flocking "boids" simulations, are concerned with the level three phenotype [62]. Some group behaviour approaches (such as Varela's autonomous systems or Minsky's *Society of Mind* agents) aim to produce level 2 behaviour through the mechanisms applied to level three, that is, the individual is defined or controlled through an emergent group behaviour.

The message of artificial life is clear: behaviour can emerge through the interaction of simple agents. However, artificial life does not offer a computational theory of intelligence, nor does it offer a computational model beyond that of the Turing machine. What has the approach to offer artificial intelligence? First, by concentrating on the mechanisms that lead to behaviour, it allows us to design agents in a bottom-up manner through the identification of simple rules, rather than by attempting an analytical approach to determine a complex algorithm and representation. Second, the artificial life approach is inherently parallel and distributed, and so can offer speed (through the execution of simple rules over many agents in parallel) and robustness (where the failure of one agent in the multiagent system may allow a graceful degradation of performance rather than a complete failure).

5.3 Emergent Behaviour Architecture

Chapter 2 showed that agents use internal state to perform in non-Markov environments, but that the most appropriate representation and algorithm is not necessarily that of a human observer-designer. Section 5.2 described how behaviour can arise as an emergent phenomenon, through the interaction of many simple processes. Suitable behaviour can be found using an adaptive machine-learning technique such as evolutionary computation to find the correct rules to effect the desired behaviour.

An agent architecture for emergent behaviour is shown in Figure 5.3. The agent is controlled by a number of parallel-executed computational processes with access to some internal state. The computational processes within the agent receive information from the environment through sensors. Each process can control the agent by sending messages to the effectors: the overall behaviour of the agent arises through the interaction of these computational processes.

Referring to the different phenotype levels introduced in Section 5.2.2, the observed behaviour is level-two, that is, the aim is to create an agent that performs intelligently in its environment. However, the mechanisms used to achieve this behaviour are those of the level-three phenotype, that is, group behaviour. To evolve agents for this distributed architecture, the composition of the computational processes and the internal state must be specified. The "computational processes" are programs, and so the genetic programming paradigm is extended to meet this architecture.

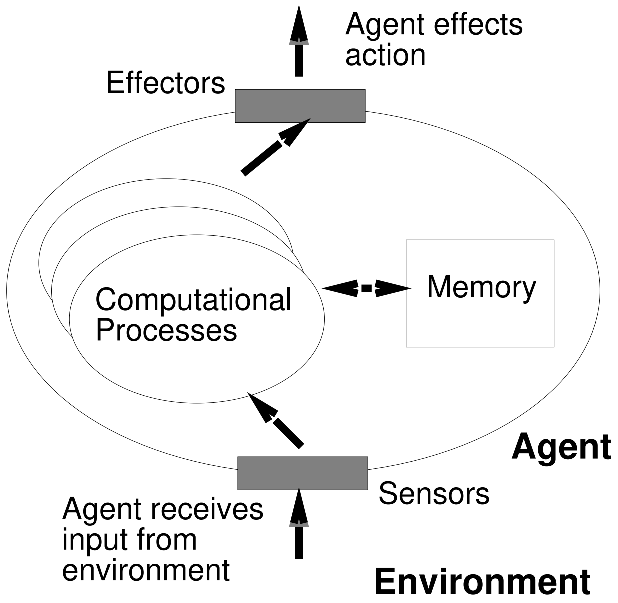

Figure 5.3: A distributed architecture for agents: the agent's behaviour is the emergent effect of a number of computational processes competing and co-operating to control the agent.

5.4 Concurrent Genetic Programming

The essence of concurrent genetic programming is the concurrent evaluation of a number of genetic programs to produce an emergent collective behaviour. As such, it borrows much from standard genetic-programming. Key issues described here are the genotype encoding, evolutionary operators, state mechanisms, and control of concurrency.

5.4.1 Genotype Encoding

The genotype encoding employed in concurrent genetic programming consists of a list of genetic programming trees: one tree for each of the computational processes that control the agent. This multitree representation is a natural choice for the evolution of agents controlled by many processes, and neatly realises the agent architecture of Figure 5.3. Sensor input is passed to the programs through sensor terminals in the terminal set. When a sensor terminal is evaluated, the current value of the corresponding sensor is returned. Each of the genetic-programs performs calculations using the sensor-input and information stored in memory to produce an output, as a numeric value. When a program produces a result, it "fires". The numeric result is then interpreted to decide what action to take using a "wrapping" function. For example, a wrapper that interprets numbers less than zero as "move left" and numbers greater than or equal to zero as "move right" might be employed.

The random creation of concurrent genetic programs proceeds much in the same way as the random creation of conventional genetic programs. Each tree has associated function and terminal sets that specify the computational building blocks that make up the tree. An individual is created by specifying an initial depth and a growth type (either FULL or GROW) and each tree in the individual is grown with this initial depth and growth type. The ramp-half-and-half method is used to create the initial population, where trees are created alternatively using the FULL and GROW growth method, and the maximum depth for initial trees varies in the range 1 to some maximum depth.

5.4.2 Operators

Three genetic operators: crossover, reproduction, and mutation are employed in the evolutionary process. Genetic operators for multitree individuals have been studied before by Haynes [26], where a single multitree individual contained the GP trees to control a number of separate agents working on a predator-prey problem. He identifies a number of different strategies for crossover and mutation of the constituent trees of his multiprogram individuals. Bruce [10, 9] uses a scheme where one of the trees in an individual is selected at random and crossed with the corresponding tree in the other parent. Here, crossover is performed by effecting the standard genetic programming crossover operator on each corresponding tree in the parents. This approach is similar to Haynes' *TeamAll* method. Mutation is applied by selecting one tree in an individual at random and applying conventional genetic programming crossover to this tree. Reproduction creates a deep copy of an individual and places it in the next generation.

5.4.3 Concurrency

Concurrent evaluation is achieved through an underlying time-sharing mechanism that allocates processor cycles fairly to each program. Two different time-sharing implementations are described in Section 5.5. To allow programs to evolve their own scheduling and time-sharing strategies over the underlying sharing mechanism, a `Yield` function is introduced. On evaluation, `Yield` informs the scheduler that the program is prepared to give up whatever cycles it has left. The program enters a suspended state, and on resumption the single argument to the `Yield` function is evaluated and returned. The purpose of `Yield` is to provide the representation with a mechanism by which novel problem-solving scheduling strategies can be evolved.

5.4.4 State

State is incorporated by adding explicit state mechanisms to the function sets of each tree. In the experiments to come in Chapter 6, indexed memory is employed by adding the `Read` and `Write` functions to the function set of each of the trees. Programs can evolve to use memory as they are evaluated. Other kinds of memory mechanisms such as stacks or queues could also be used by placing appropriate functions and terminals in the function and terminal sets.

5.5 Implementation

Ideally, the computational processes controlling the agent would be executed in parallel, however this is not possible on a single-processor machine. Parallel execution can be mimicked by time-slicing, where the processor and resources are shared among each of the agent's processes. Two different implementations of concurrent genetic programming are described.

5.5.1 Non-deterministic Implementation

In initial investigations, reported in [70], concurrency was achieved by evaluating each tree with a Java thread, leaving scheduling issues to the Java runtime interpreter. The `Yield` function maps directly to the Java `Thread.yield()` method, which informs the interpreter that the currently executing thread may be suspended and another thread allowed to execute. Then, when the thread is resumed, its argument is evaluated. By adding `Yield` to the terminal set, CGP agents can evolve their own scheduling schemes. Alternatively, the `Yield` mechanism can be used implicitly to enforce fine and coarse-grained scheduling strategies. Five such strategies were identified and are shown in Table 5.1.

When `Thread.yield()` is called in the Java runtime environment, there is no guarantee that the interpreter will suspend the calling thread and resume another thread. Nor is there any indication of what thread will be called next. For this reason, this implementation of concurrent genetic programming is non-deterministic. Solutions must evolve to solve not only the problem at hand, but also adapt to the dynamic of runtime scheduler.

A modified version Adil Qureshi's `gpsys` genetic-programming engine was used to evolve the program trees, with additional code by the author to implement the concurrent execution of program trees.

Name	Description
freeYield	`Yield` is added to the terminal set and scheduling is evolved freely;
implicitYield	`Yield` is added (as before) but an implicit Yield is performed on the evaluation of the expression. Then, every tree in the agent becomes wrapped in a Yield terminal;
fullYield	`Yield` is not added to the terminal set, but is performed implicitly on entry to every function (thus effecting fine grained parallelism);
cycleYield	where, again, `Yield` is not added to the terminal set, but is called implicitly on the evaluation of every tree. This amounts to a sequential ordering of the execution of the trees.
noYield	`Yield` is not in the terminal set and there is no implicit yield on tree evaluation. Scheduling is left entirely up to the Java runtime environment.

Table 5.1: Different scheduling strategies for evolving agents with the non-deterministic implementation of concurrent genetic programming.

5.5.2 Deterministic Implementation

Later investigations into concurrent genetic programming in [71, 72] achieved concurrency using a deterministic round-robin scheduler and the `Yield` function.

When the scheduler evaluates each program, it does so by giving each an amount of *program energy*. Program energy corresponds to the granularity of concurrency. Each genetic program is executed by traversing the program tree in an infix manner and evaluating the terminals and functions as they are encountered: the scheduler allows each tree to proceed in this manner for a number of functions and terminals equal to the energy. When the energy is "used up", the scheduler gives energy to the next tree, which is evaluated in a similar fashion. If a program completes its evaluation with energy left over, this energy is discarded and the next program in the agent is evaluated. When the last program in the sequence has been given its energy, execution resumes at the first program in the sequence. This program resumes evaluation at the last point of execution before it lost energy.

When any one of the genetic programs fires (that is, produces a result), the runtime scheduler halts execution and returns the result, which may then be used to decide the next action of an agent. The control of the agent is thus interrupt driven: as soon as a program fires an interrupt is raised, and the result is interpreted as an action. In the meantime, the scheduler interpreter maintains the program-state of each of the genetic programs, ready to resume evaluation if required. By varying the program energy, it is possible to control the granularity of the concurrency. By setting program energy to 1, the scheduler effects simulates the simultaneous evaluation of programs. In this situation, shorter programs will fire more frequently than larger ones. If the program energy is set to some $\epsilon > 1$ the scheduler effects coarse grained parallelism. Finally, if energy is set to some very large number (greater than the number of functions and terminals in any possible program tree), then the scheduler will simply evaluate each genetic program in turn. This mode of execution is termed "setting energy to infinity."

The `Yield` function allows agents to evolve scheduling schemes around that of the round robin scheduler. When the `Yield` function is encountered, it discards the remaining program energy and allows the scheduler to move on to the next program in the sequence. When the program energy is 1, `Yield` effectively acts as a "no-op". When program energy is larger, `Yield` provides the only mechanism by which

Use `Yield`	Energy	Concurrency
No	1	Fine-grained
	$\epsilon > 1$	Coarse-grained
	∞	Sequential program evaluation
Yes	1	Evolved fine-grained
	$\epsilon > 1$	Evolved coarse-grained
	∞	Evolved multi-threaded

Table 5.2: Different levels of concurrency achieved through varying program energy and use of the `Yield` function.

agents can alter the way the programs are scheduled. Combining the mechanisms of program energy and `Yield` function gives a scheduler that effects a number of different concurrent programming paradigms, as shown in Table 5.2.

5.6 Issues

Concurrent genetic programming provides an appropriate mechanism for the evolution of emergent behaviour, and opens up a number of questions with respect to concurrency, number of processes and exploitation of explicit memory towards internal representations.

The implementation of pseudo-parallel processes on a single processor machine leads to a number of issues. Fine-grained parallel execution captures best the artificial life idea of interaction and emergent behaviour in a population of semiautonomous agents. However, concurrent genetic programming may give better results if more coarse-grained execution is employed. The effectiveness of the `Yield` in allowing agents to evolve scheduling strategies around that of the round-robin scheduler must also be investigated.

With respect to the number of processes within the agent, is there a correspondence between number of processes and fitness? That is, will using more processes give fitter agents?

Finally, how do the agents exploit explicit state? Are distributed representations evolved, or is explicit state used in some other way? Should each process have a local memory, a global shared memory or both?

The experiments in the next chapter set out to provide some empirical evidence to answer these questions.

5.7 Summary

This chapter has described concurrent genetic programming, a paradigm for the automatic creation of algorithm and representation for autonomous agents through emergent behaviour mechanisms. Two different implementations are introduced, and issues worthy of investigation are noted. The next chapter describes experiments and results designed to provide answers to these questions.

Chapter 6

Experiments and Results

6.1 Introduction

This chapter describes experiments that investigate the issues raised by the concurrent genetic programming paradigm described in the previous chapter. Experiments are designed to show how the use of explicit memory, different distributed memory architectures, number of trees, and level of concurrency affect the ability of the paradigm to produce good agents. Agents are evolved to perform in two "virtual robotics" environments, described in Section 6.2. Good performance in these environments cannot be achieved using immediate sensor information alone, and so they provide a suitable test-bed for the evolution of internal state.

The experiments and results of this chapter are published in [70, 71, 72], and are presented here in chronological order. Initial investigations used the non-deterministic implementation of concurrent genetic programming, and were geared towards the examination of different distributed memory architectures and levels of concurrency. These investigations are described in Section 6.4. Results showed the development of high-performance agents, but a general indifference to memory architecture and level of concurrency. Agents exploited an implicit form of state, but the non-determinacy of the implementation made it hard to investigate how this implicit state emerged.

A deterministic implementation was developed and further investigations were carried out, as described in Section 6.5. These inquiries used a shared memory architecture, and varied the number of processes in the agent and level of concurrency used. No significant benefit was found in using more than two processes, and best agents were found by allowing the agents to develop their own scheduling strategies. Again, agents evolved to perform well while ignoring the explicit state mechanisms provided.

The emergence of implicit internal state in concurrent genetic programming is treated in Section 6.6. Essentially, every process in the agent has a number of halting states. When the processes are run together a set of composite states is formed, and the agent traces cyclical routes through this state-space to effect persistent behaviour strategies that can give good performance.

6.2 Virtual Robotics

The experiments in this chapter employ agent environments that involve a single agent residing in a small walled two-dimensional tessellated world. Typically the world contains blocks acting as movable or immovable obstacles, and the agent's task is to move the blocks around the world towards some goal configuration. The agent senses only the contents of the neighbouring squares, and does not know its position, the position of the other blocks, or the size and shape of the world.

To appreciate the difficulty of the virtual-robotics problems one must consider the problem from the perspective of the agent. The agent has eight numeric sensory inputs, and can effect a number of actions, typically "turn left", "turn right", and "move forward". It makes a number of moves, over which time the sensory inputs will change accordingly. When the agent has used up all its moves, it receives a score. The hypothetical conversation between a human player and gamemaster in Figure 6.1 illustrate the difficulty of this kind of problem.

Although virtual robotics problems are sometimes referred to as "toy-worlds", the problems they pose are demanding, and require the prospective player to find solutions solely through experience. The problems used here, described in Sections 6.2.1 and 6.2.2 are such that the next best move to make cannot be determined from the local sensory information alone. They are sophisticated enough so that random movements yield poor performance. Finally, they are attractive in that they require only modest computational load in evaluating performance, and so provide an ideal benchmark with which to experiment with a new representation such as concurrent genetic programming.

6.2.1 Tartarus and Diagarus

Teller introduced the Tartarus world as a problem requiring the intelligent use of state for good performance [68]. The world is a 6×6 grid bounded by an impenetrable wall, in which 6 blocks are randomly placed in the inner 4×4 locations. A single agent is also placed in this central area, and its task is to push the blocks to the perimeter locations. Starting with a random position and direction (either N, S, E or W) the agent can choose to turn left, turn right or move forward into the cell it is facing. Turning left or right is performed in steps of 90 degrees, so the agent can only face (and move) in the directions North, South, East and West. If the agent attempts to move forward into a cell with a block in it, if will do so only if the block can be moved into the next cell in that direction. If this next cell contains a block or the wall then the move is unsuccessful and the agent remains in place. The agent senses its world using 8 sensors that return the contents of the 8 neighbouring cells, relative to the direction of the agent. This arrangement is shown in Figure 6.2. The sensors return one of three values: 0 for an empty cell, 1 for a cell containing a block,

Game Master:	"Here's eight numbers: based on these numbers (or ignoring them completely) give me a number!"
Player:	"Ok! I'm picking the number 17"
Game Master:	"Fine! Now: You'll notice the numbers have changed. Give me another number"
Player:	"Why?"
Game Master:	"Just give a number, based on the inputs. Or ignore them completely if you like. You're not allowed any further information"
Player:	"I'm not sure I understand. But I'll play a long anyway...."
	... some time later:
Game Master:	"Well done, your score is 2."
Player:	"What does that mean?"
Game Master:	"You scored 2. Would you like to try again?"
Player:	"I don't get it. What am I supposed to do?"
Game Master:	"You're just supposed to play the game. Go on, give me a number..."
Player:	"I don't like this game!"

Figure 6.1: Hypothetical conversation between a player and game-master in a virtual robotics environment. The only information given to the player is the immediate sensor values. No information on spatial location is supplied, and positive feedback is given only at the end of the game in terms of a score.

and 2 for the wall. The agent is not given its position in the world, nor is it given the size of the world: it must perform using the immediate sensory information alone. Every action the agent makes constitutes a *move*, regardless of whether the move is successful or not, and the agent must push as many blocks to the perimeter as possible within 80 moves. The agent is awarded 2 points for every block in a corner and 1 point for every block in one of the 16 remaining perimeter cells. A number of "impossible worlds" exist in Tartarus, characterised by the appearance of a square of 4 blocks, which clearly cannot be moved giving the "push one block at a time" restriction. Agents are *not* tested in worlds containing such configurations, however, it is conceivable that such configurations could occur during the agent's work.

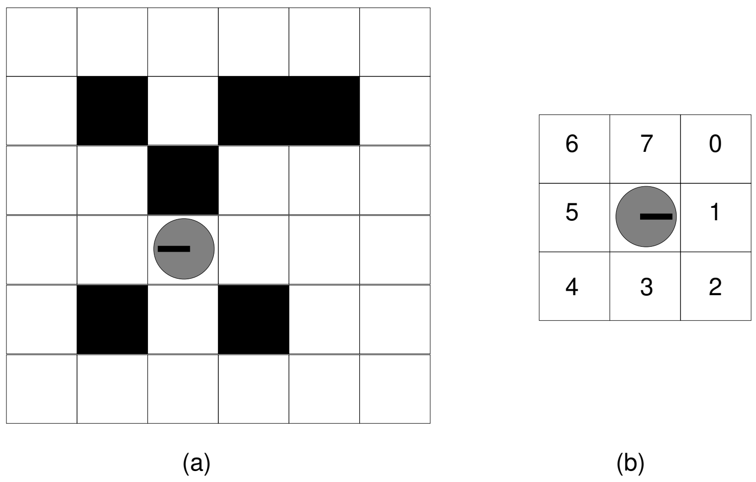

Figure 6.2: A typical initial configuration for Tartarus (left). The agent senses its environment through 8 sensors numbered relative to the agent's orientation as shown on the right.

Tartarus is an interesting problem because agents without internal state perform very poorly at it. A simple strategy such as "if a block is ahead, push it" will be ineffective as there may be no feedback from the environment to let the agent know that this move has been successful. Such an agent could expend all of its moves pushing the same block against the same wall cell. To perform well an agent must use some internal state to allow more than just reactive behaviour.

Teller originally used Tartarus to demonstrate the efficacy of linear indexed memory as an explicit state mechanism [68]. Agents, controlled by genetic programming trees with indexed memory, were evolved that performed well. Teller used genetic programming trees without indexed memory as a base-line, and these scored poorly yielding about 0.5 points per board. His best agents (using indexed memory and a single automatically defined function) gave a score of about 4 points per board, thus demonstrating that agents using indexed memory performed significantly better than those that did not. In one set of experiments, agents were evolved with access to indexed memory but *without* access to sensor information. Surprisingly, agents with good fitness were evolved, indicating that agents could evolve to use memory effectively while not storing or using the information they received from the environment.

Ashlock and Joenks [3] have proposed a different base-line for Tartarus, in which internal state is provided implicitly. The agent is controlled by a fixed sequence of actions, which is repeated over and over until the agent runs out of moves. In this manner, the agent executes a "fixed dance", where implicit state is formed via sequencing of actions. Strings of actions of length 4, 5, 8, 10, 16, 20, 40 and 80 (all factors of 80) were evolved, and it was found that strings of length 16 and 20 performed best, giving a fitness of approximately 3 points per board. A variable length string representation was also tried, and it produced an agent with an average score of 4.866 using the following "dance": FLFFLFLFFFFRFLFLFFRF. Finally, using a novel representation called ISAc lists, where agents are controlled by a number of If-Statement-Action sequences, the authors evolved agents with scores in the region of 7 points. In this representation, the controller program consists of a list of 150 if-statements that compare two sensor values, and perform a specified action before jumping to another if-statement. Each element in the list is thus a four-tuple, containing the numbers of the two sensors, the action to be taken, and the next element to be processed. While this approach performed well on Tartarus, the authors found that it did not perform as well on other problems such as decoding 7-bt Hamming code, computing four-bit parity and playing the Iterated Prisoner's Dilemma.

A variation of Tartarus has been used elsewhere by Honovar and Balakrishnan [74] to investigate the evolution of sensor configurations. They modified the problem by using more time steps (216) and allowing the agent to evolve its own sensor configurations, beyond the immediate-neighbour sensor set-up described by Teller. Using a neural network approach on this modified world they achieved fitness in the range of 3.5 to 4.5 points per board, however, given the increased number of time steps it is hard to compare these with other results.

Previous work by the author [70] used a variation of Tartarus in which agents turned in steps of 45 degrees (thus allowing movement along diagonals) and only 5 blocks on the board. This variation of Tartarus, which we shall refer to from here on as Diagarus, may be easier to solve than the original Tartarus problem. This is due to the lack of clutter on the board and the use of diagonal moves: agents evolved wide circular path strategies that cleared blocks to the perimeter.

6.2.2 Dozer

The Dozer problem is similar to Tartarus and was proposed by Ashlock in 1998 as part of a virtual robotics competition for the 1999 Congress on Evolutionary Computation. To the author's knowledge no results have been published at the time of writing. The Dozer world is larger than Tartarus (8×8) and there are nine boxes that start both away from the walls and not next to one another. The bulldozer is given 150 time steps in which to move. In each time step it may turn left or right or advance to the grid square in front of it. As the bulldozer advances it may shove one box ahead of it into an empty square. If it pushes against two boxes, nothing moves. The bulldozer has eight sensors, (similar to the Tartarus agents) that report 0 for empty space, 1 for a box, and 2 for a wall. The bulldozer is allowed no other outside information. The score, at the end of 150 moves, is the number of squares, next to any box, that are not empty. Either a wall or another box serves to make a square not empty. This makes the maximum possible score on any one world configuration 30. The goal of the robot is to take a dispersed initial configuration of boxes and both make it as compact as possible and to get the compact group of boxes against a wall or in a corner. The notion of "next to" in this problem is the one that gives a grid square four neighbours, to the north south, east, and west. The bulldozer is always pointing in one of these directions but does not know which one.

While Tartarus examines the ability of agents to clear blocks to the perimeter, Dozer requires agents to clear blocks *and* cluster them together. In both cases, the agents exist in a world where immediate sensory information is insufficient in determining an optimum strategy, and so must develop and use appropriate internal state to perform well.

6.3 Applying Concurrent Genetic Programming

To apply concurrent genetic programming to the virtual robotics worlds, the computational blocks used to construct the algorithm and representation and the mechanism by which agents receive input from the world must be specified.

6.3.1 Functions

The same functions and terminals used by Teller [68] are used in the experiments here, with the addition of the `Yield` function (described in the previous chapter in

Section 5.4.3. The functions used are described in Table 6.1. To allow agents to exploit indexed memory, the functions `Read` and `Write` are added to its function set. In cases where a number of different memory modules are used, the name of the module is appended to the function name. For example, `ReadA` means "read from memory A" and `WriteSM` means "write to shared memory".

(Or X Y)	Returns the MAX(X,Y).
(Not X)	Returns 1 if $X = 0$, 0 otherwise.
(Less X Y)	Returns 1 if $X < Y$, 0 otherwise.
(Add X Y)	Returns $X + Y$, with ceiling 19.
(If-Then-Else X Y Z)	Returns Y if $X > 0$, Z otherwise.
(Sub X Y)	Returns $X - Y$, floored to 0.
(Eq X Y)	Returns 1 if $X = Y$, 0 otherwise.
(Read X)	Returns memory$[X]$.
(Write X Y)	Returns memory$[Y]$ and sets memory$[Y]$ to X.
(Yield X)	Halt and allow next tree to resume evaluation.

Table 6.1: Typical functions used for the concurrent genetic program function sets.

6.3.2 Terminals

The agent senses its environment through 8 sensor terminals, labelled `Sense0` to `Sense7`. These sensors return 0 for empty space, 1 for object, and 2 for wall. Also in the terminal set are the integers `0` – `19`.

6.3.3 Actions

In these worlds the agent picks one of three actions, "move forward", "turn left" or "turn right". When one of the processes in the agent "fires", that is, produces a result, the result is wrapped to decide which action will be taken. Using the function set in Table 6.1, the result values are integers in the range 0–19, and the following wrapper is used: "if the result is in the range 0–6 then turn left, 7–13 then turn right, 14–19 then move forward."

6.4 Non-Deterministic Concurrent Genetic Programming

This section describes the first investigations into concurrent genetic programming, reported previously by the author in [70]. Concurrency was achieved by assigning each program in the agent a Java thread, and then letting the Java runtime environment perform the scheduling. Details of this implementation are presented in Section 5.5.1. Experiments were designed to examine the effect of different distributed memory architectures for the agents and the effect of different scheduling polices, and are described in Section 6.4.1. Results of the experiments are presented and interpreted in Section 6.4.2.

6.4.1 Experiments

The purpose of these experiments was to examine the effect of using different distributed memory architectures, and how different levels of concurrency affect performance. The Diagarus variant of Tartarus, described in Section 6.2, was used. The agent is restricted to having two genetic program trees, T_A and T_B, as this small number of trees leads to a straightforward list of memory architectures for introductory exploration. These are:

1. T_A has local memory, T_B has no memory.
2. Neither T_A or T_B have memory.
3. Both T_A or T_B have local (not shared) memory.

4. Both T_A or T_B share a single memory.

5. T_A and T_B have access to a local memory and a single shared memory.

In the above list, *memory* is taken to be a linear indexed memory of size 20.

Each of these architectures was evolved using one of the five scheduling strategies outlined above in Table 5.1, that is, *freeYield*, *implicitYield*, *cycleYield*, *fullYield* and *noYield*. Combining the five scheduling schemes with the five different memory architectures there are a total of 25 possible experiments. For comparison, Teller's original experiments employing genetic programs with and without memory and automatically defined functions were implemented using the same experimental set-up. A tableau describing the experiments is shown in Table B.1. The experiments were coded using an extension of Adil Qureshi's original Java-based GP engine, `gpsys`, and run on a farm of Pentium 133MHz machines with 16MB memory under the Windows 95 operating system, each run taking between 8 and 9 hours to complete. Each of the 28 experiments was performed 20 times, so that some notion of average performance could be ascertained.

6.4.2 Results

The fitness profiles of the best agents from each of the 20 runs were averaged and these results were used for comparison. A full table of results for the experiments is presented in Table B.3 in Appendix B,with results for the reconstructions of Teller's single tree experiments shown in Table B.2. Here, in Table 6.2, the results of the single-tree experiments and the best concurrent genetic programming experiment are presented. Fitness ranges from 0 to 360, with a higher value indicating better fitness. Comparing the mean of the single-tree agents with memory against the two-tree agents with shared memory using the *"Student's"* t test, the CGP agents are better beyond the 99.5% significance level.

Agent	Ave.	Max.	Std. Dev.
Single tree, no memory	63.21	98	14.46
Single tree + memory	176.5	212	26.20
Single tree with ADF + memory	165.10	207	25.33
Two tree + shared memory	201.45	231	12.71

Table 6.2: Table showing the results of the single-tree experiments and the CGP experiment using *implicitYield* and shared memory. A full listing of results from the CGP experiments can be found in Table B.3.

Figure 6.3 shows that the *implicitYield* scheduling scheme, where agents evolve a yielding strategy but are subject to an implicit yield on action, perform best on average. The general high performance suggests that the trees evolve some co-operative strategy to solve the problem. The other scheduling schemes have almost identical fitness profiles, as shown in Figure 6.3. For comparison, the average best-fitness curve from the conventional single-tree genetic programming experiments is also plotted on Figure 6.3.

The performance of the different memory architectures over the *implicitYield* scheduler is shown in Figure 6.4. The best memory architecture is where one tree uses local memory and the other uses none, combining reflective and reflexive processes. It performs marginally better than the other memory architectures over each of the different scheduling strategies (with the exception of the *fullYield* scheduler).

On examination of the 20 best-of-generation agents for the *implicitYield* experiments where one tree used indexed memory and the other did not, some interesting properties emerge. Many agents exhibit a tree *shrinking* property, where one of the trees shrinks to a single terminal in the range (14...20). In other words, one of the trees effects a constant *moveForward* strategy, while the other executes a more dynamic procedure to control turning and moving. An example of such a tree is

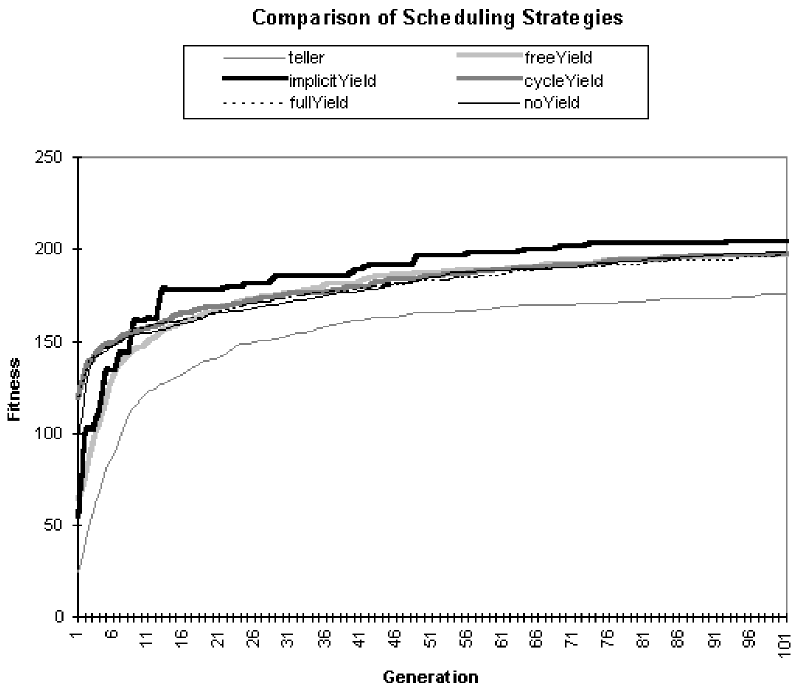

Figure 6.3: Comparison of different scheduling strategies used. The *implicitYield* strategy works best, while the others are practically indistinguishable. All concurrent genetic programming strategies perform better than Teller's genetic programming agents with indexed memory [68]

shown in Figure 6.5.

Some of the best agents exhibited very low complexity, for example, in Figure 6.6, only 5 nodes in the first tree and 3 in the second combine to give a fitness of 199. Note however that the first tree can only ever produce a *turnLeft*, as it returns a sense value which will always be in the range {0, 1, 2}, while the second can only produce a *moveForward*. Thus, this behaviour is the combination of a sequence of *moveForward* and *turnLeft* actions.

Although agents were evolved with `Read` and `Write` functions, only two of the runs evolved agents that actually used these functions, one of which is shown in Figure 6.7. This echoes the results of Paterson [56] where the introduction of indexed

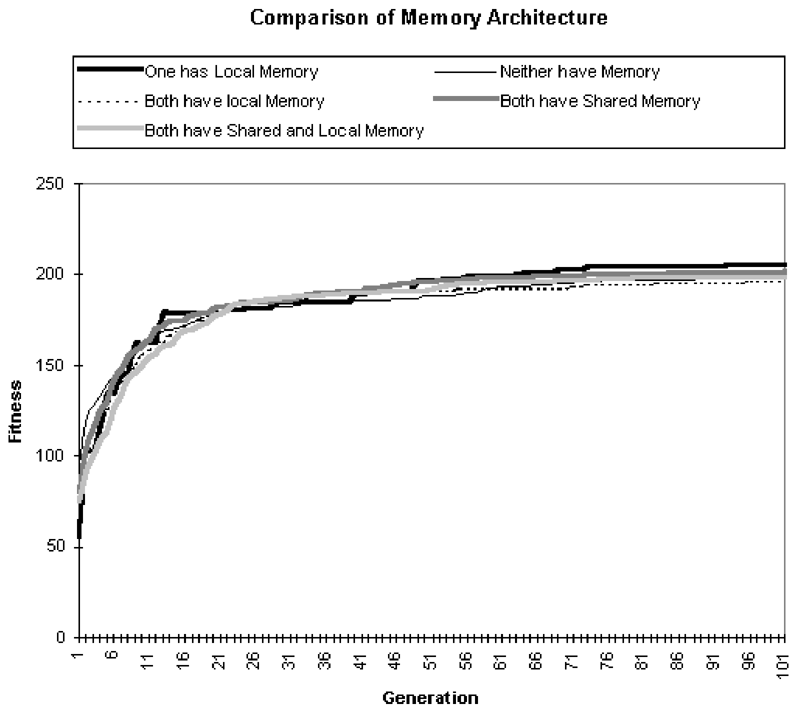

Figure 6.4: Comparison of different memory architectures using *implicitYield* scheduling. Allowing one tree memory and the other none performs best, but only marginally.

memory to the evolution of caching algorithms lead to no increase in performance. Concurrent genetic programming agents do not need implicit state for good fitness at this problem, as many of the agents evolve away from using the `Read` and `Write` terminals. Concurrent genetic programming agents without explicit state can solve this problem *better* than single-tree machines *with* explicit state, as shown in Figure 6.8. In this graph, the best concurrent genetic programming run is plotted against the best of the traditional genetic programming runs, and is clearly seen to be better.

How do such stateless agents perform so well? There are three possible answers: first, the emergent behaviour is complex enough to negate the need for state; second, the implicit state used by each tree's program stack is enough to solve the problem;

tree 0: `(14)`

tree 1: `((If-Then-Else Sense0 (Less Sense5 Sense2) (If-Then-Else (Eq Sense1 1) (Yield (If-Then-Else Sense1 (If-Then-Else Sense0 (Less Sense5 Sense2) (If-Then-Else Sense1 (Yield Sense6) Sense7)) Sense7)) (If-Then-Else Sense2 (If-Then-Else Sense7 (Eq Sense1 1) Sense3) (Yield (Sub Sense6 Sense6))))))`

Figure 6.5: Agent exhibiting the shrinkage phenomenon, where one tree reduces to a constant value that forces the agent to *moveForward*.

tree 0: `((If-Then-Else Sense1 (Yield Sense4) Sense1))`

tree 1: `((Add 18 Sense2))`

Figure 6.6: Compact agent with fitness of 199.

tree 0: `((Add (Add (Write 17 (If-Then-Else Sense0 (Yield 17) Sense7)) Sense1) (Add (Less (Read Sense4) Sense1) 18)))`

tree 1: `((If-Then-Else Sense1 (Yield (Yield Sense5)) (Add Sense1 Sense1)))`

Figure 6.7: CGP Agent showing use of memory.

or third, the trees evolve ordered action strategies to solve to problem in a fashion similar to Koza's implicit state through sequencing of action commands with the *progn* function. Whatever the case, these results indicate that for problems that traditional genetic programming finds hard, concurrent genetic programming may be better equipped to find a solution.

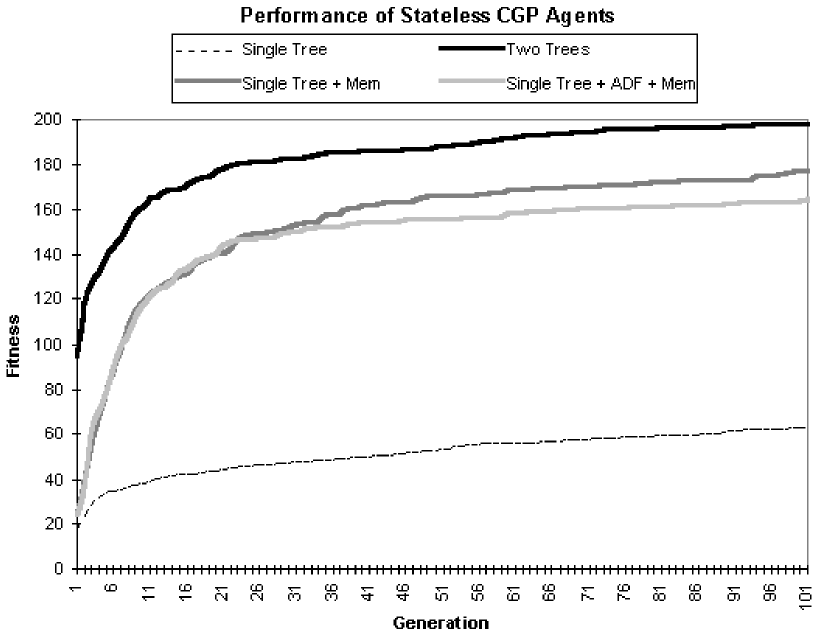

Figure 6.8: Comparison of stateless CGP agents (top curve) with Teller's indexed memory agents (middle curves) and conventional GP stateless agent (bottom curve).

6.4.3 Conclusions

The results of this section show that the concurrent approach is certainly effective for this problem, evolving agents that perform better than the conventional single-tree approach. Conventional agents perform well only when equipped with indexed-memory, interpreted by Teller to be a mechanism for the creation of dynamic models of the agent's world. Surprisingly, the concurrent genetic programming agents are indifferent to the provision of `Read` and `Write` functions, performing just as well without them. The experiments devised to test the benefit of different memory architectures, incorporating different variations of shared and local memory between the concurrent programs, revealed that the different architectures made little or no impact on fitness. Clearly, the state is developed through some mechanism other than the explicit mechanism provided.

Unfortunately the non-deterministic implementation hindered further analysis into how the implicit state was being created. A runtime investigation of the performance of the agents would require the use of blocking IO commands to communicate the agent's state, which would affect the runtime environment and ultimately impair performance. One approach considered was to model the Java runtime environment: instead, it seemed more sensible to develop a fully controlled, deterministic runtime environment, and use this for further investigation. The experiments and results carried out in the deterministic environment are described in the next section.

6.5 Deterministic Concurrent Genetic Programming

The results of the previous section indicate the ability of the concurrent approach to create an implicit form of state; however, the implementation did not easily lend itself to the investigation of this implicit mechanism. In this section this issue is addressed. A deterministic runtime environment was designed and integrated with Evolution Engine, an object-oriented framework for the evolution of arbitrary structures, described in Appendix A. The new system was applied to the Tartarus and Dozer problems, to examine the effect of different levels of concurrency and use of explicit state. Also investigated is the impact of the number of processes controlling the agent.

6.5.1 Experiments

Previous experiments in Section 6.4 showed that varying different memory architectures had little impact in the performance of the agents, but revealed that fit concurrent genetic programming agents could evolve without using the explicit state provided. To investigate how this implicit state emerges, and to examine more clearly the effect of levels of concurrency and the number of processes in the agent, a deterministic runtime environment (described in the previous chapter in Section 6.5) was employed.

The experiments are designed to shed light on the questions: "Is explicit memory useful at all in this paradigm?", "How does performance vary with the number of programs in the agent?" and "What is the best scheduling strategy?" The experimental set-up for both worlds is the same: by applying the approach to two different problems an idea of its goodness as a distributed-AI approach is investigated.

In these experiments all trees in the agent can use of the `Yield` function, and when memory is used it consists of a single linear memory element shared by all of the programs in the agent. Every tree in an agent has access to the same function and terminal sets: the same functions and terminals used in the previous section shown in Table 6.1 are used here. On each generation, 40 valid starting-worlds are generated randomly and these are used to test the agents. By testing agents against a different set of boards in every generation we ensure that they are exposed to a wide variety of initial board configurations. Raw fitness is calculated as the sum of the scores for each board. Standardised fitness is calculated by subtracting raw fitness from the maximum score: 400 for Tartarus and 1200 for Dozer.

To examine the effect of different numbers of program trees on performance we vary the number of trees in the agent, creating agents with 1, 2, 3, 4 and 5 trees. To examine the efficacy of indexed memory as a shared memory mechanism we evolve agents with and without access to the indexed memory functions `ReadSM` and

`WriteSM`. To examine the effect of different levels of concurrency on fitness we evolve agents over 1, 2, 5, 10 and infinite-energy.

This gives a total of 42 experiments for each world. Each experiment is run 20 times with different seeds: clearly, more runs would carry greater statistical significance, however, 20 was chosen to minimise the amount of computational time required produce the results, while giving some indication of the average performance under the different parameters. Runs were performed using the author's own evolution software (described in Appendix A) on 80 Pentium 233MHz machines running the Solaris operating system, taking between 3 and 10 hours for each run. Code specific to the Dozer world was that supplied by Ashlock.

A tableau summarising these genetic programming parameters is shown in Table B.4.

6.5.2 Results

Results of the 84 experiments are presented in full in Tables B.5, B.6, B.7 and B.8 in Appendix B. Table 6.3 shows how the best CGP runs fare against the conventional agents on Tartarus. In this table, the best fitness (where a lower value is better than a higher one), mean fitness and standard deviation are presented. The *Score* is the average score (out of 10) scored by the best agent over 5000 randomly generated boards. Using the one-tailed *"Student's"* t test on the mean best-fitness the CGP agents perform better than their counterparts at the 99.5% level of confidence.

Table 6.4 compares the performance of the concurrent agents with the conventional single-tree agents on the Dozer problem. Again, fitness is better when the value is lower, and the score gives the average score (out of 30) of the best agent against 5000 random boards. As could be expected, the concurrent agents perform significantly better (at the 99.5% confidence level) than the single-tree agents without memory. The concurrent agents with memory perform better than the single-tree

Experiment	Best	Mean	Std. Dev	Score
1 tree, no memory	340	359.35	5.23	0.69
2 trees, no memory, 5-energy	150	227.60	23.41	5.89
1 tree, with memory	219	249.55	15.65	3.85
2 trees, with memory, infinite-energy	157	203.25	26.00	5.58

Table 6.3: Results from best CGP experiments compared with conventional single-tree agents for the Tartarus problem.

agents with memory at the 70% confidence interval (again using the *"Student's" t* test).

Experiment	Best	Mean	Std. Dev	Score
1 tree, no memory	884	1008.75	69.21	7.20
2 trees, no memory, 10-energy	609	737.70	46.91	13.30
1 tree, with memory	622	721.70	74.58	12.45
2 trees, with memory, infinite-energy	609	717.55	51.53	13.96

Table 6.4: Results from best CGP experiments compared with conventional single-tree agents for the Dozer problem.

These results clearly show that concurrent genetic programming produces significantly fitter agents than the conventional single-program approach for Tartarus, and marginally fitter agents for Dozer.

Further analysis of the results in Appendix B reveals certain trends with respect to the effect of the level of concurrency, number of trees in the agent, and the efficacy of indexed memory.

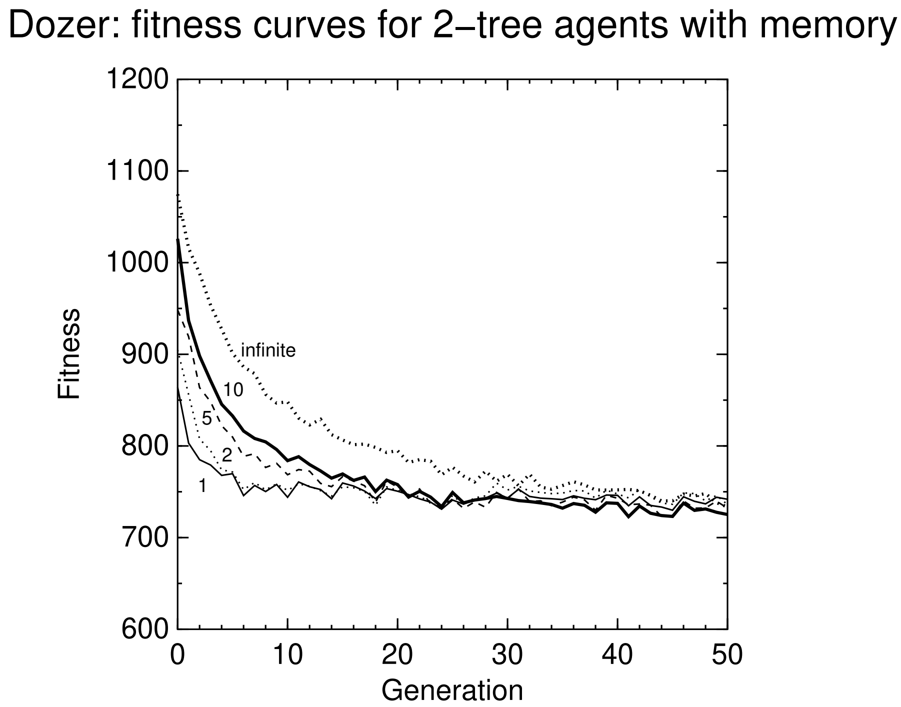

Figure 6.9: Graph showing how the fitness of 2-tree agents with memory varies with program energy.

Effect of Concurrency

The concurrent approach produces agents of good fitness, *regardless* of the level of concurrency. In general, the mean-best-fitness of infinite-energy agents is *poorer* than that of low-energy agents. However, the high standard deviation of the infinite-energy experiments indicates the presence of outliers of unusually high fitness.

The level of concurrency does affect the rate of convergence of the evolution runs, that is, agents using low program-energy converge faster than those using high program-energy. The fitness curves for the 2-tree agents shown in Figures 6.9 and 6.10 are indicative of the effect of program energy on fitness growth. In Figure 6.9 agents with low energy (1 and 2) tend to bottom out within the first 10 generations. Agents with 5 and 10-energy level out at around generation 20, and the infinite-energy agents level out at around 50 generations. Figure 6.10 shows

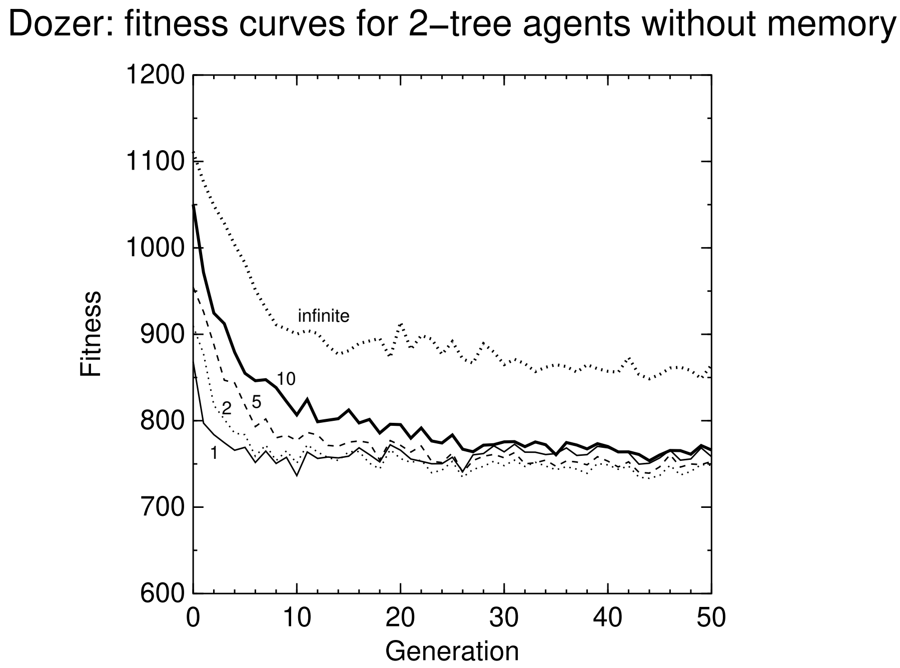

Figure 6.10: Graph showing how the fitness of 2-tree Dozer agents *without* memory varies with program energy.

average fitness curves for 2-tree agents *without* memory: they converge sooner, with the exception that infinite-energy agents perform consistently poorer than all of the lower energy agents. On average, they level out at about 100 points behind the rest.

To summarise, program energy affects the fitness of randomly generated agents in the initial population and the rate at which the runs converge. Best agents are evolved when infinite-energy is used, that is, where the agents control the granularity of concurrency through the use of `Yield`.

Number of Trees

The best performance consistently comes from 2-tree agents: there is no clear advantage in using greater numbers of trees. Agents with two trees tend to take longer to converge, while 3, 4 and 5-tree agents follow almost identical fitness curves.

Varying the number of trees affects the complexity-growth of the agent. The structural complexity is the sum of the number of terminals and functions over the program trees in an agent. In Figures 6.11 and 6.12 complexity of 1-energy (fine-grained evaluation) agents are plotted. Two things are immediately apparent: first, agents without memory are more complex than those with memory are. Second, in both graphs we notice an initial *dip* in complexity for 4 and 5-tree agents. This indicates that in the early stages of evolution agents with a large number of small trees tend to perform very well. That is, the random space of concurrent genetic programming agents contains many good randomly generated agents of low-complexity. The reason for this lies in the creation method used: using the ramp-half-and-half method many agents are created with trees with only a single node. So, when the agent is evaluated these nodes are wrapped to perform a sequence of actions. Ashlock's action-sequence baseline (see Section 6.2), has shown that action-sequences can be used to achieve good fitness in this kind of world: it follows that the 4 and 5-tree runs are converging on a pool of action-sequence solutions in the early generations, before bloat sets in.

Efficacy of Indexed Memory

Explicit state (through indexed memory) is crucial for the single-tree agents, however, the concurrent genetic programming agents exhibit only a small increase in fitness when indexed memory is used. This suggests that internal state is created using an implicit mechanism: this is discussed further in Section 6.6.

6.5.3 Conclusions

The goal of these experiments was to investigate the efficacy of memory and the effect of the granularity of concurrency and increasing the number of processes within the agent. The wide range of experiments reveal that the concurrent approach is adept at finding solutions to the virtual robotics problems presented, finding better

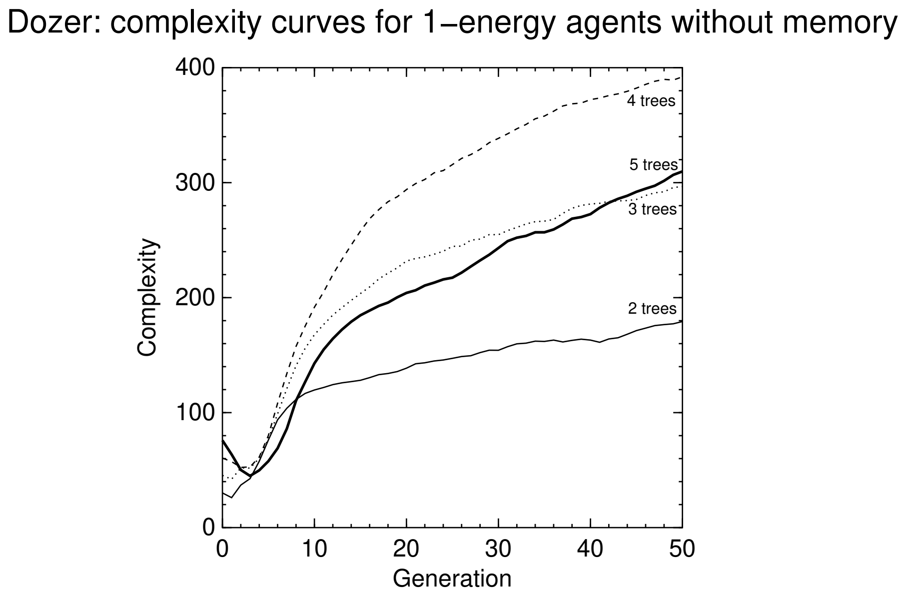

Figure 6.11: Graph showing how the complexity of 1-energy Dozer agents *without* memory varies with the number of trees.

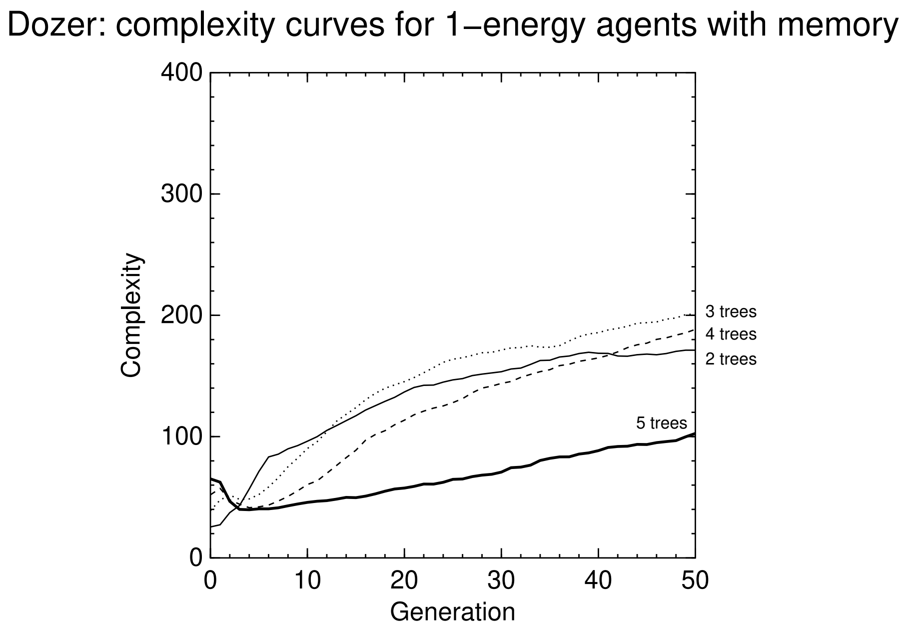

Figure 6.12: Graph showing how the complexity of 1-energy Dozer agents with memory varies with the number of trees.

solutions to the problem than a single-tree agent approach. Clearly, concurrent genetic programming works, and it works well for this kind of problem. Increasing the number of trees in an agent leads to faster convergence of the genetic algorithm; however, it did not yield a significant increase in performance. Similarly, agents with low energy (low granularity of concurrency) converged faster than those of high energy, and performed better *on average*. However, the wide standard deviation of the high-energy agents means that although the mean is lower, the high-energy runs are capable of producing outliers of high performance. To sum: two-tree concurrent agents with evolved concurrency perform best at these problems, with or without indexed memory.

The results raise an important question: if concurrent agents do not need explicit memory mechanisms, then where is the state coming from? And further, how this "emergent state" ties in with the "mental models" interpretation of indexed-memory proposed by Teller? The ability of the representation to produce state through cyclical behaviour strategies is shown in Section 6.6.1, and a discussion of the kind of state required to solve virtual robotics problems is discussed in Section 6.6.2.

6.6 Emergence of Implicit State

In this section, the emergence of implicit state in concurrent genetic programming is described, showing how behaviour emerges as sequences of actions that reflect an implicit world model.

6.6.1 Composition of Program Halt-States

This section describes how concurrent genetic programming can effect an implicit form of state, through the Cartesian product of the wait-states of the individual processes. A process in an agent is in a *wait-state* when it is about to begin execution or it has halted execution, either through a `Yield` call or through the consumption of the available program-energy.

Consider a concurrent genetic program M with n trees, each labelled t_i, where $i = 1 \ldots n$. With an energy level of ϵ, a tree t_i with complexity (number of function and terminals) $|t_i|$ will have its execution halted and enter a wait-state $\frac{|t_i|}{\epsilon} + 1$ times. That is, in the absence of the `Yield` function, a tree has $\frac{|t_i|}{\epsilon} + 1$ wait-states. Let $W(t_i)$ describe the number of wait-states for a tree t_i, giving Equation 6.1.

$$W(t_i) = \frac{|t_i|}{\epsilon} + 1 \tag{6.1}$$

The `Yield` terminal allows the evolutionary process to create suitable wait-states around those of the scheduler, by altering the position and number of wait-states. A program tree can be presented as an ordered linear list of terminals and functions encountered on the left-traversal of the tree. This string of functions and terminals (represented as γs) contains `Yield` functions, represented by Ys, as shown below:

$$\underbrace{\gamma \ldots \gamma}_{w_1} Y \underbrace{\gamma \ldots \gamma}_{w_2} Y \underbrace{\gamma \ldots \gamma}_{w_3} Y \gamma \ldots \gamma Y \underbrace{\gamma \ldots \gamma}_{w_{Y(t)+1}}$$

Each `Yield` function constitutes a wait-state, while the inter-`Yield` strings of non-`Yield` symbols, w_j will produce wait-states when energy is used up. Then, for a tree t_i using $Y(t_i)$ *Yield* terminals:

$$W(t_i) = Y(t_i) + \sum_{j=1}^{Y(t_i)+1} \frac{|w_j|}{\epsilon} \tag{6.2}$$

Equation 6.2 thus describes the number of wait-states in a tree with `Yield` functions. Then, the number of possible states an agent, M, can occupy is the product of the number of wait-states in each tree, shown in Equation 6.3

$$W(M) = \prod_{i=1}^{n} W(t_i) \tag{6.3}$$

The concurrent genetic program, M, has a finite number of states, each state the composition of the wait-states of the individual programs. As the programs are evaluated, the machine moves from one composite state to another, forming a path through the state-space of the machine. This path begins with the start-state (when all programs are about to be evaluated in turn) and will ultimately return to this state. The path through the state space may not be fixed, but rather may change due to conditional moves based on environmental input. Some of the composite states will involve one of the programs in the agent firing (and hence producing an action), and so as the programs execute concurrently they effect a sequence of actions. This kind of action sequence is similar to the embedded action terminals employed by Koza [40] and Moore [55, 54], and Ashlocks's simple action-sequences [3].

The concurrent approach thus allows the production of cyclical action sequences, that is, paths through the state-space that are repeated and effect agent-actions along the way. The paths and the actions are not fixed, but can vary according to environmental information available to the agent during the evaluation.

6.6.2 Composite State and Internal Representation

As the agent's trees are evaluated, the set of composite states (described above) is traversed. This sequence of states is cyclical, in that it begins in the start state (where all trees are in their initial state) and will inevitably return to this state. On this journey, different trees will fire producing a sequence of actions. The repeated sequence of actions can effect an appropriate goal-achieving behaviour.

It is instructive to compare this approach with how an observer-designer might have tried to solve the problem. From an observer's perspective, the agent's "world" is a walled room, filled with blocks. A meaningful mental model might be proposed, such as "The agent is facing north at position (2,3). There are blocks at positions (2,4), (0,5) ...". With this model, algorithms might then be devised to move blocks to appropriate point-scoring positions. However, the agent does not have the same global perspective as us: its "world" is an array of 8 integers in the range $0 \ldots 2$, whose values change when the agent moves. It would be naive to expect the agent to arrive at a mental model similar to ours: consequently, it is a non-trivial task for us to identify mental models within the agent.

However, an action sequence can be thought of as exhibiting an implicit world model. For example, one two-tree agent was evolved with a very simple structure: one tree consisted of five nested `Yield`s returning at the lowest level a number that when wrapped was interpreted as "turn right". The other tree contained a single number that effected a "move forward". The effect of this is to create a repeated strategy that performs something as simple as "make five moves forward, then turn right". This action sequence displays some knowledge about the world that was never made available to it explicitly: the agent has learnt at an evolutionary level that the world is no more than five moves wide, information that is implicitly encoded in the repeated action sequence. While this form of state does not represent the objects that the agent experiences, it none the less represents knowledge of the world pertinent to goal-achieving behaviour.

6.7 Summary

This chapter has applied concurrent genetic programming to a number of virtual robotics problems, with the aim of finding out how the paradigm works with different distributed memory architectures, number of processes in the agent, and levels of concurrency. The problems used where such that the agent's would have to form appropriate internal state to perform well.

A large number of experiments were carried out, showing that the paradigm prefers the implicit state due to the interleaved evaluation of the processes over the explicit memory mechanisms provided. This implicit state was exploited to effect goal-achieving sequential behaviour, reflecting an implicit knowledge of the world learnt at an evolutionary rather than agent level. Although the use of greater numbers of processes gave faster convergence, there was no increase in performance over using just two processes. The best agents evolved where those that evolved their own scheduling strategies through the `Yield` function and infinite-energy mode of concurrency.

Chapter 7

Conclusions

7.1 Conclusions

The question of how machines interact with and represent their environment lies at the heart of this thesis. I have argued that the internal representations employed by an agent should be based on the agent's experience of the environment through its sensors, and not on the world-model of a human agent-designer whose perception of reality is based on entirely different sensory capabilities. An emergent approach was proposed for the development of representation and algorithm, where the agent's behaviour arises as a result of the interactions of a number of simple processes. The design of such emergent systems can be performed by an adaptive evolutionary process that combines computational building blocks to effect an appropriate algorithm and representation. By adopting this bottom-up approach, the bias of a human designer is minimised, and attention is placed instead on the mechanisms that allow the development of internal representations when needed.

A new evolutionary approach was introduced where an agent consists of a set of evolved programs, all of which are evaluated concurrently. The behaviour of the agent thus arises as an emergent effect, due to the interaction and co-operation of the constituent programs. The use of explicit mechanisms for the development of internal state in this distributed approach was investigated, as were issues relating to the granularity of concurrency and the number of programs controlling the individual. Concurrent genetic programming produced better agents than conventional genetic programming, but did so using an implicit form of state, ignoring explicit state mechanisms. The implicit state reflects knowledge of the environment obtained at an evolutionary level, rather than dynamic representations of features of the environment during the agent's "lifetime".

A critical appraisal is presented in the next section, where concurrent genetic programming and the problem of representation are put in context. Ideas for future directions are then presented in Section 7.3.

7.2 Critical Appraisal

This thesis is in two parts, first, an inquiry into how an agent should represent the external world (if at all), and second, an investigation of an emergent approach to behaviour in an evolutionary context.

With respect to the first part, the conclusion was made that internal representations should be developed through an agent's experience of its environment. This is not to say that if an algorithm and representation are designed from a human perspective instead then they will not work, or that they are necessarily inferior. Recalling Marr's three-level framework, a computational theory may be implemented by many different algorithms and representations. The point is simply that there is a mismatch between the world model a human might propose and the ability of an agent to maintain this model using its limited sensory apparatus. If we are willing to take a leap and allow agents build their own representations, then this mismatch should not occur. The question then is how can agents build their own representations, and the investigation of a new mechanism for this constitutes the second part of the thesis.

If agents can be created through a process of analysis, identification of computational theory, and design and implementation of algorithm and representation, then why pursue an evolutionary approach? This criticism is fair, yet it is based on the assumption that the designer can find a correct computational theory for the problem at hand. If a computational theory is not forthcoming then alternative routes must be investigated: evolutionary computation is one such approach.

In the second part of this thesis, the use of concurrency in the evolution of emergent behaviour is investigated. The approach used is to apply concurrent genetic programming to a number of virtual world problems where agents must employ internal state. Virtual-robotics problems such as Tartarus and Dozer are useful precisely because they are toy problems: the focus of this investigation is not on the solution of the problem but on the mechanisms used to solve the problem. It is important to investigate the mechanism fully, to understand its strengths and its weaknesses, before attempting ambitious applications. The investigation here has shown that if genetic programming is extended to evolve mutli-threaded controllers for agents, the result is a mechanism that provides an implicit form of state that supports sequential behaviour.

The virtual robotics problems have two distinguishing features: first, the agent exists in a discrete world with restricted movement, and second, the agent's sensors always return accurate, noiseless values. This raises the question of how applicable the technique would be to the control of a real robot in a continuous real-world environment where sensors return unreliable information. Clearly, to evolve agents for the real world, the ability to deal with a noisy environment must be an integral part of the agent's fitness evaluation. However, a choice was made in this work to set the issue of sensor noise aside and concentrate on the ability of an agent to develop internal models based on accurate information. Only when it has been shown that the agents can develop internal models of a noiseless environment does it make sense to proceed and ask if the agent can generalise and deal with a noisy environment in a robust manner.

Concurrent genetic programming has not exhibited the ability to form explicit representations of objects in a noise-free environment, yet it forms an appropriate internal state to perform well. This supports the thesis that the kind of internal state employed by an agent is not necessarily equivalent to explicit internal representation of external objects. It also questions the ability of evolutionary approaches to build such explicit internal representations. Teller showed that using an unbiased memory mechanism such as indexed memory, he could evolve agents that where significantly better than agents without it, and equated this activity with the formation of internal "mental models". However, the use of explicit memory does *not* imply the development of internal world models: from experience gathered in this thesis a more appropriate explanation might be that the indexed memory was used to create behaviour through repeated action sequences.

The success of these cyclical behaviour strategies has two ramifications. First, their ability to effect goal achieving behaviour may not be limited to the virtual-robotics worlds, but may be a viable approach for autonomous behaviour in general. For example, the "internal state" of dynamic recurrent networks used in evolutionary robotics may simply act as a mechanism to provide repeated paths through a set of network states, effecting action-sequences rather than explicit representations of external objects. The state space of concurrent genetic programming provides an alternative mechanism to effect this kind of internal state. So, with respect to the question of resilience to noise, it is possible that concurrent genetic programming could produce similar behaviours for noisy agent environments. If this is the case, then an assessment needs to be made of when sequential-action forms sufficient agent-state to solve problems, and how to better exploit this approach through different evolutionary encodings.

Second, if the ability to evolve explicit representations is to be investigated, care must be taken to ensure that cyclical behaviour solutions do not distract the search process, producing behaviour through repeated action-sequences rather than manipulation of world-models. This may be achieved through a careful choice of agent environment, where goals can only be achieved through the formation of memory models that persist over time. An example of one such environment is Jacobi's T-Maze [35], where an agent must pass through a corridor lighted on one side, and at a junction turn left or right depending on which side of the corridor was lighted. In the T-Maze problem, the agent is given noisy, unreliable information, and the problem is posed so that reactive or sequential-action strategies will not succeed. Then, to perform well, the agent must develop some internal state isomorphic to information from the environment (the side the light shone) and maintain this state for an arbitrary amount of time.

7.3 Future Directions

Future work following from this thesis can go in two directions, the first relating to the concurrent genetic programming approach, and the second relating to the design of autonomous agents in general.

With respect to concurrent genetic programming, the thesis has shown both that it works and that it can perform better than traditional genetic programming on the virtual robotics problems. The next step would be to investigate how the approach fares on other agent problems, for example, wall following, the pursuer-evader problem, maze exploration, and other control problems. The key motivation of further inquiry would be to attack those problems for which no analytical solution exists, with the aim of investigating the generality and robustness of the concurrent approach.

With respect to the design of autonomous agents in general, the question of how evolutionary approaches can be applied to the development of explicit world models remains open. While the implicit state afforded by cyclical behaviour may produce good results in some environments, it does not offer a general scheme for the persistent representation of salient aspects of the environment within the agent. This suggests two paths of inquiry: first, to investigate and apply existing approaches to those environments where cyclical behaviour can succeed, and second, to identify those environments where cyclical behaviour cannot succeed, thus forcing the evolutionary search to develop persistent, explicit representations of objects in the real world.

Appendix A

Evolution Engine

The concurrent genetic programs created in Chapter 5 were evolved using the author's Evolution Engine framework (*Ee*), an extensible object-oriented framework for the evolution of arbitrary structures. *Ee* was designed to be open, that is, the evolutionary algorithm is abstracted away from the structures under evolution, so that the same framework can be used to evolve different representations, from the binary strings of genetic algorithms to the tree structures of genetic programming and concurrent genetic programming. The *Ee* framework has also been used to evolve genetic programs, concurrent genetic programs [71, 72] and a novel representation for university timetables [82].

Design patterns were used to create a clean object-oriented design [19]. *Ee* was implemented using C++ using the GNU software `g++`, `gmake`, `flex` and `bison`, developed over the LINUX and Solaris operating systems.

A.1 Core *Ee* classes and interfaces

This extraction of evolutionary algorithm from the structures being evolved is provided through a set of core classes and interfaces, shown in Figure A.1. An `Individual` is an interface describing objects with the operations `mutate()`, `crossover()` and

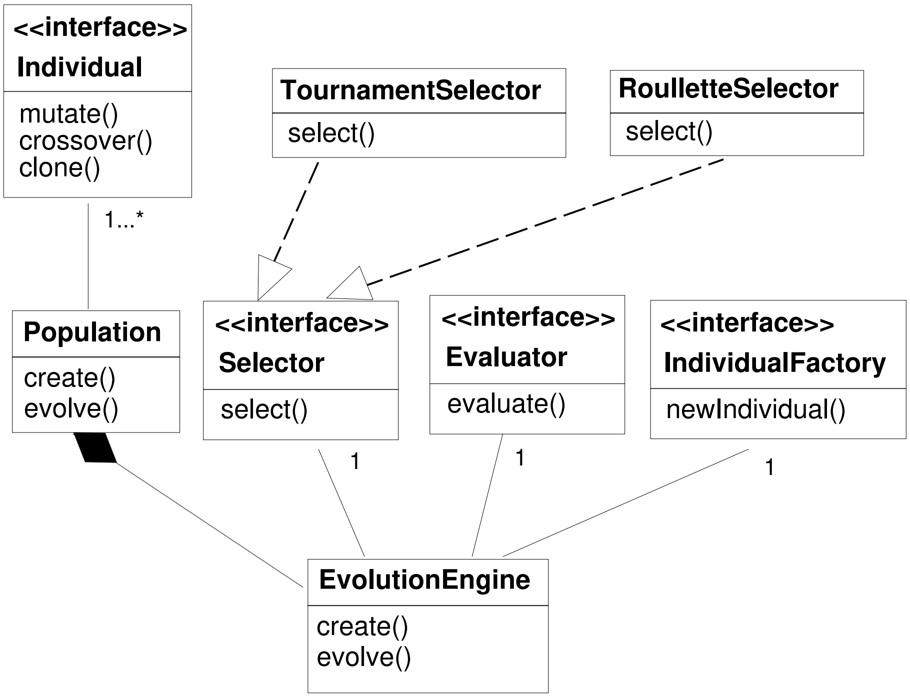

Figure A.1: Core classes and interfaces of the *Ee* framework, in UML notation

`clone()`. A list of `Individuals` is maintained by the concrete class `Population`. The `Population` uses an object described by the `IndividualFactory` interface to create the individuals in the initial random creation stage. The scheme follows the **Factory Method** design pattern [19, page 107]. During evolution, a functional object described by the `Selector` interface is used as a selection algorithm to select the best individuals for reproduction, crossover and mutation. This approach follows the **Strategy** pattern [19, page 315]. Individuals are evaluated by sending them to an object implementing the `Evaluator` interface. By abstracting away the evaluation of an individual from the individual itself, the framework allows for the distributed evaluation of individuals, where evaluator objects may reside on remote more powerful machines.

Two `Selectors` are provided with the *Ee* distribution, `TournamentSelector` and `RouletteSelector`, providing tournament and roulette-wheel selection strategies respectively. The user may use these or design a custom `Selector`, for example, to perform some form of elite selection strategy. The interfaces a user *must* implement are `Individual`, `IndividualFactory` and `Evaluator`: in the next sections the implementations for genetic programming and concurrent genetic programming are described.

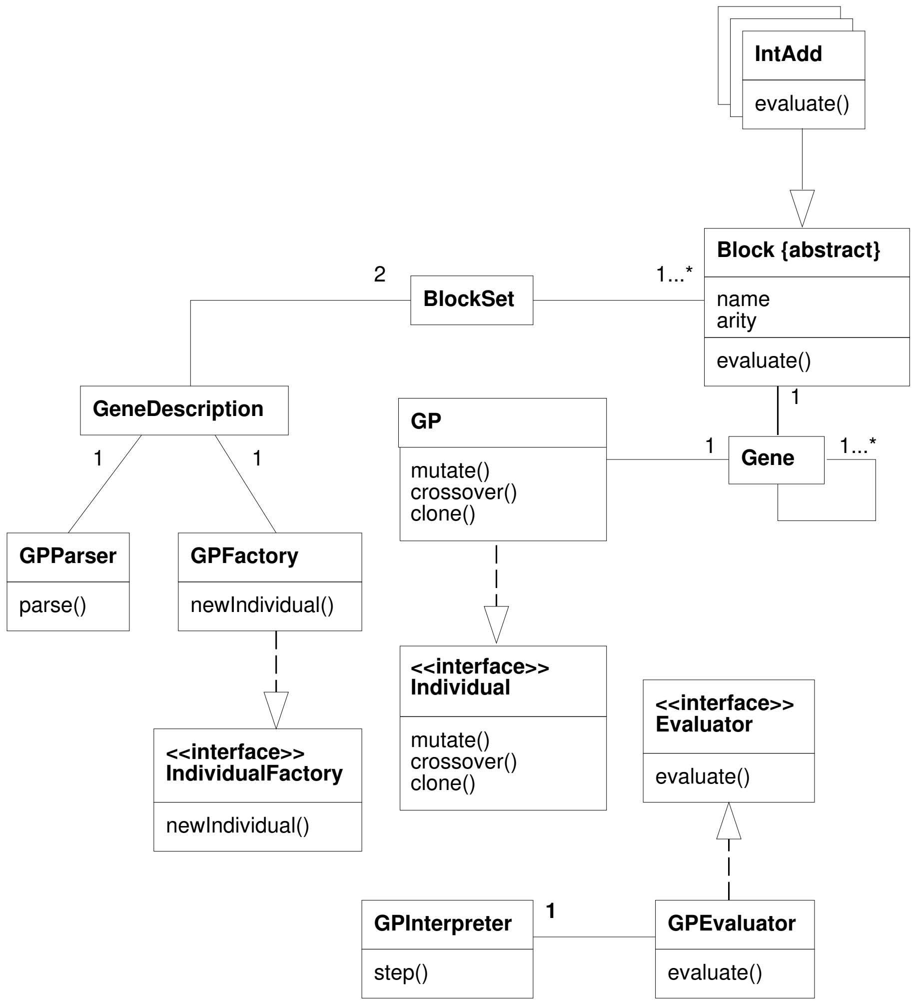

Figure A.2: Object relationship diagram for the GP integration with *Ee*

A.2 Genetic Programming

An object relationship diagram for the genetic programming components of *Ee* is shown in Figure A.2.

The class `GP`, implementing the `Individual` interface, is used to represent genetic programs. A `GP` contains a reference to a single `Gene`, the root node of the program tree. Every `Gene` has a (possibly zero-length) list of `Genes`, that is, the children of the `Gene`. Each `Gene` contains a reference to a `Block`, an object describing either a function (for example, `Add`, `Mul` or `If-Then-Else`), or a terminal (for example `PI`, `X` or a constant `13`). The `IntAdd` block is shown: this block inherits from `Block` providing functionality that returns the sum of the `Gene`'s children when the `evaluate()` method is called.

`Block` objects are kept in a container called a `BlockSet`. A `GeneDescription` describes the material that a program tree can be built from, and how program trees are created. Each `GeneDescription` has references to two distinct `BlockSets`, one containing functions and the other containing terminals. The `GeneDescription` thus contains essential information for the `GPFactory` class, used to randomly create `GPs`. `GPFactory` implements the `IndividualFactory` interface and so plugs in to the *Ee* framework. `GeneDescription` is also used by the class `GPParser`, used to create `GP`'s from genetic-programs in human-readable ASCII format.

The class `GPEvaluator` realises the `Evaluator` interface, and uses a `GPInterpreter` object to `step()` through the program tree evaluating each node in depth-first left-right traversal of the tree.

A.3 Concurrent Genetic Programming

The concurrent genetic programming integration with *Ee* uses the genetic programming objects from the previous section. The object relationship diagram is shown in Figure A.3.

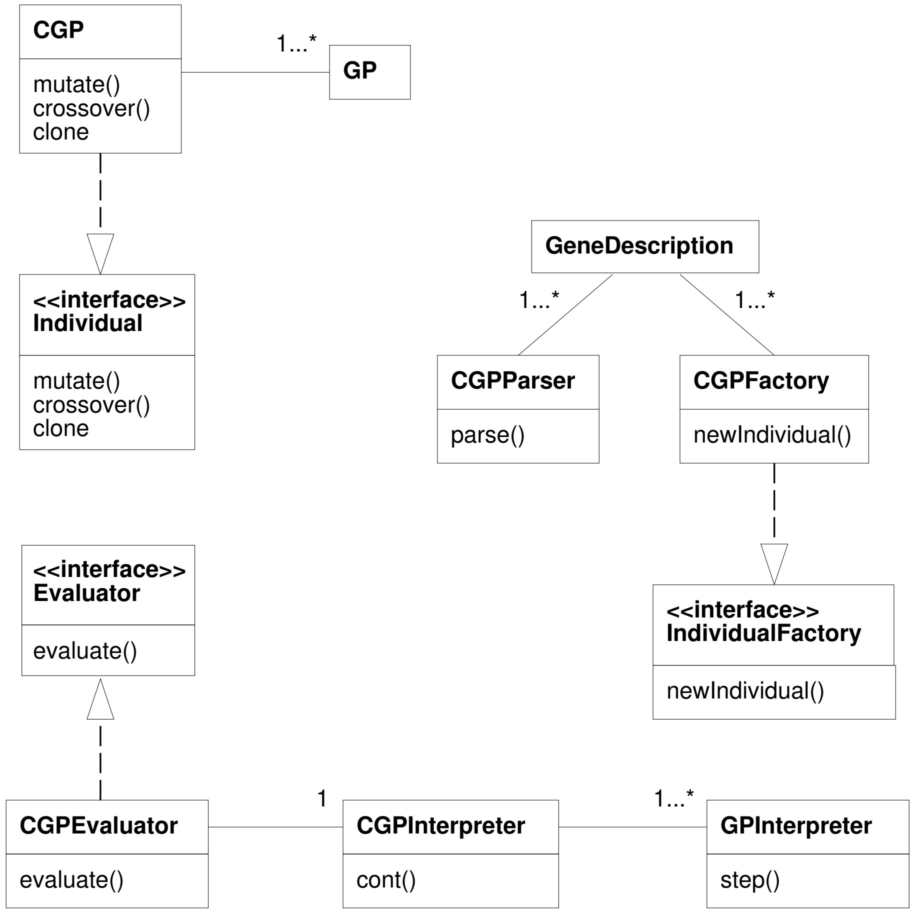

Figure A.3: Object diagram showing the CGP integration with *Ee*.

The interfaces `Individual`, `Evaluator` and `CGPEvaluator` are realised by the classes `CGP`, `CGPEvaluator` and `CGPFactory`. A `CGP` contains one or more genetic programs, that is, `GP`s. The `CGP` objects are created using either a `CGPFactory` (realising the `IndividualFactory` interface) or through the `CGPParser`, both of which maintain one or more `GeneDescription` objects, one for each of the trees in the `CGP`.

The `CGPEvaluator` maintains a `CGPInterpreter`, which itself contains a number of `GPInterpreter` objects, one for each of the genetic programs to be evaluated.

Appendix B

Results and Tableaux

For completeness, the full results and tableaux for the application of determinstic and non-deterministic concurrent genetic programming to Diagarus, Tartarus and Dozer are presented here.

B.1 Non-deterministic CGP

B.1.1 Tableaux

A Koza-style tableaux for the experiments applying non-deterministic concurrent genetic programming to the Diagarus problem is shown in Table B.1.

B.1.2 Results

Results for Teller's GP agents with and without indexed memory on Diagarus are presented in Table B.2. Results for the non-deterministic concurrent genetic programming and Tartarus are shown in Table B.3.

B.2 Deterministic CGP

B.2.1 Tableaux

A Koza-style tableau for the deterministic GGP experiments is presented in Table B.4.

B.2.2 Results

Tables B.5 to B.8 provide a full listing of the results for the application of deterministic CGP to the Tartarus and Dozer worlds.

Objective	Investigate whether concurrently executing agents using a number of different memory structures and scheduling policies can perform better than conventional GP agents.
Fitness Cases	Each agent is evaluated in 40 environments in which it and 5 blocks are placed randomly in the world.
Raw Fitness	The sum of points earned in the 40 test cases. 1 point is earned for a block against a wall, 2 points for a block in the corner. Maximum fitness is thus 360 points.
Wrapper	($0 \leq$ turn left ≤ 6), ($7 \leq$ turn right ≤ 13), ($14 \leq$ move forward ≤ 19).
GP Parameters	Population size = 800, Max. Generations = 100, Create Method = RampHalfAndHalf, Tournament Size = 7, Mutation = 0.05.
Success Predicate	Agent achieves maximum fitness, ie. 360.
Overall Program Structure	Two result producing branches, each of which produces an action according to the wrapper above. No ADFs are employed.
Explicit Memory Constructs	Agents use a size 20 indexed memory, either local or shared. Terminals to access memory are ReadL, WriteL, ReadS and WriteS.
Scheduling Functions	Yield - transfers processor to other tree: when execution is resumed it evaluates and returns its argument.
Terminal Set	0..19, Sense0, Sense11, ... , Sense7 (0..7 correspond to the compass directions NW, N, ... , W).
Function Set	Not, Or, IfThenElse, Less, Add, Sub, Eq.

Table B.1: Tableau describing the experimental set-up for the application of non-deterministic CGP to Diagarus.

Agent	Ave.	Max.	Std. Dev.
Single tree, no memory	63.21	98	14.46
Single tree + memory	176.5	212	26.20
Single tree with ADF + memory	165.10	207	25.33

Table B.2: Results for single-tree agents on the Diagarus problem without memory, with memory and with automatically defined function and memory. The maximum possible score is 360 points.

Scheduling	Memory	Ave.	Max.	Std. Dev.
freeYield	One tree has Local	197	215	7.62
	No memory	194	214	13.84
	Both have Local	185	197	16.19
	Both have Shared	189.8	212	14.95
	Both have Local + Shared	186.8	207	32.25
implicitYield	One tree has Local	205	221	11.13
	No memory	198.15	216	13.6
	Both have Local	195.47	218	13.01
	Both have Shared	201.45	231	12.71
	Both have Local + Shared	198.25	224	12.45
cycleYield	One tree has Local	197.6	206	7.14
	No memory	191.11	206	11.12
	Both have Local	189.63	211	16.42
	Both have Shared	192.42	214	13.43
	Both have Local + Shared	189.16	210	17.01
fullYield	One tree has Local	187.85	209	15.85
	No memory	196.85	216	10.23
	Both have Local	190.5	210	13.24
	Both have Shared	187.55	205	13.55
	Both have Local + Shared	192.84	209	12.57
noYield	One tree has Local	198.75	214	8.01
	No memory	195.75	206	8.78
	Both have Local	193.1	209	13.06
	Both have Shared	195.85	216	12.84
	Both have Local + Shared	187.79	205	13.79

Table B.3: Results for 25 non-deterministic CGP experiments on the Diagarus problem. The maximum possible score is 360 points.

Objective	To examine the application of concurrent genetic programming to the Tartarus and Dozer worlds as described in Section 6.2, in particular, the effect of varying the number of program trees in an individual, level of concurrency and use of explicit state.
Fitness Cases	Individuals are measured against 40 worlds randomly generated in each generation.
Raw Fitness	The sum of points earned in the 40 fitness cases. The maximum raw fitness for Tartarus is 400; for Dozer maximum raw fitness is 1200
Wrapper	When a program within an individual completes evaluation, its result is in the range 0–19. If the result is in range 0–6 then the agent turns left, 7–13 then the agent turns right, 14–19 then the agent moves forward.
GP Parameters	Population size = 800, maximum generations = 50, creation type is ramp-half-and-half, tournament selection is used with tournament size = 7, crossover occurs at 90%, reproduction at 9% and mutation at 1%.
CGP Parameters	Program energy set at either 1, 2, 5, 10 or infinite. `Yield` function allows agents to modify behaviour of scheduler (See Section 5.5.2.
Overall Individual Structure	Each program is made up of 1, 2, 3, 4 or 5 program trees.
Success Predicate	An individual scores 400 points for Tartarus, 1200 points for Dozer.
Explicit Memory Constructs	A size 20 linear indexed memory is used, accessed using the functions `ReadSM` and `WriteSM`.
Terminal Set	Integers `0-19`, sensors `Sense0`–`Sense7`
Function Set	`Not`, `Or`, `IfThenElse`, `Less`, `Add`, `Sub`, `Eq`, `ReadSM`, `WriteSM`, `Yield`

Table B.4: Tableau describing concurrent genetic programming set-up for Tartarus and Dozer worlds.

Trees	Energy	Fitness	Mean	Std. Dev.	Score
1	Infinite	884	1008.75	69.21	7.20
2	1	680	726.45	14.78	10.38
	2	681	715.05	23.27	11.58
	5	672	721.00	28.55	11.63
	10	609	737.70	46.91	13.30
	Infinite	734	809.60	44.65	10.57
3	1	666	700.25	12.83	12.63
	2	649	696.00	14.30	11.86
	5	631	690.65	21.67	12.67
	10	636	691.25	23.85	12.58
	Infinite	735	818.85	59.56	9.98
4	1	677	698.55	8.68	11.99
	2	660	698.35	13.24	11.57
	5	683	696.50	9.84	11.46
	10	674	691.60	12.12	12.21
	Infinite	706	827.55	60.95	11.46
5	1	671	708.60	13.32	12.31
	2	669	702.30	12.38	12.26
	5	657	696.10	12.89	12.28
	10	672	691.55	10.35	11.40
	Infinite	712	832.65	59.64	11.29

Table B.5: Results of the 21 Dozer experiments where agents had no memory. Each experiment was run 20 times with different initial random seeds: the table shows the best fitness achieved over the 20 runs. This is cast as *standardised* fitness, where a lower value closer to 0 indicates a better agent. Also shown is the mean and standard deviation of best fitness for each run. The most telling metric for each experiment is the *Score*, which is the average number of points scored by the best agent over 5000 randomly created boards.

Trees	Energy	Fitness	Mean	Std. Dev.	Score
1	Infinite	622	721.70	74.58	12.45
2	1	658	715.15	48.53	11.75
	2	688	716.15	15.68	11.56
	5	666	708.05	16.84	12.09
	10	652	707.45	30.76	12.41
	Infinite	609	717.55	58.53	13.96
3	1	664	707.50	13.92	12.35
	2	649	688.40	24.68	12.01
	5	645	680.85	19.58	12.01
	10	632	685.50	25.56	12.77
	Infinite	667	739.50	31.76	11.59
4	1	665	697.35	12.05	10.52
	2	654	700.75	15.55	12.51
	5	629	692.60	20.59	12.92
	10	650	693.60	14.19	12.28
	Infinite	614	734.70	41.81	12.95
5	1	679	712.15	13.11	10.73
	2	676	705.35	14.05	11.69
	5	673	702.00	12.84	11.16
	10	658	693.70	13.38	11.07
	Infinite	676	740.20	26.42	11.68

Table B.6: Results of the 21 Dozer experiments where agents had memory. Each experiment was run 20 times with different initial random seeds: the table shows the best fitness achieved over the 20 runs. This is cast as *standardised* fitness, where a lower value closer to 0 indicates a better agent. Also shown is the mean and standard deviation of best fitness for each run. The most telling metric for each experiment is the *Score*, which is the average number of points scored by the best agent over 5000 randomly created boards.

Trees	Energy	Fitness	Mean	Std. Dev.	Score
1	Infinite	345	359.35	5.23	0.69
2	1	192	232.65	21.88	4.07
	2	211	234.35	13.24	3.61
	5	150	227.60	23.41	5.89
	10	160	233.40	22.91	4.85
	Infinite	220	259.00	29.00	3.59
3	1	181	217.35	11.40	4.39
	2	207	218.05	7.57	3.39
	5	179	216.45	12.62	4.61
	10	193	219.05	10.43	4.47
	Infinite	265	286.70	18.28	2.72
4	1	214	223.90	6.97	3.57
	2	204	216.55	7.77	3.87
	5	194	213.65	6.84	4.11
	10	197	215.25	5.87	4.18
	Infinite	212	270.00	19.84	3.94
5	1	204	221.55	5.58	3.23
	2	210	221.15	6.14	3.34
	5	183	214.90	11.69	5.22
	10	199	215.75	6.06	4.33
	Infinite	208	264.05	22.37	3.87

Table B.7: Results of the 21 Tartarus experiments where agents had no memory. Each experiment was run 20 times with different initial random seeds: the table shows the best fitness achieved over the 20 runs. This is cast as *standardised* fitness, where a lower value closer to 0 indicates a better agent. Also shown is the mean and standard deviation of best fitness for each run. The most telling metric for each experiment is the *Score*, which is the average number of points scored by the best agent over 5000 randomly created boards.

Trees	Energy	Fitness	Mean	Std. Dev.	Score
1	Infinite	219	249.55	15.65	3.85
2	1	171	215.45	21.65	4.98
	2	181	208.25	17.99	4.36
	5	171	212.60	21.81	4.91
	10	165	203.65	10.94	5.02
	Infinite	157	203.25	26.00	5.58
3	1	204	221.30	13.90	3.87
	2	203	218.35	7.43	4.01
	5	191	210.85	10.00	4.27
	10	184	211.30	13.29	4.50
	Infinite	191	229.05	20.47	4.48
4	1	213	218.75	7.87	3.85
	2	210	219.25	5.73	3.63
	5	190	216.70	8.20	4.48
	10	204	216.95	5.38	4.13
	Infinite	160	224.80	25.82	5.02
5	1	213	222.75	5.24	3.89
	2	210	221.60	5.79	3.28
	5	178	214.25	12.74	4.97
	10	179	209.00	13.61	4.48
	Infinite	202	224.10	11.51	4.05

Table B.8: Results of the 21 Tartarus experiments where agents had memory. Each experiment was run 20 times with different initial random seeds: the table shows the best fitness achieved over the 20 runs. This is cast as *standardised* fitness, where a lower value closer to 0 indicates a better agent. Also shown is the mean and standard deviation of best fitness for each run. The most telling metric for each experiment is the *Score*, which is the average number of points scored by the best agent over 5000 randomly created boards.

Appendix C

Published Work

A number of papers describing work in the thesis have been published, their full references are below:

Adrian Trenaman. Further investigations into the evolution of agents with concurrent genetic programming. In W. Banzhaf, J. Daida, A. Kiben, M. Garzon, V. Honovar, M. Jakiela, and R. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, July 13-17 1999. San Francisco, CA: Morgan Kaufmann.

Adrian Trenaman. Concurrent genetic programming, Tartarus and dancing agents. In Peter Nordin, editor, *EuroGP '99, Proceedings of the* 2^{nd} *European Workshop on Genetic Programming*, Goteburg, Sweden, 1999. Springer Verlag.

Damien Woods and Adrian Trenaman. Simultaneous Satisfaction of Hard and Soft Timetable Constraints for a University Department Using Evolutionary Timetabling. In Derek Bridge, editor, *Proceedings of the* 10^{th} *Irish Conference on Artificial Intelligence & Cognitive Science*, Cork, Ireland, 1999.

Adrian Trenaman. Concurrent genetic programming and the use of explicit state to solve problems in partially-known environments. In John R. Koza et al., editors,

Genetic Programming 1998: Proceedings of the Third Annual Conference, Madison, Wisconsin, July 1998. Morgan Kaufmann.

Adrian Trenaman. Towards the evolution of stateful autonomous agents. In *Eight Irish Conference on Artificial Intelligence (AI-97)*, volume 2, pages 56–63. Magee College, University of Ulster, LondonDerry, 1997.

Bibliography

[1] David Andre. Artificial evolution of intelligence: Lessons from natural evolution. Master's thesis, Stanford University, 1994.

[2] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John Koza et al., editors, *Genetic Programming 1997 - Proceedings of the Second Annual Conference*, pages 9–19, Stanford, California, July 1997. Morgan Kaufmann.

[3] Dan Ashlock and Mark Joenks. ISAc lists, a different representation for program induction. In John Koza et al., editors, *Genetic Programming: Proceedings of the Third Annual Genetic Programming Conference*, pages 3–10, Madison, Wisconsin, 1998. Morgan Kaufmann.

[4] Wolfgang Banzhaf, Peter Nordin, and Markus Olmer. Generating adaptive behaviour for a real robot. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the Second Annual Conference*, pages 35–43, Stanford, California, July 1997. Morgan Kaufmann.

[5] Randall D. Beer and John C. Gallagher. Evolving dynamical neural networks for adaptive behaviour. *Adaptive Behaviour*, 1:91–122, 1992.

[6] Scott Brave. The evolution of memory and mental models using genetic programming. In John R. Koza et al., editors, *Genetic Programming - Proceedings*

of the First Annual Conference, pages 261–266, Stanford, California, July 1996. MIT Press.

[7] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, (47):139–159, 1991.

[8] Rodney A. Brooks. Artificial life and real robots. In Francisco J. Varela and Paul Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 3–10, Cambridge, Massachusetts, 1992. MIT Press.

[9] Wilker Shane Bruce. Automatic generation of object-oriented programs using genetic programming. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 267–272, Stanford, California, July 1996. MIT Press.

[10] Wilker Shane Bruce. The lawnmower problem revisited: Stack-based genetic programming and automatically designed functions. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the Second Annual Conference*, pages 52–57, Stanford, California, July 1997. Morgan Kaufmann.

[11] D. Cliff, I. Harvey, and P. Husbands. Explorations in evolutionary robotics. *Adaptive Behaviour*, 2(1):73–110, 1993.

[12] Dave Cliff and Geoffrey F. Miller. Coevolution of pursuit and evasion 2: simulation methods and results. In P. Maes, M. Mataric, j. A. Meyer, J. Pollack, and S. W. Wilson, editors, *From Animals to Animats 4: Proceedings of the* 4^{th} *International Conference on Simulation of Adaptive Behaviour*, San Francisco, California, July 1996. MIT Press, Bradford Books.

[13] Mark Crosbie and Eugene H. Spafford. Evolving event-driven programs. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 273–178, Stanford, California, July 1996. MIT Press.

[14] Anthony G. Deakin and Derek F. Yates. Evolving and optimizing autonomous agents' strategies with genetic programming. In John R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 42–47, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[15] Dimitrus C. Dracopoulos. Evolutionary control of a satellite. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the Second Annual Conference*, pages 77–81, Stanford, California, July 1997. Morgan Kaufmann.

[16] G. Dworman, S. O. Kimbrough, and J. D. Laing. Bargaining by artificial agents in two coalition games: A study in genetic programming for electronic commerce. In John R. Koza et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford, California, 1996. MIT Press.

[17] David B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, New Jersey, USA, 1995.

[18] Laurence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, Inc., New York, 1966.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[20] Martin Gardner. Mathematical games – the fantastic combinations of john conway's new solitaire game *life*. *Scientific American*, February 1971.

[21] David E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison Wesley, 1992.

[22] Fredéric Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the* 5^{th} *International Conference on Genetic Algorithms*, pages 318–325, 1993.

[23] Fredéric Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, l'Université Claude Bernard-Lyon 1, 1994.

[24] Inman Harvey. Species adaptation genetic algorithms: A basis for a continuing saga. In Francisco J. Varela and Paul Bourgine, editors, *Towards A Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*, pages 346–354, Cambridge, Massachusetts, 1991. MIT Press.

[25] Thomas Haynes. On-line adaptation of search via knowledge reuse. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the Second Annual Conference*, pages 156–161, Stanford, California, July 1997. Morgan Kaufmann.

[26] Thomas Haynes and Sandip Sen. Crossover operators for evolving a team. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the Second Annual Conference*, pages 162–167, Stanford, California, July 1997. Morgan Kaufmann.

[27] J. H. Holland. *Adaptation of Natural and Artificial Systems*. The University of Michigan Press, 1975.

[28] Naohiro Hondo, Koji Nishikawa, Hiroshi Yokoi, and Yukinori Kakazu. Multi-agent programming system for starfish robot control. In John R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 140–145, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[29] Brian Howley. Genetic programming of near-minimum-time spacecraft attitude manouvres. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 98–106, Stanford, California, July 1996. MIT Press.

[30] P. Husbands, I. Harvey, D. Cliff, and G. Miller. The use of genetic algorithms for the development of sensorimotor control systems. In *From Perception to Action*, pages 318–325. IEEE, 1994.

[31] Phil Husbands. Evolving robot behaviours with diffusing gas networks. In Philip Husbands and Jean-Arcady Meyer, editors, *Evolutionary Robotics '98: Proceedings of the First European Workshop*, pages 137–151. Springer, 1998.

[32] Hitoshi Iba. Multiple-agent learning for a robot navigation task by genetic programming. In John Koza et al., editors, *Genetic Programming 1997 - Proceedings of the Second Annual Conference*, pages 195–200, Stanford, California, July 1997. Morgan Kaufmann.

[33] Forrest H. Bennett III. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 30–38, Stanford, California, July 1996. MIT Press.

[34] Forrest H. Bennett III. A multi-skilled robot that recognizes and responds to different problem environments. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the Second Annual Conference*, pages 44–51, Stanford, California, July 1997. Morgan Kaufmann.

[35] Nick Jacobi. Evolutionary robotics and the radical-envelope-of-noise hypothesis. *Adaptive Behaviour*, 6:131–174, 1997.

[36] Nick Jacobi. Running across the reality gap: Octopod locomotion evolved in a minimal simulation. In Philip Husbands and Jean-Arcady Meyer, editors, *Evolutionary Robotics '98: Proceedings of the First European Workshop*, pages 40–58. Springer, 1998.

[37] Nick Jacobi and Matthew Quinn. Some problems (and a few solutions) for open-ended evolutionary robotics. In Philip Husbands and Jean-Arcady Meyer, editors, *Evolutionary Robotics '98: Proceedings of the First European Workshop*, pages 137–151. Springer, 1998.

[38] Anthony Kenny, editor. *The Oxford Illustrated History of Western Philosophy.* Oxford Universtiy Press, 1994.

[39] John R. Koza. Evolution of subsumption using genetic programming. In Francisco J. Varela and Paul Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, Cambridge, Massachusetts, 1992. MIT Press.

[40] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, Massachusetts, 1992.

[41] W. B. Langdon. Using data structures within genetic programming. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 141–149, Stanford, California, July 1996. MIT Press.

[42] William Benjamin Langdon. *Genetic Programming and Data Structures.* PhD thesis, University of London, 1996.

[43] Christopher G. Langton. Artificial life. In Christopher G. Langton, editor, *Artificial Life - Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems.* Addison-Wesley, 1989.

[44] Christopher G. Langton, editor. *Artificial Life - Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems.* Addison-Wesley, 1989.

[45] Sean Luke. Genetic programming produced competitive soccer softbot teams for robocup '97. In John R. Koza et al., editors, *Genetic Programming 1998:*

Proceedings of the Third Annual Conference, pages 214–222, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[46] Sean Luke and Lee Spector. Cultural transmission of information in genetic programming. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 209–214, Stanford, California, July 1996. MIT Press.

[47] Sean Luke and Lee Spector. Evolving teamwork and co-ordination with genetic programming. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 150–156, Stanford, California, July 1996. MIT Press.

[48] Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. In John Koza et al., editors, *Genetic Programming 1997 - Proceedings of the Second Annual Conference*, pages 240–248, Stanford, California, July 1997. Morgan Kaufmann.

[49] Sena Luke and Lee Spector. A revised comparison of crossover and mutation in genetic programming. In John R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[50] David E. Marr. *Vision*, chapter 1, pages 19–31. W. H. Freeman, San Francisco, California, 1982.

[51] Jean-Arcady Meyer, Phil Husbands, and Inman Harvey. Evolutionary robotics: A survey of applications and problems. In Philip Husbands and Jean-Arcady Meyer, editors, *Evolutionary Robotics '98: Proceedings of the First European Workshop*, pages 1–21. Springer, 1998.

[52] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384, San Mateo, California, 1990. Morgan Kaufmann.

[53] Marvin Minsky. *The Society of Mind.* Simon & Schuster, New York, 1985.

[54] Frank W. Moore. Genetic programming solves the three-dimensional missile countermeasures optimisation problem under uncertainty. In John R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 242–245, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[55] Frank W. Moore and Oscar N. Garcia. A genetic programming approach to strategy optimisation in the extended two-dimensional pursuer/evader problem. In John Koza et al., editors, *Genetic Programming 1997 - Proceedings of the Second Annual Conference*, pages 249–254, Stanford, California, July 1997. Morgan Kaufmann.

[56] Norman Paterson and Mike Livesey. Evolving caching algorithms in c by genetic programming. In John Koza et al., editors, *Genetic Programming 1997 - Proceedings of the Second Annual Conference*, pages 262–267, Stanford, California, July 1997. Morgan Kaufmann.

[57] Riccardo Poli and William B. Langdon. A new schema theory fo genetic programming with one-point crossover and point mutation. In John Koza et al., editors, *Genetic Programming 1997 - Proceedings of the Second Annual Conference*, pages 278–, Stanford, California, July 1997. Morgan Kaufmann.

[58] K. Price and R. Storn. Differential evolution. *Dr. Dobb's Journal*, April 1997.

[59] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Developmental models of herbaceous plants for computer imagery purposes. *ACM Computer Graphics*, 22(4), 1984.

[60] Adil Qureshi. Evolving agents. In John R. Koza et al., editors, *Genetic Programming - Proceedings of the First Annual Conference*, pages 369–374, Stanford, California, July 1996. MIT Press.

[61] Simon E. Raik and David G. Browne. Implicit versus explicit: A comparison of state in genetic programming. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming Conference 1996*, pages 151–169, Stanford University, July 1996. Stanford University Bookstore.

[62] Craig. W. Reynolds. Flocks, herds and schools: a distributed behavioural model. *Computerr Graphics*, 21(4):25–34, 1987.

[63] Craig W. Reynolds. Evolution of corridor following behaviour in a noisy world. In *Simulation of Adaptive Behaviour*, 1994.

[64] Conor Ryan. *Reducing Premature Convergence in Evolutionary Algorithms.* PhD thesis, University College, Cork, Ireland, 1996.

[65] J. Searle. Minds, brains and programs. *Behavioural and Brain Sciences*, 3:417–457, 1980.

[66] Sandip Sen. Multiagent systems: milestones and new horizons. *Trends in Cognitive Sciences*, 1(9):334–340, December 1997.

[67] David G. Stork. *Hal's Legacy.* MIT Press, 1997.

[68] Astro Teller. The evolution of mental models. In Kim Kinnear, editor, *Advances in Genetic Programming*, pages 199–220. MIT Press, 1994.

[69] Astro Teller. Genetic programming, indexed memory, the halting problem and other curiosities. In *The 7th Annual Florida AI Research Symposium*, pages 270–274. IEEE Press, 1994.

[70] Adrian Trenaman. Concurrent genetic programming and the use of explicit state to solve problems in partially-known environments. In John R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[71] Adrian Trenaman. Concurrent genetic programming, Tartarus and dancing agents. In Peter Nordin, editor, *EuroGP '99, Proceedings of the* 2^{nd} *European Workshop on Genetic Programming*, Goteburg, Sweden, 1999. Springer Verlag.

[72] Adrian Trenaman. Further investigations into the evolution of agents with concurrent genetic programming. In W. Banzhaf, J. Daida, A. Kiben, M. Garzon, V. Honovar, M. Jakiela, and R. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, July 13-17 1999. San Francisco, CA: Morgan Kaufmann.

[73] Alan M. Turing. Computing machinery and intelligence. *Mind*, (59):433–460, 1951.

[74] Honovar V. and Balakrishnan K. On sensor evolution in robotics. In John Koza et al., editors, *Genetic Programming: Proceedings of the First Annual Conference*, pages 455–460, Stanford, California, 1996. MIT Press.

[75] F. Varela. *Principles of Biological Autonomy*. North Holland, New York, 1979.

[76] Francisco J. Varela. Autopoiesis and a biology of intentionality. In Barry McMullin and Noel Murphy, editors, *Autopoiesis & Perception*. Dublin City University, Technical Report `bmcm9401`, 1994.

[77] David Vernon and Dermot Furlong. Relativistic ontologies, self-organisation, autopoiesis and artificial life: Part 1 – the philosophical foundations. In Barry McMullin and Noel Murphy, editors, *Autopoiesis & Perception*. Dublin City University, Technical Report `bmcm9401`, 1994.

[78] John von Neumann. *Theory of Self Reproducing Automata*. University of Illinois Press, 1966.

[79] S. W. Wilson. Knowledge growth in an artificial animal. In John J. Grefenstette, editor, *Proceedings of the first International Conference on Genetic Algorithms and their applications*, pages 16–23, London, 1988. Laurence Erlbaum Associates.

[80] Steward W. Wilson. The animat path to AI. In J. A. Mayer, editor, *From Animals to Animats - Proceedings of the first International Conference on the Simulation of Adaptive Behaviour*, pages 15–21, 1991.

[81] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley Publishing Company Inc., New York, 1986.

[82] Damien Woods and Adrian Trenaman. Simultaneous satisfaction of hard and soft timetable constraints for a university department using evolutionary timetabling. In Derek Bridge, editor, *Proceedings of the* 10^{th} *Irish Conference on Artificial Intelligence & Cognitive Science*, University College Cork, Ireland, 1999.

[83] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practise. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[84] M. Zeleny, G. J. Klir, and K. D. Hufford. Precipitation membranes, osmotic growths and synthetic biology. In Christopher G. Langton, editor, *Artificial*

Life: proceedings of an interdisciplinary workshop on the synthesis and simulation of living system, pages 1–48. Addison-Wesley, 1989.