THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

In situ Distributed Genetic Programming: An Online Learning Framework for Resource Constrained Networked Devices

PHILIP JUAN VALENCIA

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2015

School of Information Technology and Electrical Engineering

Typeset in LaTeX 2_ε.

Abstract

This research presents In situ Distributed Genetic Programming (IDGP) as a framework for distributively evolving logic while attempting to maintain acceptable average performance on highly resource-constrained embedded networked devices.

The framework is motivated by the proliferation of devices employing microcontrollers with communications capability and the absence of online learning approaches that can evolve programs for them. Swarm robotics, Internet of Things (IoT) devices including smart phones, and arguably the most constrained of the embedded systems, Wireless Sensor Networks (WSN) motes, all possess the capabilities necessary for the distributed evolution of logic - specifically the abilities of sensing, computing, actuation and communications. Genetic programming (GP) is a mechanism that can evolve logic for these devices using their “native” logic representation (i.e. programs) and so technically GP could evolve any behaviour that can be coded on the device.

IDGP is designed, implemented, demonstrated and analysed as a framework for evolving logic via genetic programming on highly resource-constrained networked devices in real-world environments while achieving acceptable average performance.

Designed with highly resource-constrained devices in mind, IDGP provides a guide for those wishing to implement genetic programming on such systems. Furthermore, an implementation on mote class devices is demonstrated to evolve logic for a time-varying sense-compute-act problem and another problem requiring the evolution of primitive communications. Distributed evolution of logic is also achieved by employing the Island Model architecture, and a comparison of individual and distributed evolution (with the same and

slightly different goals) presented. This demonstrates the advantage of leveraging the fact that such devices often reside within networks of devices experiencing similar conditions.

Since GP is a population-based metaheuristic which relies on the diversity of the population to achieve learning, many, if not most, programs within the population exhibit poor performance. As such, the average observed performance (pool fitness) of the population using the standard GP learning mechanism is unlikely to be acceptable for online learning scenarios. This is suspected to be the reason why no previous attempts have been made to deploy standard GP as an online learning approach. Nonetheless, the benefits of GP for evolving logic on such devices are compelling and motivated the design of a novel satisficing heuristic called Fitness Importance (FI). FI is population-based heuristic used to bias the evaluation of candidate solutions such that an “acceptable” average fitness (AAF) is achieved while also achieving ongoing, though diminished, learning capacity. This trade off motivated further investigation into whether dynamically adjusting the average performance in response to AAF would be superior to a constant, balanced, performing-learning approach. Dynamic and constant strategies were compared on a simple problem where the AAF target was changed during evolution, revealing that dynamically tracking the AAF target can yield a higher success rate in meeting the AAF.

The combination of IDGP and FI offers a novel approach for achieving online learning with GP on highly resource-constrained embedded systems. Furthermore, it simultaneously considers the acceptable average performance of the system which may change during the operational lifetime. This approach could be applied to swarm and cooperative robot systems, WSN motes or IoT devices allowing them to cooperatively learn and adapt their logic locally to meet dynamic performance requirements.

Declaration by Author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.

Publications during candidature

Valencia, P., Haak, A., Cotillon, A., and Jurdak, R. (2014). Genetic programming for smart phone personalisation. *Applied Soft Computing*, 25(0):86-96.

Cotillon, A., Valencia, P., and Jurdak, R. (2012). Android Genetic Programming Framework. In *Lecture Notes in Computer Science*, pages 13-24. Springer Berlin Heidelberg, Berlin, Heidelberg.

Valencia, P., Lindsay, P., and Jurdak, R. (2010). Distributed genetic evolution in WSN. In IPSN'10: Proceedings of the 9th ACM/IEEE International Conference on *Information Processing in Sensor Networks*, pages 13-23, New York, New York, USA. ACM

Valencia, P., Jurdak, R., and Lindsay, P. (2010). Fitness Importance for Online Evolution. In Proceedings of the 12th *Annual Conference Companion on Genetic and Evolutionary Computation*, pages 2117-2118, New York, NY, USA. ACM.

Valencia, P. (2007). In situ genetic programming for wireless sensor networks. In Hazas, M., Boulis, A., and Girod, L., editors, *SenSys 2007 Doctoral Colloquium*, pages 37-41.

Publications included in this thesis

Valencia, P., Lindsay, P., and Jurdak, R. (2010). Distributed Genetic Evolution in WSN. In Proceedings of the 9th ACM/IEEE International Conference on *Information Processing in Sensor Networks*, IPSN '10, pages 13-23, New York, NY, USA. ACM - Incorporated with significant modification and extension into Chapter 3

Philip Valencia	Concept (90%) Implementation (100%) Drafting and writing (70%)
Raja Jurdak	Concept (5%) Drafting and writing (20%)
Peter Lindsay	Concept (5%) Drafting and writing (10%)

Valencia, P., Jurdak, R., and Lindsay, P. (2010). Fitness Importance for Online Evolution. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 2117-2118, New York, NY, USA. ACM - Incorporated with modification and extension into Chapter 3 and Chapter 4

Philip Valencia	Concept (80%) Drafting and writing (90%)
Raja Jurdak	Concept (10%) Drafting and writing (5%)
Peter Lindsay	Concept (10%) Drafting and writing (5%)

Contributions by others to the thesis

No contributions by others.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Acknowledgements

My PhD was motivated by a passion for Artificial Intelligence that grew from working with a number of distinguished scientists at the CSIRO in Sydney who fostered very forward-looking scientific thinking by junior staff. With their inspiration and support from the CSIRO (with particular thanks to Alex Zelinsky, Steve Guigni and Peter Corke), I was able to commence a part-time PhD candidature.

As a part-time PhD student (and full time employee), I have been present “part-time” to my university and work supervisors, project leaders, family and friends. Luckily, I have been fortunate enough to have people in my life who could appreciate my predicament and support me anyway, for which I am extremely grateful. I wish to thank my past PhD supervisors, Professor Peter Corke and Professor Mikhail Prokopenko, and my work supervisors Dr Brano Kusy, Dr Tim Wark and Dr Michael Brünig for their many efforts to align my PhD with my career research. I also thank Mr Les Overs, Mr Stephen Brosnan and Mr John Whitham who designed, sourced, built and repaired various equipment used during my studies.

I am however truly indebted to my current supervisors, Prof. Peter Lindsay who has guided and stuck with me from the very beginning, and Dr Raja Jurdak for his sage-like writing advice and motivation.

Finally to my wife, my children who have only ever known a “part-time PhD dad” and my family and friends who have seen a lot less of me than they ought - *Thank You* - for understanding, supporting and believing in me.

Keywords

genetic programming, embedded systems, internet of things, wsn, online learning, machine learning, metaheuristic, exploration and exploitation, robotics

Australian and New Zealand Standard Research Classifications (ANZSRC):

080101 Adaptive Agents and Intelligent Robotics 40%

080108 Neural, Evolutionary and Fuzzy Computation 30%

080504 Ubiquitous Computing 30%

Fields of Research (FoR) Classification:

FoR Code: 0801 Artificial Intelligence and Image Processing 100%

Contents

List of Figures

List of Tables

1
Overview

This chapter briefly introduces the motivation of this research; specifically the need for a framework to allow resource constrained embedded devices to evolve their logic while simultaneously providing utility throughout their lifetimes. The overarching focus and scope of the thesis is then presented along with the major original contributions of this dissertation. Following this, the thesis structure is presented.

1.1 Motivation

Microcontrollers (MCU) are processors with vastly inferior computational capability compared to that of everyday personal computers. Yet, despite their significant resource constraints, more than 20 billion microcontrollers were sold during 2015, and the number is expected to rise in coming years. Such processors are the preferred choice for the embedded systems that are now ubiquitous throughout our society due to their low price, small size and low power processing.

In recent years, MCU's have been augmented with wireless communications technology and increasingly sensing and actuation capability. This has enabled "networked" embedded systems such as swarm robotics, smart phones, Wireless Sensor Network (WSN) devices and more generally, the Internet of Things (IoT).

The logic for the devices comprising these systems is represented as programs, or code. These instructions allow the device to read sensor values and perform logic to determine what actions or communications should be performed. Such programs are handcrafted by professional programmers who attempt to consider the many real-world conditions these devices will be subject to. Unfortunately, fixed logic can fail for a variety of reasons and will typically remain broken until human intervention occurs. Excluding failure of hardware or software bugs in the code, the most likely reason for failure is an unanticipated change in the environmental conditions or a change in the context to expect different performance. Predicting and adapting to such changes is a complex task since the changes are often context or locality specific which cannot be known in advance by the programmer.

There is however a means for automatically generating programs known as Genetic Programming (GP). GP has been demonstrated as a mechanism for producing human-level competitive behaviours and ideas. However, employing GP on such resource constrained devices would be a significant challenge. GP requires a population of programs to be evaluated over a number of generations in order to evolve programs that can achieve acceptable performance. However, executing a population of various programs with unknown behaviours would be unacceptable in many situations. As such, not only must the

issues of resource constraints be addressed, but a mechanism for achieving acceptable performance while learning is also necessary.

This thesis attempts to address this challenge by answering the question: "**Can logic be evolved via genetic programming on highly resource-constrained networked devices deployed in real-world environments while achieving acceptable average performance?**"

1.2 Thesis Overview

1.2.1 Thesis Question

This thesis aims to address the question:

> Can logic be evolved via genetic programming on highly resource-constrained networked devices deployed in real-world environments while achieving acceptable average performance?

To answer this, the question is broken into the following 3 subquestions:

1. Can distributed GP be implemented on such highly-constrained embedded devices?
2. Can acceptable average performance be achieved when needed while continuing to learn?
3. Can dynamically balancing performing and learning achieve better success at meeting acceptable average performance targets during the system lifetime?

1.2.2 Thesis Outcomes

To answer the first sub-question, a framework for evolving logic on networked, yet resource constrained, devices was formulated. The In situ Distributed Genetic Programming (IDGP) framework (designed and detailed in Chapter 3) was implemented and demonstrated on arguably the most highly resource-constrained embedded class of devices known as WSN motes. The framework was used to demonstrate cooperative evolution

of programs across a network of devices using the Island Model to solve a time varying sensing-actuating task and a task requiring evolution of communications, under environments subject to real-world effects. z Online learning via GP was successfully demonstrated, however the average observed behaviour (performance) of devices however was deemed unacceptable for many situations. To address this, and the second sub-question, a novel heuristic called Fitness Importance (FI) was developed (see Chapter 4). FI is used to bias the selection of candidate solutions evaluated such that an "acceptable" average fitness (AAF) is achieved. Using FI, it was demonstrated that it is possible to continue to learn while attaining AAF (as shown in Chapter 5). However it was also found that the learning capacity decreases as the AAF target is increased.

This contention, captured within sub-question 3, motivated a final investigation (in Chapter 6) into whether dynamically adjusting performance in response to AAF would be superior to a constant balanced approach to performing-learning. Dynamic and constant strategies of balancing learning versus performing were compared on a simple problem where the AAF target was changed during evolution, ultimately revealing that dynamically tracking the AAF target can indeed yield a higher success rate of meeting the AAF.

The combination of IDGP and FI offers a novel approach for achieving online learning with GP on highly resource-constrained embedded systems. It also enables simultaneous learning while attempting to meet an acceptable average performance which may change throughout the operational lifetime. As such, this thesis presents a novel approach that could be applied to small robots, WSN motes or IoT devices to allow them to cooperatively learn and adapt their logic in order to meet dynamic performance requirements.

1.2.3 Thesis Scope and Limitations

From a broad perspective, this research is interested in evolving logic on constrained devices and achieving acceptable performance. There are many avenues one could take to achieve this and so some limitation of research scope is necessary.

First, Genetic Programming (GP) will be used as the logic evolving mechanism. This is largely justified by the fact that GP can automatically generate programs and that the

processors of embedded systems can run programs. Hence, this is an automated means for generating logic. Other approaches will not be investigated.

Second, this thesis focusses on how highly resource-constrained devices might achieve evolution of logic. As such WSN mote devices will be employed as representative of this highly resource-constrained class of embedded systems and devices with significant computational resources (PC-class) will not be considered.

Finally, achieving "acceptable" performance using online machine learning is by no means a new concept. Using GP as an online machine learning mechanism to achieve acceptable performance however has not been extensively studied. Due to the first reduction in scope limiting the learning mechanism to GP, a method for achieving "acceptable" performance with GP will need to be addressed. Further discussion and justifications of the thesis scope appears in 3.1 Scope.

1.3 Methodology

To address the thesis questions, a framework was designed to achieve distributed genetic programming with consideration of the significant computational and memory constraints of mote-class devices. Specifically, the scenario where there is less memory (RAM/ROM) available to store a typical population of programs is considered. The literature survey on related research was conducted to formulate considerations specific to a framework that evolves logic on resource constrained devices. The framework was then implemented on a collection of devices representative of the resource-constrained class of embedded devices to address problems that requires logic to perform time-dependant actions based on real-world sensor inputs. The distributed evolution capability was demonstrated through experiments on multiple devices where devices have the same objective and also where devices have slightly differing objectives. After the learning capability was demonstrated, achieving acceptable performance was then be studied. "Acceptable" performance was defined and an online learning heuristic for achieving acceptable performance was developed. Finally, the implications of applying the heuristic(s) in various modes (constant and dynamic) was studied. Section 1.5 details which chapters relevant to each of these steps.

1.4 Original Contributions

The overarching contribution of this thesis is **a framework (IDGP)** for achieving in situ distributed evolution of logic on highly resource-constrained embedded devices **and a heuristic (FI)** to bias their average performance on demand. Both the IDGP framework and the FI heuristic can be used independently, however, together they offer a framework for distributively evolving logic while achieving an acceptable average performance. This combination should be applicable to swarm robotics, IoT devices such as smart phones and WSN motes, and offer these platforms a mechanism .

A list of more specific contributions by this thesis is as follows:

- In situ Distributed Genetic Programming (IDGP) as a framework for embedded systems engineers to achieve online evolution of logic on resource-constrained devices.
- The Fitness Importance (FI) heuristic as a means of indicating “acceptable average performance” in terms of the current population characteristics as a function of time.
- A metaheuristic for population-based learners which generates a biased population to increase the expected fitness of the population based on an input parameter for the desired improvement and the optimal learning population configuration.
- The extension of the IDGP framework with FI and an analysis of an implementation of the framework on physically deployed devices.
- Demonstration and analysis of the effect of applying the Island Model with GP where entities may have slightly different objectives.
- Demonstration that balancing learning and performing to minimally meet an acceptable average fitness allows future increases to be more readily met.
- A hybrid (tree-based and Linear-GP) program representation which facilitates easy crossover operations and improved human readability.
- A program ID hashed from the frequency of instructions within the program which can be used to calculate a crude metric of diversity between programs suitable for implementation on microcontrollers.

1.5 Thesis Structure

Chapter 2 presents a survey of research relating to online and/or distributed genetic programming approaches that have been applied on resource-constrained embedded systems.

Chapter 3 uses insights from this to design the In situ Distributed Genetic Programming (IDGP) framework. The framework is implemented on WSN motes and validated by evolving logic to address some challenges where real-world effects interfere. Additionally, the distributed evolution is validated for motes with the same and similar objectives and contrasted against local-only evolution.

Chapter 4 introduces the Fitness Importance (FI) heuristic to convey the average acceptable fitness as a function of time and a metaheuristic for generating populations with a specified (using the ϕ parameter) expected average acceptable fitness. FI is then integrated into IDGP framework.

Chapter 5 conducts numerous experiments to observe the responses of applying different values of ϕ at various times during evolution to gain an intuition of how learning and performance of the devices are affected during evolution.

Chapter 6 focusses on ascertaining whether dynamic approaches to ϕ offer any benefit over a constant/balanced use of ϕ. Empirical studies using a standard genetic algorithm are employed for simplicity and without loss of generality to the framework.

Chapter 7 concludes the thesis with a summary and provides directions for possible future research.

2

Related Work: Genetic Programming on Embedded Systems

This chapter attempts to capture relevant research employing GP (or evolution of logic) on embedded systems. Much of the pioneering research applying GP on embedded systems occurred in the late 80's and early 90's when processors were becoming small, yet powerful enough to support GA approaches. Such processors were typically being deployed for small robotic platforms and so relevant research is centred around attempting to evolve useful behaviours on these robots. This is discussed in Section 2.2.

"Moore's Law" [86] suggests that the available computational capability of computers doubles roughly every 1.5 - 2 years whilst maintaining essentially the same physical space and power requirements. The corollary of this however is that the same computational capability can be delivered in smaller packaging, requiring less power and available

at very low cost. At this end of the spectrum, computational capability is realised by microcontrollers rather than CPUs. Philosophically the computational distinction is somewhat arbitrary, however microcontrollers are typically identified as having program memory, RAM, the CPU and numerous peripherals colocated on a single microchip.

By the early 2000's, WSN motes had enough computational capability, but at the low power required for their usage scenario, to become a viable option for performing GP on the devices. Furthermore, networking of PC-class machines was commonplace. The obvious synergy between the distributed capability of networks of computing elements and distributed GA approaches inspired research into applying GP on WSN mote-class devices. Key contributions in this area are discussed in Section 2.3.

With the remarkable continuation of "Moore's Law" [76] and improvements in networking technologies, there has been a proliferation of Internet-connected devices known as "Internet of Things" (IoT) devices. The computational capabilities of IoT devices varies greatly, however there will likely always be a niche for computationally contained devices since they can generally be fabricated at a much lower cost, be lower power and hence a smaller form factor. These are critical factors in numerous application scenarios. Research employing GP on IoT-class devices has only recently been attempted (including one based on the framework developed in this thesis) and are discussed in Section 2.4.

Section 2.1 is provided as a brief primer on some basic concepts used throughout this thesis, namely, Genetic Algorithms, Genetic Programming, the Island Model and the "No Free Lunch theorem". The reader with a solid understanding of these concepts may wish to skip this section while others wishing for a more comprehensive detailing of the concepts are referred to the many good textbooks covering these topics such as [107].

2.1 Basic Ideas and Concepts

The concepts described in this section are extensively detailed in most textbooks that discuss genetic programming and machine learning more generally. Nonetheless, the concepts within this section are referred to throughout this thesis and so are briefly described for the reader's convenience.

2.1.1 No Free Lunch Theorem

The “No Free Lunch Theorem” (NFL) [153] essentially states that no solution finding mechanism is better than all others over all problems. In the words of [153], "Elevated performance over one class of problems is exactly paid for in the performance over another class." However, despite this, some algorithms deliver good performance across many different types of problems, while other, highly specialised algorithms, may outperform on a smaller subset of problems [144]. Evolutionary algorithms (EA), genetic algorithms (GA) and genetic programming (GP) are generally considered as general problem solvers and demonstrate useful performance across a range of problems where finding an acceptable solution, rather than an optimal solution, is required. NFL

Acceptable solutions often exist as subspaces within the entire *search space* (or solution space) of all possible solutions. The technique of iteratively searching through the search space in order to find a better solution is known as *optimisation* and the choice of candidate solutions evaluated along the way is referred to as the *optimisation trajectory*. Evaluating every potential solution yields the *fitness landscape* . However, the best optimiser (or solution finder) will be domain, or even problem specific, as highlighted in the well known "No Free Lunch Theorem" [153]. For many real-world problems, the dynamic environment causes the fitness landscape to vary with time [14], [80], [3]. As such, online (in situ) approaches are a logical choice of solution finder for systems dealing with dynamic, real world environments.

2.1.2 Evolutionary and Genetic Algorithms

Evolutionary Algorithms (EA) describe the class of optimisers (and search algorithms) that employ mechanisms inspired by biological evolution. Typically they employ a population of candidate solutions that undergo a selection process that is biased towards better performing individuals. Successfully selected individuals either remain in the population or contribute to the generation of the next population. Individuals of this next population undergo a process of variation, thus exploring new potential solutions. The cycle of selection and variation continues until either an acceptable solution is found or the resources EA

available for evolution (usually a time constraint) have been exhausted.

GA

Genetic Algorithms are a subclass of Evolutionary Algorithms that use a gene representation (genetic sequences, codes, strings) for each individual and 'express' this genetic representation in an 'environment' to produce the phenotype. The phenotypic performance is used by the selection process with a bias towards survival of genetic material from individuals with greater fitness. Individuals selected as a 'parent' for offspring (children) in the next generation, pass on their genes through genetic operators such as crossover which takes genetic material from two (or more) individuals in a process known as crossover. Other genetic operators, such as mutation, are also applied to introduce variation of the genetic sequences which ultimately lead to variation of the phenotypes.

The ability to pass on learnings that occur or desirable characteristics that are discovered during the lifetime of an individual to its offspring is commonly referred to as Lamarckian learning or Lamarckianism. In the early 1800s [73], the French biologist Jean-Baptiste Lamarck proposed that such a mechanism may exist to account for the adaption of animals to their environments. However standard GA is based on the"Darwinian" viewpoint where genes that are passed onto offspring are not affected by what the phenotype does during its life.

LAMARCKIAN LEARNING

For further details on GA the reader is recommended to [84].

2.1.3 Genetic Programming

GP

Genetic programming (GP) is a biologically inspired, population-based search metaheuristic that attempts to generate computer programs that satisfy a high-level problem statement. GP is essentially a genetic algorithm (GA) and employs the same basic principles such as mutation, crossover and a selection pressure, however the concept is extended to genetic representations of programs.

Like GA, GP can build up substructures (referred to as schemata by Holland [58]) which offers a hyper plane exploration mechanism. The advantage of GA is then achieved through the combination of the substructures to achieve superior performance to either of the individual structures while the population becomes a reservoir of useful schema

which are recombined into new combinations with assumably higher and higher fitness. This constructive progression is known as the building block hypothesis. As such, general (domain-independent) good performance is achieved through the combination of local search (akin to optimisation through mutation) and global search (through the crossover and other genetic operators).

The syntactic richness offered by a program (defined by the terminal and function instruction set) makes GP more general than GA since it requires fewer assumptions about the structure of possible solutions [144]. A further benefit of program representation is that it is often more intelligible to humans compared to other approaches (such as Neural Networks or Support Vector Machines that use weights/numbers for thier representations). Similarly, programs can be readily coded by humans and used by the GP process. A disadvantage of the rich representation is that the additional degrees of freedom can make the fitness landscape more difficult to search, and at the extreme, this problem representation can be as bad as a needle in a haystack (NIAH) problem. Nonetheless, GP has the potential to generate novel (human-competitive[1] [71]) solutions that can be readily shared, incorporated and built upon by other entities.

Two popular program representations are that of Linear Genetic Programming LGP and Tree-based (Koza-style) GP.

Linear Genetic Programming (LGP) was proposed by Cramer in 1985 [23], and typically directly encodes the program as a string of operations in a linear fashion. The advantages of this are: no recursion, a one-to-one mapping on genotype to phenotype is easy to understand and easy to implement. Program 1 is an example of how a program may appear. Note the largely repetitive syntactic structure of “register = terminal function terminal”. LGP

Tree-based (Koza-style) GP (also known as hierarchical GP or standard GP) first appeared in [23], however was popularised by John Koza [67,68] who demonstrated that the (parse) tree representation of programs provided a method for a more meaningful

[1] GP has demonstrated an ability for devising novel solutions comparable to the creativity possessed by human thinking through generating patentable algorithms and implementations [71].

Program 1 LGP code representing the same equation shown in Figure 2.1.

```
REG1 = 4 * A
REG2 = Sin REG1
REG1 = Cos B
REG3 = 8 + REG1
REG1 = REG3 / -2
REG3 = REG2 - REG1
```

crossover operator. Koza's tree-based GP is now generally regarded as Standard Genetic Programming (SGP) method.

Figure 2.1 shows the representation of the same logic described in Program 1 as a Tree-based mechanism. Despite the fact that both approaches represent exactly the same functionality, one can see that the representations differ greatly. While Tree-based GP offers greater flexibility in programs, this comes with the inconvenience of additional logic required to ensure the code representation lies within the limits of what is possible by the hardware the algorithm is running on.

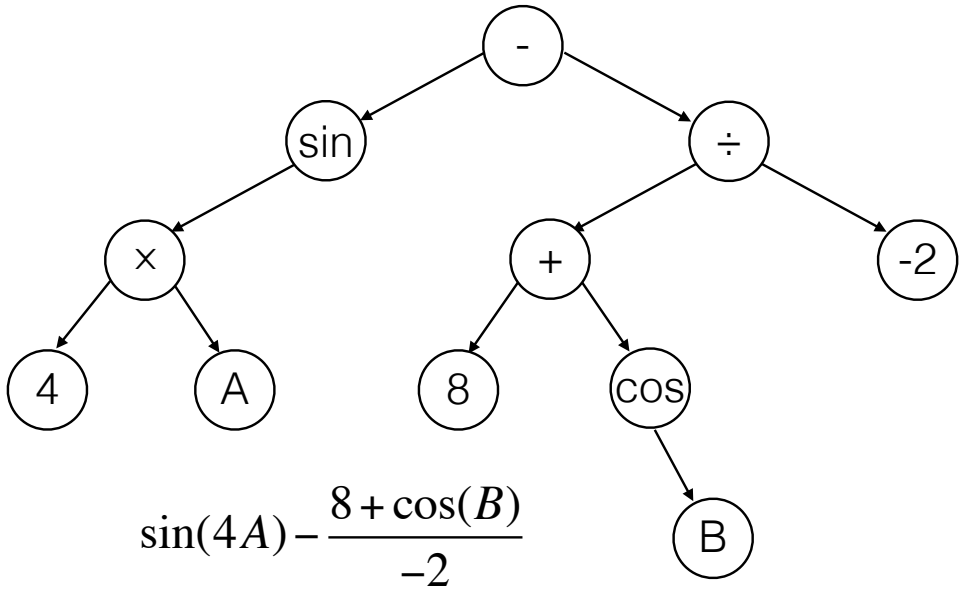

Figure 2.1: An example tree-based GP program representing an equation.

To allow the greater syntactic richness while maintaining reasonable simplicity in program representation, this thesis employs a hybrid approach which extends the tree-based representation such that nested statements form a single line of a multi-line program as

per conventional C programs. Effectively, the interpreter parses one code tree per line of code until an END_OF_PROGRAM instruction byte is reached (which need not be reached - for example if an infinite loop is entered).

2.1.4 Island Model

ISLAND MODEL

The Island Model [154], [64] concept essentially uses multiple, isolated (island) populations where occasionally (as determined by the migration rate) individuals from one island migrate to another population (island). Often an individual migrating to another population is a copy of an individual that remains in the local population. While the genetic diversity of a local population in isolation can decrease quickly, the diversity between populations is likely to remain high since evolution is largely independent of other islands. As such, a local population receives in injection of diversity each time a migrant in incorporated in the population. Prolonging increased diversity typically yields more efficient search and superior solution quality. Additionally, and equally important to this thesis, is that this algorithm can be implemented on parallel computing elements, however it also introduces the need for communication.

Approaches employing the Island Model typically exploit the divide and conquer (parallelising) approach to find a single better solution. However, the approach can also be used to expedite learning per individual island even when islands may be solving slightly different problems (due to variations in local conditions). This second usage is much less studied, however of greater interest to this research.

For more detailed explanation and analysis of the Island Model, the reader is referred to [126], [79] as suggested reading.

2.1.5 Online Learning

ONLINE LEARNING

Online learning allows a system to dynamically change its behaviour during the system lifetime in order to improve and/or adapt. Unlike offline learning, where the learning occurs before the system needs to perform, online learning has the additional challenge of achieving performance while attempting to learn. Unfortunately performing evaluations

that gain the most information (learning) typically have the lowest expected performance. Conversely, evaluating potential solutions with the best expected performance gains little or no learning. Balancing this contention is known as the Exploration Vs Exploitation (EE) tradeoff and will be discussed further in the context to the Fitness Importance metaheuristic presented in this work.

EE

2.1.6 Neural Networks

NN

Artificial Neural Networks (ANN), or Neural Networks (NN) is a biologically inspired machine learning approach capable of generalisation (function approximation, mapping inputs to desired outputs, pattern recognition and classification). Originally based on simple models of nerve cells (neurons), each neuron aggregates inputs (typically weighted) from other neurons and applies a function to determine whether the neuron will fire and subsequently supply "excitation" to the inputs of other neurons. Through various techniques (back-propagation being the most widely used) the network can be trained using the difference between desired output and actual output via supervised learning. NN can also be used in an unsupervised manner for clustering, compression, filtering, etc. For a number of decades NN were the predominant machine learning approach before being supplanted by Support Vector Machines (SVM), Bayesian Classifiers and Reinforcement Learning approaches which demonstrated superior performance (in some cases provably superior) in numerous application scenarios. However, NN have their own intrinsically desirable properties. For example, they are readily parallelisable, readily allow multi-class and real value outputs, and can be incrementally updated with additional training samples without the need to reprocess the entire training set

RNN

In recent years however, a class of NN known as recurrent neural networks (RNN) are being increasingly employed on more complex, time series data problems. The significant difference from traditional NN is that outputs from neurons can ultimately feedback into themselves through various interconnections. This provides an internal state (or memory) that can be used to generate more complex functionality particularly suited to temporal data such as sequences.

Perhaps the most exciting incorporation of NN recently is as Restricted Boltzmann machines (RBMs) which are then used in the construction of Deep Belief Networks. Deep Belief Networks and more broadly Deep learning, are receiving a lot of recent attention as they have demonstrated [25],[85] the ability to model high-level (abstract) concepts by constructing their own layers of lower-level abstractions (with various non-linear transformations, sub-formulae) from just the very high-dimensional data input and a notion of performance [12].

2.1.7 Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique that attempts to find rules for selecting actions given the current environment (state) in order to maximise a cumulative reward. Standard RL models the environment as a set of states and uses rules to choose an action given the current state. The transition to a new state has a reward associated with it and is appropriated to the rules which caused the transition (and reward). Rules, often stochastic, are employed for transitioning between states and attributing reward of a transition. A common implementation uses an action-utility function known as a Q-function which estimates the reward of a given state-action pair. Through experience this estimate becomes more precise over time. This allows actions that maximise cumulative reward to be identified and employed more frequently. RL

Despite its obvious application to online learning scenarios, RL has largely been applied to offline scenarios with many researchers considering online learning using RL as not fully addressed [151]. Nonetheless, a significant amount of research has been invested into analysis of RL for online learning. A key challenge with online learning problems is that of Exploration-Exploitation (described in Section 4.2.2). This has formulated into an idealised problem called the "Multi-Armed Bandit" (MAB) [106] problem (described in Section 4.2.3) which has been studied mostly by the RL community. The idealised nature of the problem (maximising reward from levers with unknown payout distributions) facilitates the easy comparison of approaches, however does not necessarily provide insight into applicability of RL to real world scenarios with high dimensional inputs.

2.1.8 Wireless Sensor Networks

More recently, microcontrollers are also featuring wireless communications and sensing capabilities which has made them ideal for wireless sensor networks (WSN), robotics and adding sensing, computing and communication functionality to a plethora of machines and objects. WSN devices (commonly called "motes" in the field) are often deployed as passive sensing networks for environmental [20] and industrial applications where higher spatiotemporal sensing resolution is needed over large areas or where it is prohibitively expensive or too cumbersome to deploy "wired" solutions. Like other embedded solutions, motes utilise a microcontroller for low power, though constrained, processing of logic and for other low power peripherals such as storage and sensing. By definition they must support wireless communications capability - most commonly in the form of a digital radio (transceiver) chip, though increasingly the radio and microcontroller are being placed on the same microchip which is referred to as a "System on Chip" (SoC). (see [74] for a listing of the various radio communications technologies employed by motes). The most common application for WSN is to sense data and wirelessly transmit the data back to a central data sink (base). However increasingly WSN applications are additionally performing actuation [6,72,129], which arguably overlaps with challenges faced in robotics and even IoT research. Systems that perform actuation are sometimes differentiated as Wireless Sensor and Actuator Networks (WSAN) [4], however within this thesis, WSN is treated as inclusive of WSAN.

WSN

WSAN

More recently there has been an exponential rise in the number of devices with communications capability. While these devices often communicate to humans directly, it is increasingly common for them to connect to the Internet and each other, which has resulted in the concept of the "Internet of Things" (IoT). By 2020, it is estimated that 26 billion IoT devices will be in operation [82].

IOT

2.2 GP on Small Robotic Platforms

Robots, like all embedded systems, perform sense-compute-act cycles. However robots are somewhat defined by their ability to actuate (do something physically in the world) rather than by sensing and communicating (as with WSN) or benefits gained from internet connectivity (as with IoT). Nonetheless, the system must perform computation on sensory signals in order to actuate appropriately. The logic to achieve this is typically referred to as the controller (or controller-logic, controller-function, control-process) and can be implemented via many mechanisms. Controllers evolved by GP or using GP as a method for control, have mostly been studied within the field of Evolutionary Robotics (ER). Within ER, approximately 40% of the implementations employ NN-based mechanisms while about 30% employ GP (or "evolvable programming structures") [89].

There are many potential issues with using EA approaches to evolve controllers for robots [81], largely due to complex, dynamic environments robots are deployed in which is made more complex by their ability to change their environment and move themselves within the environment. Yet, despite the dynamic environment, robots are expected to maintain a constant level of acceptable performance and in some scenarios, this can only be achieved through online learning.

Robotics research was the first to employ GP approaches on computationally impoverished platforms [45] and so has a larger body of research than GP used in WSN and IoT contexts. The following describes key robotics research utilising GP with particular interest on computationally constrained platforms and distributed evolutionary approaches.

2.2.1 SAMUEL

SAMUEL (Strategy Acquisition Method Using Empirical Learning) [45] is a complex feature-rich rule-based learning system developed over a number years (1990-1997). A brief overview of the salient features will be discussed in the following, however for a detailed understanding, the reader is referred to [43,45,46]. SAMUEL

SAMUEL is comprised of 3 main modules: A problem-specific module that interacts

with the real-world environment via sensor and actuators and also assesses the real-world performance of the system; a performance module that determines the behaviour/plan (set of rules) that is executed in the real-world and is also responsible for credit assignment to rules; and a learning module that evolves the rules and plans using a custom EA approach.

At the lowest layer, the fundamental unit of logic used by the system is a condition-action rule (also referred to as stimulus-response). A condition-action rule is a fixed IF-THEN logic structure that trigger actions based on a combination of sensor/state values (feature vector) matching a condition statement.

The general fixed-logic structure has the form:

```
IF   AND(condition1, condition2, ... conditionX)
THEN AND(action1, action2, ... actionY)
```

An example rule could be:

```
IF   time in [1,100] AND sonar in [0,30] AND bearing in[10,180]
THEN turn = 20
```

If, for example the current feature vector of (time sonar bearing) was (40 25 90), then if this rule was "fired" (executed), it would turn the system by 20 degrees.

A key advantage of this approach, and GP more generally, is that rule sets (or programs) supplied by humans can be incorporated into the evolutionary process at any time. Seeding the initial population for instance provides a useful mechanism for incorporating existing domain knowledge with the ability to evolve it as necessary to meet needs specific to the real-world problem scenario.

An ensemble of these rules comprises a candidate system plan (or behaviour). Within the learning module a population of such plans is maintained with the "best" plan available to be used by the performance module. When a plan is being employed, the rule with the highest expected fitness ("strength") that best matches the current feature vector is executed. Other (lower strength) rules whose conditions also match the current conditions are classified as being activated (but not executed). If no rules match completely, rules

with partial matching conditions are then found and fired/activated. Periodic evaluations of plans (whether in an internal simulation or in the real world) are then used to adjust the "strength" of rules that fired or were activated in the period between evaluations.

As mentioned, SAMUEL employs an EA approach for evolving rules, however unlike traditional GA approaches, SAMUEL evolves logic (both plans and condition-action rules) through a combination of Lamarckian and Darwinian evolutionary strategies. Lamarckian learning is particularly noticeable where the system is applied to learning sequential logic[45], however even in the non-sequential implementation, rules are updated based on the experiences of the system during a trial. This is achieved through a number of custom operators (specialise, generalise, cover, avoid, merge, delete) that adjust condition-action parameters based on the observed values that caused the firing of rules. For example, if the previous rule fired due to the feature vector (40 25 90), then an updated version of the rule after the "specialise" operator could be:

```
IF   time = [20,60] AND sonar = [15,35] AND bearing = [45,135]
THEN turn = 20
```

If the more general rule was assigned credit for useful behaviour, then this operator attempts to encode more concise representation of the distributions of conditions that are known to work.

Feedback of real-world fitness is intermittent since real-world evaluations occur only periodically (every nth generation). To increase the evolution rate, a simulation model of the task environment is used to estimate the expected fitness of the plans. This approach of being able to test solutions offline while running the current best known solution online is referred to as "Anytime Learning" [44] . How well solutions' performances translate into the real world will depend on the fidelity of the simulation model, however this model can be updated based on the results of the intermittent evaluations in real world. Additionally, in an attempt to make solutions more robust in the real world (i.e. avoid the 'translation' problem), noise (above what is expected in the real world) is added to the sensor models [45]. Furthermore, a consistent selective pressure for ever-improving performance is achieved by normalising the fitnesses of plans to a baseline one standard deviation less

ANYTIME LEARNING

than the mean payoff received by the population.

SAMUEL has been studied with respect to a number problems: navigating an autonomous underwater vehicle (AUV) through a simulated environment of mines [110], a cat-and-mouse "chasing" toy problem [43], robot navigation and obstacle avoidance [42,111] and evasive manoeuvring in air combat simulation [46]. Through these challenges, SAMUEL demonstrated the ability to utilise human supplied solutions and evolve new solutions and dynamically adjust behaviour during operation based on simulated and real world feedback. Interestingly, it was pointed out that SAMUEL was not designed for learning from *tabula rasa* and in fact the system would "flounder badly" in complex environments unless given sufficient initial knowledge.

SAMUEL was implemented in ANSI C and ran on a UNIX-based PCs. However plans could be executed directly on a robotic platform or remotely via a radio modem. Autonomous behaviour was demonstrated in [42,111], where an (offline) evolved program was ran on an "embedded PC"-class robot[2] in the problem scenario of navigating a robot through a room to a goal location whilst avoiding collisions. While the rule set was evolved offline, some online learning occurs due to how SAMUEL uses and updates rules during operation.

This discussion has provided a brief and simplified description of the SAMUEL learning framework. SAMUEL *partially* addresses many aspects of interest to this thesis. The framework is implemented on a UNIX workstation and not an embedded system, but potentially could be implemented on an embedded-PC-class system. It does not employ any distributed computing/learning mechanisms, but one could imagine applying the Island Model to the population of plans as well as the generated rules. It does not employ a full GP approach, but rather EA is applied to sets of condition-action rules which appears to equivalent to a constrained GP implementation. Finally, the system is not a completely online learning system, but instead requires some seeding of logic (derived offline) after which it can be modified (but not evolved) during runtime.

[2]The Nomad-200 robot used a 80486 processor running at 66 megahertz. See [19] for a detailed description of the robot platform.

2.2.2 Genetic Evolution of PDL Processes

PDL (Process Description Language) [124] is a tool for representing behaviours as process networks. A process network generates behaviours by representing sub-cognitive actions, like forward motor speed and button pressed etc, as quantitive entities that are influenced by logic (often in feedback loops) which act to drive the action quantities over a number of iterations. PDL has been implemented in LISP, for simulation experiments, and in C for robots [123–125] to generate dynamic emergent behaviours (typically using "PC-like" computers). While PDL itself is not an EA approach, further explanation of how PDL is used to generate behaviours is provided to aid understanding of how EA has been applied. PDL

Program 2 PDL code for a process to achieve a desired forward speed behaviour.

```
void up_to_default_forward_tend (void)
{
  if (value(go_forward) < 10)
    add_value(go_forward,1);
}
```

Program 3 PDL code for a reversing “reflex response” to front collisions.

```
void front_collision(void)
{
  if ((value(bumper0) > 0) || (value(bumper1) > 0) ||
     (value(bumper11) > 0) || (value(bumper2) > 0) ||
     (value(bumper10)> 0))
    add_value(go_forward,-80);
}
```

Program 2 and Program 3 are examples of behaviours using the PDL with the C language. It can be seen in Program 2 that the desired speed (10) is not simply set with `go_forward = 10` but rather the motor speed quantity (`go_forward`) is iteratively incremented until the desired speed is reached. An interesting outcome of this representation

is that different behaviours can be included that influence the same action quantities, resulting in complex emergent behaviours with smoother transitions between action states. Figure 2.2 shows an example of multiple processes feed in to reveal an emergent function which in turn self-enforces change (positive or negative) at the low layer, which ultimately drives the emergent behaviour.

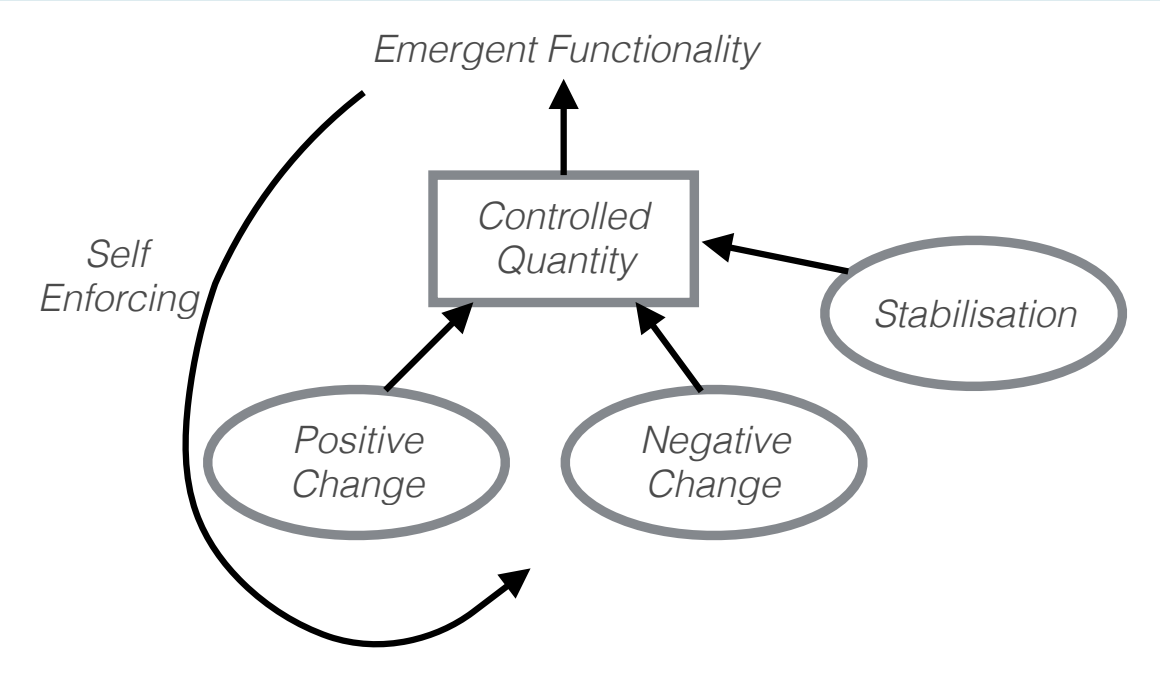

Figure 2.2: An example PDL process configuration. Recreated from Fig. 3.11 in [124]. The original figure description reads *"Emergent functionality pattern observed in process networks. There are two opposing forces, possibly stabilised. The processes impact a controlled quantity that indirectly gives rise to emergent functionality. The positive process feeds on itself."*

An example of this occurs with the inclusion of Program 3 which affects the same motor speed quantity by significantly decrementing it when a sensor detects collision i.e. `add_value(go_forward,-80)`. Since Program 2 will push the system to a limit of forward speed 10, when Program 3 subtracts 80, the system will suddenly reverse at speed -70. However, in the absence of further collisions detected from the front sensors, the system will decelerate but still travel in reverse for 70 iterations, before ramping up forward speed back to 10. Representing this same smooth behaviour with sequential logic would be significantly more complicated.

To achieve the additive parallel effect of process influences, the PDL architecture locks the most recent sensor readings then executes all the processes once, summing their effects on all action quantities, before updating the action quantities with the summed effects. After some predefined period has lapsed, new sensor readings are read in and the process repeated. This delay affects the rate that the sense-action loop is executed which significantly influences the resulting behaviours which are, by definition, a function of time.

The PDL architecture is extended to evolving behaviours on robots with online evolution in [123]. This is achieved by the introduction of a process selection mechanism, called Selectron and with a mutation operator. Selectron employs a population of PDL processes, initially constructed with multiple instances of each process. In PDL, process influences are typically additive so multiple instances or "clones" of a process effectively reinforce the single process behaviour. The Selectron mechanism probabilistically clones or deletes processes based on their contribution (positive or negative) to the average "satisfaction" of the robot's behaviour over some defined period. Note if the average "satisfaction" does not change over the period then a random increase or decrease in probability of keeping the process is performed to help overcome "dead lock" situations. The "satisfaction" quantity is also implemented as a PDL process and is periodically updated (every 10 cycles which corresponds to about 0.5 to 1 seconds).

The "satisfaction" quantity is similar to other "motivation" quantities (also PDL processes) which are used to implicitly guide the robot to perform useful behaviours. A good example of this is tying the robots propensity to travel to a charging station as an inverse square function to the remaining battery power. Thus, the lower the battery, the more the robot is "motivated" to seek charging.

This online learning mechanism was implemented on a small wheeled robot using a pocket PC and a LegoTechnics™ body and demonstrated rapid success in generating primitive behaviours. This occurs by the population composition evolving to contain only processes that positively contribute to "satisfaction". Due to stochastic reasons however, sometimes (about 20 percent of cases), subpopulations of good strategies die out early in the evolution. If this occurs the system typically relies on mutations to regenerate useful

lost processes, however in some instances, the evolutionary process is able to evolve to a stable solution by balancing subpopulations of competing (different) processes rather than relying on the system to evolve a single stable process.

While the evolution of logic in [123] is not strictly GP (more like evolution of human seeded rule structures via occasional mutation), a more recent attempt to use GP on the PDL architecture has been performed by [113]. In their experiment, they attempt to evolve a controller for the "Back Up a Tractor-Trailer Truck" problem. Additional genetic operators are introduced to provide the syntactic richness that GP offers, however like the first attempt to solve this with GP [69], it is performed in a centralised offline manner using simulation and without consideration of evolving on an embedded platform. As such, [113] falls outside the scope of directly relevant work to this thesis though it does provide an insight as to how [123] could be extended to use GP.

Nonetheless, the PDL architecture has been shown as an elegant mechanism for producing dynamic emergent behaviours and has been demonstrated on PC-class platforms. Furthermore, an online learning approach was demonstrated by employing EA to bias the representation of PDL processes within a population in order to achieve desirable, though primitive, behaviours. However, similar to GPN, the logic is distributed as parallel processing elements internal to the agent with no logic shared between robots during evolution. As such, PDL is not considered a distributed evolutionary approach.

2.2.3 Automatic Induction of Machine Code - Genetic Programming (AIM-GP) on Khepera Robots

AIM-GP

Automatic Induction of Machine Code - Genetic Programming (AIM-GP)[96,97] (formerly Compiling Genetic Programming System or CGPS [90]) performs the direct evolution of the machine code instructions which will be executed by the processor. This differs to most GP implementations which typically operate on higher level virtual machine instructions. An obvious advantage of this approach is the exceptionally compact representation of (LGP) programs and the fast execution of the code since the CPU directly executes the

instructions rather than a virtual machine interpreting instructions. The code representation (machine code) is cleverly encapsulated within a standard C function as a casted string. The simplicity of the LGP representation[3] allows the GP engine to be implemented with a small (32kB) memory footprint.

AIM-GP was first implemented on a PC (Sun-SPARC) and benchmarked on a classification task (determining if a presented Swedish word was a noun or not) and shown to outperform NN approaches [91]. Following this, the approach was then applied to the more ambitious task of evolving desirable behaviours for miniature, computationally constrained robots, in real-time and within the real-world environment (i.e. *in situ*)

The Khepera robots (c.f. [18] for a detailed description of the more recent Khepera platform) were used for numerous experiments. This mobile robot platform offered 8 range sensors on a small (6 cm wide x 5 cm heigh) cylindrical robot with 2 wheels, each with their own controller motor. The robots were placed in various irregular environments (90 cm x 70 cm typically) with configurations having walls, dead ends and obstacles. The aim was then to develop logic to enable the robots to autonomously travel fast and straight while avoiding collisions with walls, obstacles and other robots [92,93]. Subsequent experiments were aimed at more complex tasks such as seeking/following objects and locations (defined by darkness) [10,95,96,98] which implicitly required the lower level functionality of obstacle avoidance.

Various objective functions were supplied and the fitness calculated and fed back after a short 400 ms delay. The robots (whether simulated, tethered to a PC or fully autonomous) maintained a population of (50) programs each of which evolves the functional mapping of sensor values to output motor speeds (including reverse speeds). A few, typically 4, elite programs are selected from the population and were evaluated (400 ms after executing the logic) and then subject to a tournament selection which allows the best performing 2 programs to breed with their offspring replacing the 2 worst performing programs in the pool. Interestingly, it was observed that the sequential nature of evaluation could induce a motivation for programs to place the robot in a worse state than before a

[3] In this LGP implementation, each line of code is defined by the assignment of output which is the result of an instruction operating on 2 operands. E.g. Motor2 = Sensor1 + Var2; Var1 = Var2 × Sensor3 ...

program is run in order to sabotage other programs. However, it was not clear how much detriment this competitive behaviour caused to the collective system performance. As is common in scenarios where GP is employed, careful construction of the fitness functions was needed to elicit evolution of desirable behaviours. For example negative fitness was attributed to the sensor values (larger implies closer to objects, hence more likely to collide) which had to be balanced with a positive fitness for moving straight and fast. Without the correct biasing of the positive and negative feedback the robot would simply, and uninterestingly, not move at all. Nonetheless, the approach did evolve complex behaviours, such as backing up after collisions, and ultimately, mostly avoiding collisions altogether. It was reported to take about 200-300 generations (equivalent) to converge to a population where collisions were infrequent.

A later enhancement [94–96,98] of the system incorporated an "event memory" which recorded the fitness achieved for a particular action taken given various sensor inputs. A second simultaneous GP process was then employed to learn a fitness prediction function that predicts the fitness of an action given the current sensor values. This process uses the current "event memory" table as an input training data set where the differences between the predicted and actual fitnesses experienced are treated as an error that should be minimised by symbolic regression. Interestingly, similar approaches by others [109] coevolving NN as fitness predictors have also been shown to significantly improve the time to evolve to a good solution by reducing the number of candidate solutions needing to be evaluated in the real world.

The incorporation of the fitness predictor allowed the removal of the 300-400ms delay used to provide fitness feedback for each program. While the idle delay was removed, the fitness for a given program is still assigned 500 ms after the program is executed. Because the rate of evaluation of programs increased by 2000 fold, the size of the "event memory" and subsequently the program population, needed to be significantly increased (to 10 000). The limited RAM (256 kB) permits a small population to be employed causing less robust behaviour and more frequently getting stuck in local optima. This was demonstrated on the tethered experiments to dramatically reduce the time required to achieve good system performance (down to about 1.5 minutes from originally 40-60 minutes).

An interesting discussion about the need for a "childhood" period was presented based on experience that when the system needed exploratory motivation early in evolution to achieve good performance later. While some of this occurs intrinsically due to the random initial population, they did additionally introduce stochastic behaviours (noise) early on to assist the exploration. It was observed that when the system was exposed to more diverse situations in the early period, that this generally led to better behaviours later providing there was a bias to retain earlier experiences in the "events table". It is possible that providing the "events table" has a good representation of significant scenarios (i.e. the problem space) that it does not matter when the experiences are acquired.

Further investigation is needed to understand the dynamics of the population diversity at different stages of evolution and how this affects performance. Of particular interest would be how this impacts on the plasticity of the system to adapt to new environments or changed objectives. There is also no explicit analysis of meeting an "acceptable" performance, however the system clearly demonstrates "desirable" behaviour in a relatively short period. The approach does not leverage any parallel mechanisms of evolution (i.e. no Island-model sharing of programs or events), however it is likely that it could benefit from their application.

This research demonstrated the ability to learn desirable robot behaviours online and in situ using simple LGP programs generated by the AIM-GP approach combined with a coevolved fitness predictor. It is one of the few examples where GP has been deployed on a computationally constrained platform and performed online evolution in non-simulated environments. As such, the AIM-GP approach provides rich stimulus for the research of this thesis.

2.2.4 Genetically Programmed Networks (GPN)

Genetically Programmed Networks (GPN) [7] are a distributed logic representation where inputs are mapped to outputs through a connected graph of processing nodes. Each processing node within an agent's internal processing network is a self-contained program evolved using GP in a manner similar to graph-based GP systems (similar to Parallel GPN

Algorithm Discovery and Orchestration (PADO) [132]). The nodes are broadly structured into layers in a manner similar to Neural Network (NN) topologies, with input nodes (for sensor and other inputs), hidden nodes that take the inputs and outputs from other nodes (including delayed outputs similar to RNN), and output nodes which supply the output of the GPN. The connectivity topology is implicitly constructed by evolving programs that include specific instructions (terminals) that represent the outputs from other nodes as well as the inputs to the network. The instruction set however also includes additional instructions (mathematical functions and terminals) that do not correspond to the inputs or outputs of other nodes. Another GP process operates on the network of nodes (using a separate instruction set) which performs the final transformation combining the nodes to generate the outputs.

The generic nature of programs allows each node to represent practically any processing element of any optimiser. This versatility was demonstrated by evolving functionally equivalent implementations of distributed programs, rule-based systems and recurrent NN by simply changing the node and network wide instruction sets.

GPN was designed primarily for developing controller logic for intelligent agents, but has been applied for proactive aggregation protocols in WSN. The approach was empirically evaluated on the Ant [7] and Tartarus [117] problems and shown to require less evaluations than standard GP approaches to evolve to a near optimal fitness. While the "NN with memory" GPN implementation (similar to a recurrent NN approach) outperformed the distributed program (similar to GP) representation [118], the flexibility GP offers may be beneficial over NN representations in particular application scenarios.

Similar to PDL, the logic of GPN is distributed (internally to an agent), however the evolution of that logic occurs in a centralised manner. Hence GPN is not a distributed evolutionary approach. Additionally, GPN is an offline learning heuristic since the final elite solution is used as the metric of performance with no consideration of the performance of evaluated solutions during evolution. In Figure 3 of [118], an obvious difference exists between the elite and pool fitnesses during evolution, suggesting that if GPN was to be applied to an online learning scenario, then a mechanism may be required to achieve acceptable performance.

2.2.5 Embodied Evolution (EmEvo)

Embodied Evolution (EmEvo[4]) [32,36,141,142] sets out with the objective to create a population of physical robots that evolve autonomously, as well as perform their tasks autonomously, using a distributed evolutionary algorithm. EMEVO

Through a mechanism called Probabilistic Gene Transfer Algorithm (PGTA), they evolve the weights of a simple artificial neural-network architecture and demonstrate that the agents are able to achieve some level of performance on a number of simple tasks for small robot solution.

Each robot is effectively a member of a larger population and through sharing weights in a decentralised manner, the robots are able to evolve novel solutions to a non-trivial search space.

Embodied Evolution aims to achieve evolution-based learning with performance considerations in the physical environment using a distributed approach. While this is a key aspiration for this research (IDGP), Embodied Evolution is not a GP approach and furthermore each agent does not evolve a local population, but rather they are themselves members of a population searching for better solutions.

2.2.6 Distributed Agent Evolution with Dynamic Adaptation to Local Unexpected Scenarios (DAEDALUS)

Distributed Agent Evolution with Dynamic Adaptation to Local Unexpected Scenarios (DAEDALUS) [50,55] is a framework for achieving online learning for swarm robotics. In experiments employing DAEDALUS [51–54,56,57], the primary focus has been on navigating a swarm of small mobile robots to a succession of waypoints (referred to as goals) whilst avoiding obstacles and each other. DAEDALUS

The control mechanism for the robot is represented by force laws[5] such that the parameters of the force law dictate the attraction (or repulsion) of the robots to each other,

[4]Embodied Evolution is abbreviated as EE by the authors, however within this thesis EE refers to Exploration-Exploitation and so EmEvo has been used

[5]Lennard-Jones force laws that model forces between molecules and atoms were employed to allow the swarm to behave as a liquid or solid (or even gas).

obstacles and the goals. The force laws only considered observable entities so that robots and obstacles occluded by a nearby obstacle do not contribute to the aggregate behaviour of a robot. The goals however were always assumed as observable.

In an attempt to avoid the undesirable behaviours during the early phases of evolution, DAEDALUS employs offline EA, in a simulated environment, to learn parameters for the controller that will likely produce desirable behaviour (attaining goals whilst avoiding things). The real robots are then "seeded" with a slightly mutated version of the offline-evolved solution. It is important to note that the real environment is similar, though usually more complicated, than the simulated environment the offline solution was evolved in. Since the robots have slightly different control behaviours, there will be variations in the response to various real-world scenarios. This achieves online learning since if a neighbouring robot has superior performance, then its control parameters (and potentially epigenetic parameters such as mutation rate) will be adopted. This strategy is also employed by the DOWSN framework discussed later. The framework also allows for multiple "species" of robots to coexist in the same environment, however exchange of genetic material only occurs between individuals of the same species.

Interestingly, in a number of scenarios the performance of the online strategy performed worse than the offline evolved logic, however the online learning did demonstrate better behaviour in environments that were significantly different to the simulated environment used by offline evolution.

In [55], a different example problem space is investigated where L-systems[6] are grown in a resource-competitive environment. The L-systems must compete for resources until they reach maturity and then reproduce, however the threshold at which they reach maturity depends on the resource availability within the environment. Additionally, the fitness landscape is dynamic in that the energy consumption rate is varied to discrete levels at various stages during evolution. This means entities consume more resources while they

[6] A Lindenmayer system (L-system) is a formal grammar represented as a string of symbols, which when expressed (in a genotype to phenotype sense) produces a larger string of symbols. This in turn can be expressed and repeated such that the system appears to “grow”. It was devised by Hungarian biologist Aristid Lindenmayer in 1968 to model the growth processes of plants and is still researched today.

are searching for more resources. Entities can mutate the reproduction threshold parameter (which is passed on as epigenetic information to offspring) and interestingly this responds to the changes in the fitness landscape in a reproducible manner (See Fig 6, [55]). Despite being a simulation-only study, this is of particular interest to this thesis since the maturity threshold is effectively a satisficing constraint (i.e. fitness above the maturity threshold isn't important), and furthermore this threshold changes over time (similar to the study in Section 5.2.3). As such, this dynamic fitness satisficing metric could be represented by the FI metaheuristic presented in this thesis.

While this framework does not employ genetic programming, it does offer an online learning mechanism for resource-constrained robots and preliminary insights into achieving acceptable performance in dynamic fitness landscapes.

2.3 GP on WSN Motes

The untethered nature of the WSN motes afforded by wireless communications requires the device to have its own power source. For reasons such as size, cost and weight, the self-contained energy source is typically finite (i.e. a battery) or at best a small (but renewable) energy budget. Unfortunately, wireless communications can consume significant energy relative to other mote functions impacting on the energy budget and ultimately longevity, and usefulness, of the devices. This contention has been a prominent research focus of the field since before 2000 [31], with many low power communications protocols and algorithms being developed for specific network topologies in various scenarios. Accordingly, this has somewhat biased ML approaches to be mainly applied to addressing energy and communications related challenges [6,13,26,34,72,83,88,104].

Of the ML approaches applied to WSN, few employ CI approaches [60] and fewer employ EA based implementations. As discussed in Section 2.1.3, this is largely due to the significant challenges of performing EA on such highly constrained hardware and the programmers of WSN not typically being cognisant of EA approaches suitable for WSN. Nonetheless, evolving logic within WSN offers a promising approach for achieving desirable adaptive behaviour of motes in real world environments.

Within the subset of EA-based implementations on WSN devices, we focus on the approaches that employ Genetic Programming to achieve online and/or distributed evolution of logic on the mote which is typically not mentioned in most of the survey and literature reviews (e.g. [6]). In agreement with [59], the significant (and potentially only) published research employing distributed GP for WSN are Johnson et al. (2005) [62], Weise et al. (2006,2008,2009,2014) [143,145–150], Valencia et al (2007,2010) [134,137] (research within this thesis) and Iacca (2012,2013) [59,60].

These are now discussed (excluding this research) with particular attention paid to whether they are:

- In situ (i.e. implemented on the embedded devices)
- Distributed (in its learning mechanism)
- GP (employs genetic programming as the learning representation)
- Online learning (updates operational behaviour to achieve acceptable performance)

2.3.1 Broadcast-Distributed Parallel (BDP) Genetic Programming

The first investigation reporting GP approaches for WSN motes was [62] in 2005 where it was recognised that the intrinsic parallel nature of EA approaches and WSN should be an intuitive fit. Additionally, [62] identified that employing the Island-model architecture could be one way of realising this potential fit. They proposed the Broadcast-Distributed Parallel genetic programming model (BDP) as a framework suitable for WSN motes.

BDP

BDP is essentially the Island-model architecture applied to GP (and EA more generally) where the evolution of local population is performed asynchronously with respect to other populations. For the WSN scenario, this means each mote maintains its own population for local evolution and broadcasts the current elite solution to neighbouring motes (at the end of each local epoch). Motes asynchronously receive neighbouring elite solutions, however store them in a separate "mating" population until completion of the local epoch, after which a selection process occurs, which considers candidate solutions from the mating list, to generate the next population. They note that exchange of genetic

material used for crossover is roughly equal between neighbours. As has been shown by [79]), employing the Island model can significantly reduce the time to find an acceptable solution. It is not clear however why sharing only elite programs had no negative effects on the genetic diversity (as observed by which would typically result in converging to a suboptimal solution.

Various GP representations suitable for mote-class devices are also discussed. Prefix notation using virtual machine instructions is suggested as the preferred approach as it keeps the program code size small which is beneficial given the population of programs is likely to be a significant RAM consideration for memory-limited devices. Additionally, transmitting smaller programs via radio will reduce energy consumption which is another important constraint on motes. This representation was implemented with each instruction having exactly two operands to keep the representation simple. It is noted that bounds on the maximum allowable size of an individual still need to be considered (which affects the maximum nesting level) given the memory constraints of mote-class devices and the likelihood that there will be a fixed amount of memory for storing the population of programs.

BDP was implemented in simulation only and an analysis was conducted on the effects of the number of motes in the network and the population size. The objective function was a symbolic regression task ranging from a simple 3-variable version to a 10-variable problem. Training data for the experiments were generated via a clustering algorithm to ensure the training data was sparse and had good coverage of the search space. This may not be possible for online and real world problems. Additionally, while BDP was shown to evolve to an acceptable (elite) solution, there is no discussion of the acceptability of performance during evolution (i.e. online performance) Nonetheless, the empirical results showed that using the Island-model architecture for GP on WSN was more efficient than the same computational resource used as a single population.

Implementation on actual motes in real environments was suggested as future research, however this does not appear to have occurred. The authors also suggest investigating whether motes evolving solutions specific to local conditions would benefit from receiving genetic material from neighbouring motes sharing similar, though different, local conditions. The outcome of such research is likely to be relevant to real-world scenarios.

2.3.2 Distributed Genetic Programming Framework (DGPF)

DGPF

The Distributed Genetic Programming Framework (DGPF) [150] provides a flexible, open-source infrastructure for developing logic for networks of devices. The flexibility of the framework stems from its very modular design which clearly separates the choice of solver, the objective function, the simulation of the devices and the simulation of the environment. Key components of the system include: the Common Search API, Task Distribution System, Automaton Simulation and Network Simulation. The Automaton Simulation and Network Simulation components reside within the Genetic Algorithms layer and provide an emulation of physical mote characteristics of computing and communications.

While the framework itself is programmed in Java, each mote is simulated as an automaton running a virtual machine with a fixed-sized memory architecture, asynchronous IO, and a Turing-complete instruction set[7] employed in a LGP representation (i.e. no genome representation). The final evolved assembler-like program can be cross-compiled into native code to run on the actual target platform as needed, however while running on the virtual hardware, instructions can be stepped through (or back) for debugging. The fitness score of a candidate solution (program) is determined by the average performance of a number of repeated simulated networks of interacting automata, rather than from just a single entity. Each simulated network is generated with a random network connectivity topology with a guarantee of no network partitions. Communications between entities occur through receive and transmission buffers (memory transactions) subject to a number of simulated real-world issues such as packet collision, range limitations, packet loss , etc. Local conditions and responses can be incorporated by the user-supplied fitness function. The Genetic Algorithms layer also provides a number of default genetic operators (and their parameters) which can be augmented by user-defined operators if desired.

Each of these network simulations and evolutionary processes are encapsulated as tasks that can be distributed across multiple machines via the Task Distribution System. Additionally, load balancing processes can be used to distribute tasks according to the available computational resources of other machines.

[7] Includes constants, binary and unary expressions and direct and indirect memory addressing

Finally, the Common Search API, provides an abstraction between problem spaces and search algorithms. This allows a single search algorithm to be benchmarked across multiple problems, or multiple search algorithms to be tested on a particular problem or potentially multiple search algorithms to be tested across multiple problems. Within this context, GP can be viewed as one of potentially many search algorithms, however the vast majority of research performed with DGPF in context to WSN has employed GP [143–149]. These experiments demonstrated challenges with multi-objective optimisation (such as factoring in code size, correctness, network traffic etc) on problems that have the potential to be solved in a distributed fashion like the election problem [144,145], greatest common denominator problem [148] and an aggregation protocol [149]. A salient point on such problems was how easily the problem could become a needle in a stack (NIAH) problem if no intermediate rewards were given. This serves as a reminder that EA (and most other) approaches are not suited to all-or-nothing fitness/error feedback.

In summary, DGPF is a flexible research oriented framework for exploring using GP for generating logic for WSN motes. It uses GP to evolve mote logic in an offline and distributed (yet centrally coordinated) fashion using clusters of computers to simulate networks of motes and using. Once an acceptable solution has been evolved the logic then can be transferred to mote class devices.

2.3.3 Distributed Optimization for WSN (DOWSN)

The Distributed Optimization for WSN (DOWSN) framework [59] was developed as an online, distributed optimisation process for WSNs specifically targeted to address online dynamic problems requiring local computations on the motes (e.g. energy-aware routing, localisation, clustering, data fusion, scheduling, security, Quality of Service). DOWSN

At the core of the framework is an "Algorithms Database" (A-DB) which is populated with a selection of optimisation algorithms. These algorithms are preferably lightweight, memory-efficient (or fixed memory use) algorithms and so were implemented as inlined C macros to achieve a smaller memory overhead (and faster execution time). As a proof of concept, four of these algorithms were implemented: Random Search (RS), Intelligent

Single Particle Optimisation (IPSO), Nonuniform Simulated Annealing (nuSA) and 3 Stage Optimal Memetic Exploration (3SOME)[61]. Due to the simplicity of these optimisers, all except 3SOME required only 2 candidate solutions (as n-dimensional arrays) be held in memory; one for the current best known solution (i.e. elite) and one for the candidate (trial) solution. 3SOME however requires an additional candidate solution in memory for the "initial elite" which is needed for replacements when using the short distance operator (c.f. [61]).

3SOME

Each mote locally evolves (i.e. in situ evolution by the mote) by selecting an algorithm from the A-DB at each iteration and potentially using the current elite as an input to the optimiser to generate a candidate solution. If the trialled candidate solution demonstrates superior performance to that of the current elite, the elite is replaced and a local search/optimisation is performed (e.g. gradient descent optimisation). It is important to note that the same optimiser need not be selected each iteration, and this could potentially allow the optimisation process to maintain search diversity or break out of a local optimum. This was subsequently investigated [60] using the COOJA WSN simulation environment on a set of benchmark mathematical problems and interestingly found that employing only one algorithm (3SOME) was better than selecting a multiplicity of algorithms throughout the evolution.

In addition to the local optimisation, DOWSN supports distributed evolutionary search by employing an Island Model process to share elite solutions with neighbouring motes. While elite solutions are broadcast every iteration by each mote, the receiving motes will only probabilistically accept the incoming solution if it is better than the current local elite. The probability of accepting a solution (if it is better than the current elite) is referred to as the imitation rate within this framework. The Island Model was shown to unequivocally speed up the time to evolve a good solution [60]. It also demonstrated that sharing only elites ultimately generated superior solutions which is counter to the expected behaviour of this mechanism as reported by [79]. Interestingly the authors claim, "DOWSN is efficient even (and especially) when a small number of nodes is employed", which is possibly due to the fast diffusion of a good solution which would quickly reduce the diversity of solutions across the network and therefore narrow the search considerably thereafter (i.e.

premature convergence). Arguably the population on each node is so small (i.e. an elite(s) and 1 candidate solution) that it cannot be regarded as a population per se, but rather as a single candidate solution (albeit with local optimisation) within the population of WSN. Viewed like this, one shouldn't be surprised that a population-based search outperforms a single-point search and optimisation in the same number of epochs. Additionally, some real-world robustness may be lost by maintaining only the one elite solution. Real-world fitness landscapes are often slightly noisy and/or dynamic such that a candidate solution could stochastically be (incorrectly) evaluated as having superior fitness to the current elite. In this scenario, due to the replacement strategy used by DOWSN, the good solution would be lost and would need to be rediscovered. Nonetheless, the memory footprint of this approach is significantly smaller than population-based metaheuristics, making the framework suitable for implementation on mote-class devices.

Perhaps an even more significant advantage of the minimalistic population size is the potential to easily parameterise how far each mote searches away from its current best known solution into a time-dependant exploration-exploitation parameter. After some time (which could be greatly reduced by the Island Model architecture), the behaviour of all motes within the WSN could evolve to stable, yet good, performance which is ideal for online learning scenarios. In these experiments, mathematical optimisation problems were used as the objective function and the fitness of the WSN was calculated as the average of the final elite fitness value from each of the nodes. It is important to note that the average of the final elite fitness is not an online performance metric since it does not take into consideration the fitness of each evaluated candidate solution during evolution.

DOWSN was implemented on TelosB motes[8] using the Contiki OS[9] and demonstrates in situ, distributed, online optimisation on mote-class devices as aimed. Because it uses an optimisation heuristic however, it may not be suitable for more complex problems that require sense-compute-act behaviour to address them. For these problems, more sophisticated algorithms that can map sensory data into time-varying actions would be required (such as GP, NN, Reinforcement Learning).

[8]See [102] for a detailed description of the TelosB platform
[9]See [29] for a detailed description of the lightweight OS for WSN devices

It is not clear how GP could be integrated into the DOWSN framework, however a link can be drawn between DOWSN being a memetic framework and as a GP logic sharing memetic concept. As such, the idea of using GP with DOWSN could be worthy of investigation.

Genetic program representations would unlikely be compatible with optimisation representations and so the A-DB would probably need to be populated with solely GP algorithms instead (i.e. a GP-DB). Given the finding that a single algorithm typically outperformed the multiple algorithms approach [60], it is doubtful that a GP-DB would offer any significant advantage over a single "best-of-breed" GP algorithm. Additionally the local evolution with GP would likely need to maintain a larger population (than a single non-elite) to maintain sufficient genetic diversity in order to realise the benefits expected from the building block hypothesis.

2.4 GP on IoT Devices

Excluding WSN research, which is a subset of IoT systems, there are few reported implementations of GP deployed on IoT-class devices. IoT devices must necessarily be connected to the internet and common methods for achieving this include direct wired tethering (such as ethernet connections) and increasingly wireless approaches such as WiFi and Bluetooth. IoT devices face the same challenges present to other embedded systems such as resource constraints, limited power and issues with communications [129], however their internet connectivity is conducive for offloading computation (and potentially evolution) to the “cloud”. Nonetheless, local evolution on these devices still offers benefits. One potential benefit is the increased privacy gained by not sharing data that could identify the user, or information about them, with another party (cloud service). For wireless devices, significant energy savings can be gained by not sending data to cloud services (particularly over 3G/4G connections) and furthermore continuous connectivity to such services cannot be guaranteed.

Applications on IoT devices tend to employ hand-crafted logic by humans or primitive

ML such as case-based reasoning [74]. For non-smart phone IoT devices, the most dynamic logic implementations have been achieved using Fuzzy Logic and NN approaches [129], however none were found that employed GP. For smart-phones, which represent the vast majority of IoT-class devices[10], two preliminary attempts at using distributed evolution on the devices has been reported and are discussed in the following.

2.4.1 Ulfsark Framework

The Ulfsark Framework [37] is an open source[11] Bluetooth Peer-to-Peer (P2P broadcast communication layer for distributed applications on mobile phones. The framework provides a Java-based (Java ME and Java SE) API which offers programmers a simple interface for asynchronous data transfer of "packages" using a client-server model. ULFSARK

An application running on the Ulfsark framework uses a unique ID to establish a P2P network with other devices running the same application. Using this network, they demonstrate a distributed EA via Island Model architecture, though it could be any application that would benefit from peer to peer communications. For the distributed EA, each phone maintains and evolves a local population of candidate solutions and shares the best solution to another device in the network at regular generation intervals.

The EA implementation using the Ulfsark framework is demonstrated on mathematical problems (variants of the wave function and the Travelling Salesman Problem) by successfully finding solutions. The times to reach a near optimal solution are aided by the Island Model architecture which is readily implementable due to the Ulfsark framework.

Ulfsark demonstrates a mechanism to parallelise an offline optimisation problem since the performance of the EA during evolution is not considered. Arguably, such tasks are better suited to be solved by using cloud services or supercomputer clusters. Evolving phone specific behaviour however would benefit from this approach providing other devices were mutually seeking similar phone/application behaviour. This application scenario is discussed in the next section.

[10] This is expected to change as consumer goods such as fridges, televisions, lights, dishwashers, washing machines, garage door, etc) increasingly obtain internet connectivity as a standard capability.

[11] Source available at `http://ulfsark.sourceforge.net`

2.4.2 The Android Genetic Programming (AGP)

AGP

The Android Genetic Programming (AGP) framework [22] is an open source[12] java framework for evolving application behaviours on Android-based phones. It is the first reported smart phone implementation employing GP to evolve application behaviours on the phone. This first implementation [22], though based on the IDGP framework, did not perform distributed evolution, but rather maintained a local population of programs. It demonstrated the ability to evolve application behaviour to the user preferences in a news reader application and was also demonstrated to evolve improved energy efficient localisation through context-awareness.

AGP was subsequently extended [135] to a distributed evolution architecture (employing the Island Model) using bluetooth and WiFi for communications. Similar to the results of Section 3.4.2.3 and others [79], using the Island model reduces the time to find an acceptable solution, through the sharing of logic across multiple populations (each phone maintained a single population).

The research also highlights an advantage of distributed GP in that it can share logic constructs without needing to share the data they were trained on. Potentially. this helps preserve users' privacy whilst aiding others evolution of logic. In contrast, batch learning approaches typically require access to the data collected by others in order to achieve general improvements in learning. More specifically, access to a classifier's weights or parameters is often unhelpful for improving another classifier since the internal representations may differ between the classifiers.

The AGP research provides an initial proof-of-concept demonstrating that GP can be used on IoT devices to enhance usefulness over time through online learning. Research on applying EA approaches on IoT devices is still in its infancy

[12] Source code is available at `http://sourceforge.net/projects/agpframework/`

2.5 Summary of Related Work

Various works for achieving evolution (though with a focus towards GP) on embedded systems such as small robotic platforms, WSN motes and IoT devices were reviewed. It is clear that GP on small robotic platforms pioneered research in this field with research dating back to the early 1990s and is still continuing today. GP on WSN motes, while less developed, has a natural synergy with the intrinsic parallel nature of GP, and EA more generally [59]. Research with mote-class devices started during the 2000s and received recent research interest by this thesis and the others with some success. GP on IoT devices, unsurprisingly, has received only recent attention due to IoT technology becoming widely available within the last decade. Nonetheless, given the proliferation and anticipated growth of IoT-class devices in the coming years, ML approaches (including GP) are likely to play an important role in delivering device and application behaviours that improve and adjust to users' preferences as well as adapt to changes in the environment (or fitness landscape) over time. Expectations of such behaviour already exist for numerous web services like Google search, website advertising, Amazon recommendations, etc.

As can be seen from Table 2.1, many attempts to construct frameworks to achieve evolution of logic on embedded systems have been made. Each of these approaches demonstrated their utility by addressing the various aspects useful for evolving logic on embedded devices, often residing within networks. The significant aspects of these attempts can be categorised into the following: evolving on the embedded device hardware (i.e. *in situ*), performing evolution in a distributed manner, employing GP is the logic discovery mechanism and learning within the deployed environment while considering performance (i.e. online learning). Whether these approaches address these aspects is shown in Table 2.1.

Evolving logic with GP on embedded systems within networks is a desirable property for a number of reasons. First, *in situ* evolution avoids the transference problem - the brittleness of logic devised offline through simulation when placed into the real world. Second, there is a natural synergy of the distributed nature of population-based evolution with the "population" of agents within a network. Furthermore, employing mechanisms

TRANSFERENCE PROBLEM

such as the Island Model are known to significantly aid in reducing the time for an individual (population) to discover acceptable solutions. Third, computer code can represent potentially **any** behaviour capable by the device. GP offers an automatic mechanism for generating such code and a syntactic richer representation than that of traditional EA.

Approach	Reference(s)	**In situ** (CPU)	Distributed	GP	Online
SAMUEL	[111]	✗[13]	✗	✔	✗
	[42]	✗[13]	✗	✔	✔
PDL	[123]	✔[14]	✗	✗	✔
	[113]	✗	✗	✔	✗
AIM-GP	[10,92–96,98]	Khepera Robots (MC68331[15])	✗	✔	✔
	[97]	PC Class[16]	✗	✔	✗
GPN	[7,117,118]	✗	Internally	✔	✗
EmEvo	[32,36,141,142]	Handy Cricket (PIC16C715)[17]	✔	✗	✔
DAEDALUS	[50–57]	Handy Board [18] (MC68HC11)	✔	✗	✔
	[55][19]	Simulated	✔	✗	✔
BDP	[62]	✗	✔	✔	✗
DGPF	[150]	✗	✔	✔	✗
IDGP	[137]	Fleck3b (Atmega1281) [20]	✔	✔	✔
DOWSN	[59]	TelosB (MSP430) [21]	✔	✗	✔
	[60]	COOJA (Simulated MSP430)	✔	✗	✔
Ulfsark	[37]	Nokia 6288[22]	✔	✗	✗
AGP	[22]	HTC Magic [23], Nexus S[24](ARM)	✗	✔	✔
	[135]	Nexus S [24] (ARM)	✔	✔	✔

Table 2.1: Summary of related research. IDGP (this research) included for comparison.

This richness can be exploited to implement nearly any online learning mechanism, as was demonstrated with Section 2.2.4. Additionally, it is easier to understand logic represented in a familiar language like C, rather than weights of NN or support vectors, etc. Finally, customised and adaptive logic can offer longer-term acceptable solutions or enhanced user experiences by requiring less human in the loop intervention to maintain or improve logic within changing environments and contexts. *All* of these aspects are

[13] Intel 80486 (Nomad-200 robot) however evolution of programs did not occur on the platform
[14] Intel 80186 7.91Mhz, 1MB RAM *Assumed HP 200LX as “Pocket PC” based on the year of publication
[15] MC68331 16 MHz, 256 kB RAM
[16] SUN-SPARC, Power-PC, Intel 80X86, Sony PlayStation, Java Byte-code
[17] PIC16C715 20 MHz, 128 bytes RAM, 4096 bytes ROM
[18] MC68HC11 2 MHz, 32 kB RAM
[19] The second problem experiment involved L-Systems
[20] Atmega1281 8 MHz, 8 kB RAM
[21] MSP430 8 MHz, 10 kB RAM
[22] ARM9 236 MHz,2048 kB RAM
[23] MSM7201A (ARM) 528 MHz, 288 MB RAM
[24] ARM Cortex A8 1 GHz, 512MB RAM

desirable for embedded systems and this is supported by the numerous research efforts reviewed in this chapter that attempt to address various combinations. However, none of these research efforts addresses *all* of these aspects. (Excluding this research, IDGP, which is included for comparison purposes, and the AGP work which is also based on the IDGP framework.) The challenge of addressing *all* of these aspects simultaneously provides the motivation of this thesis.

3

In situ Distributed Genetic Programming Framework

This chapter details the design and evaluation of the In situ Distributed Genetic Programming framework (IDGP) [136]. IDGP is a core contribution of this thesis designed to assist embedded systems engineers in achieving distributed evolution of logic on their deployed devices. Design considerations and an example implementation of the framework are provided as a guide for embedded systems engineers to develop their own distributed genetic programming implementations on embedded devices. The implementation and evaluation of the framework revealed undesirable performance of the devices evolving logic. This motivates the need for a mechanism to balance the exploration and exploitation aspects of evaluations in real world application scenarios.

This chapter details the design and evaluation of a framework for providing the beneficial capabilities captured within Related Work: Genetic Programming on Embedded Systems. These capabilities, listed in 2.5 Summary of Related Work, can be summarised into the following objective:

Achieve distributed GP on a network of deployed constrained embedded devices

However, to ensure the framework is generally applicable (across a range of embedded systems), the framework will focus on operating with WSN mote-class devices since they typically represent the most constrained of the embedded systems. Additional specifications on the thesis scope are also presented in Section 3.1.

Within this defined scope, a framework designed to achieve distributed genetic programming on mote-class devices and is detailed in Section 3.2. An implementation of the framework on actual motes within a network of devices is subsequently described in Section 3.3. The mote implementation is then evaluated on a time-varying sensing-actuation problem (Section 3.4) and a second challenge requiring the evolution of communication (Section 3.5). Performance, benefits and deficiencies with the implementation are discussed in Section 3.6 followed by concluding remarks in Section 3.7.

3.1 Scope

Embedded systems encompass a diverse range of devices and hardware with greatly varying capabilities. In this section, the scope of embedded systems that the framework must support is focussed to microcontroller-based systems with similar characteristics to those discussed in Chapter 2. Specifically, this includes the 3 classes of embedded systems (small robots, WSN and IoT devices), however the scope is further narrowed to devices with communications capability as this will be essential for the requirement of distributed evolution of logic. Wireless communications is preferred as it offers additional autonomy and freedom for the device to move unencumbered from any tethered communications. The learning approach of the framework is limited to Genetic Programming for

reasons identified in Section 2.1.3. A specific summary of what constitutes a “mote-class" device and additional research directions outside the scope of this thesis are provided in 3.1.5 Scope Summary.

3.1.1 Mote-class Embedded System

As stated in Section 2.1.8, there is likely to always be a niche for computationally contained devices since they can be fabricated more cheaply, run with lower power and be smaller than their more computationally endowed counterparts. Taken to the extreme, one can imagine nano-scale devices capable of performing sense-compute-act cycles, and indeed significant research advances are being made towards this goal. Currently however, energy, sensing, computation, communication and actuation capabilities at this scale are very limited. Importantly, logic is not readily reprogrammable, the instruction set (if there is one) is extremely limited and there is typically no general working memory (like RAM that can be used for general computation. As such, running a GP engine on such devices is not currently a viable option.

At the micro and millimetre scales, traditional programming techniques can be employed through the use of microcontrollers. While devices are obviously larger, they are still sufficiently small to enable many application scenarios that simply weren’t possible a decade or so ago. The next scale of device is the PC-class machines, and while they offer significantly superior processing speeds, their size and energy requirements limit their deployment from the application scenarios that low power microcontroller-based solutions address. For the target embedded systems of interest to this research, microcontrollers were the predominant processor of choice and is therefore a target processor for this framework.

Of the embedded platforms discussed in Related Work: Genetic Programming on Embedded Systems, WSN devices appear to have the strongest motivation for being small and low cost since their utility is often coupled with their small size and ubiquity. This is evidenced in Table 2.1, by the trend of robotic and IoT solutions typically employing

higher performance microcontrollers (or even PC-class processors) than the microcontrollers employed by WSN devices from within a similar period. This is further supported by surveys [2,30] of WSN devices showing the typical RAM and ROM for such devices ranges from 4kB - 64kB RAM and 48kB-128kB ROM respectively[1]. Since the majority of devices typically resides within these ranges, this will be used as a minimum target specification for the framework. Similarly, processor speeds of the devices surveyed ranged from sub MHz up to 200 MHz and so this will also be treated as a minimum target range for the framework.

3.1.2 Communications Capability

Since an objective of the framework is to achieve *distributed* evolution, communications capability is implicitly required in order to distribute programs. WSN radios support the tradeoff between lower data rates and longer range versus faster data rates and lower range. Of the devices that support wireless communications in [30], the majority offer data rates from 10 kbps up to 250 kbps. Therefore the developed framework should operate with available data rates within this range.

3.1.3 Distributed Autonomous Evolution

Distributed evolution of logic does not imply that the evolved logic produces network-wide, coordinated or emergent behaviours. Furthermore, evolving collective network behaviour (requiring multiple entities to achieve desirable system performance) is likely to be a more significant challenge than evolving desirable behaviours for individual entities. For this research, the scope of interest is on embedded systems that can operate *independently*, but can also utilise neighbouring or networked entities to aid evolution of logic wherever possible. Nonetheless, achieving desirable network-wide behaviour is an interesting research challenge warranting some preliminary investigation.

[1] with some exceptions, notably the Intel mote2 which has 256kB RAM with 32MB ROM

3.1.4 Tree-based GP with Human Readable Program Representation

GP is one of many ML approaches that could potentially be employed for generating desirable behaviours on embedded systems, however various benefits of GP were identified in Section 2.1.3. Summarised, GP is a learning mechanism that can generate novel solutions using code as the logic representation and since processors natively use code, it can be argued that GP offers a mechanism to automatically generate any logic the device is capable of performing. Another advantage of using code as the logic representation is that it offers an intuitive mechanism for including human-devised logic (hence injecting domain knowledge with human programs) which can then continue to evolve. Furthermore, the resulting evolved logic is typically more human understandable (particularly to programmers) than most other ML representations which can be used to gain insights about logic that generates desirable behaviour. Like most EA approaches, GP is also readily parallelised or distributed and this has a natural synergy with the multiple entities in WSN, IoT and swarm/cooperative robotics applications. There are many variations of GP which could be employed. However, determining the best GP mechanism appropriate for embedded systems is outside the scope of this thesis. Instead, the prototypical tree-based GP with mutation and subtree crossover will suffice for the purpose of achieving GP-based learning on resource constrained embedded devices. For these reasons, the scope of learning metaheuristics for the framework is limited to standard tree-based GP.

3.1.5 Scope Summary

The aforementioned requirements and scope foci require a framework that can evolve logic using Genetic Programming (GP) on “mote-class" devices. “Mote-class" is defined as a device which employs a microcontroller with 4kB - 64kB RAM and 48kB-128kB ROM with wireless communications supporting data transfer rates from 10 kbps up to 250 kbps. The framework needs to support evolution of individual agent behaviours via distributed GP, however evolution of collective, emergent or system-wide behaviours is not within the scope of this research. Additionally, optimal configuration of the GP system will not be investigated but rather an attempt should be made to ensure a reasonable, useful

configuration is employed. For convenience and improved understandability, programs will be representable in standard 'C' language format, though not necessarily encoded or stored in this format.

3.2 Design

This section describes the design of the IDGP framework for achieving distributed genetic programming on resource-constrained, networked devices. Since GP implementations are typically memory intensive, memory constraints must first be considered. This is done in Section 3.2.1. As a result, Section 3.2.2 details a method for achieving compact program representation. For evolution of logic, a choice of selection and genetic operators is required. How operators for the compact program representation can be achieved are discussed in Section 3.2.3.1. Section 3.2.4 provides a guide for the execution and evaluation of programs, while Section 3.2.5 details considerations and recommendations for a communications subsystem.

3.2.1 Memory Considerations

The scope defined in Section 3.1 necessitates a GP engine capable of running on microprocessors with significantly constrained RAM and ROM. Unfortunately, since GP is a population-based metaheuristic, significant RAM and ROM resources are typically required to maintain the population of candidate solutions [138]. For example, if an individual program was 1kB (which is quite small for normal compiled programs), then a population of 100 individuals would require at least 100kB of RAM. This would not fit within the platform memory requirements defined in Section 3.1.5 and so a more sophisticated approach is warranted. However the framework will employ distributed evolution, and such mechanisms (like the Island Model) can evolve solutions using a small local population per agent, but yield the equivalent performance of one large population roughly the size of all the small populations combined.

Nonetheless, if the population per entity is too small, the lack of local genetic diversity will often cause the evolutionary process to prematurely converge to a poor solution [65,140]. One should also be aware that too large of a population may require an excessive number of function evaluations before the system will converge (though this is less likely to be an issue for constrained platforms). The appropriate population size for EA implementations is commonly determined by rule of thumb based on the complexity or dimensionality of the problem being addressed. As a result, the population size must be prescribed by the user since the framework has no *a priori* knowledge of the problem complexity. Nevertheless, the amount of RAM required for a standard GP population is not only a function of the number of individuals in the population but also the size of each individual. As such, keeping the representation of individuals as small as possible will maximise the number of individuals that can be supported within the memory constraints. Considerations for achieving compact program representation are discussed in Section 3.2.2.

Once the appropriate program representation has been decided, how the programs will be evolved must also be determined. The memory required will depend on the mechanism employed to generate a new population (discussed in Section 3.2.3). The generation of the next generation will require at least some additional memory, potentially as much as needed for the current population unless in place substitution is employed. In place substitution however runs the risk of prematurely reducing the population diversity. It is possible to use other memory storage mechanisms such as flash (program memory), however these typically have a limited number of write cycles, so this is not recommended for embedded devices. Discussion of a basic selection of (prototypical) GP operators is provided in Section 3.2.3.1.

A small amount of additional working memory may be required for the execution of programs. Specifically, memory for variables used by the programs need to be supplied by the execution process (c.f. Section 3.2.4) or virtual machine (c.f. Section 3.2.2.1). Ideally, if the population is already in RAM, then the execution process should be able to execute the program in place (via pointers) and not require an additional copy in memory.

Finally, the implicit memory (and processing) savings by performing *in situ* evolution

should be considered. As [16] states “the world is its own best model” and so evaluating solutions with it potentially eliminates the need to simulate the world. This is suitable for resource-constrained platforms, however it does come at a cost. Simulations tend to idealised, which often makes them faster than realtime to evaluate solutions with, though it can introduce the “transference problem”. Additionally, parallel simulations can be conducted without interference, unlike in the real world, and so this also increases the number of evaluations that can be performed within any period. Evolution typically requires many generations before converging or evolving to good solutions. Ideally, light-weight simulation that improves its accuracy based on the outcomes of intermittent real-world evaluations should be employed if the device has enough resources to do so. Examples of such approaches can be found in [96,109].

In summary, the memory constraints of microcontrollers will likely restrict the population size. Therefore, every effort should be made to ensure the representation of individual programs is kept as small as possible.

3.2.2 Compact Program Representation

This section provides considerations and recommendations for achieving compact program representation. Essentially, it is recommended that a virtual machine Section 3.2.2.1 be employed to abstract physical hardware and allow the execution of high level instruction in programs represented using prefix notation (Section 3.2.2.2). Additionally, program metadata is discussed in Section 3.2.2.3 and a novel program identifier presented as a useful means for fast, course approximation of diversity between programs.

3.2.2.1 Virtualisation of Hardware

As discussed in Section 3.2.1, achieving compact program representation is important. One method for achieving this is evolution of direct machine code as discussed in Section 2.2.3 since this is a one to one mapping with the instructions of the device architecture. However, the low level representation has drawbacks with management of the program execution. Significant care must be taken to ensure that the processor cannot execute

code that does not yield, accesses memory it shouldn't or execute instruction that could halt or crash the device. Additionally, there may other (framework) services being ran on the microcontroller that should not be interfered with by the evolved logic. Furthermore, in networks of devices with different processors (which are becoming more common due to convergence of communications standards), direct machine code will not be able to be shared usefully to other devices. For these reasons it is recommended to"virtualise" the hardware so that programs are sandboxed and can be safely terminated and cannot access restricted operations or memory.

A process virtual machine (VM) offers a software mechanism to easily abstract the sensing, computing and actuation capabilities of any platform. It can also provide a compact representation of complex logic by employing high-level instructions. Each VM instruction, which is usually represented by only a few bytes, can potentially map many native instructions to a single instruction. This means the number of bytes required to represent high level complex programs is likely to be far less than the equivalent machine code representation and furthermore such programs could run on different architectures. Additionally, a smaller number of high level instructions are typically easily understood than many low level instructions. There are two disadvantages with employing VMs however. Firstly, a VMs usually requires significant resources (in RAM and ROM and CPU utilisation). Secondly, using high level instructions potentially reduces the novelty of solution that will be achieved since these are likely to be used as building blocks for more complex behaviour rather than other novel solutions emerging from *tabula rasa*. VM

However since we aim to achieve complex behaviour on networks of devices (which may have different microcontrollers), we recommend a VM approach. A light-weight VM can largely be implemented with an "execute" function that takes the program and iterates over the instructions and executing the appropriate function calls that they map to.

3.2.2.2 Prefix Notation

Assuming a VM approach is adopted, the program representation needs to be decided. How the programs are likely to be used and manipulated should be taken into account.

Like any modern architecture, the native architecture size needs to be set. This dictates the size of each instruction, so ideally we wish to have a small size. Typically these range from 8-bit architectures, 16-bit, 32-bit and more recently on PC-class computers 64-bit architectures. 8-bit architectures are becoming less common, even for microcontrollers these are often too restrictive since it is more difficult to access memory locations larger than the native architecture bit-width. Nonetheless, a VM is not restricted since it can emulate wider-bit architectures. The choice of bit-architecture should reflect the maximum complexity expected for the problem but in general one should choose the smallest possible that will meet the needs of the problem at hand.

Keith and Martin [63] suggests that a modular representation is that of a genome interpreter that uses a prefix ordering scheme, general data support, a 2-byte node representation, and a jump-table mechanism. This is a clean and modular approach, though not necessarily the most efficient. We also recommend the prefix notation scheme (over tree-based style using node pointers), since this will ensure the program size remains small which impacts the number of programs that can be instantiated as well as the number of packets (or bandwidth) required to transmit a program. A further advantage of prefix notation is that it can be used to guarantee syntactically correct programs. This is possible since the instruction set and number and argument types are known for each instruction.

To avoid the complexity of memory management of variables, we recommend fixing the number of variables available by effectively reserving instruction numbers (or “Op codes”) for the variables to be made available. With a case statement implementation the size of each instruction and the number of parameters can be stored in a lookup table so that the program tree can be quickly traversed without the need to evaluate any instructions. This is particularly useful for conditional branching, generating a program listings or determining the size of the code (or parts therein).

The crossover operation for standard prefix notation if fairly complicated compared to LGP [148]. With prefix notion (or node representation) a branch needs to be identified that can be cut and another branch inserted. However this requires type-checking before the branches are swapped and after the swap the maximum tree depth may have exceeded.

With LGP on the other hand, if programs are the same number of lines, one simply cuts at the same randomly selected line and swaps the 2 portions to yield 2 offspring of exactly the same size as the parents. Thus, LGP can avoid the computationally expensive machinery typical to tree-based representations [103] since the crossover point is between any line and connecting fragments is always valid.

Since we wish to employ tree-based representation however due to the benefit keeping complex nested functionality, we propose a hybrid light-weight representation that is a hybrid of the 2 schemes. This linear-tree-based-hybrid representation is effectively the same representation most programmers are accustomed to. An example of programs represented in this form is shown in Figure 3.1, as is the result of a crossover operation where different lines on the parent programs were used as the crossover point. The reason for this is that the number of instructions within a tree (the program is effectively a sequence of traditional tree-based programs) may differ. Therefore, ideally the crossover point in each program should roughly be before the line number where the total cumulative instructions up to that line number (summing all the trees up to the line number) added to the other program portion would exceed the maximum program size. Put simply, attempt to make the crossover points (line numbers) such that offspring do not exceed the maximum program size.

The instruction set will vary based on the platform's capabilities, and by the capability needed to address the system objective. The choice of bit architecture will limit the number of instructions that can be enumerated, however this is typically not an issue. A set of low-level instructions will likely take longer to evolve useful or complex behaviours, however it is likely to evolve more novel and possibly more efficient behaviours due to not being seeded with high-level functionality. As stated before however, more low level instructions will be required to represent complex behaviours, which in turn requires larger program representation. For this reason, we recommend biases the instruction set with instructions representing complex functionality, however including some low level mathematical and logical instructions can often be enough to generate novel solutions.

Name	Bytes	Purpose
Program ID	4	Unique program identifier which embeds information on the distribution of functions within the program
Mutation Rate	2	Specifies the mutation rate during program generation
Program Bytes	1	Specifies the length of the program in bytes
Program	1 - L_{Max}	The instructions ordered in prefix notation format

Table 3.1: IDGP Program metadata structure.

3.2.2.3 Program Metadata

In addition to the program instruction code (VM byte code), additional information (program metadata) may be desirable to have stored with the program. This is particularly useful for keeping any contextual information with a program if it is sent externally.

We propose as a minimal set of metadata that of Table 3.1. It is recommended that the program metadata block be placed before the byte code since when the full data is received, the metadata can be quickly parsed to ascertain how large the program is and potentially whether the program will be kept or not without even assessing the byte code.

The proposed metadata block includes a program identifier which is preferably unique to every unique program. That is, if 2 programs are identical, then they will have the same ID and so one might wish to reject such a program since it offers no genetic diversity to the local population. Extending this further, we propose that this ID be generated based on the frequency of instructions within the program using a histogram representation. For example, if we use the instruction set available to the yellow program fragment in Figure 3.1 and treat the fragment as a program, then it would have the following frequencies of instructions as shown in Table 3.2. Note that no structural information is conveyed, however this could make a good extension to the ID.

The difference between IDs (sum of the absolute differences for each instruction count) can be used as a crude metric of program diversity. This implementation uses 4 bytes for the ID and is simply calculated with a single parse of the program. Thus this is a useful, compact and easily computed unique ID that can also be used for diversity calculations or simply ascertaining the distributions of terminals and functions.

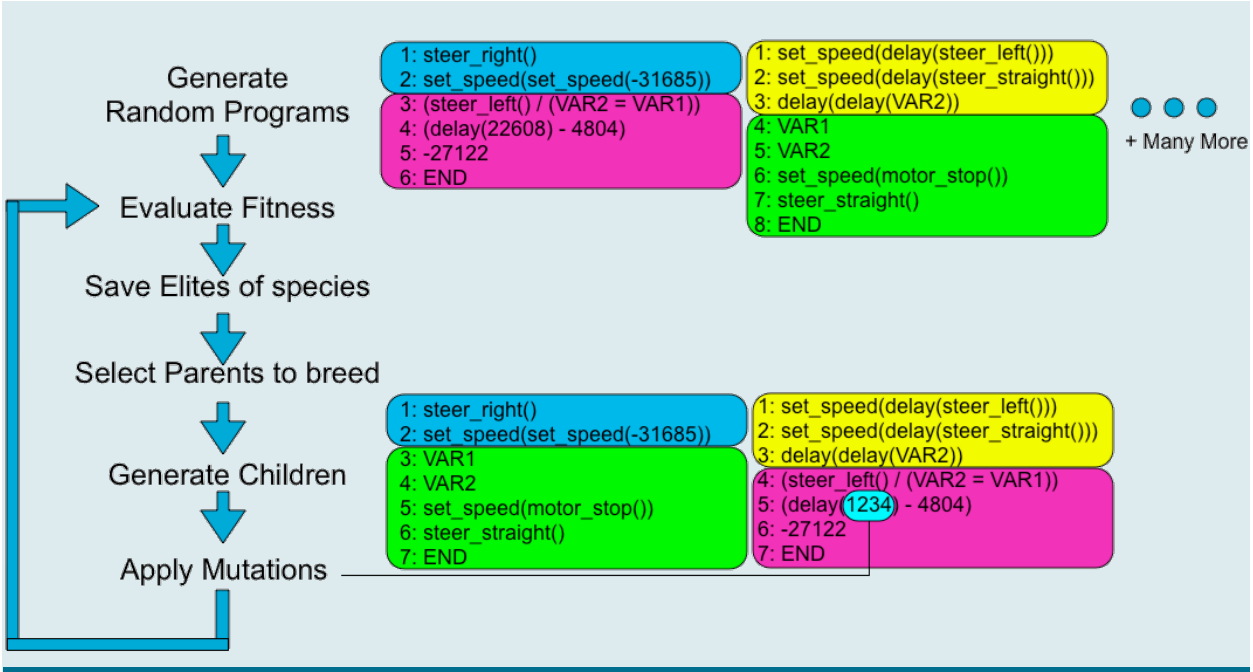

Figure 3.1: An example of 2 programs using the linear-tree-based-hybrid representation (top) and how crossover and mutation can be applied to generate offspring.

An epigenetic metadata field is reserved for the mutation rate of programs when they act as parents. This epigenetic information can enable faster learning and faster rediscovery of good solutions [130] when unexpected events in the environment cause a dramatic change to the fitness landscape, however this feature is not utilised in these experiments. The final metadata field stores the number of bytes in the program (which typically differs from the number of instructions due to optional data fields) and is used by the framework for transmitting programs as multiple packets. In total, only 7 bytes (L_{Meta}) are used for metadata, however this information is extremely useful to receiving nodes.

3.2.3 Program Generation

3.2.3.1 Generating Random Programs

Due to the limited memory of motes, there is a tradeoff between the number of programs that can be stored in the population and the size of the programs in that population. This tradeoff will have an effect on the evolution performance, however this impact is not studied here. Interestingly there is evidence to support that in many scenarios small

Instruction	Yellow	Blue	Magenta	Green	ID bits
const_int()	0	1	0	0	0-1
var_int()	1	0	2	2	2-3
steer_left()	1	0	1	0	4-5
steer_right()	0	1	0	0	6-7
steer_straight()	1	0	0	1	8-9
motor_stop()	0	0	0	1	10-11
delay(Arg1)	4	0	1	0	12-13
set_speed(Arg1)	2	2	0	1	14-15
+(Arg1,Arg2)	0	0	0	0	16-17
−(Arg1,Arg2)	0	0	0	0	18-19
×(Arg1,Arg2)	0	0	0	0	20-21
/(Arg1,Arg2)	0	0	1	0	22-23
=(Arg1,Arg2)	0	0	1	0	24-25
unused					26-31
ID (in hex)	0000B114	00008041	01401018	00004508	0-31

Table 3.2: Example instruction histogram for the code fragments in Figure 3.1 (top).

populations are still very effective [140]. Rather, the limited memory of the device is accepted and effort made to utilise the memory to produce an acceptable population size.

For simplicity, and without loss of generality, IDGP currently implements a user-defined maximum program length and fixed population size to address this design decision. As the IDGP engine is assembling the program, it uses the number of remaining instructions to select instructions whose number of required arguments does not cause the program to exceed the maximum program length. This approach induces a small syntactic bias towards functions with fewer parameters and the bias is obviously greater when the maximum program size is nearer to the maximum number of arguments for any function. However, as the maximum program size increases with respect to the maximum number of arguments, the syntactic bias diminishes.

A function selection bias, represented as a discrete probability distribution function, is used to probabilistically select instructions. This is currently user-supplied and used to bias the generation of programs away from extremely nested programs. If all functions have equal probability of being selected, the likelihood of generating deeply nested programs becomes high while the number of generated program lines becomes low.

Finally, a user-defined maximum nesting (depth) level constraint is also imposed. During program generation, the IDGP engine passes terminals into all arguments of the current function if it reaches the maximum nesting. The combination of these constraints and biases generate programs with a number of nested statements that is comparable to human generated code. Section 3.2.3.1.3 elaborates on the motivation for multi-line programs.

Figure 3.2 shows the relationship between nesting and the average lines of code per program for *any* given maximum program length. This relationship is controlled by the probability of selecting a constant P(C). While the plot indicates the theoretical relationship between nesting and lines of code, the X mark near the plot indicates the achieved program nesting and number of lines for an example of P(C)=0.5 and a program length of 42 bytes.

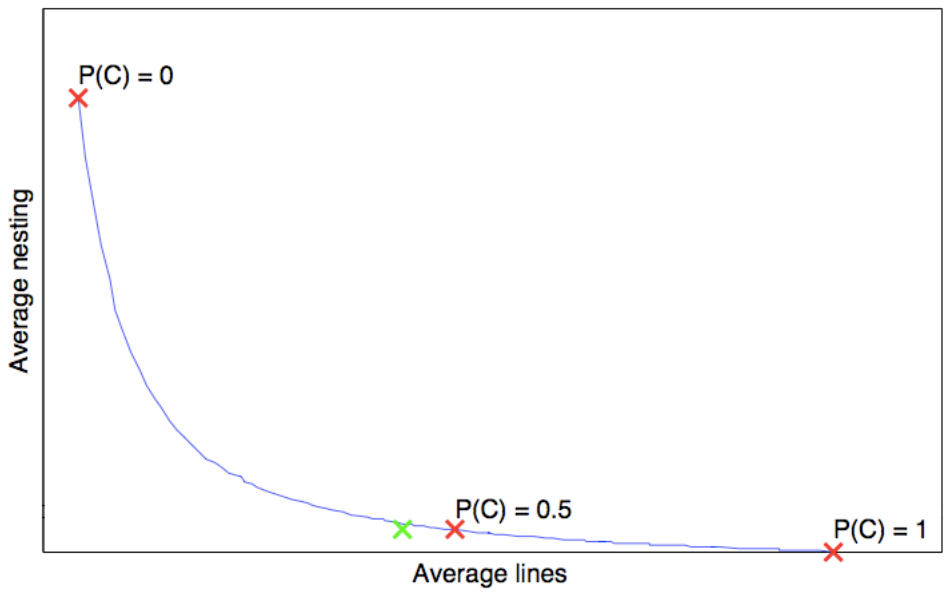

Figure 3.2: Relationship between nesting level and the average lines of code for generated programs. The nesting level and number of lines for 5a (c.f. page 78) is indicated by a cross.

The memory footprint required for the gene pool is N_{pop} multiplied by the maximum program length, L_{Max}. However two identically sized populations of programs are allocated so that one population represents the current generation being executed, while the other is used to store the programs for the next generation. This approach requires

$2 \times N_{pop} \times (L_{Max} + L_{Meta})$, where L_{Meta} is the length of any program metadata for a particular program.[2] Consideration in this manner appears somewhat novel, and we refer to the investigation of classes defined by the combination of operators used to generate portions of a population as **demographic analysis**. While demographics appear to have been little studied, the standard genetic operators of mutation, crossover and cloning/elitism have received considerable attention.

Using the operators as previously discussed, we generate five "typical" classes and define them as follows:

For ease of notation, the population composition is expressed as $[N_E \ N_H \ N_C \ N_R] \ N_O]$ representing the number of individuals *to be generated* in each of the following classes:

- E: Elites - no operators used. Top ranked individual is cloned N_E times into the new population unchanged
- H: High Ranked Mutants - mutation operator used on biased selection (top few) of individuals
- C: Children - single-point crossover using biased parent selection, mutation applied to child
- R: Random - randomly generated individuals
- O: Other - individuals supplied by "other" sources such as immigrants from other populations via the Island Model or human-devised solutions

For reasons that will be discussed later, these are ordered by their expected (not evaluated) fitness. The expected fitness of "classes" is less studied since typically how effectively they explore the search space would be of interest rather than their average (expected) fitness with respect to the rest of the population.

By including other operators, many other combinations, and hence "classes", can be realised. There is likely an optimal balance of class representation which maximises

[2]With fixed maximum program sizes it is possible to create the next generation with only 2 additional code slots of size $(L_{Max} + L_{Meta})$ providing knowledge of which programs are to be used in generating the next population is known/stored in memory.

learning, however it will be problem and representation specific. Determining the appropriate distribution for optimal learning is typically achieved through rule-of-thumb heuristics and/or insight of the specific problem. This challenge is outside the scope of this research, however the design parameters still need to be considered and an appropriate default learning distribution chosen.

3.2.3.1.1 Elites

The Elitist Strategy, or Elitism, introduced by De Jong, helps local search at the expense of the global search (essentially helping to remember where the local search is, but that "memory" could have been used for further exploration instead

After it executes all (N_{pop}) programs in the current generation, the IDGP engine ranks these programs (step 3) according to their achieved fitness. It then copies the N_E programs with the highest scores into the next generation (elitist selection).

3.2.3.1.2 Mutants of Highly Ranked

Often simple mutations can assist the speed to evolve an acceptable solution, so we also generate N_H slightly mutated programs with high rankings. This employs a strongly biased selection operator, then followed with the mutation operation.

3.2.3.1.3 Children

Next, N_C children programs are generated from a biased (fitness proportionate) selection of N_C parents as the single-point crossover , and recombination yields 2 children programs for each pair of parents chosen via fitness proportionate selection without removal. Fitter programs may be selected multiple times, and even selected to breed with themselves. This may not necessarily generate a clone of itself since the crossover point does not need to be symmetric.

This ability to create two new solutions from the same solution represents a significant added capability of GP relative to typical Genetic Algorithms (GA) [70].

When using prefix notation representation, crossover points need to be carefully chosen to ensure the resulting programs are syntactically correct. The current implementation uses a simple approach of cutting programs at a zero nesting level to ensure the generation of syntactically correct offsprings, whilst guaranteeing that the genetic information from both parents is completely passed into the next generation. This is somewhat similar to how humans "evolve" their code, by manoeuvring portions of code rather than modifying a single monolithic line of code.

During the evaluation of programs, *other* programs may be transmitted from neighbouring motes. These programs have the potential to be worse than random if their objective is anti-correlated with the local mote's. Communications is also not guaranteed and so it is unknown how many *other* programs will migrate to this mote within one generation. We set an upper bound on how many *others* can be received within one generation and simply drop additional *other* programs if the allocation has been reached. We term the number of *other* programs that do arrive as N_{others}. Finally, the remaining ($N_{pop} - N_{elite} - N_{highrank} - N_{children} - N_{others}$) programs are randomly generated programs. The newly created population is then ready to be evaluated.

The genetic operations and generation of the new population scales linearly with the number of programs in the population. The IDGP overhead for the current population size of $N_{pop} = 21$ is less than 100 ms per generation. This overhead is typically relatively small compared to the population evaluation time. For example, with a program evaluation time of 1 second per program, the overhead is <0.5%.

3.2.3.1.4 Others

The final class of programs is referred to as "others" indicating that they have been supplied external to the local genetic operators. Mechanisms by which others can be supplied include immigrant programs via the Island Model and potentially seeded programs provided by a user. The expected performance of programs of this class is unknown since they do not evolve naturally within the population, however they can be a useful way to inject domain text or inject genetic diversity from other populations.

3.2.4 Program Execution and Evaluation

Each VM instruction runs for a set maximum time of execution before it must yield, then there is a delay up to the next regular interval.

The evolutionary process however requires an assessment of individual (program) fitnesses. The best indication of an individual's fitness can be had from evaluating the individual in the real world. However, since there is a population of individuals to be assessed, a mechanism is needed for scheduling the evaluation of programs and so this is discussed in Section 3.2.4.

3.2.4.1 Program Scheduling and Execution

To ensure that IDGP programs are isolated from core functionality, so that runaway programs can be safely killed if necessary and so that actuations are kept within safe ranges. Similarly, the architecture ensures that the IDGP engine can obtain feedback about how the system is performing, in order to evaluate IDGP-generated solutions. Development of architectures appropriate for IDGP deployment is an ongoing research issue which will not be addressed here.

The *user policy*, which includes the system fitness function and other initialisation parameters, serves as a key input into IDGP. With the framework running on every mote, neighbouring motes can exchange locally generated programs in a form of cooperative evolution which may reduce the time to settle on an acceptable solution.

The genetic program engine initially generates a pool of programs in a random fashion, biased by the function selection probabilities, as shown in step 1 of Figure 3.3. Each program in the current generation is evaluated in turn (step 2) where the GP engine launches a virtual machine interpreter thread to run the evolved code and another to evaluate program fitness. The fitness evaluation thread runs in parallel with the execution thread since fitness may need to be calculated over the execution period of the program. After a specified evaluation period, the GP engine stops and kills the evaluation thread, to avoid potential deadlocks in execution.

3.2.5 Communications Subsystem

Providing a communications subsystem is essential for distributed evolution, and managing communications more generally will be required for devices with wireless communications capability. The framework will need to support sharing of programs between devices as well as other non-program data such as fitness scores and engineering data and remote procedure calls (RPC). Additionally, it may be desirable for the devices to evolve logic that also performs communications. As such, the communications subsystem should be transparent to the evolving logic. This can be achieved by virtualising the communications functions in a way that avoids the GP process having to evolve dealing with subsystem packets. In many light-weight operating systems, this can be easily achieved by dedicating a socket (or channel) to VM transmit and receive functions. On the architecture side, this may be achieved via software (usually an identifier in the packet header and a receive process which multiplexes the packets to the appropriate sockets) or in hardware (by using a different physical channel, or MAC address filtering by the transceiver). Furthermore, broadcast packets (rather than unicast or routed) are recommended for simplicity and that other types of communications could be evolved from the basic broadcast capability. In some circumstances non-broadcast packets may be better suited, however in the absence of sufficient domain knowledge suggesting otherwise, we recommend employing broadcast. Importantly however this does not imply that broadcast is suitable for framework or RPC communications.

As with any wireless communications system, issues such medium access control, bandwidth utilisation, and data corruption need to considered. Many of these issues are increasingly handled with hardware and by the operating system and so will not be discussed relative to the “engineering” packets. Nonetheless, the evolved logic can impact on these issues also and must be considered. If the evolved logic is allowed to communicate freely, then this could lead to congestion and impact on the ability of the framework to function correctly. Specifically, programs may not be shared and global fitness feedback would also be effected. As such, it is recommended that the VM ensure that a maximum rate of packet transmission is imposed.

Digital radio transceivers often have a maximum packet size that can be sent. While this size is increasing as the technology advances, having the ability to break a program into multiple packets and reconstruct the full program on the other side may be essential in order not to restrict the program size. There are many methods for achieving this, including some that achieve this with the transceiver hardware (which is preferable). Regardless of the solution employed, it is recommended that a checksum calculation like CRC be employed to ensure the integrity of the received program. Again, the system user should ensure that the rate of transmission of programs will not saturate the available bandwidth as to cause detriment to other services.

In summary, the communications subsystem needs to appropriately prioritise communications resources in the following order. Remote procedure calls should have highest priority so that the system state can be probed and any user action (such as halting the framework or execution of an evolved program) can be affected immediately. It should be attempted to ensure framework messages not interfere with the communications from evolved programs being executed and vice-versa. However, the framework should take precedence in terms of bandwidth utilisation in order to avoid evolved programs impacting on framework packets (such as dropping packets with global fitness information). Finally, framework and RPC messages should not be visible to evolved logic.

3.2.6 IDGP Design Overview

The recommendations in this section culminate in the In situ Distributed Genetic Programming (IDGP) framework which can be used to implement genetic programming on resource-constrained platforms. Figure 3.3 provides a simplistic representation of the IDGP framework's core architecture. A fixed size population (N_p) of compact programs of fixed maximum size are sequentially executed simultaneously evaluated. The fitness can be supplied externally or be local to the evaluation thread all be a combination of both local and external (global) fitnesses. At the end of a generation, the programs, including any received during the evaluation of programs in this generation, are ranked and various

IDGP

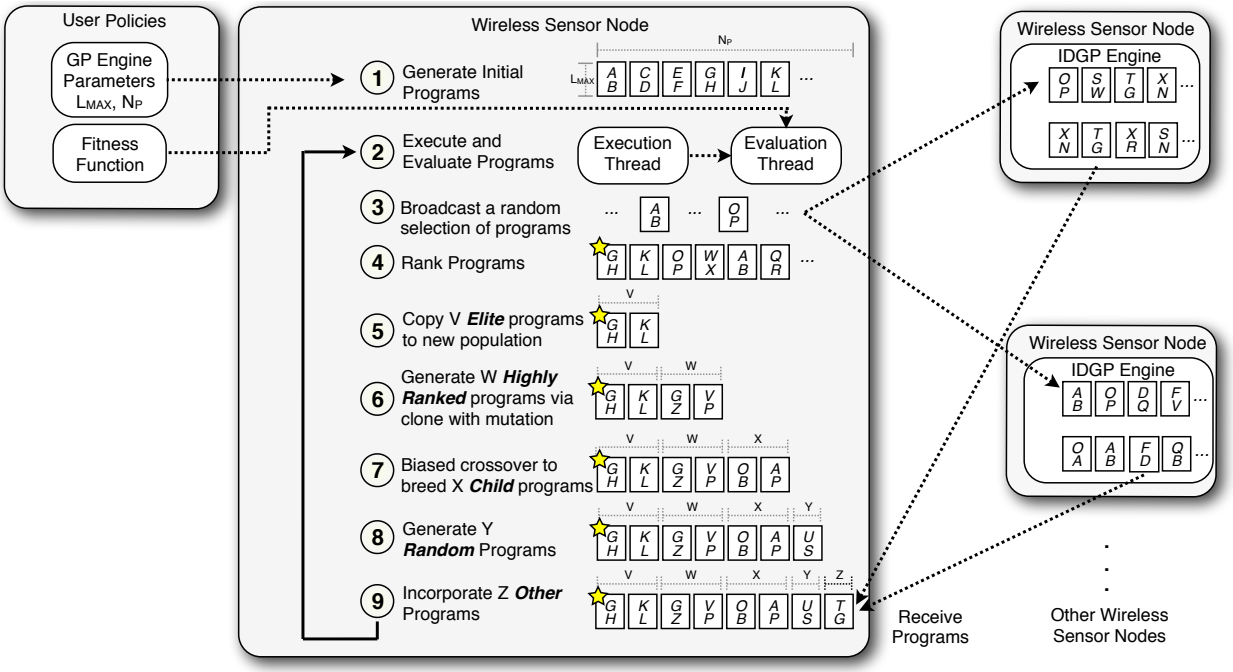

Figure 3.3: An overview of the In situ Genetic Programming (IDGP) Framework.

classes of programs are generated to comprise the new population. Additionally, in accordance with the GP settings supplied by the user policy, a number of programs will be broadcasted to neighbouring nodes. Note, this may require fragmenting the program and reconstructing it on the receiving side. The cycle is then repeated according to the user policy. This GP process occurs on all nodes in an asynchronous manner.

3.3 Implementation on Motes

Using the framework modelled in Section 3.2, we now look to create a physical instantiation of the IDGP framework. This requires first choosing the device hardware which is described in Section 3.3.1. Upon deciding the hardware, Section 3.3.3 specifics of how the framework is implemented (in software) on the specific architecture.

3.3.1 The Fleck3b Mote Platform

There are many WSN platforms available [2], however the CSIRO Fleck™3b platform [114] was readily available, and importantly, it meets the criteria identified in Section 3.1.5.

The Fleck™3b employs the Atmega1281 microprocessor running at 8 MHz with 4kB EEPROM, 8kB SRAM, 128kB program flash. The low power microcontroller is complemented with the Nordic NRF905 low power digital transceiver (<100mW during transmit, < 43mW during receive) which enables the Fleck™3b to communicate at 50kbps across 1 km using a 915 MHz quarter-wave whip antenna. The Fleck™3b can measure many external sensors though the ADC connector block and also measure onboard sensors such as the power used by the mote. The platform has features 3 onboard LEDs, a red, a green and a yellow, which are software controlled. An overview of the board layout and functionality is shown in Figure 3.4.

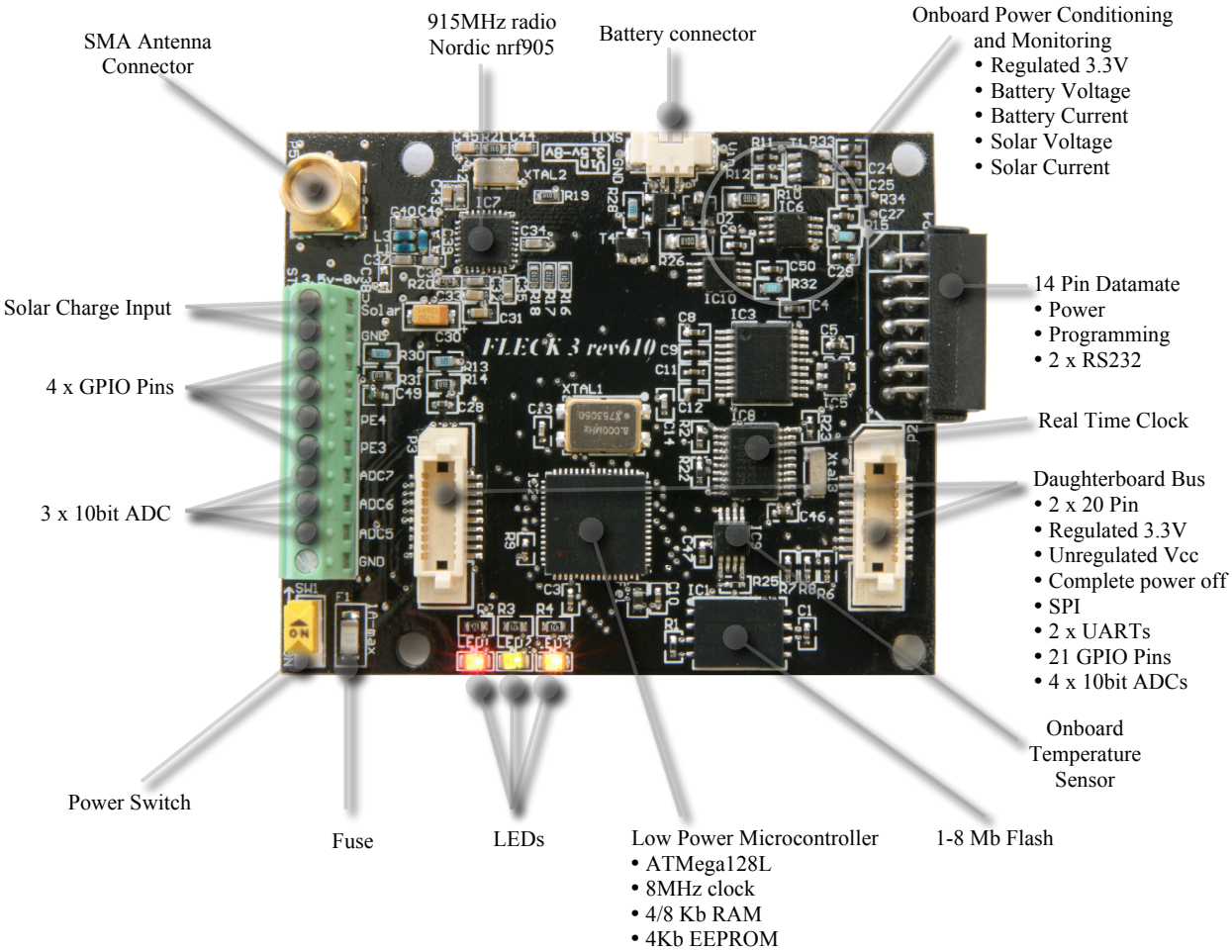

Figure 3.4: Fleck3b hardware functionality overview.

Figure 3.5: Fleet of “Fleck cars” (circa 2008).

The Fleck™3b platform has been used for numerous research and real-world applications [20,39,47,48,99,101,115,120,137,139] though mainly for outdoor applications such as environmental monitoring and animal tracking/monitoring.

Preliminary testing of the IDGP framework was performed with Fleck™3b devices that were embodied in small remote control cars where the original motor controller circuitry was replaced by the mote platform. An infrared range sensor (Sharp GP2Y0A21YK) was attached to the front and connected into an ADC and the motors were controlled with a specific motor controller daughterboard. The initial implementation of IDGP was performed on these small robots and a number of behaviours were evolved. The “Fleck cars” (shown in Figure 3.5) were trained to avoid crashing into walls, however they also demonstrated online learning capability. This was achieved by using the range sensor as the fitness feedback. When the behaviour of the car was desirable, the trainer could place their hand in front of the range sensor to provide a level of fitness feedback including not putting the hand in front (lowest fitness). Through this, one could evolve a behaviour such as mostly driving forward, but then later retrain it to drive mostly in reverse by providing appropriate feedback when the desirable behaviour was displayed. This investigation was

not performed under experimental design and so have been omitted from this thesis. The maintenance, due to the cheap remote control car components frequently failing, was deemed too excessive. As such, the Fleck™3b devices had light sensors attached to the onboard ADCs to provide sensing input, while the onboard LEDs provided actuation capability that could affect the environment and also be sensed by the light sensors. This configuration (shown in Figure 3.7) provided a low maintenance mote-class device with sensing and actuating capability suitable for experimentation.

3.3.2 Program Representation

The current IDGP implementation employs a hybrid tree-based program representation as suggested in Section 3.2.2.2, and supports the int16 type and functions with up to 3 arguments (though is easily extensible to additional types and functions with more arguments). For ease of implementation each op code is passed to a switch statement and the corresponding code called. The Prefix, Jump-Table (PJT) approach suggested in [63] requires slightly more complexity in constructing a program, however the PJT approach is more modular and cleaner.

The op code representation differs from typical GP C++ representation such as that used in [63] in that variables and constants are not memory references. Instead, IDGP directly embeds variables and constants into the code as either the variable number or the actual constant value. The prefix notation implemented uses 1-byte instructions and an optional 2-byte datum field if required by the instruction. Constants (signed 16-bit integers) are currently the only instruction which use this datum field, and are therefore represented by 3 bytes. The advantage of the datum field is that constants are directly embedded in the program rather than pointing to an external reference which would have to be moved with the program when it is shared. Two bits of the instruction byte are reserved for a variable index which removes the need for the 2-byte datum field when the variable instruction is used. When only 1 byte is used for the op code, this approach limits programs to a maximum of 4 variables and a maximum instruction set of 64 instructions. The obvious advantage to this representation is however an extremely compact program

Table 3.3: Instruction Representation

Bits	0:1	2:7	8:23 (Optional)
Use	VAR index	Op Code	Datum Field

with nearly all instructions using only a single byte. Table 3.3 shows how instructions are formatted.

3.3.3 Program Evaluation on Fleck OS

The Fleck Operating System (FOS) [21] is a small code footprint, cooperative threaded operating system designed for low power, embedded devices with limited ROM and RAM. FOS was originally implemented for the Fleck™3b hardware previously described in Section 3.3.1. FOS applications are written in C and support numerous high level functions such as routed radio messages, sensing of onboard sensors and external sensors and remote procedure calls via radio. The cooperative threading model allows complex programs to be written in a compact and elegant manner which is easily understood. IDGP was implemented as a FOS application with each program running as a thread and supervised (spawned and killed) by the 'evaluation thread'. Since FOS is not preemptive threading, the evaluated programs have to explicitly yield to give control back to the 'evaluation thread'. This was achieved by placing yield statements between each instruction (in the virtual machine program execution) and ensuring that all instructions either completed or yielded within a short period.

Programs that are distributed for the Island Model are broken into fragments and broadcast (single-hop). Once all fragments of a program are received, the program is reconstructed and pushed into the queue of immigrant programs where it can be used by the GP engine during construction of the next generation of programs.

3.4 Evaluation on a Sensing-Actuation Problem

To illustrate various aspects of the IDGP framework, we have devised a problem with a known optimal solution and implemented the IDGP framework on a mote-class device

in order to solve the problem. However, unlike many GP systems which perform evolution offline and are validated through simulation, we perform all evolution *in situ* (i.e. on the physical motes). Thus experimental results are based on online performance values rather than simulation.

The first part of this section introduces the mote class hardware platform (Section 3.3.1) and describes of the designed objective function (Section 3.4.1). The second part of this section evaluates the IDGP framework for individual mote evolution, homogeneous evolution, heterogeneous evolution, and response to unexpected events (Sections 3.4.2.1-3.4.2.5).

3.4.1 LED - Photodiode "Blink3" Problem

This experiment aims to evaluate IDGP on a non-trivial WSAN application that requires the sensing and actuation to be performed on a time-varying basis and demonstrate resilience of logic under changed conditions.

There are a number of well known "benchmark" optimisation problems within the GP community [70] and many in the broader optimisation community [100]. These problems typically have well defined solution spaces and are useful for comparing differences in optimisation techniques. However, none of these would demonstrate the capacity of the IDGP framework to generate WSAN solutions: in isolation, based on local conditions; in a neighbourhood of motes with the same objective but each with their own local characteristics; in a neighbourhood of motes with differing objectives, and despite environmental changes.

3.4.1.1 Time-Varying Sensing-Actuation Requirement

To demonstrate evolution of WSAN logic, a sensing-actuating objective function is designed that uses the 3 onboard LEDs as the actuation capability and a light sensor (photodiode) as the sensing capability. The objective is to maximise the light intensity received by the photodiode at three time checkpoints while penalising energy usage at 6 time checkpoints. The objective is weighted so that the optimal logic (shown in Figure 3.6)

BLINK3 PROBLEM

would result in the LEDs blinking 3 times during the evaluation period of 500 ms For this reason, we call this the “Blink3” problem.

The light sensor (photodiode) is oriented roughly facing the onboard LED (Figure 3.7) and is measured by the onboard 10-bit ADC. Ideally ,the photodiodes should have a reasonably narrow pass band filter as shown in Table 3.4, and therefore more responsive to some LEDs than others. In practice however, each sensor will vary slightly in its responsiveness to the same light and be effected by its orientation and distance relative to the LEDs. The photodiode has a maximum forward voltage of 0.8 V corresponding to a maximum ADC reading of approximately 248. The Fleck™3b wireless sensor platform

Centre Frequency	Bandwidth (FWHM)
550nm (green-yellow)	70.0nm (515-585)
650nm (red)	70.0nm (615-685)

Table 3.4: Both with filter transmissivity of 75% and using the Intor T5 detector. Note that LEDs that do not correspond to the centre frequency of the filter will still contribute to the ADC reading however, the value is attenuated substantially compared to the matching LED colour.

has 3 different coloured LEDs - red, green and yellow. A photodiode with a filter centred around green (550nm) or red (650nm) is fed into an ADC. The photodiode is placed to face towards the LEDs (see Figure 3.7). The photodiode measures a reading which ideally is responsive strongly to the LED matching the filter and weakly responsive to the other LEDs.

As each configuration will vary in position of the sensor and responsiveness of the sensor, each configuration has been placed into a light absorbing bag and the individual LEDs excitation voltage (ADC reading) measured and presented in Table 3.5. Note that the filter has a 70nm (FWHM) bandwidth so some light from the orange sensor contributes to the sensor reading of both green and red light sensors.

Basic idea is to have some minimal in situ example.

Situatedness (or embeddedness) in context to Embodied embedded cognition, refers to the idea that physical interaction between the body and the world strongly constrain the possible behaviours of the organism, which in turn influences (indeed, partly constitutes) the cognitive processes that emerge from the interaction between organism and world.

	set_leds(0)	set_leds(1)	set_leds(2)	set_leds(3)	set_leds(4)	set_leds(5)	set_leds(6)	set_leds(7)
ID	- - -	- - Y	- G -	- G Y	R - -	R - Y	R G -	R G Y
81	13.8 (5.8)	49.0 (0.98)	41.6 (0.96)	51.7 (0.92)	17.7 (4.56)	49.0 (0.97)	41.6 (1.03)	51.6 (0.97)
85	11.7 (4.7)	48.0 (0.99)	45.2 (0.97)	51.9 (0.77)	21.3 (2.55)	47.8 (0.85)	45.2 (0.90)	52.0 (0.77)
87	3.0 (3.4)	48.3 (0.91)	41.1 (1.04)	51.3 (1.09)	15.4 (3.55)	48.4 (0.92)	40.9 (1.01)	51.0 (1.03)
88	3.7 (4.6)	40.3 (0.83)	35.5 (1.10)	43.7 (0.88)	31.1 (1.52)	42.4 (0.79)	39.0 (1.01)	45.0 (0.83)
90	3.4 (4.0)	43.0 (1.19)	40.4 (1.17)	47.5 (1.09)	15.9 (3.89)	43.3 (1.12)	40.9 (1.16)	47.7 (1.15)

Table 3.5: Mean ADC readings (std) for all combinations of LEDs for the 5 nodes with no other external light sources present. R,G,Y indicates which LEDs (red, green and yellow) were on. Note each mean value was calculated from 200 samples taken periodically every 100 ms.

Program 4 The "*Blink-3*" objective function

```
fos_leds_set(0)
idleOffset = fos_power_battery_current()
delay(50)
isOdd = TRUE
WHILE program_is_executing
  IF (isOdd)
      fitness = fitness + fos_adc_read()
  END
  LEDmA = fos_power_battery_current() - idleOffset
  fitness = fitness - 6 * LEDmA
  isOdd = not(isOdd)
  delay(100)
END
```

In this these we refer to this property as being "in situ"[3] One of the objectives of the framework is to work within real world domains. Hence we want to subject the nodes to "real world" effects. This implicitly occurs since it requires actuation into the world (LED) and sensing from the real world (photodiode).

The LEDs affect the local environment by providing illumination but are not necessarily the sole influence of light in the environment. Sunlight, room lights and reflections are just some of the "external" light sources that can contribute to the ambient light, and in turn affect the value read by the light sensor. These "external influences" are often unpredictable, however are useful for demonstrating responses to unanticipated environmental changes. At first instance, we avoid exposing the motes to external light by placing the

[3] *in situ* is a latin phrase meaning "in place", "in position", "locally", "on site" etc

motes in a box, lined with a light absorbing material and spaced such that the effect of the LEDs from any mote would have minimal effect on other motes' light sensors. This enables us to focus on the learning and performance of the IDGP framework. The light absorbing material surrounding each mote was depressed to further reduce the influence of light generated from neighbouring motes. Because the position and orientation of the mote relative to the light absorbing material can affect the sensor readings, no contact with the motes was made during the main experiments. Optimal solution fitness for each mote was measured before each experiment and used to normalise fitnesses for subsequent experiments.

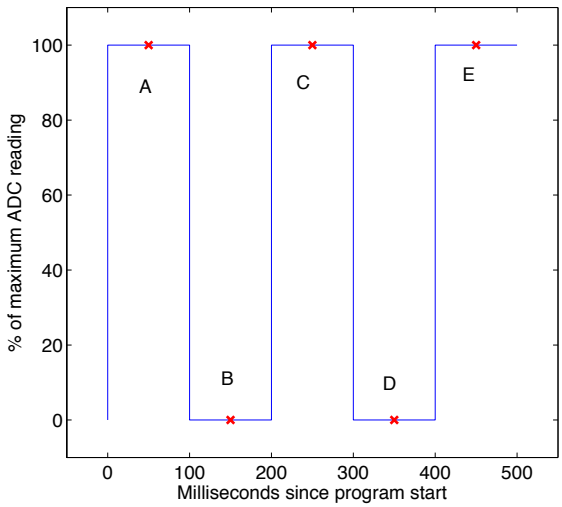

Figure 3.6: Sample points shown for the Blink3 objective on an optimal solution (Program 5c)

The physical set up of each mote is to have a light sensor (photodiode) connected to an onboard 10-bit analog-to-digital converter (ADC) on the mote (known as a Fleck3) with the photodiode roughly facing the onboard LEDs as shown in Fig 3.7. The "Blink3" objective function is structured in such a way that the optimal solution is to blink the LED corresponding in wavelength nearest to the pass-band of the light sensor, 3 times within the 500 ms evaluation period as shown in Figure 3.6. With the reduced instruction set used and optimal selection probabilities, the probability of generating the exact intended optimal program can be calculated as $1.0E^{-34}$, however the probability of generating a functionally equivalent program is much higher. Due to the syntactic richness of the GP engine (multiple lines of code, nesting and a selection of functions), calculating the exact

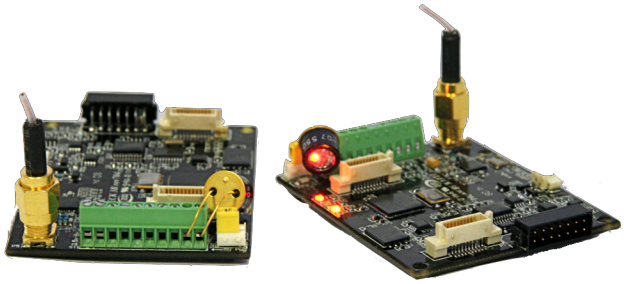

Figure 3.7: Fleck™3b motes with a photodiode connected to an ADC input (shown left) which is facing the onboard LEDs (shown right)

probability of generating a random program that is functionally equivalent to an optimal solution is intractable.

The photodiodes are passive sensors and do not draw any power from the Fleck. The onboard LEDs consume approximately 3 mA each and can be measured by the onboard current sensors. Note that LEDs are turned off before any program is executed and evaluated. The penalty of the LED current draw is weighted such that a maximum fitness is achieved by toggling only the LED that most closely matches the light sensor filter band.

ADC readings at 50 ms, 250 ms and 450 ms, and subtract the weighted current draw at 50, 150, 250, 350 and 450 ms. This produces an objective function described in pseudocode by Program 4. An optimal solution to this objective function is to have only the LED corresponding to the filter on at 50, 250 and 450 ms and all LEDs off at 150 and 350. As there are 3 off-to-on LED transitions in the solution, we coin the name *Blink-3* for this objective function. A visual representation of this optimal solution is shown in Figure 3.6.

3.4.1.2 An Optimal Solution

The instruction set required by a program to achieve this optimal fitness requires only constants and the delay(x) and set_leds(x) functions. The IDGP instruction set was reduced to these 3 instructions in order to reduce the time to evolve to an acceptable solution. To ensure programs never exceed the maximum program size (in bytes, rather than in number of instructions), a “NOP” (no operation) instruction was introduced. The effect of

this “NOP” instruction is typically negligible since it merely introduces a delay in the order of microseconds.

Program 5 (a) is an optimal solution to the *“Blink-3”* objective where X=1 and X=4 for motes with 550nm and 650nm pass filters respectively. Programs (b) and (c) are common local maxima.

```
 1: set_leds(X)
 2: delay(100)
 3: set_leds(0)
 4: delay(100)
 5: set_leds(X)
 6: delay(100)
 7: set_leds(0)
 8: delay(100)
 9: set_leds(X)
10: delay(100)
11: END
```

Program 5a

```
1: set_leds(X)
2: delay(125)
3: set_leds(0)
4: delay(250)
5: set_leds(X)
6: delay(125)
7: END
```

Program 5b

```
1: set_leds(X)
2: delay(250)
3: delay(250)
4: END
```

Program 5c

Using this instruction set, we design an optimal solution (Program 5a) to the *Blink-3* objective and note that it is 10 lines and an average nesting level of 1.

3.4.2 Results and Discussion

3.4.2.1 Random Search Baseline

Uniform random selection of instructions will generate programs with a calculable distribution of number of lines per program and average nesting level. Typically, the optimal program length and average nesting is not known *a priori*. To construct a best case benchmark of random search we set the instruction selection probabilities to maximise the probability of randomly generating the known optimal solution Program 5(a). The selection probabilities of 0.43 for functions and 0.57 for terminals yields programs with an average of 10.0098 lines and an average nesting level of 1.2850. Based on these probabilities we calculate the probability of exactly generating Program 5(a) to be equal to 6.57 x 10^{-34}. While this is practically zero, the probability of generating a functionally equivalent program is much higher. Due to the syntactic richness of the GP engine (multiple lines

of code, nesting and a selection of functions), calculating the exact probability of generating a random program functionally equivalent to Program 5(a) becomes intractable. For example, since set_leds only uses the lower 3 bits of the value supplied, then one in every 8 values will yield the exact same LED configuration with no loss of functionality. Just considering this alone increases the probability of generating an equivalent program to 6.92 x 10^{-25} Consider various combinations of the delay periods which yield functionally equivalent programs increases the probability of generating a functionally equivalent optimal solution to more than 6.92 x 10^{-15}.

We therefore establish a baseline empirically by allowing motes to generate populations of random programs (equivalent to the population used by the IDGP engine each generation) and record the maximum fitness, pool average and standard deviation for each population. Scores are normalised against the respective optimal fitness for each mote. In total, over 3.5 million randomly generated programs were evaluated by the 8 motes using over 21 000 populations with 21 programs per population. The maximum fitness found was 82.22% of the optimal solution fitness and the average fitness of a random population (pool fitness) was 4.15% of the optimal solution fitness. This confirmed that finding an equivalent optimal solution to the *Blink-3* objective is a non-trivial problem. We generate a curve which reflects the typical maximum (elite) program fitness after i generations (evaluated populations) by randomly sampling the 21 000 elite scores i times and taking the maximum fitness of those samples. The process is duplicated 10 000 times and the results averaged for each generation value. This curve and the average pool fitness provide a comparison baseline for the experiments in Section 3.4.2 and we refer to it as “Random”.

Using the maximum normalised scores of the 21 000 populations, we also construct a probability density function for generating a program with a particular performance using a single population, as shown in Figure 3.8. The large initial value at zero can be explained by programs which have no delay instruction calls since programs such as these finish executing before the first objective function evaluation (i.e. before 50 ms) is performed. As such, the objective function will not execute and the program will yield a default fitness score of zero. Following from this is the striking feature is the 3 peaks. This can be

explained similarly by various execution durations. Since positive fitness values are only attributed at the 3 ADC readings at 50 ms, 250 ms and 450 ms, then programs which run for at least these periods will obtain some fitness due to noise on the ADCs which provide some positive values even in the absence of LED light. Finally, the most important feature of this PDF is the very low probability of generating random programs with high fitnesses. This confirms that evolving an optimal solution to the *Blink-3* objective is non-trivial and therefore employing GP should show a clear advantage over a random search.

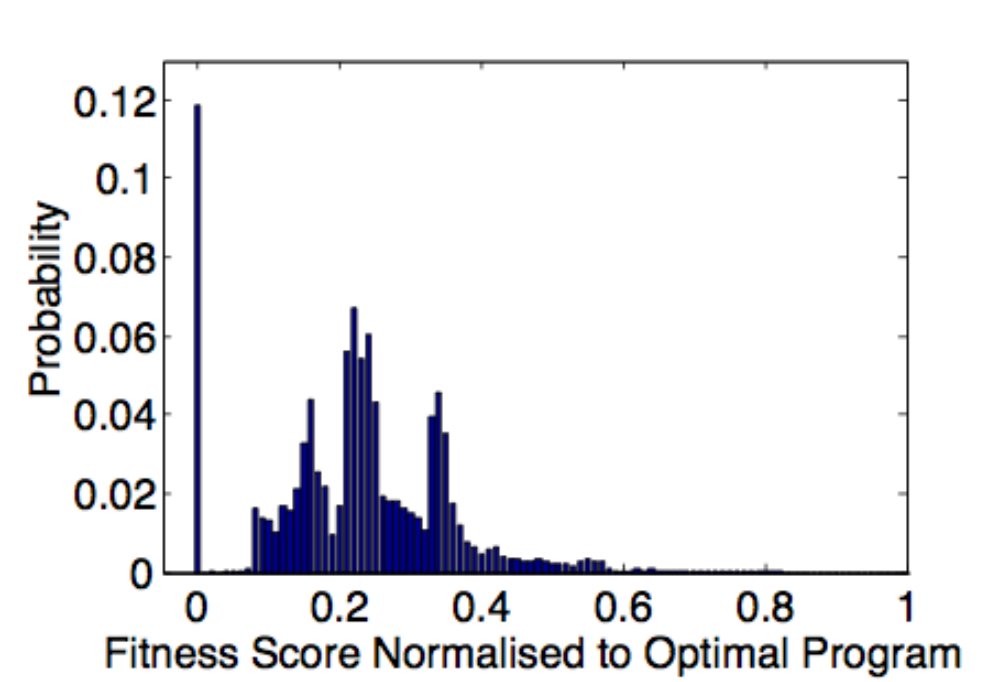

Figure 3.8: PDF of randomly generating a solution with a specific fitness.

3.4.2.2 Single Mote (Local) Evolution

Programs 5b and 5c are shown for interest since retrospective inspection of evolved code commonly revealed various forms of these suboptimal programs occurring. These solution programs fitnesses for one mote are shown in Figure 3.9 with a plot of the Elite fitness during an evolutionary run. Immediately evident are the discrete jumps typical to many evolutionary systems. Careful inspection of the code reveals many functionally equivalent solutions to Programs 5a, 5b and 5c around periods where fitnesses are close to the

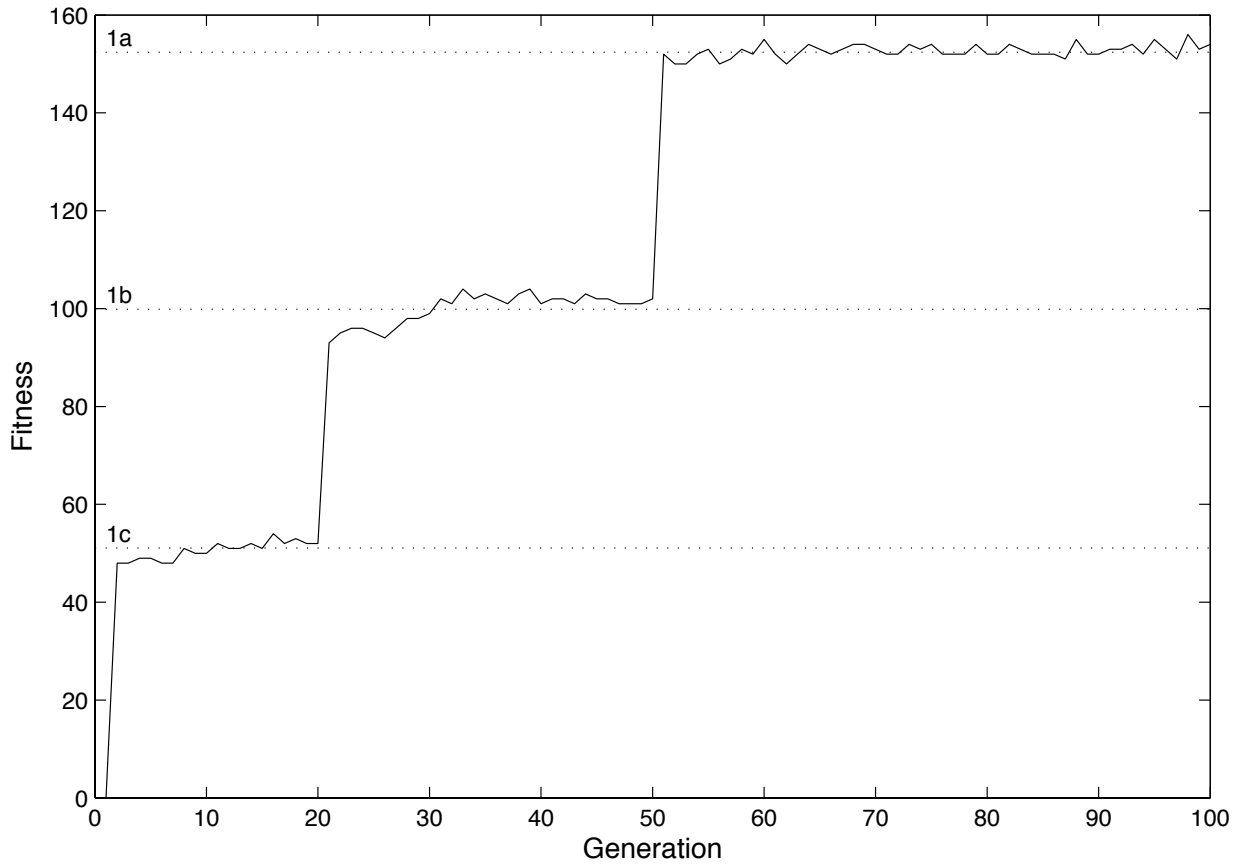

Figure 3.9: Discrete Evolution - this evolutionary run exhibits jumps in performance to discrete levels corresponding to mostly functionally equivalent solutions of Programs 5c to Program 5b and finally to an equivalent optimal solution of Program 5a. Such behaviour is typical to evolution of solutions to the *Blink-3* problem.

respective designed program fitnesses. Like all GA systems, such jumps in performance occur stochastically. We repeat experiments several times to gain statistical confidence, then normalise the results before combining them by taking the average of fitnesses at each generation. This yields relatively smooth evolutionary trajectories, keeping in mind that such curves are the aggregate of more discrete evolutionary runs.

To demonstrate that *in situ* evolution of logic can be performed in isolation, we allow 8 motes to evolve independently and thus without sharing programs. The mutation rate was set to 5% and the subpopulation distribution defined as 1 elite, 3 highly ranked, 12 children, and 5 randoms i.e. $[1\ 3\ 12\ 5]$. The size of the population (21 programs) and the distribution of subpopulations (elites, children etc) were chosen heuristically and

do not necessarily represent the optimal learning configuration. Many heuristics can be employed to construct a population distribution that achieves good learning, whereas optimising the learning configuration is still an open research question that is beyond the scope of this research. We proceed with this population distribution and allow the motes to evolve. Figure 3.10 shows a typical run with 8 individually evolving motes. Note that optimal scores have been normalised to each motes' respective optimal solution as measured before the experiment. There is an initial rapid learning phase which can be largely attributed to learning to use the delay instruction in order to reach an evaluation point, followed by learning to use the LEDs.

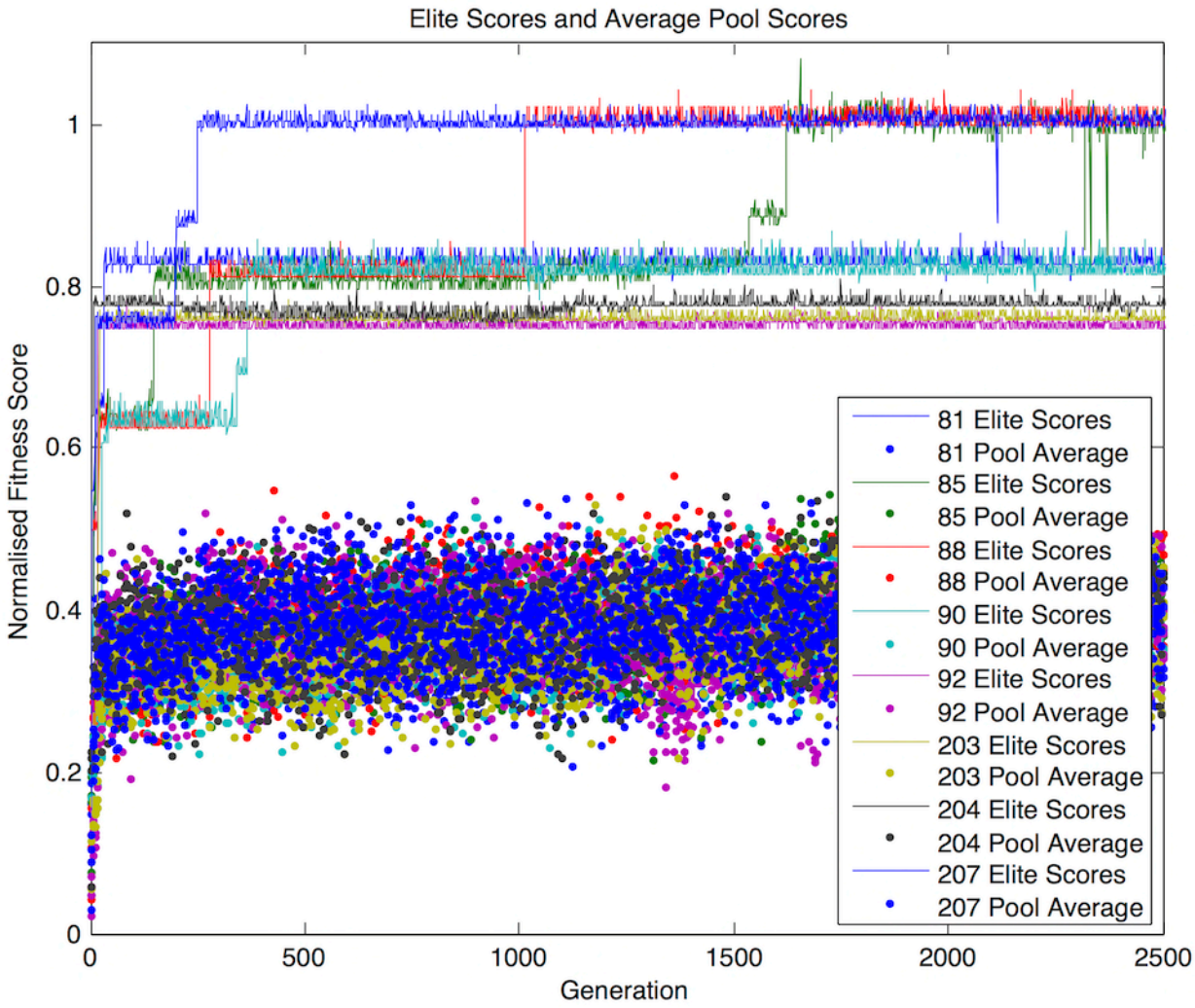

Figure 3.10: An example of typical evolution trajectories (elite and pool) from 8 individually evolving motes.

On average, motes will typically find a solution that achieves 70% of the optimal solution within 50 generations as compared to a random search which yields an average fitness of less than 55% of the optimal solution within an equivalent number of program evaluations. This indicates that the framework's GP implementation performs better than random search, however the subpopulation distribution is probably not an optimal learning strategy. As such the evolution may converge prematurely or never reach the optimal solution within an acceptable timeframe. Despite this, 3 out of the 8 motes in this run have converged to the locally optimal solution by generation 1622. Due to noise in the system such as reading fluctuations and timing differences caused by other threads, it is possible for a mote to diverge from an optimal solution. For example, the elite program score for mote 85 dips at generations 2219, 2331 and 2370 despite it being the same elite program (Program 6) being evaluated before and after the fluctuation.

Inspection of the generated code revealed that the mote was using delays of up to 2 ms in the system to achieve the first 50 ms delay. However, occasionally the mote may have less delay-causing events in other threads which then means the LED is switched off too early. By generation 2123 the mote has evolved to a delay 50 ms before eventually extending the delay even further.

Program 6 Evolved optimal programs for mote 85 at generations 2318-2320

```
1: set_leds(217)
2: set_leds(delay(55))
3: set_leds(set_leds(delay(136)))
4: set_leds(-207)
5: delay(set_leds(delay(124)))
6: delay(delay(31))
7: set_leds(-207)
8: delay(190)
9: END
```

3.4.2.3 Multiple Mote Homogeneous Logic

To demonstrate that cooperative evolution achieved via the Island Model provides faster evolution towards an acceptable solution than that of evolution in isolation, we perform

three experiments: (1) motes evolve their logic in isolation; (2) motes with a common objective function share programs periodically via their wireless radios, referred to as distributed homogeneous evolution; and (3) motes with different objective functions share programs periodically via their wireless radios, referred to as distributed heterogeneous evolution. Although motes in the homogeneous experiment share a common objective function, their perception of the environment may be different due to spatial differences in light. In the heterogeneous experiments, both the objective functions and the environmental perception of the motes can be different.

This experiment aims to explore whether motes with different objectives can still benefit from using the Island model. The importance of this arises from the likelihood that 2 motes will neither be in exactly the same environment nor have exactly the same sensing configuration, and as such may require slightly different solutions to obtain acceptable performance.

We test this by constructing groups of motes, each with 2 motes per group, with 650nm pass filters (responding mainly to red light) and 2 motes with 550nm (responding mainly to green-yellow light) pass filters. The 4 motes are permitted to share programs amongst each other. The mutation rate for all three experiments is 4%, and the subpopulation distribution includes 1 elite, 3 highly ranked, 12 children and 2 random programs and 3 “other” programs, which are obtained from other motes each generation.

3.4.2.4 Multiple Mote Heterogeneous Logic

In all 3 experiments, the motes evolve for a minimum of 50 generations. We then computed the aggregate normalised fitness score of each experiment for every generation in the learning process. Individual evolution was calculated on 16 individual motes, homogeneous evolution on 4 runs of 4 motes and heterogeneous evolution of 6 runs of 4 motes. Some runs were left to evolve longer than 50 generations to provide some insight as to what may happen longer term. Figure 3.11 plots the normalised fitness scores of the 3 experiments versus generation.

The results shown in Figure 3.11 support the literature [79] in that sharing genetic

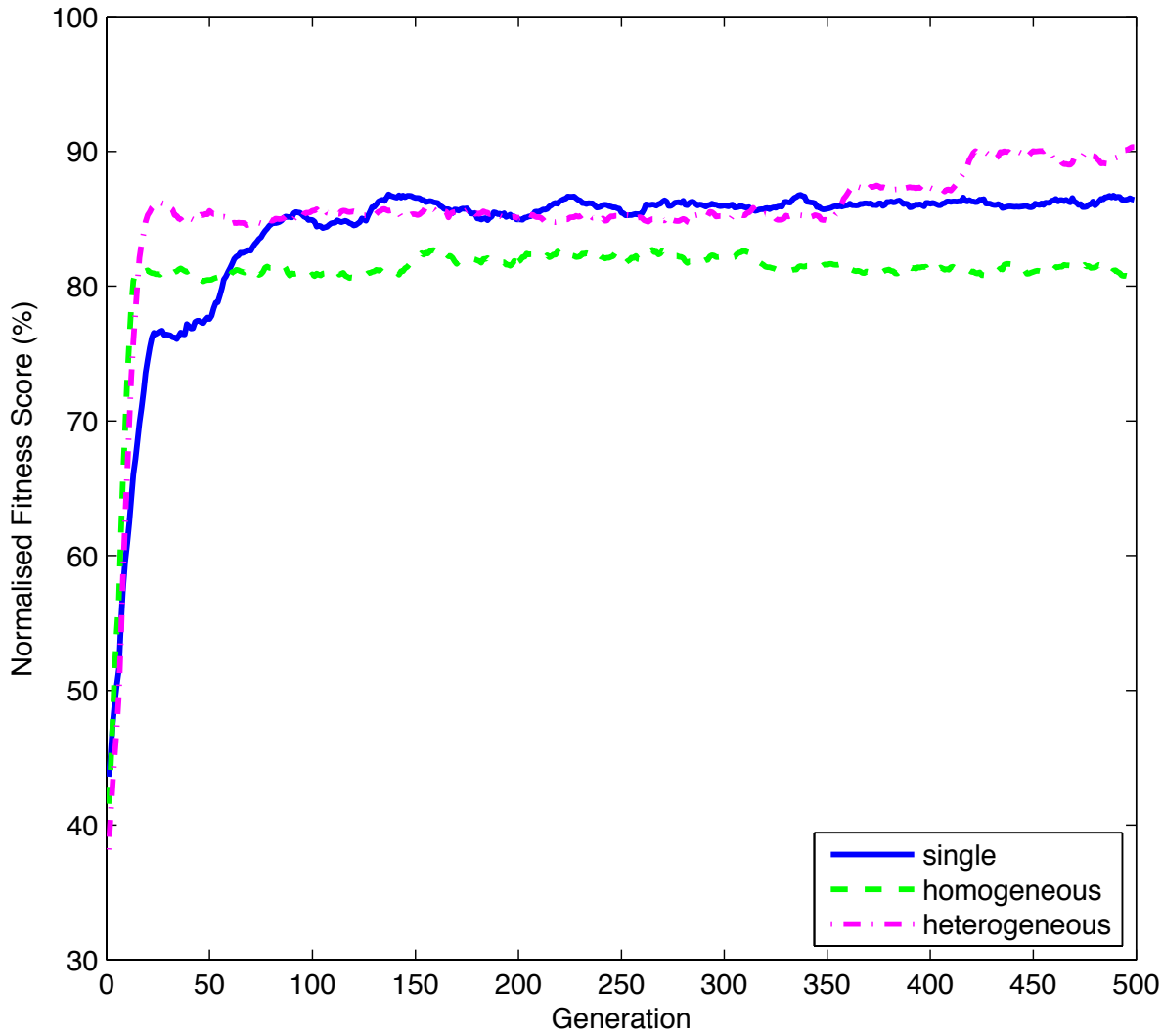

Figure 3.11: Comparison of optimisation trajectories: evolution in isolation, Island Model (homogeneous) and Island Model (heterogeneous).

material via the Island Model facilitates faster evolution towards a good solution than does evolution in isolation. Both the homogeneous and heterogeneous experiments reach a fitness score of 80% within 20 generations, while the single mote evolution takes 80 generation to reach the same fitness score. This results shows that groups of motes with similar objectives can help each other expedite evolution to reasonable performance and still further specialise to their local fitness landscape.

However it is also possible that the system prematurely converges as the evolution in isolation ultimately ended up with a higher fitness than the distributed homogeneous

evolution after about 100 generations. This indicates that it might be beneficial to initially share information, but perhaps not best to do it indefinitely since dominant solutions from neighbouring motes will tend to end up dominating the local pool and reducing the genetic diversity. Thus, despite the benefit of quick initial learning, the inherently incestuous evolution of the homogeneous experiment limits its learning potential which supports the findings of [79].

Interestingly, the fitness for motes in distributed heterogeneous networks is similar to the fitness of the isolated motes by generation 100. By generation 400, the heterogeneous evolution approach improves its fitness further (likely due to higher genetic diversity of programs in the pool), whereas the fitness of isolated motes remains the same.

3.4.2.5 Adaptation to Dynamic Environmental Conditions

The motes in the previous experiments were placed in a box and shielded from external light sources. Program 5a was designed with the assumption that there were no other external light sources present. To simulate an unexpected environmental change we place some of the motes outside the box which are then subject to other light sources, namely sunlight through room windows and the fluorescent room lighting. The optimal solution is dependant on the intensity of the ambient light into the photodiode and the energy consumed by the LEDs. As the ambient light increases, turning all LEDs off all the time becomes the optimal strategy.

Motes 85 and 207 were removed from the box after generations 51 and 52 respectively. The elite scores reveal that the fitness exceeds that of the “known solution” for that when the mote is in the box. It can be seen in Figure 3.12 that initially the improvement is due to the additional ambient light, however as the fitness increases after this it is conjectured that the mote discovers that it does not need to turn the LEDs on at all to maintain a high fitness, and in doing so saves power and hence generates a higher fitness. Manual analysis of various programs within the period where motes were out of the box reinforces this conjecture with many programs simply delaying more than the evaluation period. Interestingly, after the motes were placed back into the box, the motes almost instantly

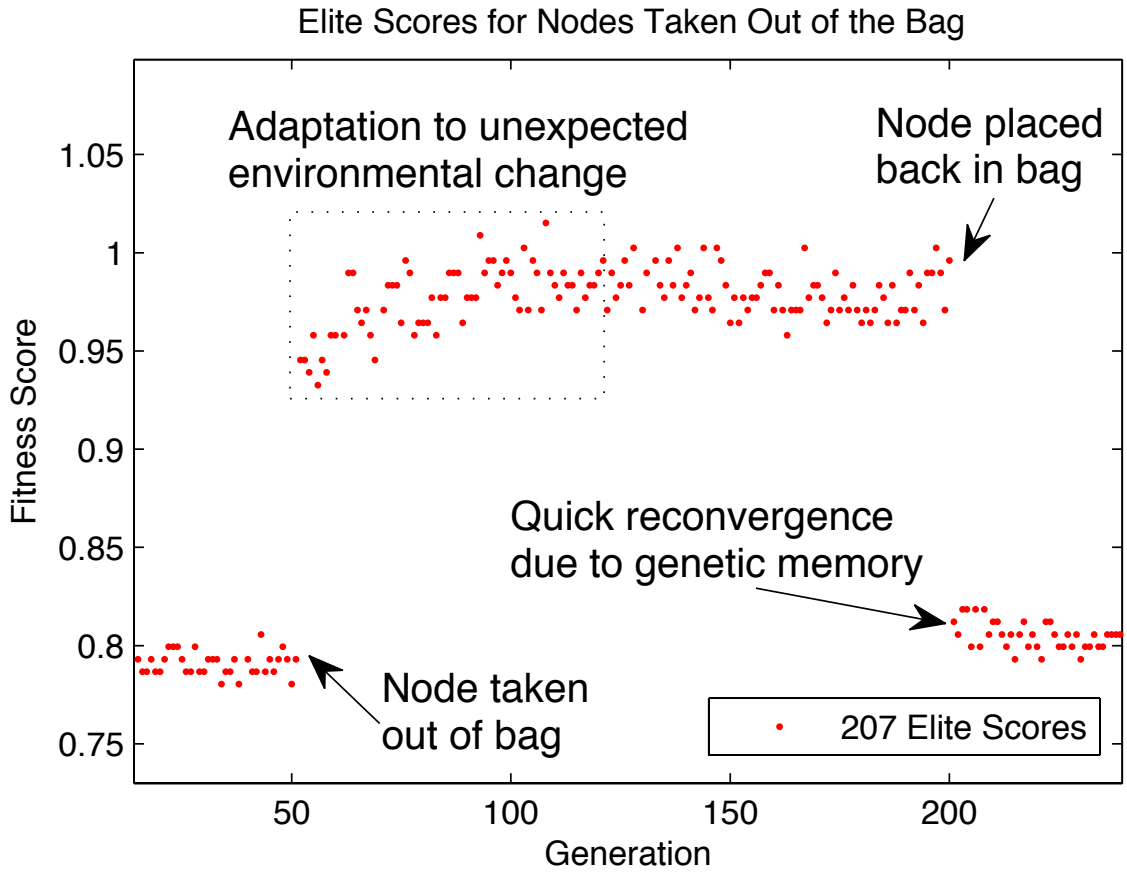

Figure 3.12: Adaptation to the "unexpected event" (change in fitness landscape) where the devices (nodes) were removed from the bag and exposed to ambient lighting conditions.

returned to a fitness of that achieved before removal from the box, indicating that the genetic material maintained some history.

3.5 Evaluation on Evolution of Comms Problem

The aim of this section is to investigate whether nodes can learn to cooperate using communications to achieve an objective. As such, an objective that *requires* communications to occur is devised and detailed in Section 3.5.1. The problem is tested in two Fleck3b motes and the findings discussed in Section 3.5.2.

3.5.1 The "PacketForwarder-RfmToLeds" Problem

This section details the design of an objective function that requires receiving of packets, transmission of packets and actuation. The design is detailed in Section 3.5.1.1 and its optimal solution provided in Section 3.5.1.2.

3.5.1.1 Packet Receiving-Sending Requirement

In the previous experiments, fitness was calculated and provided back to the evolution process entirely locally on the devices. However, since networked embedded have communications capability, they should, in theory, be able to learn to coordinate their behaviour through communications to achieve an objective that is not local to the agent (i.e. fitness is either supplied from neighbouring devices or a system-wide objective function).

To demonstrate this we desire a problem that requires devices to communicate, yet is well defined and preferably as simple as possible. Minimally, two devices (motes) are needed to demonstrate communications. For simplicity, these will be enumerated as Mote A and Mote B in this experiment.

RfmToLeds [1] is a simple radio program made popular by the TinyOS community as a tutorial and has often been used to demonstrate wireless communications to mote platforms. The program receives packets containing an incrementing counter, transmitted by another mote (typically IntToRfmM[1]), and simply displays the 3 least significant bits of the counter received using the 3 onboard LEDs.

This demonstrates receiving and actuating, however, since the packets generating the counter would need to come from an external source, it does not demonstrate transmission of packets by the motes or communication between them. To require communication, a base mote (connected to a PC) executes the IntToRfmM logic, generating packets at a rate of 1 Hz. Importantly however, the packets are addressed only to Mote A (through specify a MAC destination address in the MAC layer). All combinations of LEDs are cycled through every 8 seconds. This is also used as the evaluation period for programs in the experiment.

The LED status of Mote B is measured (via a remote procedure call transparent to

any evolved logic) and used to calculate a fitness based on the counter value sent to Mote A and the LED status of Mote B. Frequent periodic fitness scores are supplied to both motes and summed locally over the execution time of 8 seconds per program. Importantly, communications is not explicitly rewarded, but rather communications must evolve in order to achieve the acceptable objective (LED state of Mote B given the counter sent to Mote A).

Ideally, the system would learn to receive the counter packet at Mote A, forward this to Mote B which in turn would receive it and set its LEDs accordingly, thus demonstrating the evolution of primitive communication. Therefore we term this as the *"PacketForwarder-RfmToLeds"* problem. Figure 3.13 provides a diagrammatic representation.

The function set is limited to {radio_tx_int(x), set_leds(x), radio_rx_int()} since all of the essential aspects reside within this set. Based on these instructions, an optimal pair of programs (since the motes will require different behaviours) is shown in Section 3.5.1.2. The population structure was set to $[5\ 3\ 10\ 2]\ 1]$ (5 elites, 3 highly ranked, 10 children, 2 randoms and 1 other). The mutation rate of set as 5% and single point crossover on code line (tree bases).

3.5.1.2 An Optimal Solution

Programs 7a and 7b were conceived as the minimal, ideal solution for the *"PacketForwarder-RfmToLeds"* objective.

Program 7 (a) and **Program 7** (b) are the optimal programs for Mote A and Mote B respectively to the *"PacketForwarder-RfmToLeds"* objective.

`radio_tx_int(radio_rx_int())` `end`	`set_leds(radio_rx_int())` `end`
Program 7a "*PacketForwarder*"	Program 7b "*RfmToLeds*"

Simply rewarding the devices for when the correct sequence is displayed on Mote B's LEDs results in the local optima of Mote B leaving its LEDs unchanged for the entire evaluation period. This was ultimately negated by generating local negative fitness and supplying external (global) positive scores such that the inert behaviour (Mote B leaving

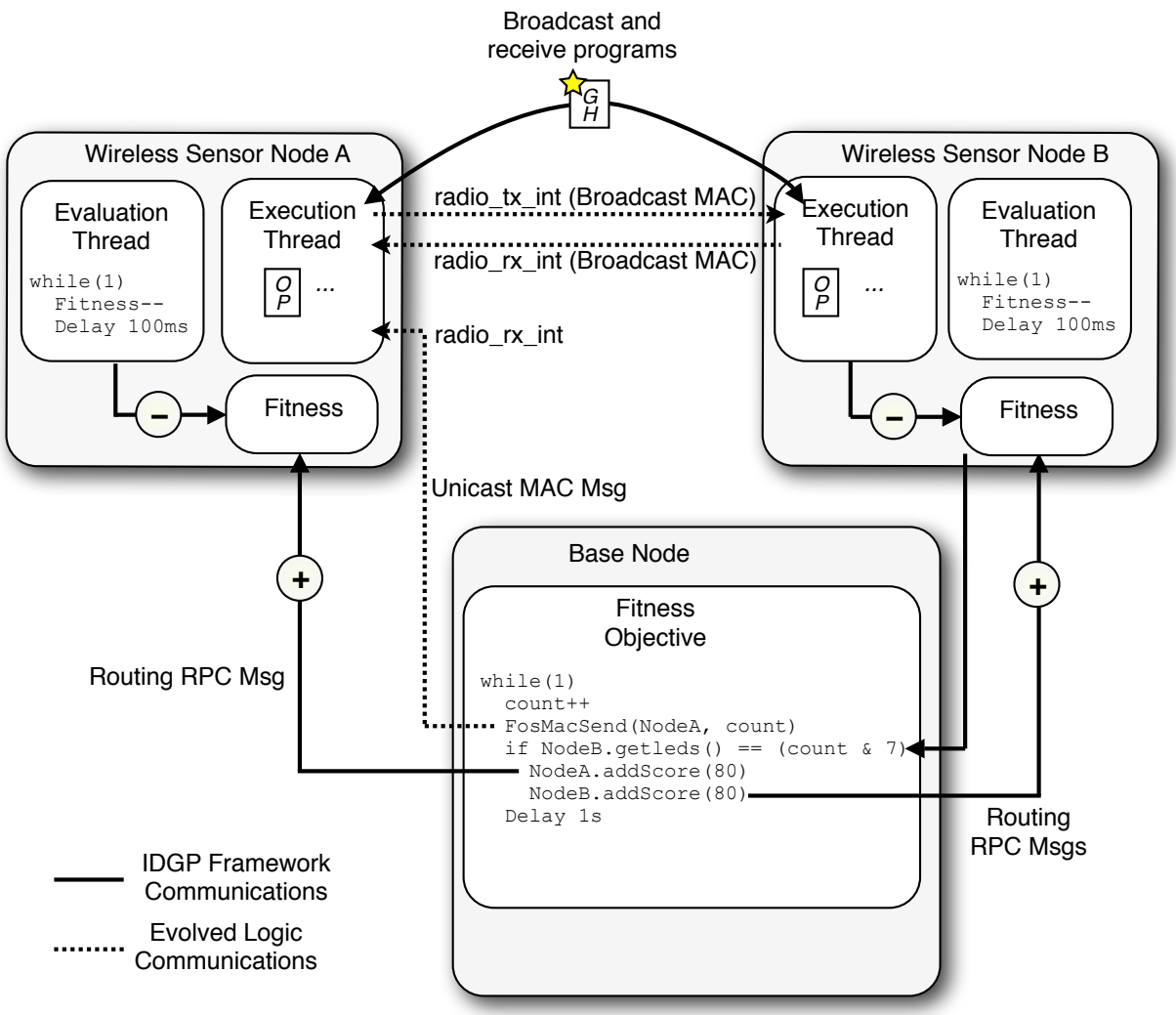

Figure 3.13: Communications topology for the *"PacketForwarder-RfmToLeds"* experiment.

its LEDs unchanged for the entire evaluation), would yield an overall fitness of zero. This motivates the system to attempt to get more sequence matches during the evaluation period.

The final evaluation function can be expressed as follows:

- subtract 80 from the mote fitness for each evaluation (over 8 seconds)
- add 80 to the mote fitness for each correct LED sequence on Mote B matching the value sent from the base to Mote A

Therefore the optimal score is $(8 \times 80 - 80 \times 1) = 560$.

3.5.2 Results and Discussion

The novelty supplied by GP was underestimated for this experiment and served as a reminder of the importance of specifying the fitness function carefully. Initially, global negative fitness feedback was supplied to both motes when Mote B's LED status did not match the acceptable pattern, and positive feedback supplied when it did match. While this appears reasonable, it did not consider that the motes' communications use the same medium (radio frequency) as the framework that supplies the fitness feedback. Through the GP evolution, motes were able to evolve a "denial of service" strategy by congesting the channel, maximising packet collisions and ultimately reducing the number of received negative fitness packets. This converged to a fitness of zero, where no positive or negative fitness was being received by the mote, however it was more optimal than the initial condition of receiving predominantly negative fitness. One can imagine this as the equivalent of a child learning that shouting when being reprimanded prevents the "negative feedback" from being heard.

Several approaches were experimented with to address this issue. Eventually it was determined that a combination of small, though frequent, negative feedback generated locally with positive larger global/external feedback, allowed the motes to converge to the optimal solution. The local frequent negative feedback is term "self deprecating" behaviour. This combination does not "reward" motes that congest the channel since only positive fitness values are transmitted wirelessly.

With the updated fitness function employing both local negative and global positive feedback, the evolution proceeded more successfully. The evolutionary trajectory of the elite and pool fitness for both nodes is shown in Figure 3.14.

Again, the novelty of the generated solution was surprising. Both nodes were able to achieved elite fitness scores of 640 which is higher than the expected maximum fitness of 560 $(8 \times 80 - 80 \times 1)$. This "cheating" appears to result from extra instructions during the evaluation prolonging the mote execution process just long enough so that the fitness accumulated occasionally receive 9 positive fitness values. Intriguingly, this strategy has a lower average (pool) fitness than that of Program 7(a) each time this strategy becomes

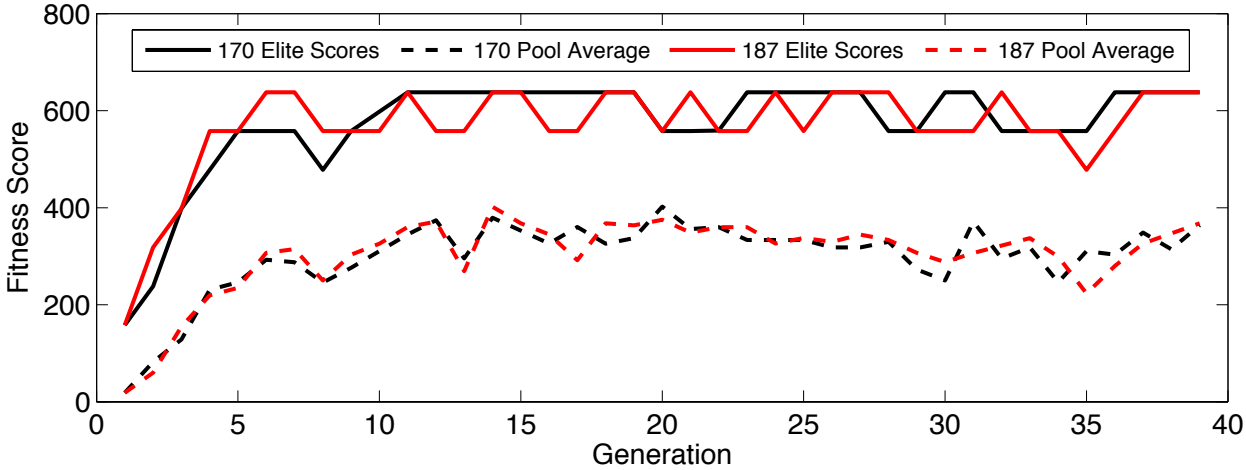

Figure 3.14: *"PacketForwarder-RfmToLeds"* fitness evolution

the elite and subsequently dominates the population. It was identified that this strategy relies on the evaluation period being near the beginning and end of the evaluation of the program. This is a risky strategy since most of the time it will not be beneficially aligned and the program may miss one or both of the positive feedback scores and will experience more local negative feedbacks. This causes the perpetual oscillation of both elite and pool fitnesses (lagging in phase) as the pool and elite toggle between the 2 solutions. Eventually the conservative strategy becomes elite again, however genetics of the risky strategy remain in the pool and eventually the conditions (timing) makes the risky strategy become the elite again perpetuates indefinitely explaining the oscillatory nature of the elite fitnesses displayed in Figure 3.14.

Nonetheless, the experiment did successful show the evolution of communications between nodes and also demonstrated the framework using a combination of both local and global fitness feedback simultaneously.

3.6 Discussion

3.6.1 Challenges of In Situ Evolution

In our solutions we were able to ensure that the system does not enter a state where it is impossible to evolve to an acceptable solution. Since the actuation, LEDs and radio,

cannot permanently alter the environment (including itself), an acceptable solution for these problems always exists and therefore the GA can always potentially find one of these solutions. In more general applications it will be important to design node hardware and software architectures such that IDGP programs are executed and evaluated in isolation from core functionality to ensure that evolved programs cannot hog critical resources or cause undesirable actuations.

This raises the issue of which problems are suitable targets for IDGP implementation as a topic for further research. In some cases it may be possible to impose constraints during evolution to ensure solutions stay within safe behaviour. However setting constraints inherently reduces the solution space that can be searched meaning that better solutions may exist that cannot be evaluated.

Evolving logic post deployment (in situ) has both opportunities and disadvantages. In this section we discuss some of the lessons learnt from the challenges faced when evolving logic *in situ* across a WSAN.

The evaluation of logic *in situ* avoids reliance on synthetic (simulated) models of the environment, and there is no better representation of the environment than the environment itself. Not only does this prevent the evolution of logic being made brittle by exploiting artefacts in the simulation, it also means that the mote does not need to perform any simulation at all which is a great benefit for resource-constrained devices.

A significant disadvantage of this approach however, is that there is only one single shared environment for all programs across all motes. It is impossible to reset the environment back to *exactly* the same initial conditions prior to the evaluation of every program. Hence, each program may leave a “footprint” which may help or hinder other programs and this generates a credit assignment problem which can slow or even prevent the system from converging to a specific fitness. Even more important is whether the environment could be changed in a way that permanently prevents the acceptable objective to be fulfilled. Unlike offline evolution, one cannot go back in time and trial another optimisation trajectory if the current one has failed.

3.6.2 The Performing-Learning Paradox

Offline learnt logic can deliver immediate high performance upon deployment, however due to unexpected changes in the environment, may not perform well over the longer term. Online learning, on the other hand, allows the system to adapt to unexpected changes, and so it should, in theory, be able to provide better performance than that of offline developed logic under unanticipated environmental conditions. Unfortunately, it requires time to learn how to achieve the acceptable performance and there is no guarantee that an acceptable solution will be found within an acceptable timeframe. Furthermore, learning implies that the system is not performing optimally.

Paradoxically, to achieve better performance, logic that is likely to perform worse than the current best known solution must be executed in order to provide a chance of discovering better performing solutions. To an external observer, one would see this as variation of performance over time. Implicitly, performance is dependant on the fitness as measured within the observer's critiquing period. Ideally then, the best current performing logic would be executed within this critiquing period, and learning (evaluating and executing other programs) would occur outside of this period. If performance is calculated continuously however, the average observed performance will match the pool fitness. Here, optimising the average (pool) performance is desirable.

For many problems satisficing is acceptable, however within that there is an additional class where additional learning is beneficial. However the rate of learning could be impeded. Meet the need and maximise learning with available resources (slack). There is little disadvantage to utilise unused resources to maximise learning.

3.6.3 Challenges of Distributed Evolution

There are many challenges with evolving logic on devices simultaneously across a network. Perhaps the most important aspect is setting the fitness objective appropriately and supplying fitness information to the motes. Section 3.5 highlights the importance of this with the unexpected evolution of the "denial of service" strategy. The global (external) fitness feedback needed to be altered to a combination of strong positive reinforcement

with small, but frequent, self-provided negative reinforcement. We recommend this approach in systems where the framework and evolving code share the same communications medium. Unfortunately identifying whether the problem representation is the main reason why a system becomes stuck in a local-minima is usually non-trivial.

In systems where motes maximise a purely local fitness, evolution is reasonably straight forward since it is essentially many motes evolving independently of each other. Motes can still benefit from sharing optional information (such as programs), and will often converge on the same solution for a given objective. With a high selective pressure acting on local environmental differences, speciation of logic will occur and motes will essentially revert to individual evolution as demonstrated in section 3.4.2. Not to be confused with niching which is a popular mechanism for deliberately divided a search space across nodes. Note this requires some level of coordination (whether implicit or explicit) [17].

However, when the fitness objective requires cooperation or feedback from other motes, the potential for information to be lost or misleading then arises. For example, in section 3.5, motes evaluate their programs asynchronously which can reward “leech programs” which rely on the prior program to perform well. With strong elitism in the population, such programs can easily dominate the pool. Randomisation of the order in which programs are executed will quickly penalise leech programs, but not until after significant damage to the pool fitness. Some form of fitness memory may be useful to reduce this effect.

The random order of programs (synchronised or not) causes a “macro-crossover” of programs, where various combinations of programs from different subpopulations are executed simultaneously. For example, Mote A may be executing its elite program while Mote B is executing a random program. For distributed evolution, the probability simultaneously executing programs that perform well together needs to be significantly high otherwise good fitness genetic material could be “forgotten” (since a combination of a good program executing with a poor program is likely to collectively result in poor performance). Clearly, biasing the population composition and/or evaluation schedule will be important towards achieving evolution of a good solution. As such, further investigation into this topic is recommended.

3.6.4 IDGP Configuration

The configuration of IDGP framework parameters impacts evolution performance (speed to find good solutions and quality of final solution). Design considerations, such as using homogeneous or heterogeneous nodes, are important to how the system will achieve its purpose. The size and composition of the population directly affects both evolution speed and performance, we require configurations that maximise performance within the size constraints imposed by the memory limitations of the device. For some objective functions, all acceptable solutions may be larger than the maximum program length L_{Max}, in which case we either increase L_{Max} by reducing the number of programs N_{pop}, we increase the physical memory size, or we adopt a human coded solution which can afford to be up to N_{pop} times the size of an IDGP program.

Richer syntax allows more complex problems to be generated and ultimately richer evolved behaviours. Additionally, higher level instructions allow more compact representation of logic. Ideally, IDGP programs should be as syntactically rich as possible in order to maximise the efficiency and performance of the solution programs. This will allow more programs (i.e. a larger population) to reside within the memory constraints of the platform, which in turn allows for greater genetic diversity to be maintained which will assist learning. Determining the optimal instruction set and program size for a particular objective remains an open issue for future investigation.

In this section we used a small instruction set of 3 instructions to demonstrate IDGP for simple well-defined objectives. Subsequent experiments were however performed using a larger instruction set of 13 instructions to solve the *"PacketForwarder-RfmToLeds"* objective and demonstrated similar evolutionary trajectories.

The NFL theorem states that no solution finding mechanism is better than others over all problems. IDGP is well suited to complex WSAN problems with no well known solution, or where adaptivity of logic is likely to be necessary. IDGP is not a solution for every WSAN problem, and we recommend the designer to estimate whether an alternative approach, such as human-crafted logic, could produce acceptable logic more efficiently.

3.7 Conclusion

We have presented In situ Distributed Genetic Programming (IDGP) as a framework for the automated online creation of logic on WSAN nodes by using a light weight genetic programming engine suitable for resource constrained sensor node devices. The in situ evolution of logic was demonstrated on a network of physical sensor nodes which were able to evolve their logic in order to adapt to local conditions as well as unanticipated environmental changes. Empirical results indicate that sharing evolved logic with neighbouring nodes typically provided faster learning initially, however potentially converging prematurely.

The 3.6 Discussion however highlighted the need for being able to make the system perform better on average. This motivates the need for a mechanism to achieve this and motivation for developing a heuristic

In conclusion, the IDGP framework provides a method for tasking WSANs with a high level system objective. Through continuous online evolution, nodes learn to sense, communicate and actuate in order to optimise for a specified objective.

4

Fitness Importance

This chapter presents Fitness Importance (FI or $\Phi(t)$) as an online constraint satisfaction (satisficing) heuristic defined by the acceptable average fitness as a function of time. The need for such a heuristic is discussed, before the heuristic is formally described. An implementation of the heuristic, γ_{simple}, biases the generation of the population in such a way that the average performance meets the Fitness Importance specified, while attempting to retain as much learning capability as possible. FI is ultimately incorporated into the IDGP framework and the enhanced framework presented and discussed.

This chapter introduces the concept of Fitness Importance (FI) as a heuristic to describe the varying importance of exhibited fitness of a learning system during its life. Furthermore, an accompanying metaheuristic, γ, is described for generating populations to achieve the level of fitness deemed important by FI whilst attempting to retain learning capacity.

The motivation for the FI heuristic and how it is envisaged to be utilised are presented in Section 4.1 with Section 4.2 detailing how it differs from existing heuristics. A definition for acceptable average fitness AAF is provided and then used to define an analytic model of the FI heuristic in Section 4.3. A metaheuristic for using FI to generate populations with an AAF while attempting to retain learning capacity is then presented Section 4.4. Challenges to implementing an ideal form of the metaheuristic are discussed.

Section 4.5 demonstrates how FI can be realised by providing a simple (non-ideal), yet pragmatic implementation of a population generator metaheuristic called γ_{simple}. This is incorporated into the IDGP framework and an overview of the enhanced framework presented.

Discussion of general issues that may arise when employing IDGP are provided in Section 4.6 along with summary remarks in Section 4.7.

4.1 Motivation

The experiments in Chapter 3 demonstrated in situ evolution on resource constrained embedded devices. However it was quickly realised that finding a solution, using a population-based metaheuristic, is vastly different from being a solution. Essentially the system was learning and not performing (at least acceptably anyway).

Nonetheless, it makes intuitive sense that learning logic (or behaviour) in situ should be possible while achieving acceptable performance, since arguably we, and many other biological systems, learn in our environments while “performing acceptably”. This basic intuition largely motivates the search for a heuristic to achieve this, however it is elaborated on in the following sections.

4.1.1 Online Learning with a Minimum Performance Constraint

Such scenarios occur where anything above the necessary 'performance' is deemed as *excess*, and importantly, the excess has negligible cost associated with its production otherwise it would be included in the objective function. This can commonly occur in win/lose scenarios.

4.1.2 Inspiration from Natural Systems

Many animals, humans included, have the ability to gauge the current context and perform to a level that is acceptable for that context. This also implies the agent has an understanding of what fitness is (or the objective function). Put simply, the agent understands that acceptable performance is a satisficing target which resides on the scale of fitness and performing above this target is no better than minimally meeting the target. However, an important additional trait of many higher-order animals is their ability to deliberately trade excess performance capability for online learning capacity. The following allegory attempts to exemplify how excess performance capability can be used for learning.

4.1.2.1 Sprinter Allegory

Let us assume a fictional sprinter named Mr Bolt has the goal of being titled as the fastest man on Earth. To achieve this title, he must consistently win the events in which he competes, where winning a race is a satisficing objective since he would be deemed first regardless of how much he wins by.

When he is training at the local park, he uses the fitness metric of how fast he can run 100 m as an optimisation target. In this non-race context he understands that not meeting the fastest time will not impact on his ultimate goal. In fact, the metric of winning makes no sense at all - there is no race to win, no satisficing target. Nevertheless, his speed and strategies can be optimised through unrestricted freedom to explore/play with strategy variations. During each sprint he trials various strategies, changes to gait, stride length, arm movements , etc.

Through a combination of innate (seeded) abilities and optimisation (at the park), Mr Bolt does not need to perform his current most optimal strategy in order to win at the national heats. This affords him the luxury of performing various strategies in a context where performance to a level of meeting the satisficing goal is important. He first applies his optimal strategy for the first half of the race until he is significantly in front. He now realises that he can comfortably afford to evaluate other less optimal strategies, but in a race-context that could not be achieved at the park, while still achieving the *acceptable average performance* which wins him the race (just). From this, he learnt the responses of his fellow sprinters to various strategies in an in-race context.

Now, having qualified for the olympics, he exploits his learnings by executing his best strategy with no variations for the entire 100 m. This results in the win and attaining the title as the fastest person on Earth.

In this allegory, we see 3 modes of using the contextual information. At the park, the knowledge that no performance was necessary allowed complete freedom to learn and explore strategies. At the nationals, the knowledge that good (but not optimal) performance was necessary allowed some online learning to occur while still meeting the satisficing target (i.e. winning the race). At the Olympics, the knowledge that the best performance was likely necessary, prevented any learning from occurring, but ensured the satisficing target (i.e. winning the race) was met.

In each of this situations, the time to complete 100 m is a separate metric of fitness, however the importance of what time was actually achieved varies depending on the context.

Ideally, we wish to employ a mechanism that **captures the importance of fitness in the current context** and **use any excess performance capability for online learning**.

4.1.3 Population-based Performance

The IDGP framework employs GP as the learning metaheuristic for generating novel logic. GP is a population-based approach and as such not normally used for online learning.

Traditionally, GP approaches evaluate solutions in simulated environments that do not

affect the real world (i.e. tie up resources, consume energy, move things, etc.) In this scenario, the performance of programs during evaluation is of little interest. Instead, the final elite solution is focussed on as a measure of performance. However, for online population-based learning, every evaluation performed in the world contributes to the agents observed behaviour and ultimately its performance. While it is desirable for offline GP to have high diversity within the population for optimal learning, high diversity also means that the average fitness evaluated will be, by definition, the pool fitness, which is typically vastly inferior to the best (elite) solution normally deployed after an offline learning process. Hence, it would be desirable to **bias the pool fitness closer to the elite fitness where possible**. Some GP approaches can avoid explicitly instantiating a full population, such as EDA-GP approaches. However, these are highly complex [112] in their representations and their implementation would require computational resources far beyond that of a small microcontroller.

Furthermore, defining a novel GP algorithm is not necessary to demonstrate the usefulness of the IDGP framework, since we wish to demonstrate that distributed online learning using existing prototypical GP implementations (as described in Section 3.3) is possible on mote class devices.

There appears to be no reported mechanisms for applying standard GP approaches to online learning contexts (prior to this research). As such, to achieve the advantages of GP discussed in Section 3.1.4 within the IDGP framework, we must attempt to devise a mechanism to permit GP to be used online by biasing the average evaluated fitness closer to the elite fitness rather than the pool fitness.

4.1.4 An Ideal Fitness Importance Heuristic

In the previous sections, we identified that a mechanism for conveying the importance of fitness in the current context could allow the system to perform at an acceptable level while achieving online learning. It was also identified that for GP to be used online, a mechanism is needed to bias the average evaluated fitness (i.e. the pool fitness) closer to the elite fitness. Combining these requirements, we propose the development of a

heuristic called Fitness Importance (FI), to describe the importance of fitness over an entity's lifetime.

The heuristic implicitly implies that if the average evaluated fitness exceeds the current level of importance, then excess performance capability can be traded for learning capability. This should maximise the learning potential (while delivering acceptable performance) which can then be exploited in the future should the acceptable performance threshold be increased.

Ideally, the heuristic would be represented as a normalised (or percentage-based) quantity representing no importance minimally as 0, and maximum importance as 1.

4.2 Related Heuristics

Achieving acceptable behaviour is certainly no new concept. In fact Humans have likely been achieving this since the dawn of the species. Despite this, a formal definition of satisficing was only presented by Herbert Simon in 1955 [119]. Satisficing is discussed in Section 4.2.1 and is a useful description for the aspect of the desired heuristic to represent acceptable performance.

4.2.1 Satisficing

SATISFICING

The concept of satisficing was pioneered by [119] and distinguishes between requiring optimality and requiring "acceptable" payoffs (satisfactory, satisficing and "good enough" were also used as descriptions).

Expressed formally, satisficing can be described as an indicator function:

$$I_S(s) := \begin{cases} 1 & , \quad s \in S \\ 0 & , \quad s \notin S \end{cases} \quad , \; s \in X \tag{4.1}$$

Were a solution s from the set of all solutions X is deemed optimal (indicated by 1) if it belongs to the set of satisficing options (i.e. an element of S), otherwise it is non-optimal (indicated by 0). Another interpretation of the Indicator function is "success".

Interestingly, “aspiration” goals were also suggested which could be used in conjunction with a discrete satisficing conditions allowing the agent to pursue a payoff above the level of satisfaction. In [119]’s use cases, he inferred that the higher fitness carries some benefit, such as more payoff, but it may be for nothing else other than description. For example, one could say that a person “easily won” which adds some additional information to the result of a “win”. It does not change the payoff from winning (i.e. the reward for meeting the satisficing condition to the agent is the same either way), however the additional information could then be used by the agent to reinforcing internal logic more strongly, etc.

It is important not to confuse satisficing with satisfiability. Satisfiability (in computer science) usually refers to whether a set of sentences in propositional logic is satisfiable.

Constraint satisfaction however does have similarities to satisficing in that constraints can be viewed as multiple satisficing conditions that need to be met simultaneously in order to be feasible.

CONSTRAINT SATISFACTION

This pioneering perspective describing “acceptable” payoffs as different from a “aspiration” goals has provided a foundation for the developed heuristics. However, satisficing does not describe the intent of *deliberately* sacrificing performance for additional learning. Hence, this alone cannot address the needs for the meeting a target fitness while performing online learning with GP.

4.2.2 Exploration-Exploitation (EE)

It is not immediately obvious how acceptable performance is achieved using population-based approaches in online learning contexts. As with all search heuristics, potential solutions (program logic in the context to this thesis) with unknown performance need to be evaluated in order for knowledge to be gained (i.e. for learning to occur). However, in an *in situ* online learning context, **the fitness of solutions evaluated in the real world is the perceived performance** of the system, mirroring the adage “it’s what you do that defines you.”[1]. Therefore, at each moment a fundamental choice needs to be made:

[1] From the movie “Batman Begins” (2005) Warner Bros. Entertainment Inc.

either to perform an evaluation that has been previously evaluated and hence will have a high chance of performing similarly well (this is known an **exploitation**), or to take a chance evaluating something new and potentially perform worse, in the hope of finding better solutions (this is known an **exploration**).

If only previously evaluated solutions are reevaluated, then little, or no, learning occurs, however the performance is likely to be predictable. This is known as a "pure exploitation" strategy. If only previously unevaluated solutions are evaluated, then learning occurs, however the performance will likely be unpredictable. This is a "pure exploration" strategy. If a combination of evaluated and unevaluated solutions are evaluated, then some learning can occur, which will result in unpredictable performance, but superior solutions which have been learnt can be exploited and this is likely to continue to increase over time as learning occurs. Striking the balance between exploration and exploitation is known as the Exploration Exploitation (EE) tradeoff and has been well studied in Reinforcement Learning (RL) research using the Multi-Armed Bandit (MAB) problem [28,66,128,131] which is discussed in the following section.

EE

EE is commonly discussed in an offline search context where it refers to the allocation of resources used for search intensification (exploitation) and diversification (exploration) phases. In this scenario, the fitness of the evaluations is not the performance metric, but rather the number of evaluations required to find a solution.

For this research however, EE is used in the online learning context.

4.2.3 Multi-Armed Bandit (MAB)

> " Bandit problems embody in essential form a conflict evident in all human action: information versus immediate payoff. "
>
> - Peter Whittle (1989)

MAB

The *Multi-armed Bandit Problem* (MAB) problem first originated as an idealised experiment for analysis in sequential design problems by [106]. It has since been refined into a generic problem reflecting common real-world EE challenges along with many variations

to highlight specific aspects of interest [78]. In recent times the problem has typically been used for benchmarking advances in RL algorithms.

The MAB problem name originates from the colloquial term of the 'One-armed bandit' referring to a slot (or poker) machine which have one lever (one-armed) and takes your money (like a bandit). The idealised MAB problem is one where there are K levers (or machines) and each one has its own payoff $r_i(t)$ where i is the machine number and $r_i(t)$ is the return based on a unique probabilistic (usually Gaussian) distributions with an average payout of μ_i and a standard deviation of σ_i.

Ideally, the machine with the highest mean would be continually sampled as this would maximise the expected average reward. However, since the payout by each machine is probabilistic, a number of samples may be needed to "learn" accurately enough about each machine as to determine which lever has the best underlying payout distribution. Given a finite number of evaluations, where one lever is selected at each time step t over the duration T where $t = [0..T]$, then the ideal payoff is one that maximises:

$$\sum_{t=0}^{T} r_{s(t)}(t) \tag{4.2}$$

This turns out to be a very challenging task, nonetheless effective strategies that converge on the optimal machine have been developed [38],[152] under certain conditions.

In practice however, simple algorithms like the ϵ-greedy are often employed. The ϵ-greedy approach exploits the lever believed to have the best long-term return with probability 1-ϵ, otherwise (with ϵ probability) it chooses a lever at random. The ϵ parameter, where $0 < \epsilon < 1$, is a tuning parameter, which can be fixed or dynamically updated (scheduled or respond adaptively based on heuristics [133]) throughout the run.

ϵ-GREEDY

Rather than maximising payoff, solving the MAB problem can alternately be viewed as attempting to minimise bad choices resulting in "potential" losses. The concept of regret is defined as the lost potential gain due to not taking the globally optimal action for a particular play during the run. The cumulative regret is often referred to as the loss function. For the standard MAB problem, regret grows at least logarithmically with the number of plays [9].

REGRET

LOSS FUNCTION

This and other methods for addressing the MAB problem [8,11,33,49,116,122,127] focus on the objective of maximising the cumulative payout and/or minimising the regret. However there appears to be no consideration of achieving *satisfactory average* performance, nor maximising learning given the need to meet a satisficing constraint. As such, these heuristics cannot be used to achieve acceptable performance with GP in online learning scenarios.

4.2.4 Performance Metrics for Learning Algorithms

One can consider performance of a machine learning algorithm from 2 perspectives. One is from the perspective of how "good" a learning algorithm is with respect to various qualities, usually in context to how other algorithms perform, or against an idealised algorithm. Frommberger [35] suggests the following performance measures for learning systems:

- Guaranteed convergence to the optimal
- Time to converge to the optimum
- Time to reach near-100% success.
- Time to reach near-optimality
- Minimal regret
- **Success rate of attaining acceptable performance**
- Optimality of solution
- Robustness of the learning process
- Avoiding forbidden states during learning
- Number of parameters required by the learner

Frommberger [35] however cautions that "All of the given performance measures have their application in certain scenarios, and which one to focus on will generally depend on the task at hand". The task at hand for this thesis is achieving desirable average performance during operation using the IDGP framework (as identified in Section 3.6.2). This goal-oriented focus is the other perspective, which focusses on whether a goal can be or is being achieved and not on whether it could have been done more efficiently

or more optimally. Of the performance metrics previously listed, only the success rate of attaining acceptable performance is a true goal-oriented metric. The other metrics, while no doubt beneficial to have optimised, may or may not be significant to whether the system's performance is or was acceptable. Frommberger's performance metrics were clearly envisaged from the perspective of benchmarking learning algorithms as they implicitly apply after learning has finished. When a system is deployed however, the current or achievable performance will be of interest.

"Local online performance" is proposed by De Jong [24] as a practical population based learning performance metric. It is a function of the sum of weighted fitnesses from all evaluations that have occurred up to the current generation. Specifically, it is defined as:

$$x_e(s) = \frac{1}{\sum_{t=1}^{T_e} c_t} . \sum_{t=1}^{T_e} c_t . u_e(a_t) \tag{4.3}$$

where $u_e(a_t)$ is the performance of the population a at time t (i.e. pool fitness), which is weighted by c_t and averaged over the time period T_e (i.e. from the start of evolution to the current epoch). c_t is provided as function to bias the performance measure based on time (e.g. one could weight later performance over initial performance), however [24] employed uniform weighting such that $\forall t : c_t = 1$ resulting in the local online performance being equal to the mean of all the evaluations.

Success rate of attaining acceptable performance is a practical real-world performance metric since it does not require knowledge of the optimal solution or even its fitness, but rather, only the determination of whether a goal has been reached or not [35]. *Success rate* employs a *success function*, $S(a, b)$, which is used to represent this binary outcome of 1 (for success) if the condition $a \geqslant b$ is satisfied, otherwise the result is 0. Note that condition b which defines success or not is context specific and can in fact be supplied via the Fitness Importance heuristic. Formally, success if defined as:

SUCCESS RATE

$$S(a, b) = \begin{cases} 1 & \text{if } a \geqslant b \\ 0 & \text{otherwise} \end{cases}$$

Success rate of attaining acceptable performance will be used as the online performance metric for this research. Specifically, the average performance within a generation meeting or exceeding the acceptable performance criteria

4.3 FI Analytic Model

FI

Fitness Importance (FI) is a context information sharing heuristic that describes the desired level of performance of a population of programs. The desired performance can be expressed in either absolute terms as a target average fitness score to be achieved by the population or as a relative fitness to the current population's performance characteristics. Since it is common for EA performance to be measured in absolute terms, FI will be discussed in absolute terms as an acceptable average fitness before the relative (to the current population characteristics) definition of the Fitness Importance Function (Φ) is detailed.

4.3.1 Acceptable Average Fitness

Implicitly, performance is dependant on the fitness as measured within the observer's critiquing period. To an observer with a short critiquing window observing sequential evaluations of programs from a population, performance would vary over time. If the critiquing window is aligned to the epoch (period of evaluation of all programs within the population), then arguably the pool fitness can become the observed fitness. This is particularly the case where the fitness effect of programs in the environment is additive over time. In this case, optimising the average (pool) performance is desirable. However, when an acceptable fitness threshold is introduced, optimising the average performance is no longer optimally desirable since further fitness improvement does not yield more success but rather potentially reduces the exploration that could have been achieved.

Fitness Importance is defined by the threshold of acceptable fitness *relative* to the current population's performance capacity. However, it is also possible to consider the

effect of FI in *absolute* terms as an acceptable average fitness threshold. Specifically,

$$F_{accept}(x) = F_{pool}(\Phi(x), x) \tag{4.4}$$

where $F_{accept}(x)$ is the *Acceptable Average Fitness* (AAF) function which can be expressed as a pool fitness (average population fitness) for generation x given a desired FI of $\Phi(x)$. If $\forall x, F_{accept}(x) = K$ where K is a constant (absolute fitness), then we further simplify the notation to F_{accept}. AAF

4.3.2 Fitness Importance Function, Φ

Having motivated the need for FI and demonstrated its potential utility in managing the exploration/exploitation tradeoff, we now formally present the model for Fitness Importance.

This thesis proposes that $\Phi_{accept}(t)$ could be a more practical and pragmatic method for setting acceptable performance since it is calculated relative to the known-to-be-achievable fitnesses of the current generation (i.e. $F_{elite}(g)$ and $F_{pool}(g)$).

The absolute value of F_{accept} can be expressed in terms of Φ_{accept} by Equation 4.5 which readily serves to provide a sanity check about whether the acceptable performance is realistically immediately achievable, in that if $\Phi_d(g) \leq 1$ then it is likely that it can indeed be obtained.

$$\Phi_d(g) = \frac{F_{accept}(g) - F_{pool}(g)}{F_{elite}(g) - F_{pool}(g)} \tag{4.5}$$

If F_{accept} is constant then providing evolution/learning is occurring, then Φ will decrease over generations, hence having a constant Φ or ϕ makes less sense,

In practical implementations it is easy to set Φ so that the next generation target performance is achieved with a maximal allocation of learning whilst meeting the performance constraint. Leaving it at this value will continue to meet the performance requirement (so long as it remains unchanged) and keep learning (at a rate that is not optimal however)

A Genetic Algorithm (GA) is a population-based heuristic learning mechanism that successively applies genetic operations on a population. Let the set of all possible unique populations be $\mathcal{P} = \{p_1, p_2, .., p_{N_\mathcal{P}}\}$ where p_i is an arbitrary unique population. Note that

the syntactic richness of the instruction set may mean the number of unique programs, $N_{\mathcal{P}}$, is practically infinite.

Let g represent the generation number of the population generated by the genetic algorithm. Note that time can be quantised into discrete units defined by the time taken to evaluate all programs within a population of a particular generation. For simplicity we use the generation as our timebase. i.e. $\Phi(g)$.

We introduce the short notation of ϕ_{accept} to represent a constant $\Phi(g)$ at a particular period of interest which expresses the acceptable performance improvement as a percentage of the difference between the elite and pool fitnesses without FI applied. If we express the elite and pool fitnesses as functions of ϕ and g, i.e. $F_{elite}(\phi, g)$ and $F_{pool}(\phi, g)$ respectively, then the elite and pool fitnesses without FI applied are obviously $F_{elite}(0, g)$ and $F_{pool}(0, g)$.

$$\begin{aligned} E(F_{pool}(\phi_d, g)) =& F_{pool}(0, g) \\ &+ \phi_{accept}[F_{elite}(0, g) - F_{pool}(0, g)] \end{aligned} \tag{4.6}$$

The difference between the elite and the pool at every generation dictates the extent of what can be exploited. We define the exploitation potential function at generation g as

$$F_{\Delta_{ep}}(0, g) = F_{elite}(0, g) - F_{pool}(0, g) \tag{4.7}$$

and so equation (4.6) becomes,

$$E(F_{pool}(\phi_d, g)) = F_{pool}(0, g) + \phi_{accept} F_{\Delta_{ep}}(0, g) \tag{4.8}$$

The achieved performance improvement $\phi_{achieved}$, can be expressed in terms of the current elite and pool fitnesses as

$$\phi_{achieved} = \frac{F_{pool}(\phi_{accept}, g) - F_{pool}(0, g)}{F_{\Delta_{ep}}(0, g)} \tag{4.9}$$

Ideally $\phi_d = \phi_{achieved}$, however practical considerations, explained in the next section, can

prevent this from occurring.

A value of 0 corresponds to no importance being placed on the current fitness of the system, but it provides optimal learning potential (i.e. it should provide the minimum time to discover an optimal solution). It is worth noting however that $\phi = 0$ does not mean that the fitness of the system is also 0. In fact, $\phi = 0$ represents the conventional GP approach for maximising exploration. A value of 1 for ϕ however corresponds to a maximum importance of fitness, meaning that the system should perform as best as possible with the current knowledge of programs and their fitnesses. To achieve this, the program with the highest expected fitness (i.e. E_1) should be exploited. This is similar to offline evolutionary approaches where the best solution evolved offline is placed into the online environment and then not altered. Setting $\phi = 1$ at generation g renders $F_{pool}(g) = F_{elite}(g)$, but it also prevents any exploration (learning), because of the lack of diversity in the program population. FIF

The FI function $\Phi(t)$ is defined to lie in the range $0 \leq \Phi(t) \leq 1$. $\Phi(t) = 0$ indicates that the importance of fitness is minimal and thus there are no penalties for executing poor performing logic. This provides an opportunity for maximum exploration.

While $F_{pool}(\phi, g)$ provides an instantaneous [2] performance measure, NAP provides a definition for a longterm performance metric. We now wish to analytically explore the impact of FI on longterm performance, however to achieve this we need to better understand how the pool fitness varies over time in the absence of FI being applied.

The longterm pool and elite fitnesses are tied to the optimisation trajectory of the population. As stated in Equation 4.31, GA uses selection and genetic operators on a population to generate a new population which hopefully has a high chance of discovering a program with higher fitness than the current elite program. If we assume $\mathcal{G}(\phi, p)$ produces new populations in a deterministic manner (i.e. no stochasticity is involved) and the environment is not dynamic in the short term (i.e. the fitness metric is constant and there is no noise in measuring fitness), then the population at generation g can be described in

[2] $F_{pool}(\phi, g)$ is calculated over the duration of the evaluation time for a generation of programs. However, as we quantise time into generations, this measure occurs in 1 unit of time in our timebase.

terms of an initial population p_i as:

$$p(i,g) = \mathcal{G}(\mathcal{G}(....\mathcal{G}(\mathcal{G}(p_i))...)) \tag{4.10}$$

where the genetic operator G is applied successively $g-1$ times, thus $p(i,1) = p_i, p(i,2) = \mathcal{G}(p_i)$ and so on. The set of all optimisation trajectories under non-stochastic operations and in a non-dynamic environment would therefore be a function of the set of possible initial populations, P, and therefore there must also be N_P possible optimisation trajectories.

In the real world, changing environmental conditions and stochastic processes employed by $\mathcal{G}$ result in the actual number of possible optimisation trajectories being more than N_P and not deterministic on p_i alone. As the effects of environment and stochasticity increase, the optimisation trajectory dependance on p_i becomes less significant. The IDGP framework employs a number of stochastic processes (random program generation, mutations, random crossover points, fitness proportionate parent selection), and combined with real-world dynamics such as sampling noise, thermal variation and other potential environmental variations, the optimisation trajectory dependance on p_i is assumed to be very weak. Hence, we remove the dependance on p_i from the notation, i.e. $p(i,g) \simeq p(g)$. Non-deterministic processes include: syntactic biases, such as instruction set and programming language syntax, and procedural biases such as the population composition, genetic operators, and changes to the environment.

We now introduce L to describe the fitness improvement rate during evolution. L can be thought of as how much is “learnt” between generations and as such is defined as the derivative of the elite and pool fitnesses with respect to generation, g.

$$L_{elite}(\phi,g) = \frac{\mathrm{d}}{\mathrm{d}g} F_{elite}(\phi,g) \tag{4.11}$$

and similarly for the pool learning rate,

$$L_{pool}(\phi,g) = \frac{\mathrm{d}}{\mathrm{d}g} F_{pool}(\phi,g) \tag{4.12}$$

The elite and pool fitnesses start to deviate from the $F_{elite}(0, g)$ curve at generation k where FI is applied. FI reduces the population size used for exploration and increases the number of elites which exploit the known good solutions. We term the population pool used for exploration as $pool^*$. With $\phi = 0$, the population size of $pool^*$ is $N_{pop} - 1$ as one program is used to facilitate the elitism mechanism. When $\phi = 1$, the population size of $pool^*$ is 0. With the current defined $\mathcal{G}_{simple}(p_i)$ function, when $0 < \phi < 1$, $pool^*$ has a population size of approximately $1 + (1 - \phi)(N_{pop} - 1)$.

Since $F_{elite}(\phi, g) = \mathrm{MAX}(F_{pool}(\phi, g))$, any improvement is the result of learning occurring in $pool^*$. Hence, increasing ϕ will reduce the size of $pool^*$ and its learning rate. This reduced fitness is represented by a learning rate scaling function, $\lambda_{elite}(\phi_d, g)$, such that,

$$\begin{aligned} F_{elite}(\phi_d, g) =& F_{elite}(0, k) \\ &+ \lambda_{elite}(\phi_d, g) \int_k^g L_{elite}(0, x)\mathrm{d}x \end{aligned} \tag{4.13}$$

Similarly the reduced population $pool^*$ fitness can be expressed as

$$\begin{aligned} F_{pool^*}(\phi_d, g) =& F_{pool}(0, k) \\ &+ \lambda_{pool^*}(\phi_d, g) \int_k^g L_{pool^*}(0, x)\mathrm{d}x \end{aligned} \tag{4.14}$$

However, of most interest is the calculation of the new pool, $F_{pool}(\phi_d, g)$ which represents the post-FI performance. The new $pool$ is a composition of ϕ $elite$ programs and $(1 - \phi)$ $pool^*$ programs and so can be expressed in the previously defined terms as

$$F_{pool}(\phi_d, g) = \phi F_{elite}(\phi, g) + (1 - \phi) F_{pool^*}(\phi, g) \tag{4.15}$$

We rearrange this equation, to highlight the salient contributions to the post-FI performance.

$$\begin{aligned} F_{pool}(\phi_d, g) =& F_{pool}(0,k) + \phi\Big[F_{elite}(0,k) - F_{pool}(0,k)\Big] \\ &+ \phi\lambda_{elite}(\phi_d, g)\int_k^g L_{elite}(0,x)\mathrm{d}x \\ &+ (1-\phi)\lambda_{pool^*}(\phi_d, g)\int_k^g L_{pool^*}(0,x)\mathrm{d}x \end{aligned} \tag{4.16}$$

Effectively, the new $pool$ deviates at the point $F_{pool}(0,k)$ with an immediate performance improvement of $\phi\Big[F_{elite}(0,k) - F_{pool}(0,k)\Big]$ as described in the instantaneous section. Note, when $g = k$, Eq. (4.17) reduces to Eq. (4.6). However, when $g > k$, the $pool$ performance continues to improve with the new pool learning rate of

$$\begin{aligned} L_{pool}(\phi, g) =& \phi\lambda_{elite}(\phi_d, g)L_{elite}(0,g) \\ &+ (1-\phi)\lambda_{pool^*}(\phi_d, g)L_{pool^*}(0,g) \end{aligned} \tag{4.17}$$

$L_{pool}(\phi, g)$ will exhibit non-linear behaviour, even if $\lambda_{elite}(\phi_d, g)$ and $\lambda_{pool}(\phi_d, g)$ are constants, since $L_{pool}(\phi, g)$ depends on both $pool$ and $elite$ learning rates. This makes determining NAP a difficult problem. Equation 4.16 can be simplified as:

$$\begin{aligned} F_{pool}(\phi_d, g) =& F_{pool}(0,k) + F_{\Delta_{ep}}(0,k) \\ &+ \int_k^g L_{pool}(\phi, x)\mathrm{d}x \end{aligned} \tag{4.18}$$

We now consider the relationship between the learning rate reduction factor λ and fitness importance more closely. It is well known that reduction in diversity reduces the learning potential of the population. At the extreme, setting $\phi = 1$ will completely remove learning capability, leading to $L_{pool}(g)|_{\phi=1} = 0$. Populations are composed solely of the elite program from the previous generation, effectively limiting performance variation due to learning.

However, by selecting the elite from the previous generation, we typically maximise the probability of good performance of the current generation since the probability is low that

any variation due to genetic operations will result in a phenotype with higher performance than the previous elite program. Consider the long term effect of ϕ through an example. Each generation includes 20 program that run sequentially. A node runs the programs for 5 generations using the IDGP framework and ranks the programs according to fitness. In a normal GP approach, achieving a solution with optimal fitness would require on average 100 generations. In our example, we apply a $\phi = 0.5$ at generation 6. In response, the node sets half of its population to the current elite program, while it continues to perform genetic operations on the remaining 10 programs to learn new behaviour. This results in a step increase in performance so that the observed performance during generation 6 is midway between the elite program performance at generation 5 and pool fitness at generation. This step increase comes at the cost of an increase in convergence time towards the optimal solution. Instead of converging after 95 generations, we hypothesise that the logic now requires on average 2 x 95 = 190 generations to converge. This stems from the subpopulation size that is effectively used for learning being halved [3].

Generalising the example above, we hypothesise that the learning rate after applying ϕ is reduced by a factor of $(1 - \phi)$. As such, over time the learning rate reduction factor should tend towards $(1 - \phi)$. i.e.,

$$\lambda = 1 - \phi \tag{4.19}$$

Experiments were performed to validate this hypothesis using empirical results obtained via a parameter sweep of ϕ and k with 20 repeated experiments of the Blink-3 problem for each $\phi \in \{0, 0.05, 0.15, 0.4, 0.5, 0.7, 0.95, 1\}$ applied starting at generations $k \in \{5, 10, 15, 25, 50, 250\}$. The results obtained from 1 month of evolution were stored to log files and the analysis of this data is discussed in the remainder of this section.

[3] Section 5.3.2 will empirically test this hypothesis on the impact of learning rate upon applying ϕ.

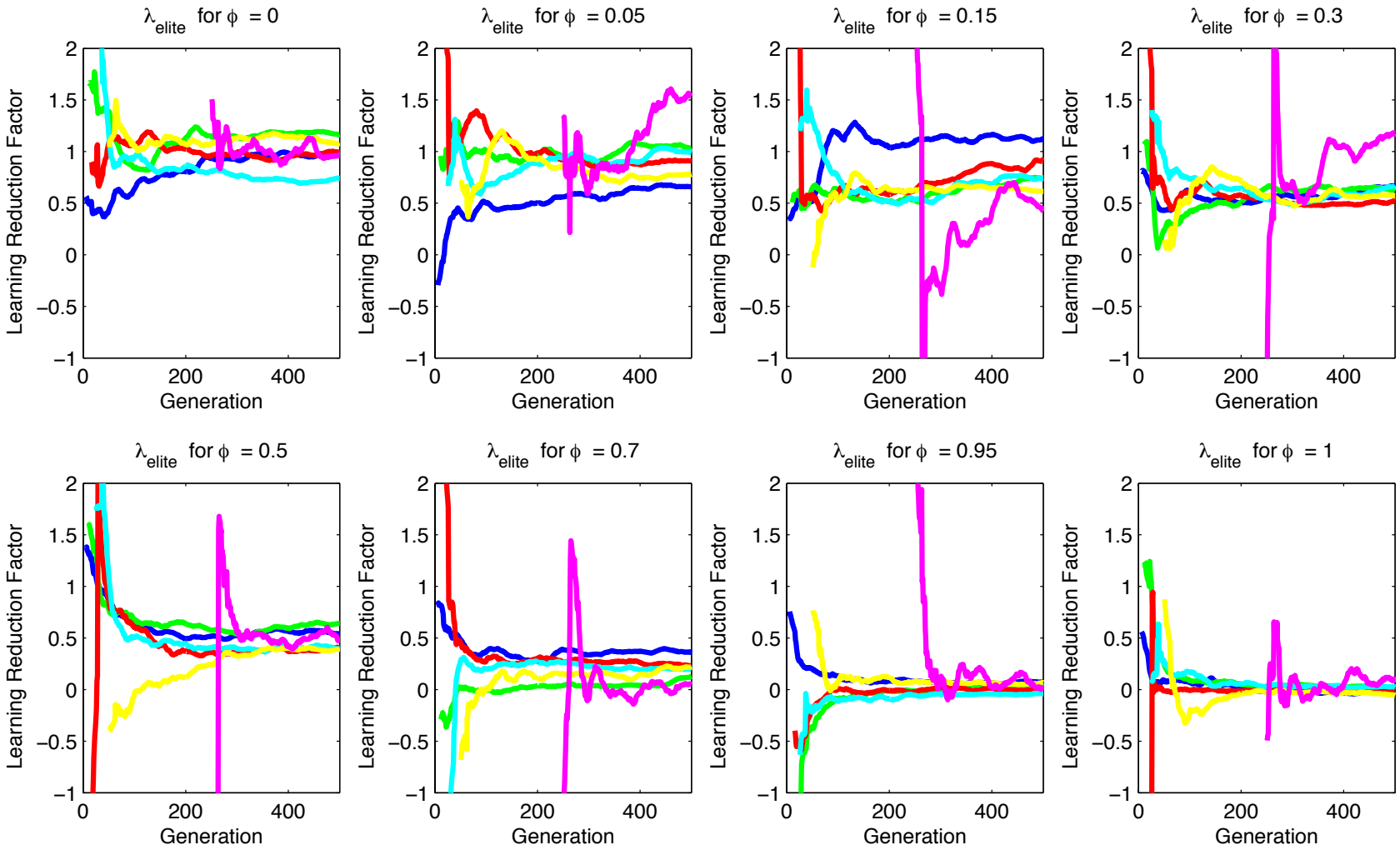

Figure 4.1: λ_{elite^*} for various values of ϕ_{accept} showing that the Learning Reduction factor can be estimated more accurately later during evolution. The analytic model suggests that $\lambda_{elite^*} = 1 - \phi$, however due to the stochastic and discrete nature of an evolutionary population performance, the trend is hard to see until close to convergence.

Figure 4.1 shows the evolution of $\lambda_{elite^*}(\phi_{accept}, g)$ for various values of ϕ_{accept}. Each individual line represent the smoothed average of the final 20 generations of each run where ϕ_{accept} was applied at generation k. Since $\lambda_{elite^*}(\phi_{accept}, g)$ is a function of how much is learnt since FI was applied, the initial calculations of λ after k fluctuate significantly with noise. However, the cumulative nature of the integral of the learning rate means that the variation decreases over time, and provides a better insight to what the underlying real value of $\lambda_{elite^*}(\phi_{accept}, g)$ converges. There is a variation in the convergence value of λ that is most pronounced for smaller values of ϕ.

To examine the relationship further, Figure 4.2 plots λ versus ϕ. While each of points in the figure refers to an individual run, the solid line is the (1-ϕ) line. It is clear that this function does capture the trend in the relationship between λ and ϕ, and it does so more accurately for larger values of ϕ.

The post-FI pool performance will contain ϕ elite programs while the remainder of the population will have fitnesses that tend to the average program fitness, regardless of the method used. As expected, Figure 4.3 shows the learning rate of the diminished size learning population generally trends downward as ϕ is increased (during the runs where $0 < \phi_g < 1$). Counterintuitively, for some low values of ϕ however there are occurrences where better learning occurs with a smaller learning population. This implies that despite the reduction in population size used for learning, the learning rate has actually increased. However, any improvement in the elite will be magnified by the size of the elite population portion in $pool^*$, and so while the learning rate of the pool is reduced, the amplification of any improvement by the number of elites more than compensates for the reduced learning rate (at least while $\phi < 0.95$ for this particular problem). Figure 4.4 shows the full population learning rate

The acceptable effect of FI is that it will increase the pool fitness of future generations to reside above the current pool fitness by at least the percentage (ϕ_{accept}) of the difference between the current elite and pool fitnesses. This is expressed as Equation 4.20 for any generation $g > k$, where k is the generation at which ϕ is applied.

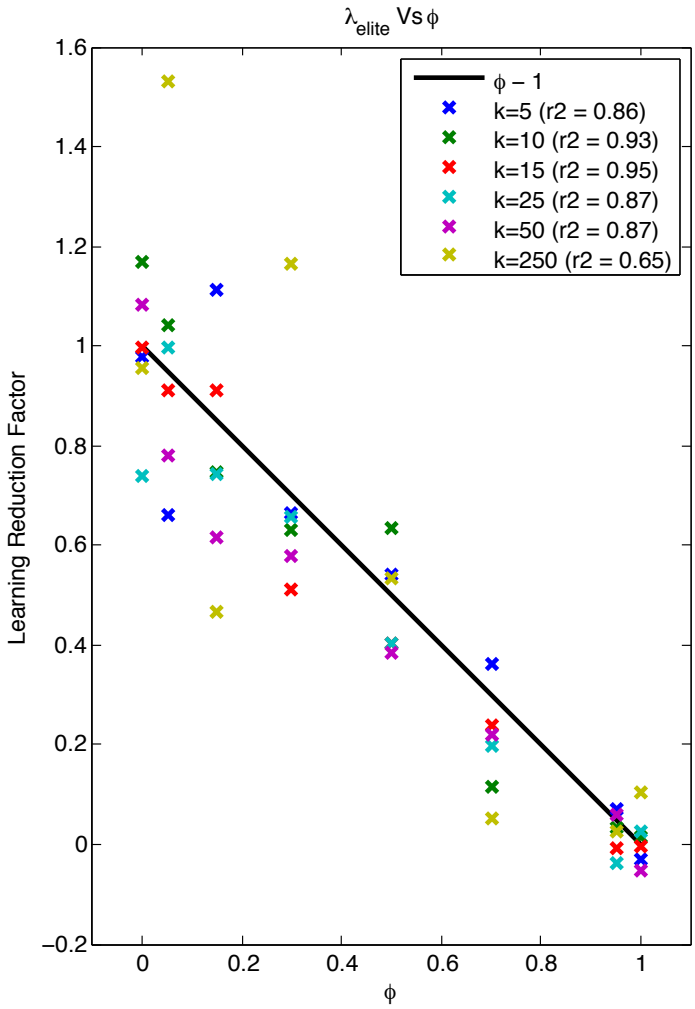

Figure 4.2: This figure shows λ_{elite} Vs ϕ. Also shown is the line of $1-\phi$. There is good correlation with this indicating that $F_{elite}(\phi, g) = (1-\phi)\int_{x=k}^{g} F_{elite}(0,x)\mathrm{d}x$ is reasonable assumption.

$$F_{pool}(\phi_d, g) = F_{pool}(0,k) + \phi_{accept}[F_{elite}(0,k) - F_{pool}(0,k)] + \int_{x=k}^{k+n} L(\phi_d, x)\mathrm{d}x \tag{4.20}$$

Note that Equation 4.20 is the general form of Equation 4.6. The main difference is that Equation 4.6 only captures the expected step increase in pool fitness, while the generalised Equation 4.20 also includes the additional improvement in fitness (learning)

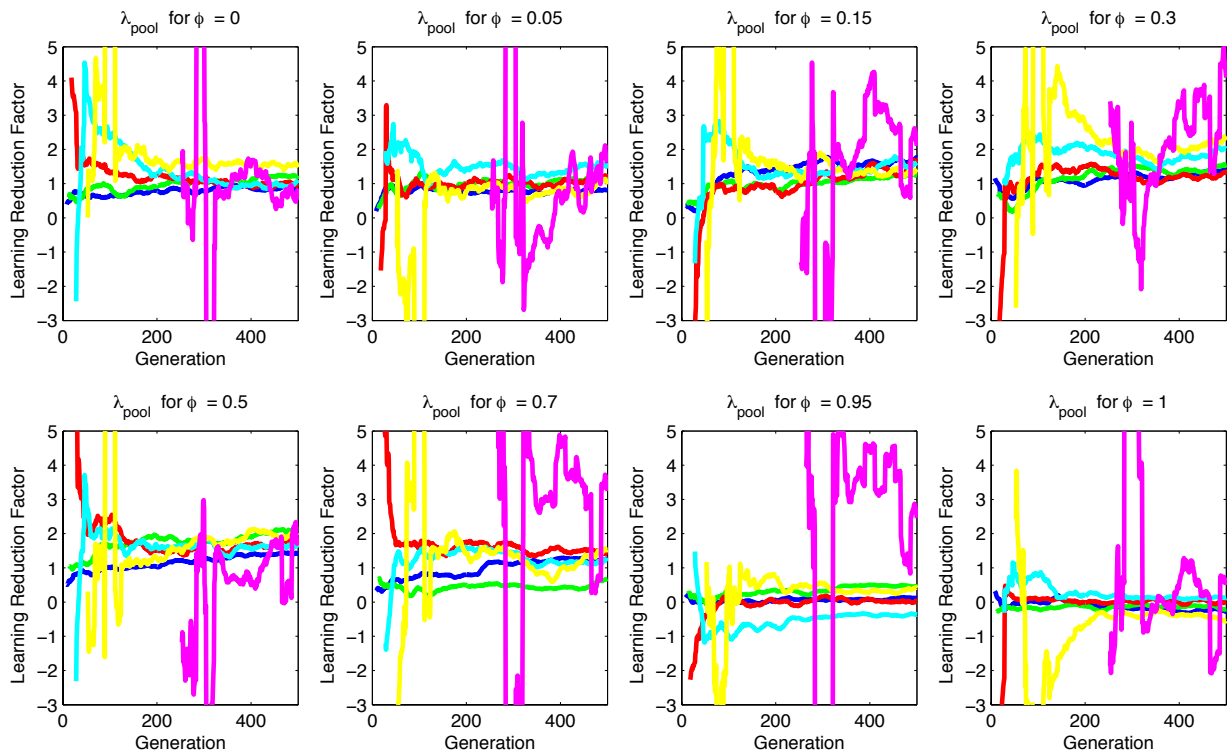

Figure 4.3: λ_{pool^*} for various values of ϕ.

that occurs in the generations that follow the application of ϕ_d

In order to deliver the acceptable increase in pool performance (as per eq. 4.6), FI biases the population diversity by increasing the number of elite programs in the population to approximately $\phi_{accept}N_{pop}$. This provides the minimum acceptable pool performance of $\phi_{accept}F_{elite}(g)$, while the remainder of the population is configured for optimal searching. If the learning rate after applying FI $L(\phi_d, g) \geq 0$, then the performance achieved at generation g will be higher than at generation $k+1$. However, for specific cases where devolving can occur to significant changes in the environment, $L(\phi_d, g)$ may be negative, resulting in a lower fitness at generation g than generation $k+1$. However, in most cases where the environment does not drastically change, the learning rate will remain positive, yielding a monotonically increasing fitness after applying ϕ_d.

With this expanded definition of learning rate, we can revisit the projected pool fitness as a function of the performance gain and reduced learning after applying FI. Let the fitness exploitation potential, which is the difference between $F_{elite}(0, g)$ and $F_{pool}(0, g)$, as

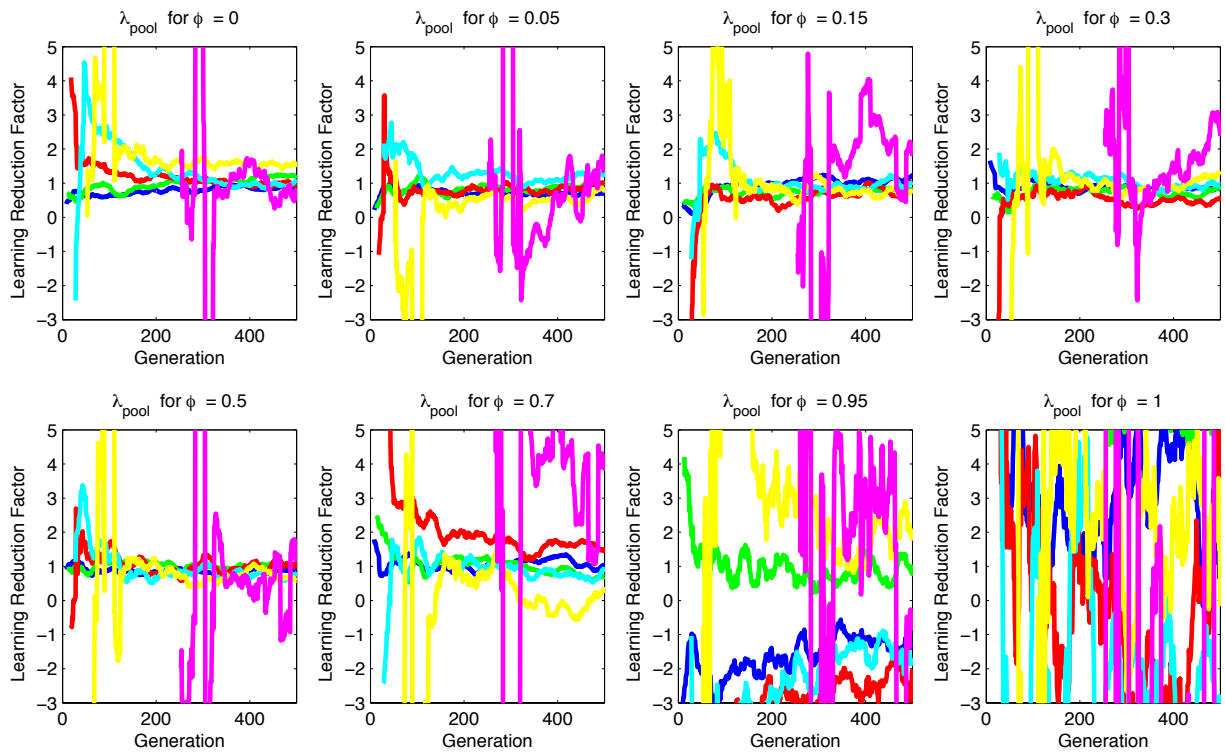

Figure 4.4: λ_{pool} shows the learning rate reduction in the population including the Elites. The calculation requires division by $(1-\phi)$ so as ϕ increases, small stochastic changes become highly magnified leading to "noisy" calculated values. This is particularly evident for $\phi = 1$ (bottom right).

$F_{\Delta ep}(g)$. This describes how much fitness can be potentially exploited by FI at generation g. Assume $F_{pool}(g)$ is a monotonically increasing function. Applying FI after generation k results in a step increase in fitness by $\phi_d F_{\Delta ep}(k)$ as of generation $k+1$, representing an immediate performance gain. Using the revised definitions with Equation 4.20 yields:

$$\begin{aligned} F_{pool}(\phi_d, g) = \phi_d \left[F_{elite}(0,k) + (1-\phi_d) L_{elite}(0,k)\right] \\ + (1-\phi)\left[F_{pool}(0,k) + (1-\phi_d) L_{pool}(0,\phi_d)\right] \end{aligned} \tag{4.21}$$

More detailed derivation of Equation 4.21 can be found in Appendix B.

Equation 4.21 quantifies the step performance gain and the reduction in learning rate when ϕ is applied. Achieving an appropriate balance between performance and learning is highly depend on the time on the application scenario, so the next section explores this relationship for the maximum performance scenario.

4.3.3 Application Scenarios

The target application scenarios are a subset of the real world problems as discussed in Section 4.1.1.

Achieving an appropriate balance between learning and performance is a highly application-specific problem that depends on the window of time for evaluating performance and on the level of required performance. We consider six different application scenarios, as shown in Table 4.1.

#	Problem Type	Time	Performance	FI	Description
1	Maximum Online learning	unbounded	none	$\phi = 0$	Continuous learning with an infinite time observation window
2	Constrained perpetual learning	unbounded	minimum threshold	$0 < \phi \leq 1$	Continuous learning that imposes a minimum requirement on performance while learning
3	Maximum perpetual performance	unbounded	maximum	$0 < \phi \leq 1$	Require maximum aggregated performance over an infinite time window
4	Offline learning	bounded	none	$\phi = 0$	Maximise learning in a finite time window to exploit it later
5	Constrained learning	bounded	minimum threshold	$0 < \phi \leq 1$	Maximise learning in a finite time window while meeting some minimal performance threshold
6	Maximum performance	bounded	maximum	$0 < \phi \leq 1$	Maximise aggregated performance over a given time window

Table 4.1: Learning problem type

The first application scenario simply aims to maximise the learning capacity of the system for an infinite duration, and it represents a maximum online learning application corresponding to the minimum FI value of 0.

The second and third application scenarios also observe the system over an infinite time window, but they place constraints on the system's performance. The constrained perpetual learning scenario aims to maximise learning while delivering a minimum level of user-defined performance, while the maximum perpetual performance requires that the system's aggregated performance over time reaches its highest possible levels. Both of these scenarios significantly benefit from setting FI to a value greater than 0 in order to deliver the acceptable level of performance.

Scenarios 4, 5, and 6 all have a bounded time window. Scenario 4 allows a system to maximise learning within the bounded time window, in anticipation of exploiting the evolved behaviour beyond the time window. This is the offline learning scenario. Similarly to maximum online learning, this scenario uses a FI of 0. Scenarios 5 and 6 resemble

scenarios 2 and 3 above, except that they use a bounded time window for evaluating learning and performance.

Previous work, such as [50], have explored in situ evolution (scenarios 1 and 4) for online learning in the real-world. Significant research efforts have been made for maximising performance within a bounded time interval, such studies on the bandit problem. To the best of our knowledge, we are the first to propose a method that allows the system to meet a minimum performance criteria while maximising learning (scenarios 2 and 5).

4.3.4 FI Performance Metrics

4.3.4.1 Normalised Average Performance

Symbols

Symbol	Description
p_i	Unique population i where $1 \leq i \leq N_P$
P	Set of all possible unique populations $p_1, p_2, ..., p_{N_P}$
g	Generation number where $g \geq 1$
$\mathcal{G}(p_i)$	$\mathcal{G}(p_i) \mapsto p'_i$
$p'_i(g)$	Population at generation g given the initial population p_i
$\phi_{g,n}$	Desired Fitness Importance from generation g to $g+n$
$F_{pool}(p_i, g)$	Pool Fitness at generation g, given the initial population p_i
$F_{\Delta ep}(p_i, g)$	Exploitation Potential defined by $F_{elite}(p_i, g) - F_{pool}(p_i, g)$
$L_{pool}(g, n)$	Pool Learning Rate defined by $F_{pool}(p_i, g) - F_{pool}(p_i, g-1)$
$L_{elite}(g, n)$	Elite Learning Rate defined by $F_{elite}(p_i, g) - F_{elite}(p_i, g-1)$

Table 4.2: Symbols

In Section 5.2 we saw the effect of applying FI on the pool fitness in response to a desired immediate performance improvement. However, if we consider performance over period of time during evolution, then it is likely that the performance will vary over that period due to learning and/or changes to the environment. An external observer will see the performance resulting from the execution of every program in the pool. Extending this to a period of time described by multiple generations, an external observer would also see every program across every generation in the period of interest. Recall that the pool fitness represents the average fitness of sequentially run programs during a particular

generation. Since all generations have the same number of programs, the average of all pool fitnesses provides the average fitness of all programs during the period of interest.

For our toy-problem, we know an optimal program and can therefore execute it to determine the optimal fitness. This fitness score varies from node to node due to differences in hardware and can also vary in time due to temperature and other varying environmental factors. In most real-world problems, the optimal fitness may not be calculable or known, but for convenience of comparing performance across multiple nodes, the fitnesses is normalised to the fitness of the optimal program (recorded at some time before the experiments were conducted). We define the Normalised Average Performance, $F_{NAP}(i,n)$ as the average pool fitness across generations i through to $i+n-1$. i.e. NAP

$$F_{NAP}(i,n) = \frac{1}{n}\sum_{g=i}^{i+n-1} F_{pool}(g) \tag{4.22}$$

This definition of NAP considers that the long term performance of an online learning system depends on the mean of the system's observed performance over the observation time window of length n. For a learning system that aims to maximise performance over an *entire* bounded time window, NAP represents the observed system's performance over the whole time, where $i = 0$ and $n = window\ length$. For a similar learning system that requires maximising performance over the *remainder* of the time window, i is set to the current point in time, while n is set to $window\ length - i$. The following sections demonstrate the utility of this NAP definition further.

$$\mathrm{NAP}(i,n) = \frac{1}{n}\sum_{g=i}^{i+n-1} F_{pool}(\Phi(g),g) \tag{4.23}$$

4.3.4.2 FI Success Rate Metric

The Success Function $S\big(\Phi_{achieved}(x),\Phi(x)\big)$ defines 'success' as achieving the desired fitness importance, $\Phi(g)$, at generation g.

The *success rate* is defined by the fraction of evaluations that meet the criteria of 'success'. For this research, the *Success Rate of Satisficing* $P_{success}(k,g)$ is defined as the percentage of generations in which $\Phi_{achieved}(x) > \Phi_{desired}(x)$ where $k \le x \le g$. This can

SUCCESS RATE OF SATISFICING

be expressed relative to Fitness Importance as:

$$P_{success}(k,g) = \frac{k-g}{g} \sum_{x=k}^{g} S\big(\Phi_{achieved}(x), \Phi(x)\big) \tag{4.24}$$

However, Fitness Importance can be expressed in terms of absolute fitnesses as:

$$\Phi(x) = \frac{F_{accept}(x) - F_{pool}(0,x)}{F_{\Delta_{ep}}(0,x)} \quad \text{and} \quad \Phi_{achieved}(x) \approx \frac{F_{pool}(\phi(x),x) - F_{pool}(0,x)}{F_{\Delta_{ep}}(0,x)}$$

Through substitution the success condition can be expressed follows:

$$S\big(\Phi_{achieved}(x), \Phi(x)\big) = \begin{cases} 1 & \text{if } F_{pool}(\phi(x),x) \geqslant F_{accept}(x) \\ 0 & \text{otherwise} \end{cases}$$

This now allows the success function to be expressed in terms of absolute fitnesses as:

$$P(k,g) = \frac{k-g}{g} \sum_{x=k}^{g} S\big(F_{pool}(\phi(x),x), F_{accept}(x)\big) \tag{4.25}$$

Hence, performance can be measured as the percentage of generations from k to g in which the achieved pool fitness equalled or exceeded the acceptable average fitness (AAF).

The maximum value of $P(g) = 1$ occurs when $\forall g : F_{pool}(\phi_{achieved}, g) \geqslant F_{accept}(g)$ and conversely the minimum occurs when success is never obtained, i.e. $P(g) = 0$ when $\forall g : F_{pool}(\phi_{achieved}, g) < F_{accept}(g)$.

Note that F_{accept} is an absolute value which can be set independently of the current fitnesses ($F_{pool}(g)$ and $F_{elite}(g)$). Unlike $\Phi(g)$ which can theoretically be obtained for all generations, an *a priori* value for $F_{accept}(g)$ could easily lie outside of an achievable fitness, i.e. $F_{accept}(g) > F_{elite}(g)$.

Online evolution provides a mechanism to adapt (converging and reconverging) to dynamic environments and time-varying fitness functions. Figure 4.5 illustrates typical elite and pool fitness curves for an evolutionary run. The vertical axis represents the fitness while the horizontal axis represents time (measured by generation number). Suppose

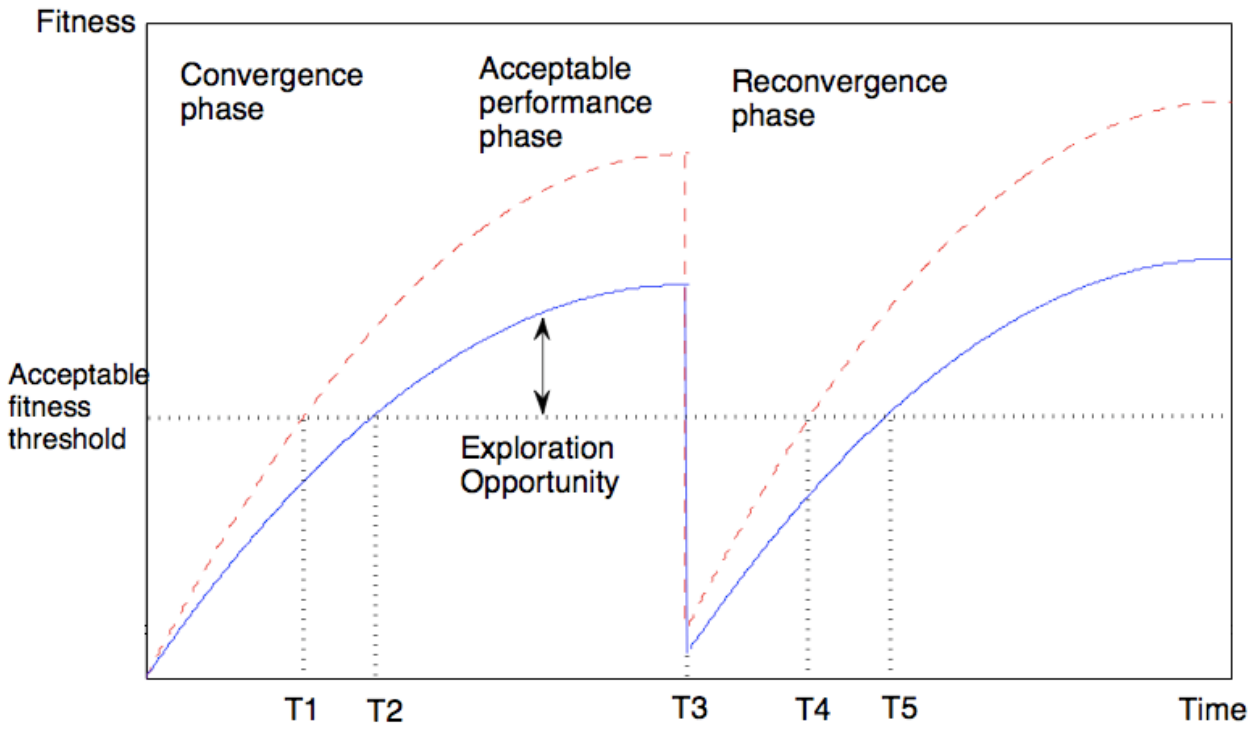

Figure 4.5: Expected evolutionary trajectories (of pool and elite fitnesses) with IDGP in a scenario where the fitness landscape suddenly changes (at T3).

there exists some minimum performance threshold, at some point, T_1 , the evolutionary search may find solutions which satisfy this constraint. Once a solution has been discovered, one could potentially exploit this solution to deliver acceptable performance, providing the environment and fitness objective remain constant. If we allow the evolution to continue, the elite will exceed the minimum acceptable threshold and potentially at some point, T_2, the average (pool) fitness may exceed the minimum acceptable threshold. In most problem scenarios, the pool fitness will be significantly less than the elite fitness and often the pool fitness will asymptote much earlier than that of the elite.

T_3 represents a point in time where an unexpected event occurs in the environment or there has been a change in the fitness metric, which results in a large drop in elite and pool fitness. The extent of the drop is inversely correlated with the diversity of the population prior to the event. With high population diversity the system is likely to experience a lesser drop in fitness. The system may drop below an acceptable fitness, however the evolutionary process will allow the fitnesses to reconverge to a new solution which

hopefully meets or exceeds the acceptable performance under these new conditions. Re-convergence may lead to the elite programs reaching acceptable fitness again (at T_4), and potentially the pool fitness reaching an acceptable fitness once again at time T_5.

The concept of FI is to provide a means to manipulate the evolutionary trajectories above, in particular, when the elite fitness exceeds an "acceptable fitness" threshold. Whenever the elite fitness is below the acceptable threshold (i.e. the convergence phase), the GP engine should maximise learning by setting FI to zero which will allow for the fastest possible evolution to an acceptable performance. [4] Starting at T_1, values of FI above zero can be considered which should cause the pool fitness to increase quickly towards the elite fitness, but it also limits subsequent fitness improvements.

Figure 4.6 illustrates this effect through an example from empirical experiments on a simple evolutionary problem, whose full details are explained in Section 5.2. It shows the trajectories of the pool fitness for a range of FI values, all applied at generation 41 for the remainder of the evolutionary run. One can see the immediate performance gains for all non-zero value of FI at generation 41. However, because of the reduced fitness improvement with the application of FI, it is clear that there is a crossover point at which exploiting via FI yields worse performance than not applying FI in the long term. The impact of Fitness Importance is largely dependant on where the system is within the search space and the optimisation trajectory that it is travelling along. Identifying the appropriate value of FI and the appropriate time at which to apply it is a challenging problem that we explore in this paper.

4.4 Population Generator Metaheuristic

FI is a heuristic that supplies domain-specific knowledge, namely what level of average fitness is acceptable. This information is then used by a metaheuristic which attempts to generate a population that will meet the AAF while attempting to maximise learning given the performance requirement.

[4] In exceptional cases that require immediate performance improvements even if performance is less than the acceptable threshold, the FI can be set higher during the convergence phase.

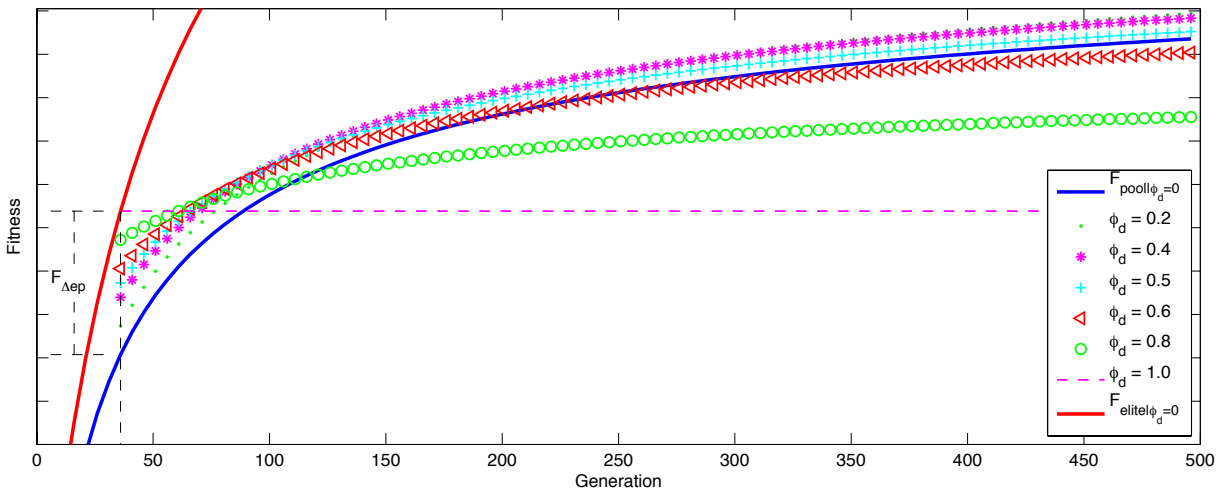

Figure 4.6: FI, $FI = 0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0$, is applied to idealised the pool and elite fitness curves to demonstrate the complex trade-off for performance and learning over time.

With the acceptable pool fitness defined, we now describe a population manipulation mechanism that, at minimum, generates a population with an expected average fitness as in Equation 4.8 while attempting to retain the maximum learning potential.

As is with most genetic algorithms, the initial state of the In situ Distributed Genetic Programming framework can begin with randomly generated entities or seeded entities provided by a user. Specifically for this implementation we start by constructing a population of random programs.

Using a supplied fitness function (or performance evaluation metric), each program is evaluated in turn and the performance (or score) for each program is recorded. Note that external to the system, the performance appears as a function of the aggregate of performances across all programs in the population.

The next generation population is constructed from a number of subpopulations, hereafter termed as classes. Classes are defined by various operations such as (but not limited to) cloning, mutation, crossover on programs from the previous generation or other sources (such as programs from other nodes, human-devised (injected) programs, or even randomly generated programs). CLASSES

The expected performance for programs in each subpopulation can be calculated based knowledge of the programs' previous performance and knowledge of the effect

of the operations on the performance of programs. For example, the expected performance of an elite (clone operation, which means the program is copied unaltered) in the next generation will be the same as the performance from the previous generation (in the absence of any other knowledge about changes in the environment). In contrast, the expected performance of generated children ("Children Subpopulation", defined by crossover and mutation operations on a set of probabilistically biased selected programs) will typically be less than the performance of either parent. However the expected learning potential for "Children" programs will be greater than that of "Elite" programs since "Children" programs will provide more knowledge about the fitness landscape than evaluating a program that has been tested recently.

Based on the expected performance and expected learning capability, subpopulations are constructed and selected to generate a total population which maximises the learning capability given an acceptable expected performance constraint. We term this acceptable expected performance constraint as the "Fitness Importance", since it reflects the level of importance of performance (fitness) over the importance of optimal learning. Since the optimal performance that can be achieved given the current knowledge of programs and their fitnesses is to use the best performing program for the entire population and presumably a population configuration is known for optimal learning, we then normalise "Fitness Importance" to range from 0 to 1, where 0 signifies a population of optimal learning is acceptable, 1 signifies an optimal performing population is acceptable, and values between these limits signify an optimal performing population given the constraint of expected performance being a scalar portion of the expected optimal performance. We achieve an acceptable expected performance (as defined by the "Fitness Importance" parameter) by increasing the size of the "Elite" subpopulation and scaling the remaining subpopulations over the fixed population size. The specific implementation is explained in Section 4.3.

Finally, the new population is constructed from the subpopulations and the evolutionary process of evaluated and constructing new populations repeated. This evolutionary process allows continual (life long) optimisation and learning to occur simultaneously. At any point in time, the "Fitness Importance" may be varied to bias the system to either better performing or faster learning as acceptable.

The IDGP engine maintains a population structure of a 5 subpopulations, defined by a set of genetic operations as per Table 4.3. Let Q be the set of subpopulation types, noting that the number of subpopulation types, N_Q, need not be limited to that of Table 4.3.

$$Q = \{E, H, C, R, O, ...\} \tag{4.26}$$

Table 4.3: Expected Fitness for Specific Demographics

	Copy	Mutate	Crossover	Generate	Import
Elite	✔				
High Rank	✔	✔			
Child		✔	✔		
Random	✔	✔	✔	✔	
Other					✔

$\longleftarrow$ ..... $\longrightarrow$

Higher Expected Performance
Lower Learning Capability

Lower Expected Performance
Higher Learning Capability

Let the demographic sizes be N_E elite programs, N_H highly ranked programs with mutation, N_C children programs generated through crossover and mutation, N_R randomly generated programs and N_O "other" externally generated programs. In the absence of FI, the size of each subpopulation is constant, although typically $N_E \neq N_H \neq N_C \neq N_R \neq N_O$. Hence, the total number of programs in the population N_{pop}, within the IDGP framework is $\Gamma(\phi = 0) = N_E + N_H + N_C + N_R + N_O = \sum_{S \epsilon Q} N_S$

Each subpopulation will have an expected fitness $E(F_{subpop_E}), ..., E(F_{subpop_O})$ and an expected learning potential which we will investigate further in section 5.3. Thus the expected pool fitness can be expressed as,

$$E(F_{pool}(0, g)) = \sum_{S \epsilon Q} \frac{1}{N_S} E(F_{subpop_S}(0, g)) \tag{4.27}$$

Let Γ be the set comprised of the N_Γ unique population structures γ_x that can be generated from the various combinations of subpopulation sizes where (4.4) is satisfied.

$$\Gamma = [\gamma_1, \gamma_2, ...\gamma_{N_\Gamma}] \tag{4.28}$$

Let the population fitnesses given a specific population structure γ_x and specific instance

of a population p_i be $F_{pool}|p_i, \gamma_x$ Note that the subpopulations generated with the same population demographics but from different population instances will likely have different fitnesses. i.e. $F_{pool}|p_i, \gamma_x \not\equiv F_{pool}|p_j, \gamma_x$

Given the current population, p_i, it is possible to exhaustively compute the expected fitnesses for the entire set of possible population structures and identifying those which produce an expected performance improvement to satisfy ϕ_{accept}. This set is defined as,

$$\Upsilon(\phi_{accept}, p_i) = \{ \gamma_x \mid x \in \mathbb{N}, E(F_{pool}|p_i, \gamma_x) \geq E(F_{pool}(\phi_d, g)) \} \tag{4.29}$$

Within this set of population structures $\Upsilon(\phi_{accept}, p_i)$ that satisfy the performance constraint, there will be a structure(s) that provide optimal exploration (i.e. fastest evolution towards the optimal solution). Unfortunately, finding the optimal population structure that meets the performance constraint with maximal learning capability is an extremely challenging problem and beyond the scope of this research. The number of possible population structures, N_Γ, which must be assessed scales exponentially with increasing population size N_{pop} (depending on the number of subpopulations N_Q employed). Specifically,

$$N_\Gamma = \frac{(N_{pop} - 1)!}{(N_Q - 1)!((N_{pop} - 1) - (N_Q - 1))!} = \frac{(N_{pop} - 1)!}{(N_Q - 1)!(N_{pop} - N_Q)!} \tag{4.30}$$

Let us assume that a function $\Theta(\phi_{accept}, p_i)$ exists which generates the optimal learning population structure that meets the acceptable expected performance. $\mathcal{G}(\phi_{accept}, p_i)$ can then use this function to produce the appropriate subpopulation sizes that it must generate with its selection and genetic operations specific to each subpopulation type.

Given the difficulty in finding the optimal Γ, we return to a variant of our simple heuristic of biasing the population with elite programs. However, instead of directly manipulating the population, we manipulate the population structure to be biased by elites and attempt to redistribute the remaining structure as per an “optimal” learning population structure

that mimics a scaled version of the $\phi = 0$ pool structure. We term this function that generates populations according to this structure as $\mathcal{G}_{simple}(\phi_{accept}, p_i)$. $\mathcal{G}_{simple}(\phi_{accept}, p_i)$ subsequently employs $\Theta_{simple}(\phi_{accept})$ to generate a population structure with $1 \leq N_E \leq N_{pop}$ elite programs used to achieve the acceptable pool fitness (performance), and the remaining distribution of subpopulation types scaled and quantised in an iterative manner starting using a ceiling function. Note, we order the remaining subpopulations by highest expected fitness (in this case *highly ranked*) before iteratively applying the scaling and ceiling functions until we reach the subpopulation with the lowest expected fitness or until N_{pop} programs have been allocated. This introduces a bias to keep programs with higher predictable expected performance, and removes programs with low predictability in performance (lower expected performance). This bias tends to make the achieved performance improvement greater than the acceptable performance improvement. i.e. $\phi_{achieved} > \phi_{accept}$.

After all the programs within a population have been executed and their fitnesses calculated, IDGP then performs selection and genetic operations, $\mathcal{G}$, given ϕ_{accept} on p_i), on that population to generate the new population. By applying the genetic operation(s) $\mathcal{G}$ to p_i which generates a new population, p_j, which is also a member of $\mathcal{P}$.

$$p_j = \mathcal{G}(p_i), \quad p_i, p_j \in \mathcal{P} \tag{4.31}$$

Note that $\mathcal{G}(0, p_i)$ is the best available learning strategy, as it uses all the available resources towards exploration. Let k be the generation at which FI is applied, implying we desire to have the pool fitness improve by $\phi_{accept}[F_{elite}(0,k) - F_{pool}(0,k)]$ above what would have been the pool fitness without applying FI. To achieve the acceptable change in pool fitness $\mathcal{G}(\phi_{accept}, p_j)$ must generate an alternate population with an expected average fitness of that inEquation 4.8. A simple heuristic to achieve this is to replace a fraction (ϕ_{accept}) of the pool programs with elite programs. Thus the new population consists of ϕ_{accept} elite programs and $1 - \phi_{accept}$ non-elite programs, which we will call $pool^*$. Let us select which programs to replace such that the unaffected programs in $pool^*$ preserve the average program fitness of $F_{pool}(0,k)$. This generates a population with an expected

performance of $\phi_{accept}F_{elite}(0,k) + (1 - \phi_{accept})F_{pool}(0,k)$ which can be rearranged to (4.8) and hence delivers the minimum acceptable performance gain.

However, selecting which programs to replace based on preserving the pool fitness does not necessary provide optimal learning capability. To achieve good learning, it is important to maintain diversity in the population, which is implicitly achieved through the right combination of "subpopulations", each of which is defined by selection and genetic operations. This is however beyond the scope of this research and unnecessary to demonstrate that a desired average fitness can be achieved while learning by judiciously selecting the (sub)population composition.

4.5 Integrating FI into the IDGP Framework

We extend the IDGP Framework with FI as shown in Figure 4.7. Note that FI function can be computed locally or supplied externally. IDGP is currently implemented for memory and resource constrained low-power wireless sensor network devices. Each physical device locally runs a C-based GP-engine to perform in situ evolution, however the framework allows for programs to be shared wirelessly between neighbouring nodes (using the Island Model) which can aid evolution as well as allow distributed and cooperative evolution.

The user policy, which includes the initial program code (often random programs), fitness function, and fitness importance, serves as an input to this framework. IDGP then employs a traditional elitist mechanism in conjunction with genetic operations to provide genetic diversity in the population which facilitates learning. FI is then used by the IDGP engine to generate populations with an expected average fitness which meets a relative acceptable performance criteria. This is the responsibility of the population generator function as described in section 4.4. For ease of implementation, a computationally simple algorithm termed $\mathcal{G}_{simple}$ is employed and is described in the following section.

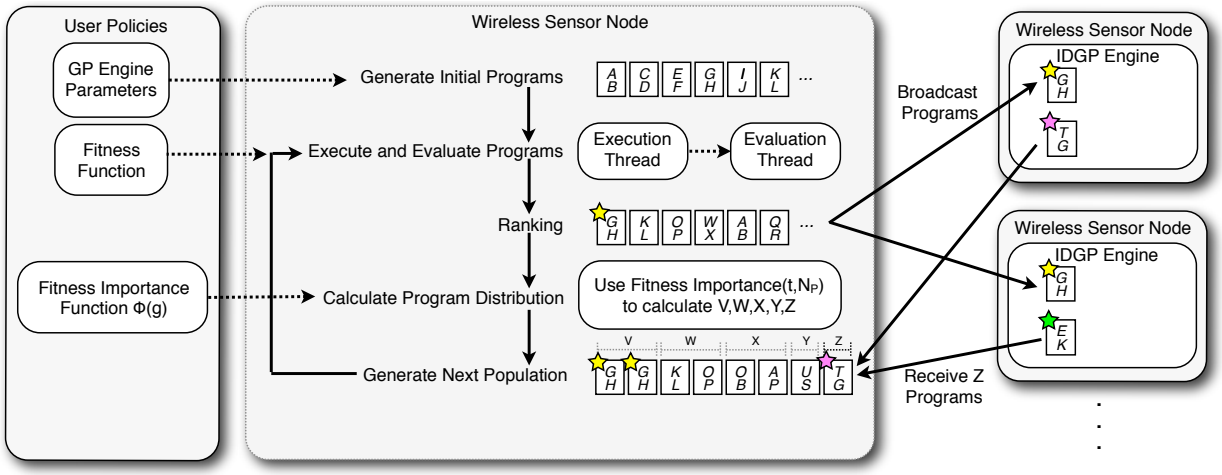

Figure 4.7: An overview of the In situ Genetic Programming (IDGP) framework extended to use the Fitness Importance heuristic. Note, the population generation is the same build order as shown in Figure 3.3.

4.5.1 Using $\mathcal{G}_{simple}$ to achieve desired performance

The number of possible population structures that could be employed scales exponentially with respect to the population size as described in Equation 4.30. Specifically for when $N_Q = 5$, the set of possible population structures, $N_\Gamma(x)$, scales with population size x as $N_\Gamma(x) = x(x-1)(x-2)(x-3)/4! = x^4 - 6x^3 + 11x^2 - 6x/24$.

Ideally, the expected fitness for all possible population structures would be computed, however this is unlikely to be practical for computationally constrained devices with any significant population size. To demonstrate FI however, one only needs to show that the desired average performance (AAF) can be achieved and that learning can still occur. Since the elite subpopulation has the highest expected performance, increases in this subpopulation will rapidly shift the entire population performance towards meeting the desired performance. The remaining subpopulations could then be distributed as per a typical 'optimal' learning distribution. An additional advantage of this approach is that is also computationally simple to generate and so we term this approach as $\mathcal{G}_{simple}$.

Due to the significant RAM limitations of the devices used for the demonstration, a

small population size $N_{pop} = 21$ is employed. Determining the optimal learning population distribution is outside of the scope of this thesis and not essential to demonstrating the functionality of using FI. As such, we will assume that the subpopulation sizes $\{N_E, N_H, N_C, N_R, N_O\}$ of $\{1, 3, 12, 2, 3\}$ respectively correspond to a reasonably good learning population. Let us also assume that the respective expected fitnesses for individuals of each population are $\{47, 45, 27, 5, 3\}$.

The "simple" subpopulation distribution generator function γ_{simple} creates subpopulation distributions by linearly increasing the number of elites from $\phi = 0$ to $\phi = 1$ and scaling the remaining subpopulation distributions. Table 4.4 illustrates the $\Theta_{simple}(\phi_{accept})$ population generator with N_{pop} unique discrete distributions corresponding to $1 \leq N_E \leq N_{pop}$. Since $\Theta_{simple}(\phi_{accept})$ employs elites until the average fitness is achieved, it follows that $\phi_{achieved} \geq \phi_{accept}$. However what the impact is to learning is unclear and will be investigated further in Chapter 5 and Chapter 6.

N_E	$\Theta_{simple}(\phi_{accept})$	$\phi_{achieved}$	ϕ_{accept}
1	EHHHCCCCCCCCCCCCRROOO	0.000	0.00
2	EEHHHCCCCCCCCCCCCRROO	0.095	0.05
3	EEEHHHCCCCCCCCCCCRROO	0.139	0.10
4	EEEEHHHCCCCCCCCCCRROO	0.182	0.15
5	EEEEEHHHCCCCCCCCCCRRO	0.277	0.20
6	EEEEEEHHHCCCCCCCCCRRO	0.320	0.25
7	EEEEEEEHHHCCCCCCCCRRO	0.364	0.30
8	EEEEEEEEHHCCCCCCCCRRO	0.368	0.35
9	EEEEEEEEEHHCCCCCCCCRO	0.459	0.40
10	EEEEEEEEEEHHCCCCCCCRO	0.502	0.45
11	EEEEEEEEEEEHHCCCCCCRO	0.545	0.50
12	EEEEEEEEEEEEHHCCCCCRO	0.589	0.55
13	EEEEEEEEEEEEEHHCCCCCR	0.684	0.60
14	EEEEEEEEEEEEEEHHCCCCR	0.727	0.65
15	EEEEEEEEEEEEEEEHCCCCR	0.732	0.70
16	EEEEEEEEEEEEEEEEHCCCR	0.775	0.75
17	EEEEEEEEEEEEEEEEEHCCC	0.866	0.80
18	EEEEEEEEEEEEEEEEEEHCC	0.909	0.85
19	EEEEEEEEEEEEEEEEEEEHC	0.952	0.90
20	EEEEEEEEEEEEEEEEEEEEH	0.996	0.95
21	EEEEEEEEEEEEEEEEEEEEE	1.000	1.00

Table 4.4: Subpopulation distributions generated using γ_{simple} linearly increasing numbers of elites, N_E, then iteratively scaling and quantising the remaining subpopulations

.

4.6 Discussion

This section discusses opportunities and challenges for using Fitness Importance with the IDGP framework for evolving logic *in situ* and online.

4.6.1 Slack: Excess Energy and Time

The paradigm of learning and performing highlights 2 important points. Firstly, "excess energy" and "excess time" must be present to facilitate non-critical periods where learning can occur. If the best known performance is always demanded of a system, then there is no opportunity for variation from the current best solution and hence learning cannot occur. The second important point is that performance is not the same as fitness. Performance is the perceived fitness over a period where the system behaviour matters to the perceiver.

For all problems we can make the very general claims that:

- a problem must come into existence at some point in time or is identified at some point in time
- once the problem is known it will take some finite amount of time to devise a solution
- once an acceptable solution is believed to have been found, it could be applied immediately or further time could be spent devising an even better solution

There will be a class of problems where it makes most sense to generate solutions by interacting with the world directly (i.e. *in situ*) and not via a simulation of the world (i.e. *in silico*). Therefore, regardless of whether a solution is devised offline or online, the real-world will be used for feedback during the generation of the solution. Providing the same solution generating mechanism is used, then the time to generate an acceptable solution should be the same for either approach. We will continue to employ evolutionary algorithms (though not evolutionary programming) as the solution generating mechanism. For the problems used in this analysis we can think of *in situ* evolution being equivalent to

requiring each evaluation taking a finite period of time and, most importantly, evaluating a solution affects the success rate of the system when FI is greater than zero.

Let us suppose that a problem presents itself at some specific time, g_0. For convenience we will quantise time into discrete periods, namely generations, corresponding to the period of evaluation of the population for the respective generation. Additionally, let g_i correspond to the point in time immediately after the completion of the evaluation of the ith generation. Hence the period over which the first generation is evaluated is $g = 1$ which occurs between g_0 and g_1.

Let us additionally assume that we are keen to deploy a solution when it becomes available. More specifically, that we have the flexibility to wait for an acceptable solution to be evolved but not so much as to be able to wait for evolution to an optimal solution. Let g_{start} be this point in time after an acceptable solution has become available. Similarly, let g_{finish} be the point in time when solving the problem is no longer of interest or the importance of performance is once again zero (i.e. $\Phi(g) = 0$ where $g \geqslant g_{finish}$).

Thus, for the given problem, we have a fixed period of utility from t_{start} and t_{finish} defined by when $\Phi > 0$. The importance of fitness during the period of utility will be lifted such that success over the time epoch of one generation requires the pool fitness to be between $F_{pool}(g_{start})$ and $F_{elite}(g_{start})$. This means that success is known to be achievable as it will less than the current known elite, but also not so trivial that any population can achieve success. The lower limit of $F_{pool}(g_{start})$ is a logical bound since in theory this will be the observed performance of the optimal learning population and so without *any* loss of learning potential, this level of performance can be achieved. Setting an acceptable performance target lower than this will tend to be trivial to achieve providing the fitness landscape remains constant.

For experiments where the FI function changes during the period of utility, the change will be limited to only one occurrence. Where the fitness landscape also changes, we will coincide the change of fitness importance to the same generation that the change in the fitness landscape occurred.

In summary, we wish illustrate scenarios where using the FI metaheuristic provides

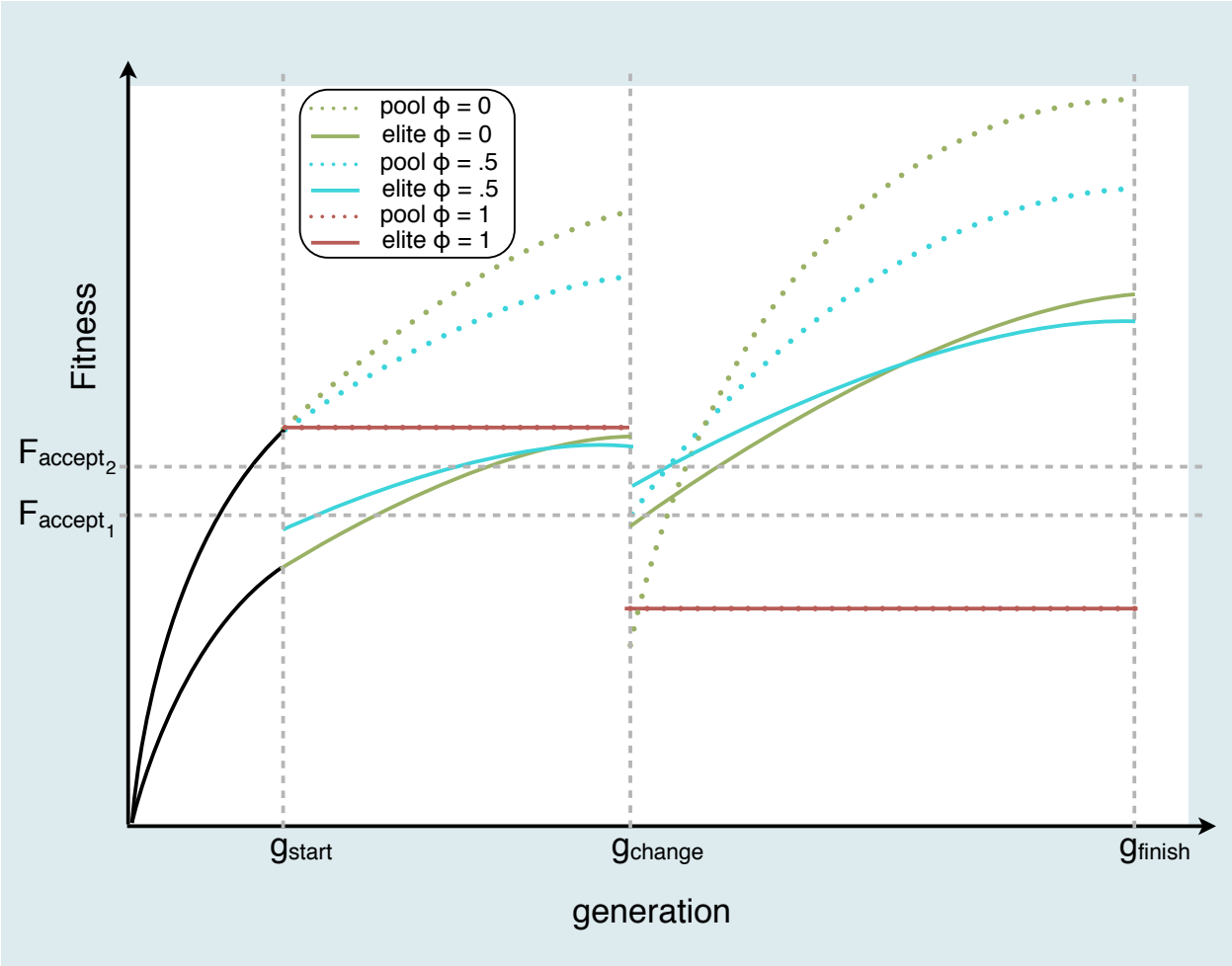

Figure 4.8: Hypothesis of how experiments will be evaluated. Show g_0, g_{start}, g_{finish}, change of landscape, F_{accept_1}, F_{accept_2}

benefit (in terms of success rate) over the offline and pure learning approaches. Practically, we need to set a start time, even if the timing is somewhat arbitrary. In an attempt to construct a fair comparison, we will employ only one method for solution finding, namely genetic evolution, and allow all approaches to evolve for the same number of generations before the importance of fitness is increased from 0. This will occur at generation, g_{start}, at which the importance will be lifted such that success requires a pool fitness that lies between the $F_{pool}(g_{start})$ and $F_{elite}(g_{start})$. Furthermore, g_{start} will be set prior to convergence (i.e. F_{elite} nearing $F_{optimal}$, if known) such that online learning can provide benefit.

4.6.2 Practicalities of Online Learning

Practical implementations of online learning will need to consider various real world constraints. For an acceptable solution we need to know:

- how to prevent the system from entering a state where it is impossible to evolve to an acceptable solution
- whether it will converge to an acceptable behaviour within an acceptable period
- whether it is possible to evolve an acceptable solution given the IDGP configuration constraints such as memory footprint, computational ability, available energy and syntactic richness of the instruction set.
- that it will not perform any actions outside of what is acceptable (safe evolution)

In more general applications it will be important to design node hardware and software architectures such that IDGP programs are executed and evaluated in isolation from core functionality to ensure that evolved programs cannot hog critical resources or cause undesirable actuations. This raises the issue of which problems are suitable targets for IDGP implementation as a topic for further research. In some cases it may be possible to impose constraints during evolution to ensure solutions stay within safe behaviour - in a sense “sandboxed learning”. However setting constraints inherently reduces the searchable solution space, meaning that better solutions may exist yet are never evaluated.

The general intuition from the “No Free Lunch Theorem” [153] leads us to believe that it is unlikely that one learning (optimisation) technique will be better than all others for all online learning problems.

The “No Free Lunch Theorem” [153] states that no solution finding mechanism is better than others over all problems and it follows that IDGP is a solution for a niche of problems. We recommend the designer to estimate whether an alternative approach, such as human-crafted logic, could produce acceptable logic more efficiently.

IDGP is well suited to distributed systems with complex multi-objective problems with no well known solution, or where adaptivity of logic is likely to be necessary during the life

of the system. FI is useful to problem scenarios that exhibit excess energy and time which can be exploited to learn with reduced or without penalty. We expect the combination of IDGP and FI to be useful to many future pervasive computing scenarios.

4.6.3 Challenges of In situ Evolution

The evaluation of logic *in situ* avoids reliance on synthetic (simulated) models of the environment, and there is no better representation of the environment than the environment itself. Not only does this prevent the evolution of brittle logic by exploiting artefacts in the simulation, it also means that the system does not need to perform any simulation at all which is a great benefit for resource-constrained devices. However, a significant disadvantage of this approach is that there is only a single shared environment for all programs across all motes. It is impossible to reset the environment back to *exactly* the same initial conditions prior to the evaluation of every program. Hence, each program may leave a “footprint” which may help or hinder other programs, generating a credit assignment problem which can slow or even prevent convergence. Even more important is whether the environment could be changed in a way that permanently prevents the acceptable objective to be fulfilled. Unlike offline evolution, one cannot go back in time and trial another optimisation trajectory if the current one has failed. These are inherent drawbacks of online learning. FI provides a means to control the learning/performing balance within a dynamic physical environment.

4.7 Conclusion

This chapter described the phenomena that the importance of fitness can vary over time i.e. something can be “fit” even when that “fitness” is not appreciated as performance. “Fitness Importance” (FI) was devised as the metaheuristic for decoupling the fitness function from desired performance for online learning. This allows various levels of average system performance to be minimally met, while allowing ongoing evolution towards finding higher fitness solutions.

An analytical model of how FI impacts learning and performing was presented and then finally demonstrated through an implementation of a simple population biasing algorithm called γ_{simple}. γ_{simple} manipulates the number of elites evaluated per generation in order to meet an acceptable performance as specified by Fitness Importance Function. This demonstrated the first implementation and example of a mechanism for responding to FI. Future research on more complex implementations are recommended in order to achieve greater learning capacity and/or more reliably meeting the minimal performance criteria.

5

Constant FI Parameter Management Strategies

This chapter describes and analyses experiments where FI is set to a constant value at various times during evolution. The experiments are performed on motes deployed in real world environments (as opposed to simulation) to gain an intuition of how learning and performance are affected during in situ evolution on this class of device.

5.1 Introduction

The problem scenario where learning is desirable whilst simultaneously achieving acceptable average fitness has previously been enumerated as Table 4.1Learning Problem Type 5 and characteristics of this problem type are discussed in Section 4.3.3. Fitness Importance (FI) will be varied, and the FI control parameter ϕ will be set to a non-zero constant during the evolutionary run and the effects on meeting acceptable fitness analysed. Performance of the strategies (detailed in Section 6.2.5) is measured via the success rate, $P(g)$ over a finite period, which is essentially is the portion of generations that the average fitness exceeded a required fitness score (c.f. Equation 4.24). In addition to the achieved performance during the interval, the elite fitness of the final generation is considered as a measure towards the system's potential to meet future increases in acceptable fitness requirements.

5.2 Instantaneous Performance Control

With the definition of FI and a mechanism for generating populations with the expected acceptable performance, we now demonstrate the application of FI on a simple problem evolving under real-world environmental conditions. The immediate impact of FI is analysed and discussed.

5.2.1 Experimental Setup

We demonstrate the effects of FI in this section with 3 experiments using a time-varying, light-sensing and light-actuating problem referred to as the "Blink3" problem which has been studied in detail in Section 3.4 and published in [137].

An empirical study using 3.5 million randomly generated[1] programs found a maximum fitness of only 82.22% of the optimal solution fitness and the average fitness of a random population (pool fitness) of 4.15% of the optimal solution fitness. The "Blink3" problem is

[1]Not necessarily unique, however the likelihood of the same program being generated randomly is extremely low ($1.0E^{-34}$) due to the multiple choices of instructions available at each step during construction.

subject to a number of real-world conditions such as local, global and temporal variations in environmental conditions as well as noise on the sensor reading. This was a deliberate decision to evaluate online in situ evolution under real-world conditions.

A baseline of learning and performing was established by setting FI to 0 for the entire evolution up to 500 generations with this repeat for 50 runs. Each experiment where the value of FI was altered had 20 repeated runs and so average elite and pool performances at each generation were calculated from 20 points.

5.2.2 Analysis

Because of changes in the surrounding environment, the achieved pool fitness at the next generation, $k+1$, after ϕ_d is applied (at generation k), may deviate from $F_{pool}(\phi_d, k+1)$. Extending equation (4.9) to the next generation $k+1$, the achieved fitness importance, $\phi_{achieved}(k+1)$, that accounts for such deviations is:

$$\phi_{achieved}(k+1) = \frac{F_{pool}(\phi_d, k+1) - F_{pool}(k)}{F_{elite}(k) - F_{pool}(k)} \tag{5.1}$$

Ideally the difference between ϕ_d and $\phi_{achieved}(k+1)$ is kept minimal, such that maximum diversity is maintained while achieving an acceptable average fitness. Hence, when $\phi_{accept} = 0.5$ at generation k, it indicates the desire for the expected average fitness to lie exactly half way between the average fitness of a standard GA, $F_{pool}(0, k+1)$, and the average fitness of using only the current elite solution $F_{elite}(k+1)$.

In practice, the application of FI is best suited to occur at the end of the evaluation of a generation of programs so that the knowledge of distribution of fitnesses for the subpopulations can be used to generate a new population with an acceptable expected performance. Within the current implementation, there is only one population and so if FI is used to alter the structure of the population, then $F_{pool}(\phi_{accept}, g)$ can be calculated, however $F_{pool}(0, g)$ cannot, unless of course $\phi_{accept} = 0$.

Using $Eq.5.1$ and given that $E(F_j)|_{\phi=0}$ and $E(F_j)|_{\phi=1}$ can be estimated based on the

generation prior to FI being applied, then the expected fitness becomes:

$$E(F_j)|_{\phi_j} = \sum_{k=1}^{N_{E_j}} E(f_{Ek_j}) + \sum_{k=1}^{N_{H_j}} E(f_{Hk_j}) + \sum_{k=1}^{N_{C_j}} E(f_{Ck_j}) + \sum_{k=1}^{N_{R_j}} E(f_{Rk_j}) + \sum_{k=1}^{N_{O_j}} E(f_{Ok_j}) \tag{5.2}$$

where f_{Sn_j} is the fitness from the nth highest ranked program in the subpopulation S for generation j.

5.2.3 Impact on Learning and Performance

FI extends the IDGP framework (see Figure 3.3) to balance performance and learning during the life of the nodes by altering the number of elites in the program. Experiments we performed, such as Figure 5.2, to investigate various aspects and effects of the FI parameter have also highlighted a number open questions for online learning systems.

In many real world systems the importance of the system's performance varies over time. In Chapter 4 we introduced the "Fitness Importance" (FI) parameter $\Phi(t)$ as a time varying function to reflect the acceptable balance between system performance (as defined by the population fitness) and learning potential (as defined by improvement of the population elite) at any point in time. Typically, $\Phi(t)$ will be application specific, possibly not known in advance, and even potentially a function of local events and sensed data. The practical use of FI is as an input to the subpopulation distribution generator γ which attempts to bias the distribution of programs in the population in order to achieve a pool performance which is ϕ of the way between the pool fitness of a population designed to achieve optimal learning and a pool fitness composed entirely of the elite program (optimal performing based on current knowledge). The improved system performance however comes at a price of decreased learning potential due to a reduced pool diversity.

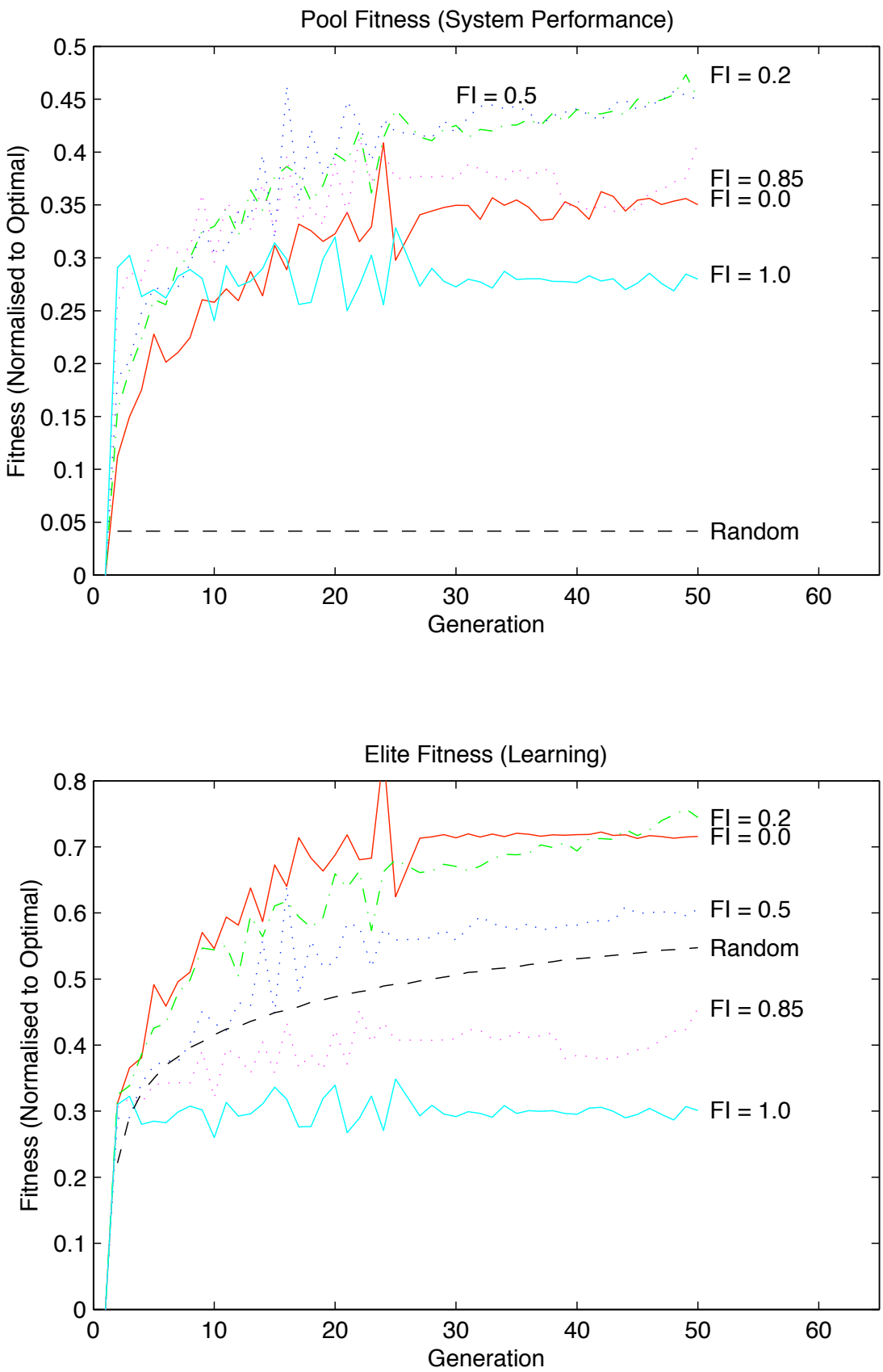

Figure 5.1: Evolution of system performance (pool fitnesses) and learning capability (elite fitnesses) with fixed FI.

To explore the effects of FI applied at the start of an evolutionary run, we conducted an experiment with various values of FI set at generation 0. Figure 5.1 plots the results of these experiments. We can observe that the plot $\Phi(g) = 0$ exhibits the fastest learning, yet the slowest improvement in pool performance. Conversely, the $\Phi(t) = 1$ population distribution outperforms all other distributions immediately but never improves since no learning can occur. Both of these observations confirm the conjectures in section 4.1.3. However, the marginal difference in learning rates between $\Phi(t) = 0$ and $\Phi(t) = 0.2$ indicates that the population distribution for $\Phi(t) = 0$ may not necessarily be the optimal learning configuration.

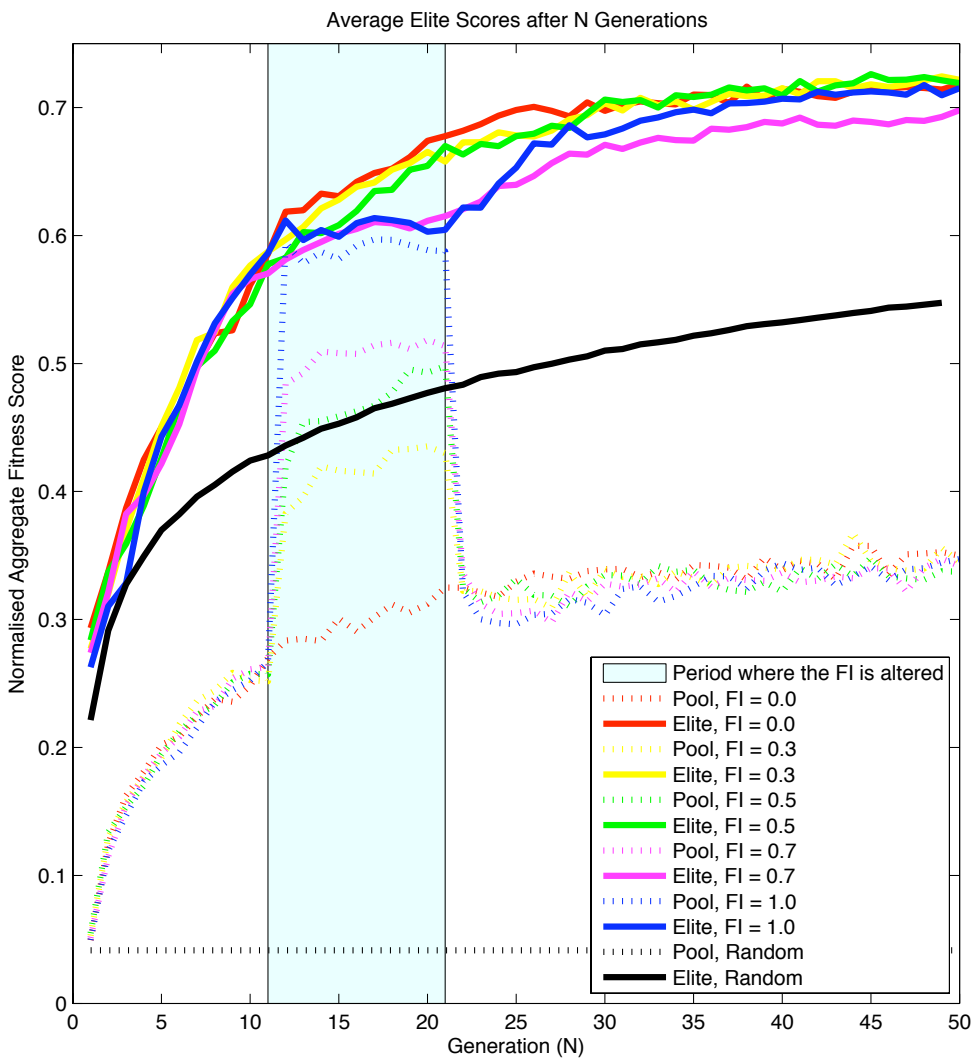

Figure 5.2: Cooperative evolution of system performance (pool fitnesses) and learning capability (elite fitnesses) with various values of FI.

Figure 5.2 shows the cooperative evolution of 8 nodes which then all simultaneously subject to the same FI. The resulting elite and pool curves are averaged producing the lines shown. Random search (best found so far) is shown to provide a baseline. Since a random pool is generated every generation, $F_{pool}(random, g)$ is close to constant. By generation 11, $FI > 0.5$ can produce a pool fitness greater than the best random search solution for the same number of evaluations. i.e.

$$F_{pool}(0.5, g) > F_{elite}(random, g), \quad g > 11 \tag{5.3}$$

From Figure 5.2 we see that when FI=0, the average elite fitness increases from an initial 25% of the optimal solution fitness to about 70%. The average pool fitness increases from an initial 4% to about 37%. This clearly performs outperforms random search for an equivalent number of program evaluations which is also provided as a reference.

To understand the impact varying FI has on learning, we allow the nodes to evolve for 11 generations with $\phi_{accept} = 0.0$ then change FI to another value for 10 generations, before returning FI back to 0 for the remaining generations. We use $\phi_{accept} = 0.3, 0.5, 0.7, 1.0$ as the alternate values of FI for the period between generations 11 and 21 as the input into the subpopulation distribution function γ.

Figure 5.2 shows the aggregate elite and pool fitnesses for each run where ϕ_{accept} was altered between generations 11 and 21. As expected, the pool fitness immediately improves in response to the acceptable FI. The actual immediate performance improvement $\phi_{achieved}$ achieved by taking the ratio of the pool fitness at generation 12 and the difference between the elite fitness at generation 11 and the pool fitness at generation 12. $\phi_{achieved}$ is calculated and shown in Table 5.1. Because the system is in a period of rapid learning, the pool fitness is also rises. Hence even when $\phi_{accept} = 0$, the pool still improves yielding an equivalent $\phi_{achieved}$ of 0.043. Since the current subpopulation distribution generator (γ) does not take into consideration the learning occurring within the current population, the resulting subpopulation distributions typically generate higher than expected performance (evidenced by the positive error between $\phi_{achieved}$ and ϕ_{accept}).

The pool gain over any period is a complex function of the population distribution and

the optimisation trajectory through the search space.

To illustrate this, imagine we have identical 2 systems, A and B, in exactly identical states. The pool size is 10 programs with average program fitness of 50% of optimal. We now change the population distribution of system A to E X X X X X X X X X and the other (system B) to E E E E E E E E E X where E is reserved for elite programs and X is reserved for a program that explore the solution space (whether it be mutation, crossover, random or some other method). Let us say that the probability of X being a solution that is 100% fit is 90%, however the average fitness of X remains 50%. System A will almost certainly (99.99999%) find the optimal solution and therefore the elite will be 100%, but the pool fitness will be 55% (an improvement of 5%). System B will have a 90% chance of finding the optimal solution. If it does, then the pool fitness will rise to 95% (an improvement of 45%). In this situation, system B may be the more favourable configuration since there is a high chance of the system becoming near optimal. However, if we change the probability of X being the optimal solution to say 1%, then system A has a 8.64% chance of finding an optimal solution while system B has only a 1% chance. In this situation one might prefer an So here the 8.64% chance of a system improvement of 5% over a 1% chance of 45% improvement.

Unfortunately, the probability of finding solutions is not normally available in advance, even knowledge of what the optimal fitness is, is often not known. However, it is clear that varying the population distribution based on FI impacts on the optimisation trajectory which in turn affects the evolution of the system (performance and learning) which will likely feedback into the FI.

The graph (in Figure 5.2) of FI=1 during generations 11 to 21 is an extreme example which highlights both the benefit and disadvantage of non-zero FI. The system (pool) performance more than doubles as a result of the application of FI, improving from approximately 26% (OSF) at generation 11 to 59% (OSF) at generation 12. However, at generation 21, the pool fitness has remained constant since no learning is occurring, while lesser FI allow the pool fitness to improve throughout the same period. For example the pool fitness for FI=0.5 improves by nearly 8% in the same period. If this learning trend was linear (although it clearly isn't in this case), then the FI=0.5 pool performance would

eventually exceed the pool performance for FI=0 after about another 10 generations or so. In that situation, the learning configuration has the advantage of continual optimisation while performing better than the system with a fixed FI of 1. This demonstrates that worse performance in the near term allows learning which results in better performance in the long term. Hence, fixing FI high for an extended period of time is not recommended.

Interestingly, the subpopulation distribution generated by FI = 0.5 for generations 11-21 provides a higher gain in the elite performance than when FI is set to 0, indicating that the subpopulation distribution for FI=0 may not yield optimal learning during this period. However, the learning rate in this period is influenced by a number of factors. One important factor is the current elite performance, which is actually higher for FI=0 than FI=0.5 throughout the period. In essence the FI=0 curve is ahead of the FI=0.5 curve and given the typical tapering of the curve, one would expect the slope to be less when the elite fitness is higher. The standard error of the mean during this period is typically about 2% of the optimal solution fitness during this period. The elite gain for FI=1 is practically zero as expected.

ϕ_{accept}	$\phi_{achieved}$	$\phi_{achieved} - \phi_{accept}$	Pool Gain %	Elite Gain %
0.0	0.043	0.043	4.1	5.9
0.3	0.391	0.091	4.7	6.1
0.5	0.507	0.007	7.8	8.7
0.7	0.719	0.019	3.0	3.4
1.0	1.025	0.025	-0.7	-0.7

Table 5.1: Note gains are represented as percentages of the optimal solution fitness during the period when the FI was altered.

As expected, the system fitness increases as better programs are found and the genetic material within the population improves. This can be seen in Figure 5.3 during the initial period where $\phi_{accept} = 0$ (and the packet index is less than 410).

5.2.4 Discussion

In scenarios where immediate performance improvement is required, we have demonstrated that FI can be used to deliver an instantaneous performance gain. However it

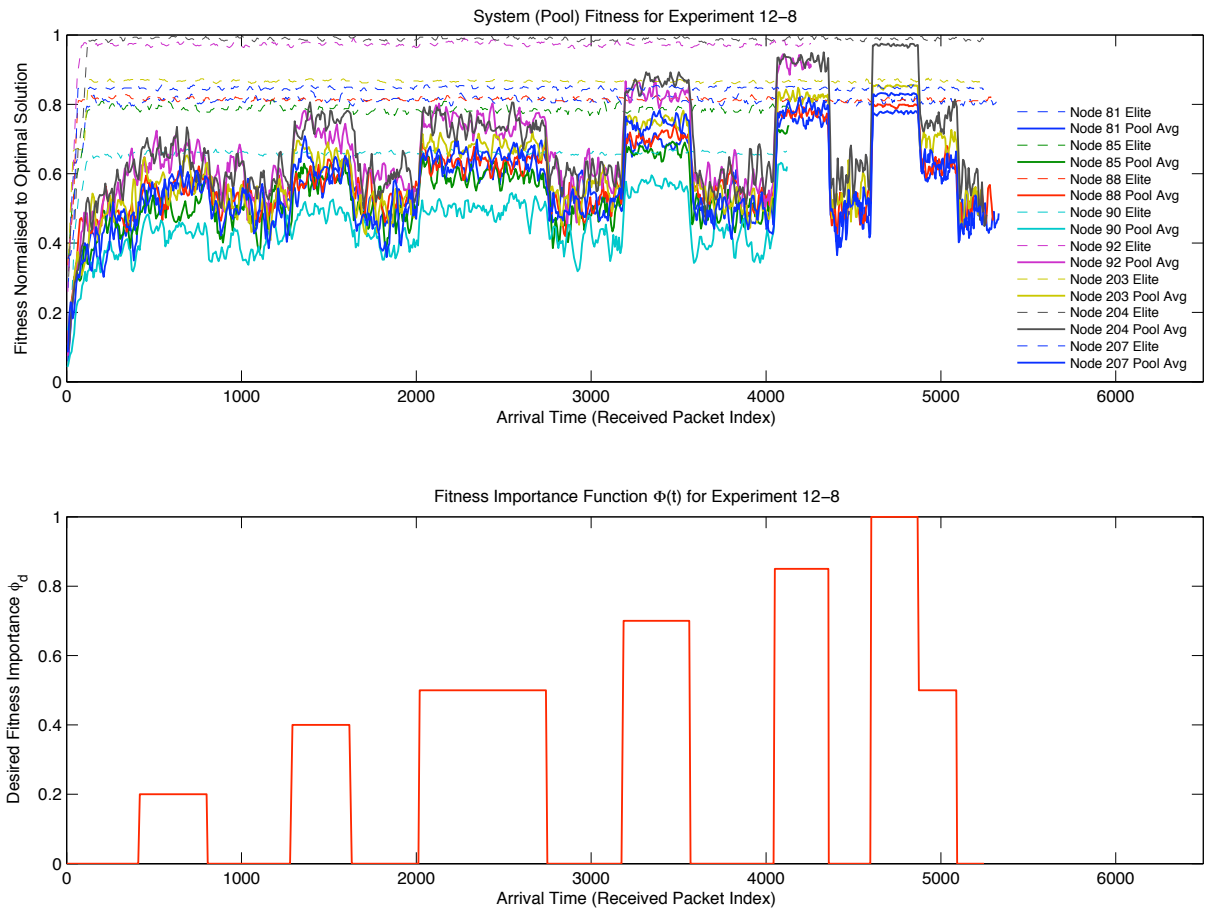

Figure 5.3: As the Fitness Importance is varied, the performance of the nodes, as measured by the average normalised fitnesses of all programs within a generation, responds accordingly. The change in acceptable performance is produced by employing a population distribution generated by the simple γ function (refer to Table 4.4).

clear that setting $\phi_{accept} > 0$ has a negative effect is on learning capability which would impact on long term performance and is the focus of the next section.

The results also showed that the range of $\phi_{achieved}$ can go beyond the ϕ_{accept} range of $0 \leq \phi_{achieved} \leq 1$. We attribute this deviation to changes in the environment which can be more or less favourable to the current logic. Section 4.6.3 elaborates on this issue.

The negative impact on average fitness could be interpreted as a result of applying a poorly chosen ϕ value that does not sensibly balance the exploration-exploitation into the future.

5.3 Long Term Performance Effects

The previous section has described the use of FI in delivering an acceptable instantaneous performance gain, highlighting the constrained evolution scenario that requires a minimum performance guarantee. This greedy approach for providing an immediate step increase in performance may come at the cost of reduced long-term performance. In contrast with constrained learning applications, maximum performance applications need to achieve the best aggregate performance over a longer time window into the future. The approach in section 5.2 represents a special case of performance maximisation over a window of a single time step into the future. This section presents the generalised analysis for applying FI to maximise the expected performance across any future time window.

This section is organised as follows. Section 4.3.4.1 defines the average pool fitness across multiple generations as a performance metric for a future period. Section 4.3 analytically derives FI and future pool performance as a function of the current elite fitness, pool fitness, and the pool fitness learning rate. With this analysis in place, Section 5.3.2 empirically explores these relationships through extensive experimental evaluation. The results show the effects on Normalised Average Performance (NAP) when applying various FI values at various generations. This reveals the best choice for the value of FI and when it should be applied are not deterministic for individual runs. While individual runs exhibit variations, there is a clear average long term trend. Section 5.3.2 demonstrates an example predictor function which can be used to select an appropriate value of FI online to maximise NAP over a bounded future period with high probability.

5.3.1 Empirical Evaluation

Because FI is a metaheuristic that relays the importance of instantaneous performance it is hard to quantify whether a different strategy would result in better lifetime performance.

We now use FI to maximise $NAP(k, \phi)$ over 500 generations. In other words, the application requirement is to learn a solution and to have the best possible performance over the entire period of 500 generations.

Equation 4.23 can be partitioned into the period prior to ϕ being applied and the period after ϕ has been applied as follows:

$$\mathrm{NAP}(i,n) = \frac{1}{k}\sum_{g=1}^{k} F_{pool}(0,g) + \frac{1}{n}\sum_{g=k+1}^{n} F_{pool}(\Phi(g),g) \tag{5.4}$$

where $k + n = 500$. This equation clearly shows that F_{NAP} is a function of ϕ, k, $F_{pool}(0,k)$ and $F_{elite}(0,k)$. In order to reduce this equation into a two-variable optimisation problem, we use the results from our 1-month long experiment. Figure 5.4 show the averages for $F_{pool}(k)$ and $F_{elite}(k)$ for all the runs. We use these averages in equation 5.4 in Matlab to find the optimal k and ϕ for this problem. We seek to find the values for ϕ and k (when to apply FI) over 500 generations which maximise Normalised Average Performance (NAP).

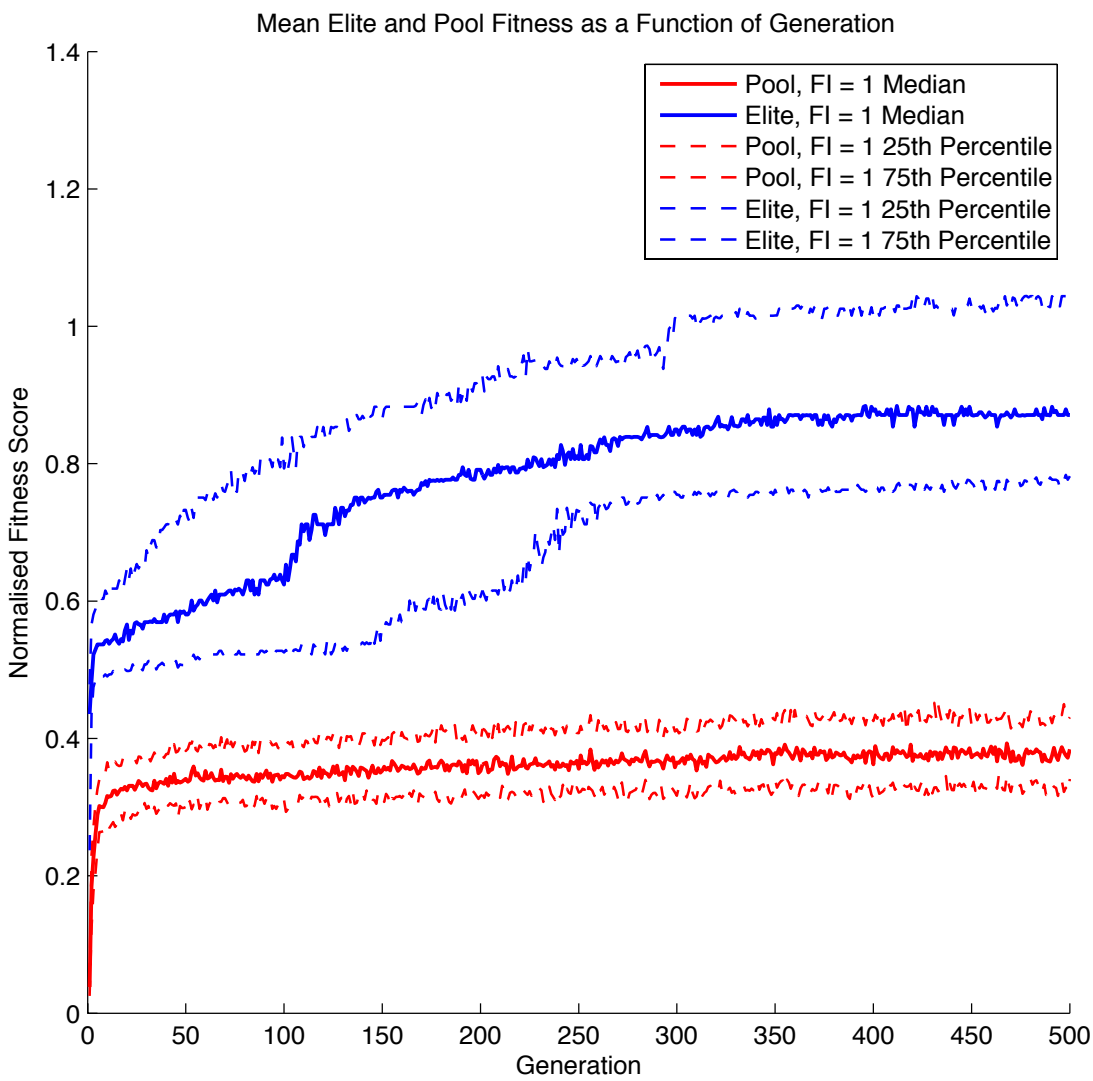

Figure 5.4: Average elite and pool fitness as a function of generation. The 25th and 75th percentiles are also shown.

Figure 5.5 shows the achieved NAP for various values of ϕ and k with the standard deviations shown in Figure 5.6. As expected, there is a general trend of higher NAP as ϕ and k increase as a result of exploiting learnt logic. Due to the relative simplicity of the Blink-3 problem, applying high values of ϕ at early generations also yield high NAP, where the bulk of learning appears to occur in the first few generations.

To further explore the learning potential, we also investigate the variance of the elite program fitness as a proxy for the learning potential. The combination of $F_{elite}(0, g)$ and

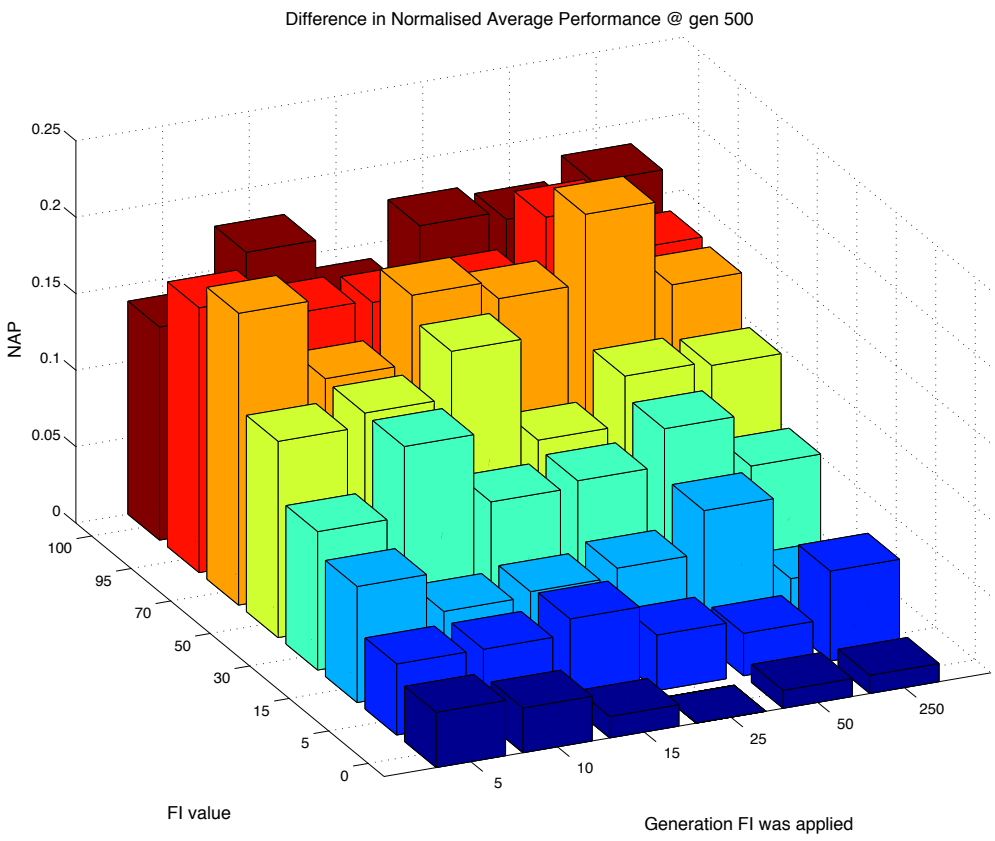

Figure 5.5: A snapshot of the Normalised Average Performance (NAP) after generation 500. Higher NAP is achieved as a result of exploiting learnt logic later in the evolution (after good solutions have been discovered) but not so late in the run that it cannot exploit the good solutions over many generations. As such, there is a "sweet spot" for values of ϕ and k which are unlikely to be knowable in advance.

its variance significantly influence the NAP obtained. By definition, the elite variance concerns the fitness of the program that could be exploited. A higher variance in the elite fitness implies that the expected performance level (NAP value) is less predictable. Thus, exploiting too early can be risky.

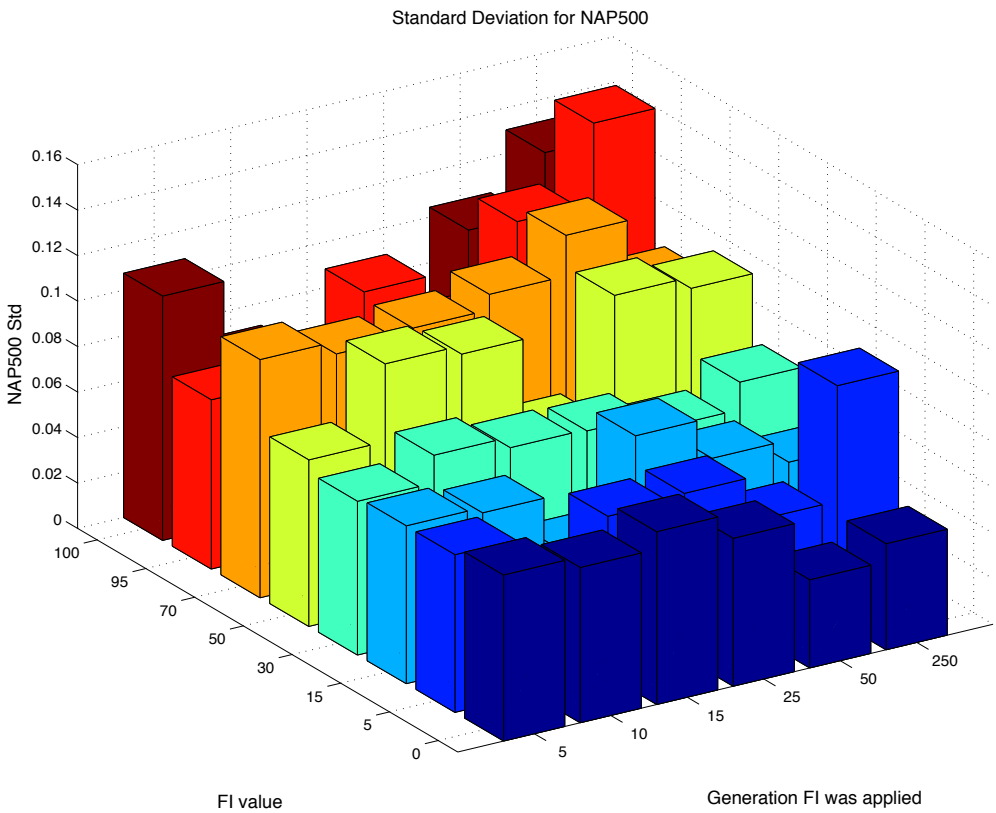

Figure 5.6: A snapshot of the standard deviation of NAP after generation 500. There is a trend indicating that there is greater variability in the fitnesses achieved with higher values of FI being applied. This can be interpreted as a “risky” strategy in that you lock in exploitation whatever was learnt at a particular time (which may or may not be high fitness).

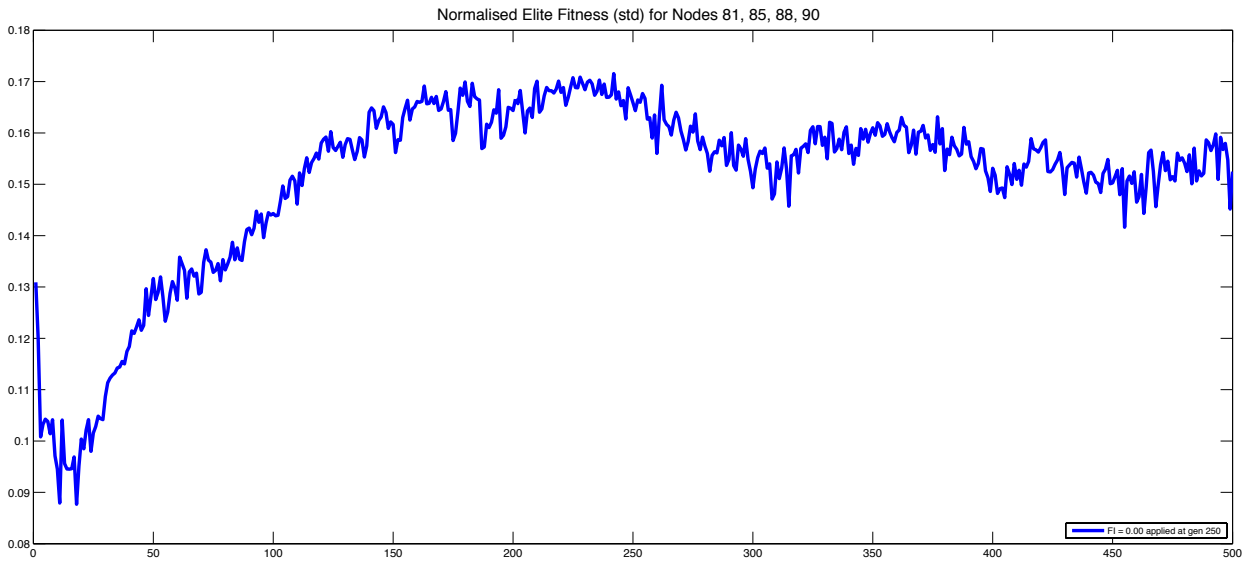

Figure 5.7: Standard deviation (N=120) of the Elite fitnesses shows how diverse the learning rate is between evolutionary runs.

Training Error	Weights	R^2 (σ)	$\bar{error}$ (σ)
$ABS(\sum_{g=k+1}^{500-k} error)$	1	0.265 (0.24)	28.94 (26.60)
$ABS(\sum_{g=k+1}^{500-k} error)$	$\frac{1}{g}$:$\frac{1}{g}$:1	0.265 (0.24)	22.33 (22.05)
$ABS(\sum_{g=k+1}^{500-k} \frac{error}{F_{pool}(g)})$	1	0.265 (0.24)	43.52 (37.07)
		0.265 (0.24)	28.17 (27.65)

Table 5.2: Goodness of fit for log curve prediction

Our results in Figure 5.7 show that the variance of the elite program fitness increases with increasing ϕ and k. Note that we generate this graph was generated from 120 runs where ϕ=0, providing high confidence in the repeatability of the results. The standard deviation for normalised elite fitness with $\phi = 0$ dips from an initial random population, which is the result of early learning typically producing similar elite fitnesses by generations 10-20. Thus exploiting from here on will give similar long term outcomes (thus lower NAP500 variance). After this dip, the variance in elite fitnesses ruses due to stochastic processes allowing some runs to learn quickly whilst other runs learn little.

5.3.2 Setting FI Online

The previous section has explored empirically how to select the fitness importance ϕ and the generation at which to apply it k by retrospectively observing experimental data. In this section, we investigate if and how ϕ and k can be set online during the evolutionary process.

We can say how well our prediction function works (measured by curve fit error against a post-processed curve or final Normalised Average Performance, NAP, error) for every (8x6x20=960) run and how well it compares to a post-processed curve fit. Table A.1 shows the goodness of fit for log and exponential curves with various error weighting schemes as predictors of fitness given the fitness information prior to FI being applied. The best fit curve (i.e. least error across all k and ϕ combinations for predicting fitness values after FI was applied) is obtained by the exponential curve with uniform weighting

of error.

For our problem, individual runs exhibit discrete fitness improvements depending on the initial population and evolution trajectory so far. However, the average of many runs tends to produce a relatively smooth asymptotic curve as illustrated in Figure 5.4. This curve form is typical of many convergence curves [75], and can be expressed well by a number of functions. We chose to investigate curves of the forms $a\log(bg)+c$ and $a10^{-\frac{1}{bg}}+c$. $y = -\frac{a}{\sqrt{bg+c}} + d$ could also have been used for example.

For most real world problems the fitness curves will not be known a priori. However, knowledge of how the $pool$ and $elite$ fitnesses are likely to progress is essential for the appropriate choice of FI if FI is to be applied for a number of generations. While absolute knowledge of the optimisation trajectory is not possible for stochastic evolution in unconstrained environments, we wish to test if it is possible to predict how the fitnesses will evolve during evolution given the fitness values prior to when FI is to be applied.

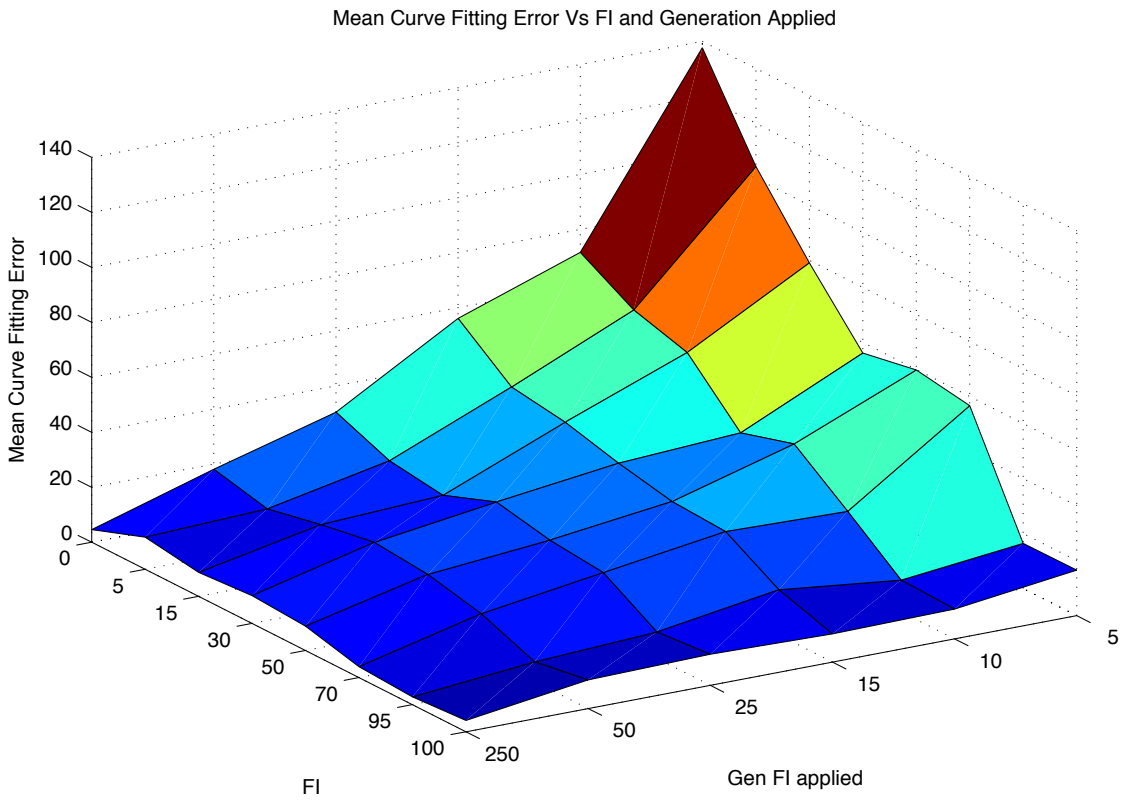

Figure 5.8: The average curve fitting error for a sweep of FI values applied at various generations.

Figure 5.8 shows the average curve fitting error for the 20 experiments at each (ϕ, g) combination. This figure shows 2 significant trends. The first is that there is less error for the prediction curves that are calculated later (higher generation). This is most likely due better curve fitting resulting from fitting against more data points, and secondly that there is less learning opportunity available, so the variation in the final result will be lower.

The second trend is that the higher the FI, the more predictable the future fitness curves are. Again, the learning potential has decreased (though for different reasons from the first trend) which reduces the variability of the final fitness.

Using a typical curve fit function, we aim to capture the evolutionary trajectory so far and to project the expected trajectory into future generations by running a series of simulation experiments. For each defined by (ϕ, k), we curve fit for both elite and pool fitness curves using an exponential function form [2] $F_{0,pool} = a.10^{-\frac{b}{g}} + c$ in order to obtain the parameters $a, b, andc$, where $1 < g <= k$. We then extend the best fit (least error) curves into the future up to $g = 500$ and calculate the expected $\mathrm{NAP}(k+1, k+n)$ where $k+n = 500$. Figure 5.9 shows the NAP results for our predictions of 20 repeated experiments of applying FI ($\phi \in \{0, 0.05, 0.15, 0.4, 0.5, 0.7, 0.95, 1\}$) at a range of generations ($k \in \{5, 10, 15, 25, 50, 250\}$). The measured average NAP across the 20 experiments achieved at each (ϕ, k) is shown by the squares. Notably in the top 3 plots, where FI is applied early, the maximum post-FI NAP ($\mathrm{NAP}(k, 500 - k)$) is not achieved by merely exploiting the elite solution solely, but through a combination of learning and exploiting. Learning potential decreases quickly due the rapid initial learning which means that by generation 25, exploiting the current elite with $\phi = 1$ becomes a good strategy for maximising NAP from that point.

The blue circles show the predicted average NAP at each (ϕ, k) for each of the curves in which the actual result of NAP was measured. Note that for each k there are 160 runs with $\phi = 0$ leading up to this generation after which 8 different values for ϕ are tested in 20 experiments. The green diamonds are the result of taking the 160 optimisation trajectories for $F_{pool}(0, g)$ where $g <= k$ and projecting them forward to estimate $F_{pool}(\phi_{accept}, g)$ where $k < g < 500$, which is then used to calculate the average predicted $\mathrm{NAP}_k^{500}(k)$. The average provides a better indication of the average predicted NAP, however does not have ground

[2] (and similarly for $F_{elite}(0, g)$)

truth to be compared against. It can be seen that the predictive power for high ϕ is good which is expected as error is caused by the inability to predict the learning rate. Thus as high ϕ reduces the learning capability, it will improve the predictability. When FI is applied later in the evolutionary run, more accurate prediction is achieved for 2 reasons: (1) there are more data points available which enhances the curve fitting; and (2) the fitness is typically higher which implies the learning capacity is lower, which also reduces uncertainty in prediction.

While predicting Normalised Average Performance (NAP) into the future would be ideal for setting ϕ online, it is also relevant to look at how the predictor ranks the considered ϕ values relative to the ground truth. If the predictor function does not correctly forecast the NAP, yet successfully determines one value of ϕ should perform better than all the others, then the online algorithm can use this value of ϕ with a high likelihood that it is the better choice.

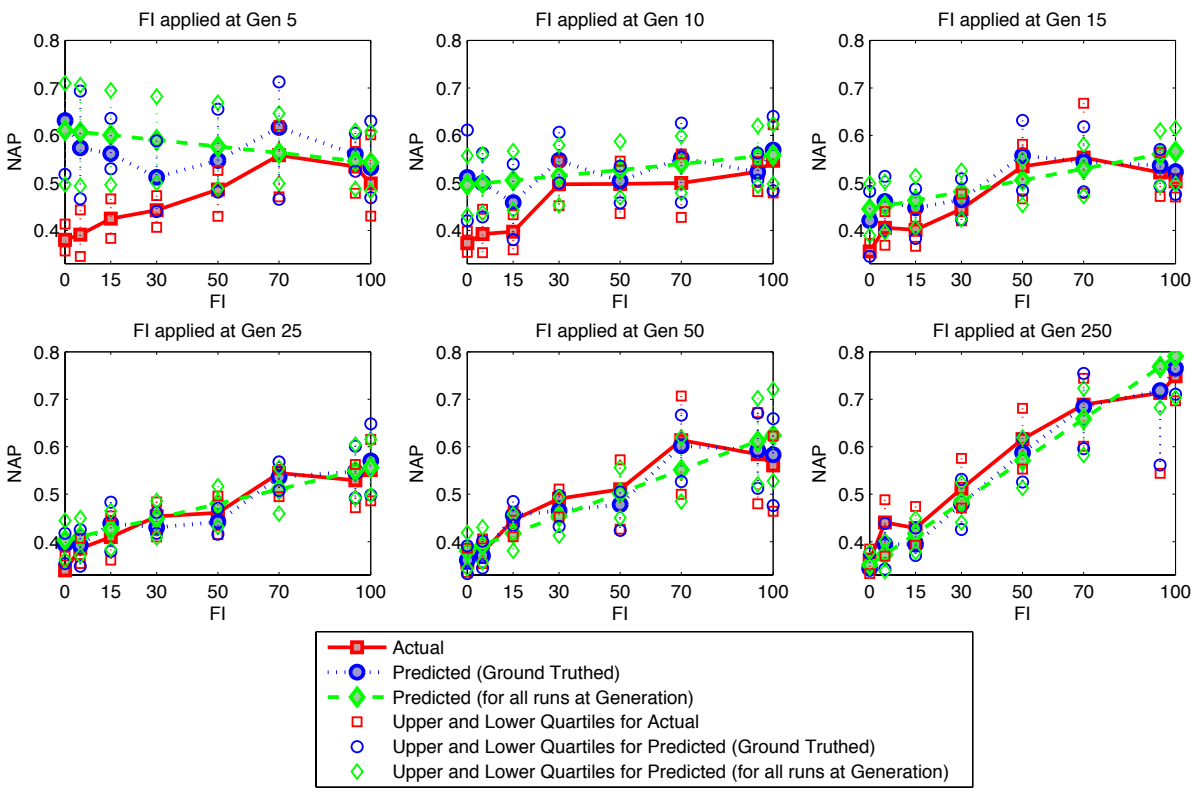

Figure 5.9: Predicted NAP versus the empirically measured NAP (N=20) with ($\phi \in \{0, 0.05, 0.15, 0.4, 0.5, 0.7, 0.95, 1\}$) applied generations ($k \in \{5, 10, 15, 25, 50, 250\}$).

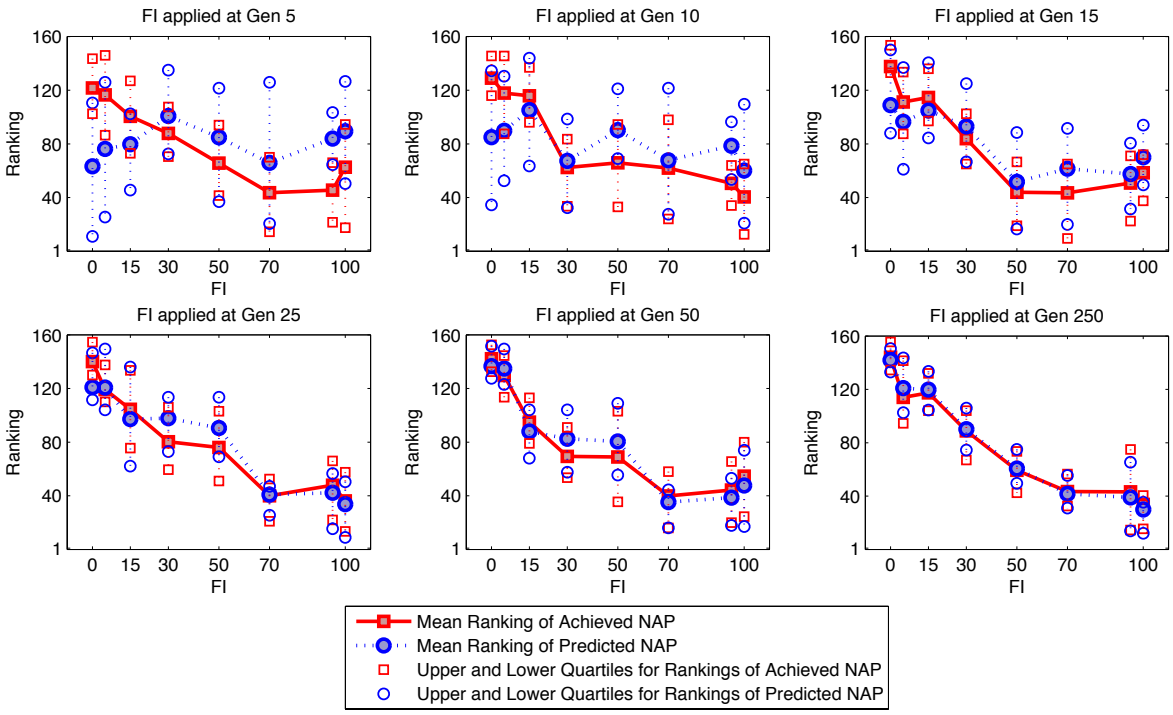

Figure 5.10: Predicted (blue) versus achieved (red) rankings (N=160) of performance of individuals in the next generation after FI is applied at various generations. The good correlation provides reassurance that the expected effect of FI reflects the actual performance impact likely to occur.

Figure 5.10 shows the average ranking of FI values that the predictor function provides if FI was to be applied at various generations. Each subgraph is the result of 160 runs (20 experiments for each $\phi \in \{0, 0.05, 0.15, 0.4, 0.5, 0.7, 0.95, 1\}$) applied at generations ($k \in \{5, 10, 15, 25, 50, 250\}$) for the actual runs (red) and the predicted rankings (blue) based on the predicted NAP generated from the curve fitting of the corresponding $F_{pool}(0, g)$ prior to k.

Again, the later FI is applied and the higher the FI applied, the more predictable the outcome. Figure 5.10 highlights that precise estimation of NAP is not necessary to reliably select a FI that maximises the NAP over the remaining generations. This is possible since the ranking relationship between the different FI curves tends to hold even if the predicted NAPs are incorrect. This provides further reassurance that the effect of FI based from the FI formula reflects what actual impact that is likely to occur.

When looking at Figure 5.10, it is important to remember if one wishes to apply FI early, that a relatively high ϕ value of 0.7 to 0.95 is a good strategy for maximising NAP

after that point, however if the objective is to maximise NAP across the whole evolutionary duration of 500 generations, then it is not a good strategy to apply so early (c.f. parameter sweep results).

5.4 Conclusion

It became evident balancing ϕ to maximise the average performance over a finite period may be possible using predictions of elite and pool fitnesses based on the exponential curve fitting of previous fitness values. Despite the predicted NAP being fairly inaccurate, the ranking of the predicted performances for each ϕ is reasonably consistent with the achieved rankings and thus provides a good method for choosing an appropriate ϕ at any generation. While this technique worked reasonably well for the 'Blink3" problem, how it performs on other problem scenarios and other optimisation parameters remains unknown.

Of particular future interest is how this approach scales to multiple, or distributed systems with global and local goals in real-world scenarios. A tradeoff was evident where the reduction in diversity of the population to obtain higher expected fitness reduced the learning potential of the system. Mitigating this effect will be the subject of future research.

For a system with limited domain knowledge or computational capacity, determining FI dynamically is a challenging problem. In some cases, it may be too difficult to ascertain FI automatically; however we have shown a process for predicting the optimisation trajectory, which can guide what value of FI to set and when to apply it. Alternately, domain knowledge of the problem could also be programmed in to automatically adjust FI.

6

Dynamic FI Management Strategies

The chapter investigates the dynamic management of FI to determine whether it can achieve better performance as measured by the success rate in meeting acceptable average fitness (AAF) over a finite number of generations.

Two dynamic ϕ-management strategies are designed, implemented and compared to less dynamic constant-ϕ-management strategies. One dynamic ϕ-management strategy, ϕ_{track} attempts to minimally meet the AAF while maximising learning capability, while a second more simplistic approach, ϕ_{greedy}, uses the elite whenever its expected performance meets the AAF, otherwise it reverts to a pure learning approach.

Performance for the ϕ-management strategies is experimentally determined using standard GA on the Concat-V *problem. Analysis revealed that the dynamic ϕ-management strategies outperformed non-dynamic ϕ-management strategies, demonstrating that dynamic (continuous) utilisation of the FI heuristic can meet AAF targets more successfully.*

6.1 Introduction

In Constant FI Parameter Management Strategies, it was demonstrated that changing ϕ to a new constant value during evolution impacted the acceptable average fitnesses (AAF). The effect was dependant on the value of ϕ and also when it was applied. In this chapter, the effect of altering ϕ in a dynamic manner (potentially every generation) in response to an AAF target will be investigated.

For this investigation, a solution that satisfices the AAF per generation most regularly will be considered the superior performing algorithm. It is important to understand that solution that attempt to optimise the average fitnesses over an evolutionary run may in fact achieve very poor performance by not utilising the contextual information about AAF provided by the FI heuristic. Solutions for such problems are effectively equivalent to the finite-time multi-armed bandit problem [15] and are not what the FI heuristic, or the control parameter ϕ, were designed to address.

To investigate whether dynamically changing ϕ to meet an AAF target is better than a non-dynamic approach, both non-dynamic and dynamic strategies must first be defined. Section 6.2.1 details the non-dynamic approaches, which effectively employs a constant ϕ value throughout the entire evaluation period. While in Section 6.2.2, the implementations of the dynamic ϕ-management strategies are described.

The results of performance achieved by the various strategies are detailed in Section 6.3 with further discussions of the potential benefits and issues with dynamic ϕ-management in Section 6.4. Some brief conclusions of the outcomes are then presented in Section 6.5.

6.2 Methodology

The aim of this chapter is to determine whether dynamic use of ϕ outperforms a non-dynamic use of ϕ (i.e. as a constant value). As such, a representation of both constant and dynamic ϕ-management strategies must be developed (or supplied). These are found in Section 6.2.1 and Section 6.2.2 respectively.

Ideally, the strategies should be compared via problem that exhibits various levels of AAF since one would expect the dynamic strategies to respond by attempting to meet the new AAF target occurs. In contrast, whether the constant ϕ strategies meet the new target would depend on their current average fitness when the change occurred. Figure 6.1 depicts hypothesised evolutionary trajectories of a dynamic approach against constant ϕ strategies. If the hypothesised average fitness (pool) evolutionary trajectories are representative of their actual behaviour (shown as dotted lines), then the dynamic strategy ($\phi_{tracking}$) should demonstrate the best success rate in meeting the AAF targets (defined by F_{accept_1} where $g_{start} \leq g \leq g_{change}$, and F_{accept_2} where $g_{change} \leq g \leq g_{finish}$). Note that the success rate performance metric is a function of the *both* the fitness function defined in Section 6.2.3 *and* of FI, $\Phi(g)$ which is detailed in Section 6.2.5.1.

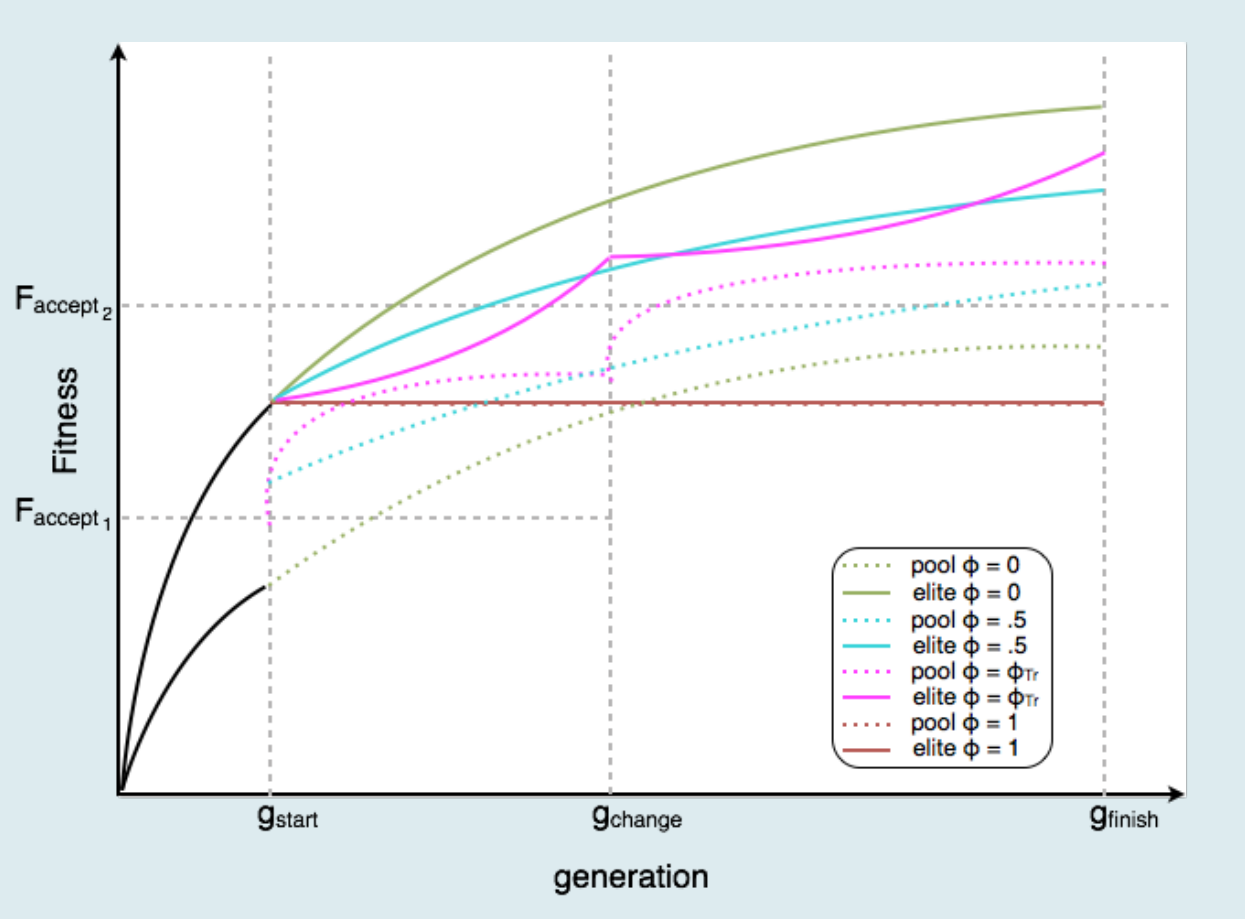

Figure 6.1: Hypothesis of the expected evolutionary trajectories for various ϕ-management strategies.

Unlike previous chapters, a standard GA is employed for simplicity of analysis and to avoid confounding unrelated issues arising from in situ distributed genetic programming on motes. The configuration details of the GA are supplied in Section 6.2.4).

As previously mentioned, the performance of the strategies will be measured with a success rate metric. This is defined in Section 6.2.5.

6.2.1 Constant FI Parameter Management Strategies

The parameter ϕ can be used each generation to bias the construction of a new population towards one where the expected average (pool) fitness is :

$$E[F_{pool}(\phi, g)] = F_{pool}(0, g) + \phi[F_{elite}(0, g) - F_{pool}(0, g)] \tag{6.1}$$

Typically[1] the value of ϕ ranges between $0 \leq \phi \leq 1$ producing a corresponding expected average fitness that lies between the fitnesses of the optimal learning population (i.e. standard pool fitness) and that of the current elite fitness, i.e.

$$F_{pool}(0, g) \leq E[F_{pool}(\phi, g)] \leq F_{elite}(0, g) \tag{6.2}$$

The effect of a changing ϕ from zero to a constant value was shown in Chapter 5 and importantly it was identified that the benefit of immediate improvement of the average fitness came at the expense of a reduced learning capability. Nonetheless, employing ϕ can increase the average fitness which could meet various AAF targets due to the higher pool fitness and the continued learning.

The simplest management of ϕ is to keep it constant throughout the entire evolutionary run. However, depending on the constant value employed, the evolutionary trajectories of the average fitnesses will be markedly different. As such, a number of constants will be employed. The various constants are introduced in order of their naivety to the FI.

[1] ϕ can be specified outside of the range of [0,1], however it makes little practical sense

6.2.1.1 Pure Exploration Strategy

Arguably the most naive strategy to managing ϕ is to keep it set to zero throughout the entire evolution. This is effectively equivalent to not using ϕ at all and so the system reverts to one of normal evolution with no regard to FI. This approach will ideally employ an optimal learning population each generation. When employing this strategy, one relies on the population fitness of the learning metaheuristic to rise to meet the acceptable average fitness (AAF). There will be scenarios where this approach is acceptable and even sensible; for example when:

- The AAF is easily obtainable by the majority classes in the population generation. For example if the expected fitness of even randomly generated programs is higher than the AAF. (i.e. $F_{accept} \lesssim E[F_{random}]$) then it follows that all members of the population are likely to have an expected fitness F_{accept} or better. However, if acceptable fitness is so readily achieved, then it is probable that the problem is trivial, or the solution finder is extremely effective at (or biased to) finding fit solutions.

- The converged pool fitness exceeds the AAF, i.e. $\lim_{g \to \infty} F_{pool}(g, \phi = 0) > F_{accept}$, and sufficient time has already passed such that $F_{pool}(g, \phi = 0) > F_{accept}$. This is likely to be less trivial than the previous scenario since it requires learning. However if a maximal learning approach can achieves the AAF, then it is still likely to be reasonably trivial or that the difference between the elite and pool fitnesses (i.e. $F_{\Delta_{ep}}(\phi, g)$) is small. In the latter case, the FI heuristic will be of little use since it exploits the difference between the elite and pool fitnesses.

- A final scenario where a pure exploration strategy could be employed is one where there will never be any importance to perform. As such the AAF is left undefined. Once could also interpret this as the system's behaviour is always 'acceptable' regardless what evaluations occur which effectively equates to the offline learning scenario.

6.2.1.2 Pure Exploitation Strategy

Exploitation implicitly requires prior knowledge so that it may be exploited. In this context, some knowledge of evaluated candidate solution fitnesses is necessary. Unless *a priori* knowledge is externally supplied before evolution commences (i.e. solution seeding), then this strategy must evaluate at least some candidates to work out which solution to exploit. Despite being a "pure exploitation" strategy, some learning occurs from the evaluation and ranking of the first (randomly generated) population. This is effectively a random search (with the initial population size) and then exploiting the best found solution from generation 2 onward.

As with all the constant-ϕ strategies, the first biasing of the population generation occurs after the first generation. One would imagine though that it would be better to exploit a solution forevermore *after* a solution that meets the AAF has been found. This however requires using FI context information in a dynamic manner (i.e. checking if it was met each generation). An approach similar to this is presented in the dynamic ϕ_{greedy} strategy in Section 6.2.2.1.

For any non-trivial problem this is likely to have extremely poor performance and so is omitted from the experiment. The performance can be determined however as the elite fitness of the random search at generation 1, i.e. $F_{random}(1)$ and assuming that value for the entire evolutionary run.

6.2.1.3 Constant FI Parameter Strategy

Intuitively, a constant ϕ value within the range $0 < \phi < 1$ would provide better performance than the extremes of the pure exploration ($\phi = 0$) and pure exploitation ($\phi = 1$) since it facilitates continued improvement though online learning and also biases the average evaluated fitness above the usual pool fitness.

After the evaluation of the first generation, $\phi = k$ will attempt to raise the pool fitness to $k\%$ of the way between the estimated pure learning pool fitness and that of the current elite fitness. However, this improvement will come at the detriment to the learning rate. The complex interplay of increased pool performance and reduction in learning rate will

generate unique optimisation trajectories depending on the value of k employed. As such, a number of values for will be explored, specifically:

$k \in \{0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.65, 0.70, 0.75, 0.80, 0.90\}$

6.2.2 Dynamic FI Parameter Management Strategies

The objective of the dynamic ϕ-management strategies is to use the contextual information provided by FI to meet the AAF more frequently. Ideally, these algorithms will monitor the AAF, and set ϕ accordingly so that it is met while continuing to learn so that future increases in AAF are readily met (faster). If a portion of the population have fitnesses that exceed the AAF then ϕ necessary to achieve the AAF will be less than 1. Recall that it is necessary that $\phi < 1$ in order for learning to occur.

If no individuals have a fitness above the AAF (as one might expect initially) then the best strategy is simply to maximise the learning rate, i.e. $\phi = 0$. However, in this scenario, when the first individual in the population has a fitness equal to or above the AAF, then it may require setting $\phi = 1$ to achieve the desired AAF. This would then prevent further learning and if the AAF was subsequently increased, then it would need to revert back to pure learning. This strategy is described further in Section 6.2.2.1. Alternatively, ϕ could be set less than 1, and while it would not achieve the AAF immediately, in time it hopefully would with the benefit of continued learning. The interplay of pool fitness improvement and decreased learning rate complicate the decision of what value of ϕ (less than 1) is appropriate. A strategy, ϕ_{track}, that attempts to achieve this is detailed in Section 6.2.2.2.

6.2.2.1 Greedy Strategy

Attempting to maximise the success rate of meeting AAF with online learning has similarities with the goal of attempting to maximise payoff in online learning scenarios. The subtle difference being that AAF is a satisficing metric which can change over time. Nevertheless, many algorithms have been developed for such problems, in particular the studied MAB problem has been studied extensively. Surprisingly, often simple, greedy (short-term gain focussed) strategies perform quite well, and in some cases, optimally [77]. In this

context, a greedy approach would use whatever value of ϕ necessary meet the current AAF without regard to the longer term aspects such as possible increases in AAF. In most situations, this myopic approach will result in ϕ being set to 1 when any individual in the population has a fitness that meets the current AAF. Hence this will also prevent any learning, so should the AAF be increased a later time, then this approach will no longer be able to meet the AAF and should return to the pure learning strategy.

Therefore, this strategy will be defined by:

$$\phi_{greedy} = \begin{cases} 1 & \text{if } F_{elite} \geqslant \text{AAF} \\ 0 & \text{otherwise} \end{cases}$$

Summarised, this strategy exploits the best solution whenever its expected fitness can meet the current AAF, otherwise it will simply revert to pure learning.

6.2.2.2 Dynamic FI Tracking Strategy

The idealised dynamic ϕ-management strategy would minimally raise ϕ to meet the current AAF target and thereby retaining maximal learning potential given the satisficing constraint. However, this does not address the situation where it is required to set $\phi = 1$ in order to meet the AAF target. i.e. when $F_{elite} = F_{accept}$. If this occurs it will prevent the system from learning, requiring $\phi = 1$ forevermore. To avoid this, and ensure at least some learning can occur at all times, it is necessary to limit ϕ such that $\phi < \phi_{max}$ where $\phi_{max} < 1$.

In order to achieve good exploitation, it is beneficial to set ϕ_{max} close to 1, however doing so will likely cripple the learning rate. Furthermore, when the system is transitioning from a population generated with $\phi > 0$, it means that the fitness of the pool as if $\phi = 0$ needs to be estimated. Stochastic effects, plus errors in estimation, mean that the system can overshoot the AAF (and lose diversity which negatively impacts on learning) and also undershoot (which negatively impacts on the success rate).

A detailed explanation of these issues and the mechanisms to address them are found in Appendix D. The resulting dynamic ϕ-management strategy, ϕ_{track}, attempts to control

ϕ to achieve a pool fitness that tracks the AAF. However, to address various practical issues, the parameters described in Table 6.1 were needed.

ϕ_{max}	Limits ϕ so that some population diversity is always retained for learning
ζ	Bias to ensure F_{pool} will likely be above F_{accept} despite stochastic variation
α	Dampens rate at which ϕ increases to avoid oscillation and maintain diversity
β	Dampens rate at which ϕ decreases to avoid oscillation and maintain success

Table 6.1: ϕ-tracking strategy parameter descriptions

Some exploration of suitable values for these parameters will be necessary to achieve desirable behaviour.

6.2.3 Problem Selection and Description

> " Problems of interest tend to fall somewhere between the Onesmax problem and the Needle-in-a-haystack problem. "
>
> - Adam Prügel-Bennett [105]

We seek a simple, well-studied fitness function that is sufficient to show the benefit of employing FI. This way, the mechanisms responsible for performance can be studied in detail without requiring significant investment into understanding intricacies of the problem. There are a plethora of well studied fitness functions (problems) for benchmarking in evolutionary computation research. However, perhaps the most quintessential "simple" problem is the OneMax Problem and is described in Section 6.2.3.1.

Unfortunately the OneMax is best handled with any number of hill-climbing approaches (incremental learners) and not aided by schemata searching operators like crossover. Since incremental search works well, the GA population tends to be dominated with mutations of an elite solution. This means that F_{elite} and F_{pool} are usually fairly similar, leaving little for ϕ to exploit (c.f. 4.3 FI Analytic Model).

A variation of the problem, known as the *Concatenated-V problem*, however, has reported superior performance with GA, and furthermore exhibits a more significant divergence between elite and pool fitnesses which can be exploited. The *Concatenated-V problem*, which is ultimately used as the problem to benchmark the various ϕ-management

strategies on, is detailed in Section 6.2.3.2. However, since it is a variant of the OneMax problem, the OneMax problem will first be described.

6.2.3.1 The OneMax Problem

ONEMAX PROBLEM

The OneMax [108] problem is defined by a binary string (bit string) of a specified length, N, where fitness is simply determined as the sum of digits within the string (i.e. the number of 1's in the string). Thus, with the objective to maximise fitness, the optimal solution is simply a string composed entirely of ones. The fitness landscape of the OneMax objective function for $N = 7$ is shown in Figure 6.2.

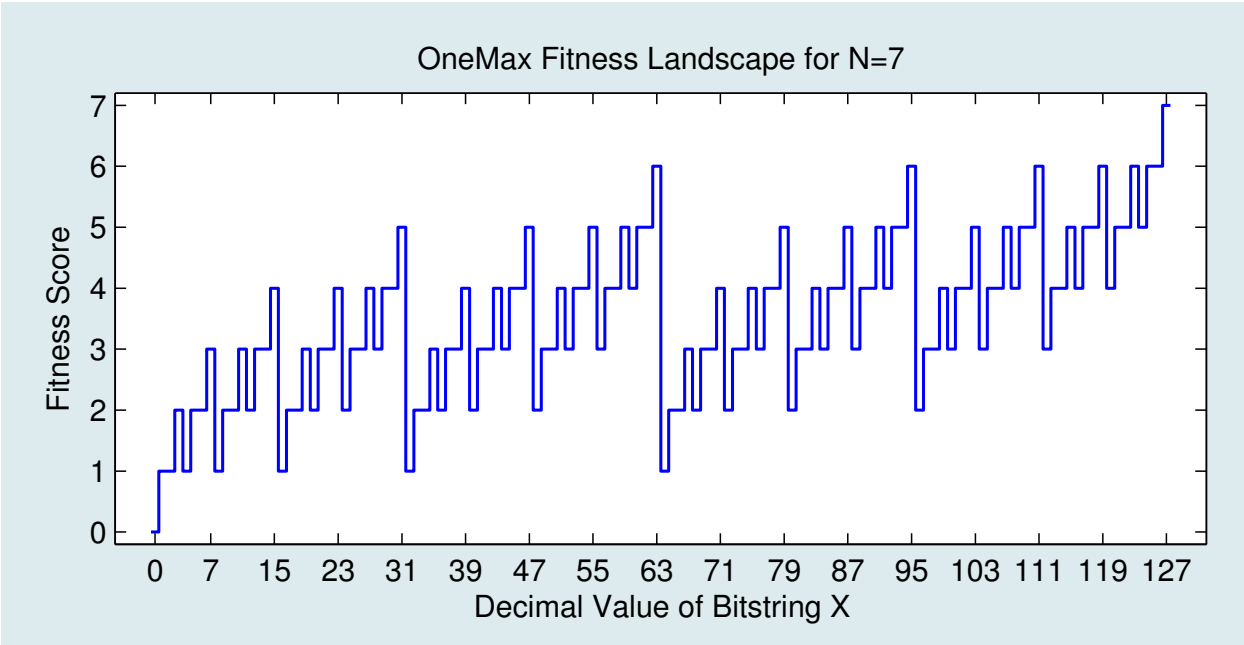

Figure 6.2: The OneMax objective function fitness landscape for $N = 7$

Formally, the OneMax objective function can be defined by attempting to maximise:

$$F_{onemax}(\mathbf{X}) = \sum_{i=1}^{N} x_i \tag{6.3}$$

where $\mathbf{X} = \{x_1, x_2, \ldots, x_N\}$ is a bit-vector composed of $x_i \in \{0, 1\}$.

Unfortunately the extremely simple nature of this problem does not demonstrate the utility of GA well [105] and since incremental search works well, GA tends to revert into

a hill climber by gaining benefit predominantly from mutation only. This biases the population towards local search behaviour, meaning the population is largely similar and so mostly similar performing. As such there is typically little difference between the elite and pool fitnesses, leaving little to be exploited by using FI. Hence, while the OneMax problem is ideally simple and well studied, it is not suitable for demonstrating the utility of FI.

6.2.3.2 The Concatenated-V Problem

CONCATENATED-V PROBLEM

The *Concatenated-V problem* [105] is a simple variant of the OneMax problem which while remains simple to describe, is better suited to be solved by the prototypical implementation of GA over simple hill-climber solvers, etc. As per the OneMax problem, the optimal solution remaining as a string composed entirely of ones. For example, when $N = 7$, the optimal solution is $\mathbf{X} = \{1, 1, 1, 1, 1, 1, 1\}$ (expressed in decimal as 127) has the unique optimum fitness of 4, as shown in Figure 6.3.

However, an important difference from the *OneMax* problem, is that there are a number of local maxima that many solvers (including hill-climbers) can easily get stuck on. One such local optima is the bit string of all zeros (expressed in decimal as 0 in Figure 6.3), which in contrast with the *OneMax* problem was the lowest fitness possible. This local optima is extremely difficult for many optimisers to escape from and get "across" to the all-ones global optimal. The difference in fitness landscape arises from the bit string being subdivided into a M blocks (substrings) each of a fixed length k such that $N = Mk$. The fitness is the summation of the partial fitnesses gained from each of the blocks, which in turn is described by a V-shaped function (hence its name of *Concatenated-V*). This V-shape function, shown in Figure 6.4 is generated by taking the absolute value of the number of ones within the block subtracted from $\frac{k-1}{2}$ where k is odd and $k \geq 3$.

Expressed formally, the *Concatenated-V* objective function is :

$$F_{CV}(\mathbf{X}) = \sum_{i=0}^{M-1} F_B(X_i) = \sum_{i=0}^{M-1} \left| \frac{k-1}{2} - \sum_{j=1}^{k} x_{ki+j} \right| \tag{6.4}$$

where $\mathbf{X} = \{X_0, X_1, ..., X_{M-1}\} = \{x_1, x_2, \ldots, x_N\}$ and $x_i \in \{0, 1\}$ and $N = Mk$ where M is the

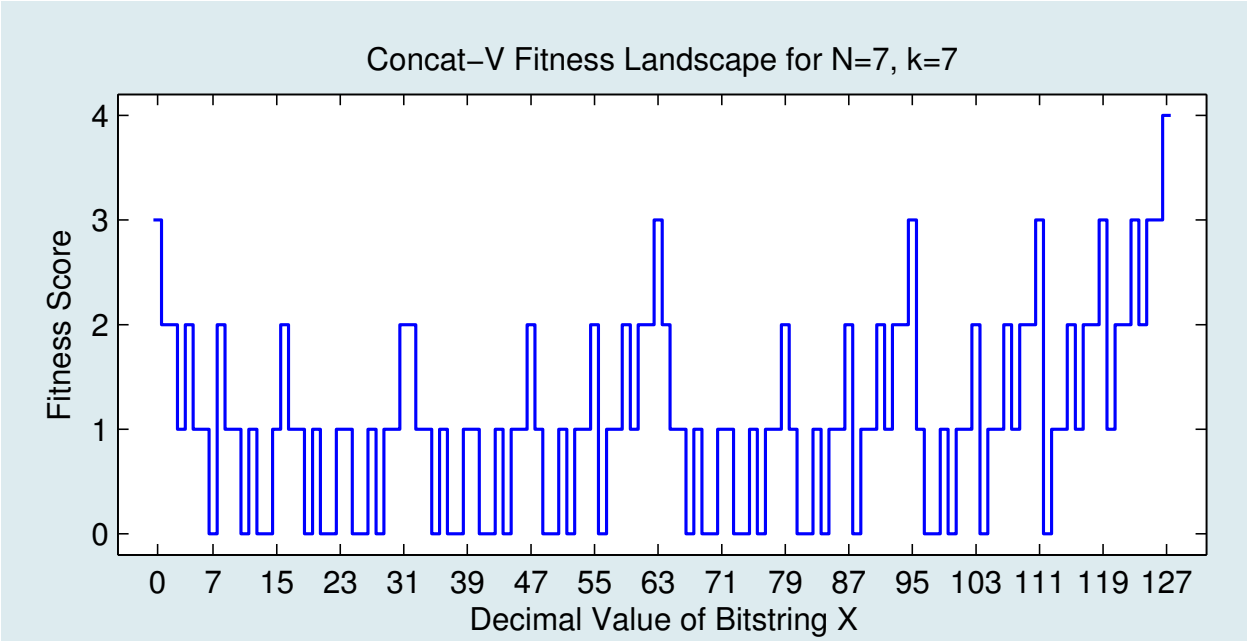

Figure 6.3: *Concatenated-V* objective function fitness landscape for $M = 1, k = 7$ ($\therefore N = 7$)

Table 6.2: Distribution of fitness scores in the *Concatenated-V* block landscape for $k = 7$. $p(F_B)$ is the probability of fitness $F_B(\mathbf{X}_R)$ occurring given a random bit string, $\mathbf{X}_R$.

Fitness	Occurrences	$\mathbf{p(F_B)}$
0	35	0.2734375
1	56	0.4375000
2	28	0.2187500
3	8	0.0625000
4	1	0.0078125

number of blocks each of size k bits.

For the specific parameters of $M = 100$ and $k = 7$, [105] claim that a hill-climber will achieve an expected final fitness of 366 ± 5 with a low probability (5×10^{-19}) of finding the optimal all-ones solution (of fitness 400). Furthermore, they show that standard GA with crossover outperforms the hill-climbing technique meaning that the problem is both hard, but suited for GA to solve.

Since $N = Mk$, the number of possible states in the landscape is $2^N = 2^{700} \approx 5 \times 10^{210}$. Within this landscape, there is one optimal solution (all ones) with a fitness of 400 and hence the probability of randomly selecting the optimal solution is $\approx 5 \times 10^{-210}$. Conversely, there is one minimum fitness solution with a fitness of 0.

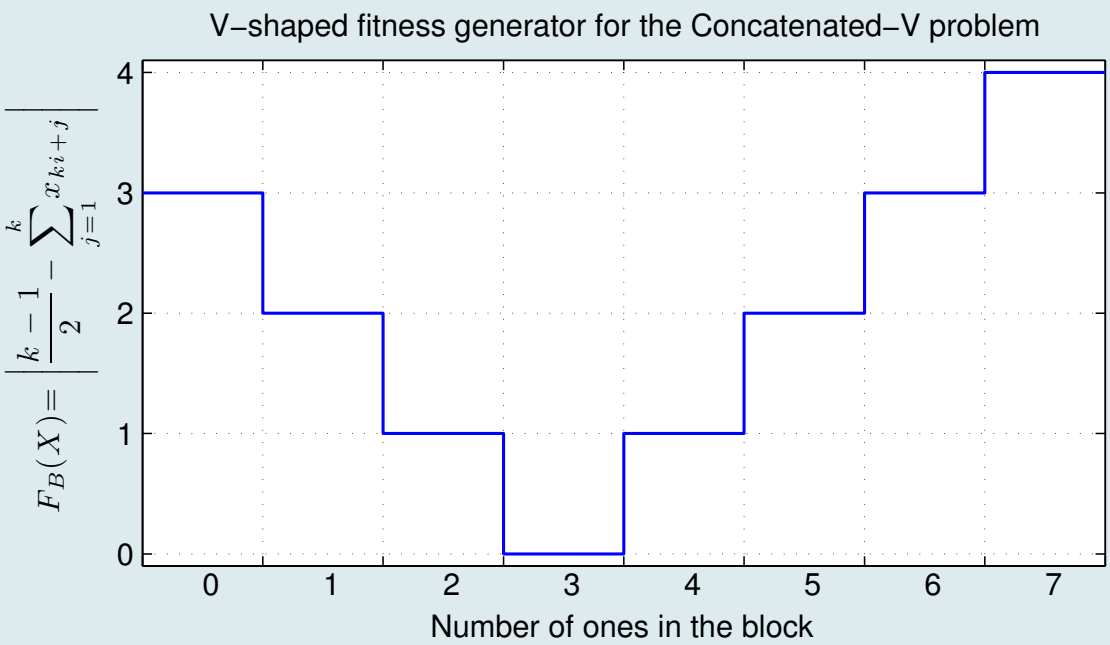

Figure 6.4: Block fitness, F_B, can be described by a V-shaped function of the number of ones in the block. The block fitnesses are combined to generate the *Concatenated-V* fitness score.

Given the block size of $k = 7$, there are 5 discrete fitness scores possible occurring with the distributions shown in Table 6.2 for a randomly generated string. The Central Limit Theorem (CLT) states that the sum of M independent random variables taken from identical and independent distributions (i.i.d) will asymptotically approach a Normal distribution with a mean of $\mu_B M$ and standard deviation of $\sigma_B\sqrt{M}$ where μ_B and σ_B are the mean and standard deviation of the i.i.d's. The fitness, F_{CV} of a random bit string, **X**, composed of substrings $\{X_0, X_1, ..., X_{M-1}\}$ is essentially the sum of random variables pulled from M samples of $F_B(X)$ (hence i.i.d).

Using the probabilities of the block fitnesses, $p(F_B)$, from Table 6.2 the expected fitness (mean) is computed as $\mu_B = 1.09375$ with a standard deviation of $\sigma_B = 0.896$ (to 3 decimal places). Thus, the distribution of F_{CV} fitnesses for random bit strings $\mathbf{X}_R$ will approach the Normal distribution of Equation 6.5 as M becomes large ($\gtrsim 30$).

$$F_{CV}(\mathbf{X}_R) = \sum_{i=0}^{M-1} F_B(X_{R_i}) \approx \mathcal{N}(\mu_B M, \sigma_B\sqrt{M}) \tag{6.5}$$

For completeness we compute the expected fitness for a random string using Equation 6.5 with $M = 100$ as $\mu_{CV_R} = E[F_{CV}(\mathbf{X}_R)] = \mu_B M = 109.375$ with a standard deviation of $\sigma_{CV_R} = \sigma_B\sqrt{M} = 8.965$ (to 3 decimal places).

This choice of the *Concatenated-V problem* is motivated by its simplicity to describe, that it has been studied by others, that the optimal solution is known and the fitness landscape readily calculable. Importantly, the problem is sufficiently challenging such that a standard genetic algorithm (the solver employed in this investigation) can outperform random search and basic hill-climbing strategies. This exploits the genetic variation gained by the crossover operator which is known to typically be "destructive" to to fitness more often than not. This creates a larger variation in the fitness of individuals, which in turn creates a larger difference between the pool and elite fitnesses. This difference makes the *Concatenated-V problem* well suited to demonstrate FI and so will be employed for the comparison of ϕ-strategies. Prior studies employing this problem [105] used the parameters of $M = 100$ and $k = 7$. This study will also use these parameter settings, resulting in a problem dimensionality of $N = 700$.

6.2.4 Solver Selection and Description

The following section explains the choice of employing a basic Genetic Algorithm as an prototypical population-based learning optimiser whose characteristics can be influenced by the population generator function.

6.2.4.1 Selection of GA as the Solver

A basic Genetic Algorithm (GA) with elitism is employed as the solver for the experiments in this chapter. The main justification for this choice, rather than Genetic Programming (GP) or within the IDGP implementation, is it simplifies analysis by avoiding confounding unrelated issues generated from other aspects such as environmental effects of being in situ and the complexities of being distributed and GP on motes (see Section 3.6). Furthermore, prior research on this problem by others [105] employed GA, so an understanding of how GA performs on the problem already exists in the literature.

The ϕ control parameter (through the population generator) employs various operators to construct a biased population to achieve a desired average fitness for the next generation. This mechanism is applicable to both GA and GP since they are both population-based evolutionary algorithms that employ the standard operators of selection, mutation and crossover[2]. As such, conclusions drawn about ϕ-management from the experiments employing GA will equally apply to GP, thereby remaining relevant to the IDGP framework.

6.2.4.2 GA Settings

Any number of GA settings could be employed to demonstrate the utility of ϕ providing the following statements hold true:

1. A population generated with $\Gamma(\phi = a)$, should demonstrate **superior learning** capability over a population generated with $\Gamma(\phi = b)$ where $a < b$.

2. The average fitness of a population generated with $\Gamma(\phi = a)$, i.e. $E[F_{pool}|\Gamma(\phi = a)]$, should demonstrate **inferior performance** (average fitness) over a population generated with $\Gamma(\phi = b)$ where $a < b$.

This implies that the population generator function Γ is a monotonic function (at least in ideal form). However, it does not imply that $\Gamma(\phi = 0)$ has to be the optimal learning population configuration in order to use ϕ (however, that is the ideal case).

Nonetheless, the key to achieving approximate monotonic learning/performance behaviour with the γ_{simple} population generator is for the user to provide a good "pure learning" $\Gamma(\phi = 0)$ population structure. In this context, population structure refers to whether elitism is employed and how many elites are copied, whether mutation and crossover are applied and in what combinations and proportion of the population they are applied to, and whether randomly generated (or immigrated for distributed GA) candidates are added and in what proportion. Typically for GA, users determine a good (though unlikely optimal) population structure by using heuristics, trial and error or another optimisation process.

[2]Cloning due to Elitism is included as it can be achieved with these standard operators.

For these experiments, the GA population structure will be kept close to an archetypal "standard" GA implementation. **Elitism** is employed keeping the best solution between generations. A proportion of the top performing individuals are cloned and have **mutation only** applied, implemented as random bit-flipping of a percentage of genes specified by a constant mutation rate. **Children** are the result of single-point crossover on two parents selected via fitness proportionate selection and then having mutation. **Random** programs will also be generated and included.

6.2.4.2.1 Population Size

One of the first considerations is often population size. Some advocate the use small population sizes, particularly in the face of computational constraints such as those expected within this thesis, whilst others suggest near optimal sizes can be achieved for specific problems [40]. Generally, where computational capability exists, it is recommended [27], [5] that the population size be up to the order of the dimensionality of the problem, i.e. $N, 2N, N \log N$ or dynamically as a function of the variance [41] or error [121] of the population fitness.

Within the IoT scenarios that the IDGP framework is directed at however, it is expected that the population size will be restricted due to the constraints on the computing capability of each entity. Additionally, the population size for optimal learning is not essential to demonstrating the characteristics of applying FI, which is the focus of this experiment and so a population size of 50 was chosen.

6.2.4.2.2 Number of Generations

For the specific parameters of $M = 100$ and $k = 7$ on the *Concatenated-V problem*, [105] state that a hill-climber will achieve an expected final fitness of 366 ± 5 with a probability of finding the optimal all-ones solution (fitness of 400) as 5×10^{-19}.

For 100 evolutionary runs employing a basic GA, the final fitness was 369 ± 6 after 5000 generations, with little progress occurring after this (see Figure 6.6). As such, the evolutionary run was set to 5000 generations.

Setting	Symbol	Value
Problem dimensionality	N	700
Population size	N_{pop}	50
Population Structure	$\Gamma(\phi = 0)$	$[1\ 3\ 43\ 3]$
Mutation rate	μ	2.5%
Crossover		Single Point
Max generations	g_{finish}	5000

Table 6.3: GA "pure learning" configuration and settings.

6.2.4.2.3 Mutation Rate

For the *OneMax* problem, [87] state that the optimal mutation rate is proportional to the size of the representation, N. i.e. $\frac{1}{N}$. However the empirical results of various mutation rates used on the *Concat-V-change* problem shows that the suggested mutation rate, $\frac{1}{700} \approx 0.0014$, does not yield optimal learning performance.

The elite trajectories for a variety of mutation rates were tested revealing that mutation rates around 0.40% - 2% do well for the first period before the landscape changes and the high mutation rate of 3% and 4% do well for the second period from $g = 2501$ on. The selection of an optimal mutation rate or population size is not essential to this study, and so the generally good performance gained from a mutation rate of 2.5% (shown as the dotted black line) with a population size of 50 was ultimately employed.

6.2.4.2.4 Population Structure

In 6.2.4, a population is defined as the composition of subpopulations (classes), that were generated by a combination of genetic operators (i.e. a population generator). Recalling using γ_{simple} requires a difference between the pool and elite fitnesses to exploit, a variety of population structures were evaluated over the period of 5000 generations and repeated 100 times for each structure. Figure 6.5 shows the average elite and pool fitnesses after 5000 generations from the experiments. While conversely, the GA configuration and parameter settings for the "pure learning" ($\phi = 0$) configuration are shown in Table 6.3.

($\phi = 0$)

Using these operators, four 'classes', or classes, are generated with each new population created, similar to those discussed in Chapter 4 (specifically Section 3.2.3.1) with

the notable exception of the '*other*' deme since all individuals are evolved locally (not a distributed approach) with no immigration from other populations.

We will use the previously defined notation for population structures defined by classes.

This is evident by comparing Figure 6.5 with Figure 6.7 which shows that the same learning population structures used on the differing objective functions of *Concatenated-V* and *Concat-V-changed*, yielded significantly different performance. From analysis of these graphs, the population structure of $[1\ 3\ 43\ 3]$ was determined as achieving reasonable performance on both problems and so was chosen as the 'optimal' learning configuration. Figure 6.6 shows the optimisation trajectories for a standard evolutionary learning approach.

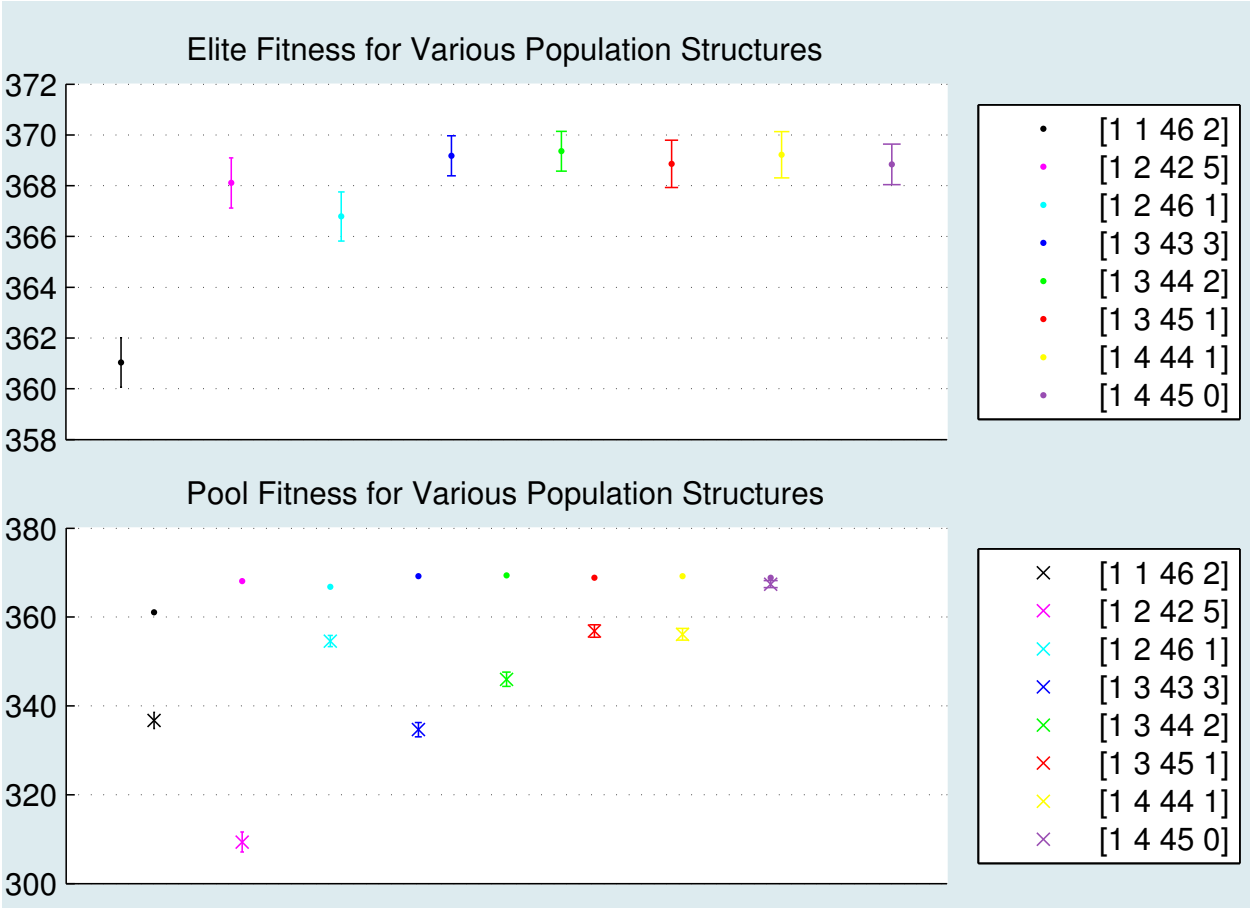

Figure 6.5: Mean ($N = 100$) elite and pool fitnesses for a variety of population structures at $g = 5000$ for the *Concatenated-V* problem. The 95% confidence intervals are indicated with the bars. Note that the elite fitnesses are also shown on the pool fitness plot to highlight how the pool fitness becomes closer to the elite fitness with the removal of 'randoms' from the population structure.

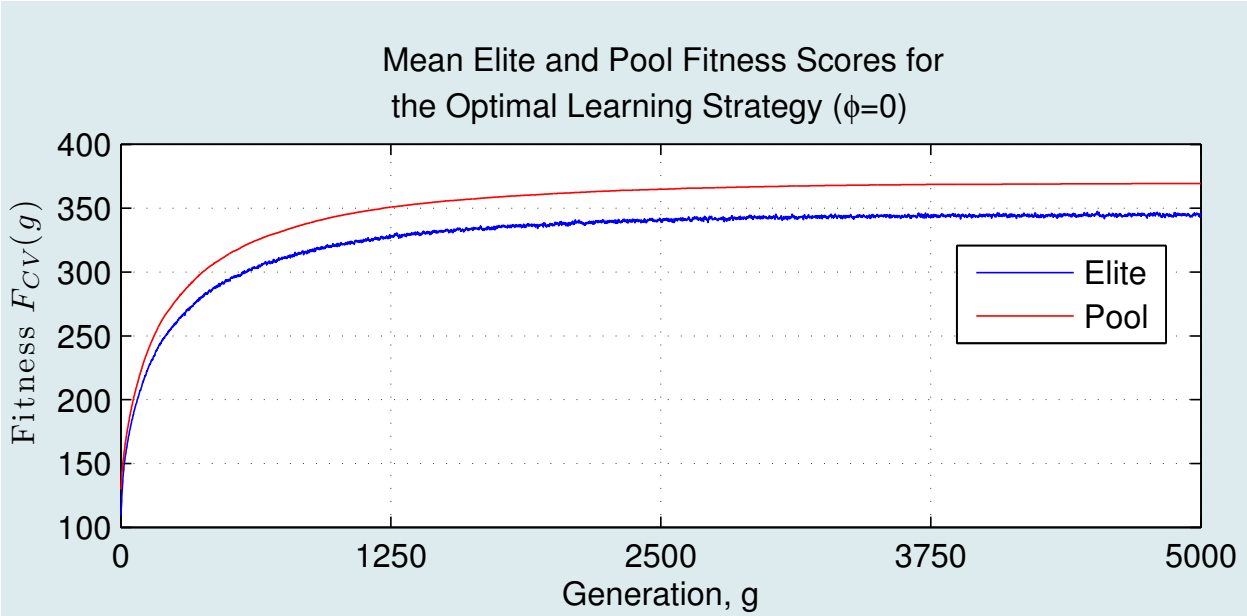

Figure 6.6: Average (n=100) elite and pool evolutionary trajectories for the standard learning configuration on the *Concatenated-V* problem. Note that the AAF threshold of 350 is exceeded, on average, after 1215 generations (about a quarter of the 5000 generation evolutionary run).

6.2.5 Evaluation

6.2.5.1 Fitness Importance Function for this Experiment

Traditionally, a description of the fitness function from which the fitness landscape can be derived, would suffice for describing a particular problem. However, this thesis postulates the decoupling of achievable fitness from the contextual information of how important achieving a particular level of fitness is. When FI is considered, the problem changes from one which focusses on achieving the best possible fitness within the constraints (such as convergence time or optimality of solution), to one that attempts to achieve an acceptable average fitness within the current context. Therefore, within this framework, to completely describe a "problem" requires *both* the fitness function, $F(t)$, and its representation by the solver, as well as the acceptable average fitness (AAF) to be known.

For a given fitness function, the fitness landscape will be a result of the problem representation employed by the solver. Therefore the choice of solution finder can significantly effect how easy (or difficult) it will be to find a solution.

The problem type of interest to this research is one where learning is desired whilst

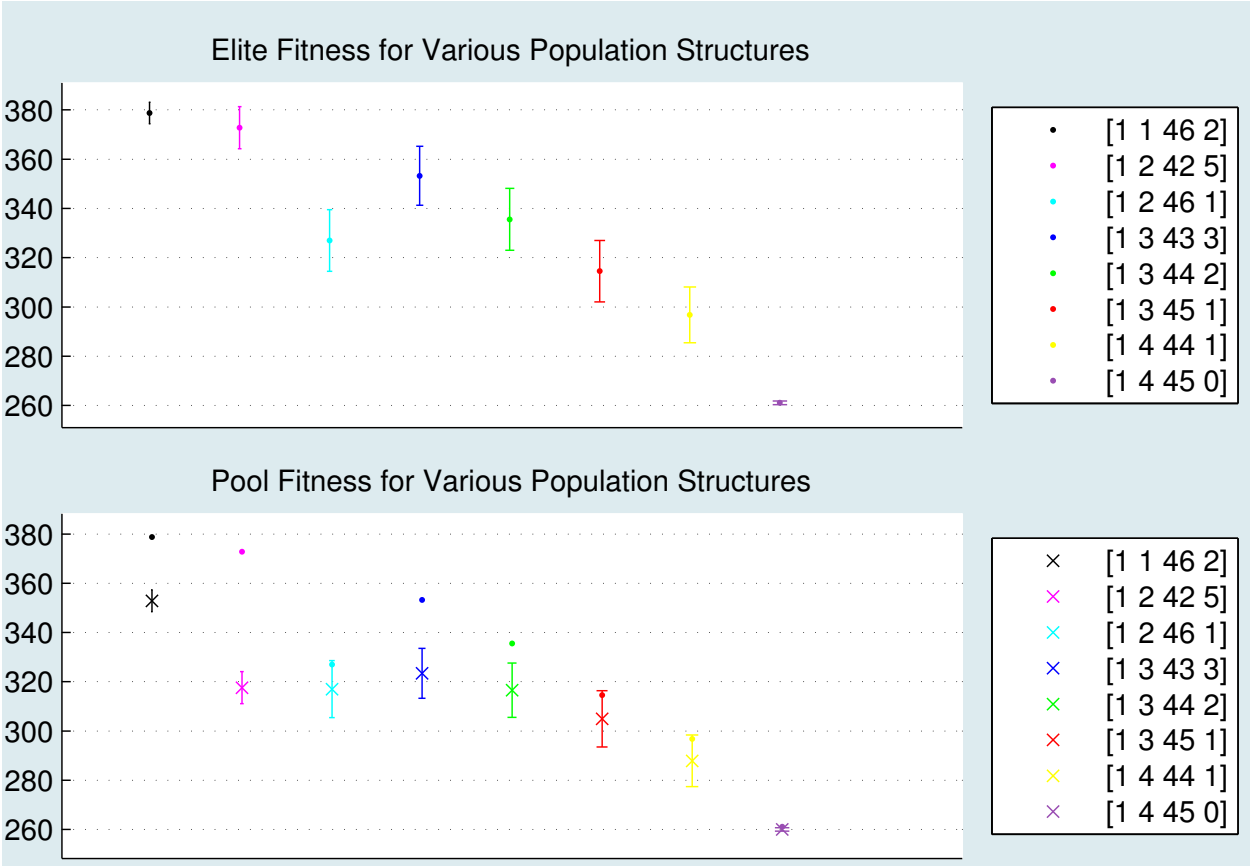

Figure 6.7: Mean ($N = 100$) elite and pool fitnesses for a variety of population structures at $g = 5000$ for the *Concat-V-Change* objective function landscape. The 95% confidence intervals are indicated with the bars. Note that the elite fitnesses are also shown on the pool fitness plot. Unlike in Figure 6.5 however, the exclusion of 'randoms' from the population structure has a significant deleterious effect on the best solution obtained during the run.

simultaneously demonstrating an acceptable average fitness. This problem type has previously been enumerated as Learning Problem Type 5 and characteristics of this problem type are discussed in Section 4.3.3. The ϕ-management strategies to be evaluated in this chapter will be studied within the context of this "learning while performing" problem class.

The prototypical GA performance on the fitness landscape described in Section 6.2.3 and shown in Figure 6.6.

For simplicity, the change in acceptable fitness is limited to the minimal case of one permanent change of the acceptable fitness occurring half way through the evaluation period after the initial non-zero setting of acceptable fitness (i.e. the acceptable fitness will be zero until the importance of performing is realised, after which at some point it is

then changed to another value).

Similar to the dynamic fitness landscape, the simplest variation from a constant AAF is one that changes from one constant AAF to another constant AAF. Again, even though the change is minimal, it still becomes dynamic in nature since the value changes during the evolution of the system.

The timing and the new acceptable value of the change will likely effect the success rate that can be achieved. For simplicity, the single change will occur half way through the evaluation period at $g_{change} = 501$, and the increased value will be an absolute fitness value as per the constant acceptability scenario. Once again, this value must be well considered otherwise the value may be achieved trivially or conversely impossible to achieve. Ideally the value should be readily discoverable by the elite and not readily achievable by the (standard learner, $\phi = 0$) pool fitness. Since the value is absolute and the actual elite and pool fitnesses at generation 501 cannot be known *a priori*, the expected values of the elite and pool at generation 501 are used instead to determine the new absolute acceptable average fitness. The mean elite and pool fitnesses at generation 501 (c.f. Figure 6.6) are 369 and 350 respectively, resulting in a mean exploitation potential of $E[\Delta_{ep}(501)] \approx 19$.

Some consideration of the dynamic fitness landscape is also necessary as the value should be achievable within the changed landscape also otherwise little insight will be gained.

With these considerations in mind, the first AAF, F_{accept_1}, is set to be 343 for the period $g_{start} \leq g \leq g_{change}$. The second AAF, F_{accept_2}, is set to 365 for the interval $g_{change} \leq g \leq g_{finish}$.

6.2.5.2 Performance Metric and Evaluation Period

The success rate performance metric, $P_{success}(g)$, corresponds to the rate at which the average fitness over each generation meets the AAF.

In an attempt to construct a fair comparison approaches, we will employ only one method for solution finding, namely genetic evolution, and allow all approaches to evolve for the same number of generations before the importance of fitness is increased from 0. This will occur at generation, g_{start}, at which the importance will be lifted such that success

requires a pool fitness that lies between the $F_{pool}(g_{start})$ and $F_{elite}(g_{start})$. Furthermore, g_{start} will be set prior to convergence (i.e. F_{elite} nearing $F_{optimal}$, if known) such that online learning can provide benefit.

Using the four application scenarios, two *FI strategies* are compared with the conventional strategies of offline evolution and online evolution. (i.e. $0 \leq \phi \leq 1$) one with a constant FI, $\phi = 0.5$, and the other with a dynamic FI such that $\phi_{tracking} \geq \phi_{desired}$ at each generation. (*Learning Strategy* ($\phi = 0, \forall g$), an *Offline Learning Strategy* ($\phi = 0$ until $F_{elite}(g) \geq F_{accept}$ and then $\phi = 1$ from then on) to that the *FI strategies* ($0 \leq \phi \leq 1$) of $\phi = 0.5$ and $\phi_{tracking}$ (where $\phi = \phi_{desired}$ at each generation).

In this experiment, $F_{accept}(g)$ starts at 0 (i.e. $\Phi = 0$) at generation 1, and remains zero up to and including generation $g_{accept_1} - 1$ which is the 749th generation. At generation g_{accept_1} (150th), $F_{accept}(g)$ steps up to F_{accept_1} and remains constant until the end of the experiment (in the constant performance scenario) or is set to a higher constant of F_{accept_2} at generation g_{accept_2} (500th). The value of F_{accept_1} is selected randomly from the uniform distribution across the range $F_{pool}(149) \leqslant F_{accept_1} \leqslant F_{elite}(149)$. All strategies are compared using the same F_{accept_1} and share the same evolutionary trajectory up to generation 149. Note that the optimal learning strategy is employed for $g < F_{accept_1}$ since there is no penalty for not performing.

After generation g_{accept_1} the evolutionary trajectories will diverge due to the different metaheuristic strategies being evaluated. The population structure is set using the "simple" subpopulation distribution generator function $\Gamma_{simple}(\Phi_{strategy})$ where $\Phi_{strategy}$ is 0, 0.5, 1 or tracking. Some divergence will occur due to the stochastic nature of the mutation and crossover operators of evolution.

Generating a reasonable value for the second value of F_{accept} is less straightforward however, since the evolutionary optimisation trajectories may have significantly diverged by g_{accept_2}. Further complicating matters is that the fitness landscape may change at the same generation ($g_{change} = g_{accept_2}$) for the changing landscape scenario. We will assume that while the fitness landscape has changed, the meaning (or units) of fitness has not. For example, if the fitness score was say profit in dollars, then even though the challenge (fitness landscape) may change, an amount before or after the change will

have the same utility. Within this in mind, then it can make sense to desire a higher acceptable average fitness relative to F_{accept_1}. However how much higher is achievable is dependant on the fitness landscape. A conservative approach would be to choose $F_{accept_1} \leqslant F_{accept_2} \leqslant F_{elite}(749)$, which in an unchanging fitness landscape, would guarantee the acceptable performance level was higher but achievable. However, this scenario is of trivial interest since $P_{success\_rate} = 1$ could be achieved without any additional learning after g_{accept_1}. A less-conservative, and more interesting scenario, would be to raise the acceptable fitness by some factor regardless of knowing whether the target is achievable or not at that point in the future, i.e. $F_{accept_2} = aF_{accept_1}$ where $a > 1$, but hoping that learning will enable F_{accept_2} to be achieved. This would be akin to an investor demanding a 10% improvement on daily profit next year compared to the current year's daily profit, and if that isn't achieved then deeming it not to be a success. One can see how this approach could easily generate a $F_{accept} > F_{elite}(749)$, which while extremely unlikely to be achievable at generation 750, may be achievable later due to learning or changed fitness landscape. This challenge is also a more compelling illustration of the potential usefulness of the FI heuristic, since it may require learning to occur in order to achieve success when the second acceptable fitness threshold is set. For this reason we will employ the scaling factor approach, setting $a = 1.1$ thereby requiring a modest 10% performance improvement from that set at generation 750.

6.2.5.3 Evaluation Period

Practically, we need to set a start time, even if the timing is somewhat arbitrary.

Even in the offline scenario, one can still observe the time-dependent change of importance of performance from very low (pre-deployment) to very high (post-deployment). This temporal variation of the importance of performance is necessary for the Fitness Importance heuristic to be useful.

For all problems we can make the very general claims that:

- a problem comes into existence or is identified before a solution can be applied
- it will take some finite amount of time to devise or identify a solution that can be

applied

- a devised solution could be applied immediately or further time could be spent devising a "better" solution

In this analysis, the problem is assumed to present itself at time $t = 0$ and for convenience time will be quantised into discrete periods (generations) corresponding to the time taken to evaluate all individuals within a generation. i.e. we will use $g = 1, 2, 3...$ as our time basis. Additionally, let g_i correspond to the point in time immediately after the completion of the evaluation of the ith generation and before the $i + 1^{\text{th}}$ generation, with the special case of g_0 which occurs before the first generation $g = 1$ but obviously does not follow any generation. Hence the period over which the nth generation is evaluated is $g = n$ which occurs between the specific times of g_{n-1} and g_n.

As discussed in Section 4.3.3, there will be a class of problems where it makes most sense to generate solutions by interacting with the world directly (i.e. *in situ*) and not via a simulation of the world (i.e. *in silico*). Therefore, regardless of whether a solution is devised offline or online, the real-world will be used for feedback during the generation of the solution. Providing the same solution generating mechanism is used, then the time to generate an acceptable solution should be the same for either approach. We will continue to employ evolutionary algorithms (though not evolutionary programming) as the solution generating mechanism. For the toy problems used in this analysis we can think of *in situ* evolution being equivalent to requiring each evaluation taking a finite period of time and, most importantly, evaluating a solution affects the success rate of the system when FI is greater than zero.

Let us suppose that a problem presents itself at some specific time, g_0. Let us additionally assume that we are keen to deploy a solution when it becomes available. More specifically, that we have the flexibility to wait for an acceptable solution to be evolved but not so much as to be able to wait for convergence to an optimal solution. Let g_{start} be this point in time after an acceptable solution has become available. Similarly, let g_{finish} be the point in time when solving the problem is no longer of interest or the importance of performance is once again zero (i.e. $\Phi(g) = 0$ where $g \geqslant g_{finish}$).

Thus, for the given problem, we have a fixed period of utility from t_{start} and t_{finish} defined by when $\Phi > 0$. The importance of fitness during the period of utility will be lifted such that success over the time epoch of one generation requires the pool fitness to be between $F_{pool}(g_{start})$ and $F_{elite}(g_{start})$. This means that success is known to be achievable, as it will less than the known current elite, but also not so trivial that any population can achieve success. The lower limit of $F_{pool}(g_{start})$ is a logical bound since in theory this will be the observed performance of the optimal learning population and so without *any* loss of learning potential, this level of performance can be achieved. Setting an acceptable performance target lower than this will tend to be trivial to achieve providing the fitness landscape remains constant.

For experiments where the FI function changes during the period of utility, the change will be limited to only one occurrence. Where the fitness landscape also changes, we will coincide the change of fitness importance to the same generation that the change in the fitness landscape occurred.

6.3 Results

Table 6.4 shows the success rate for the various ϕ-management strategies calculated over the period before and after g_{change} (i.e. generation 500), as well as the performance over the whole period. Note the performance is not calculated on the first 150 generations where $\phi = 0$ since the success rate during that period is trivially 100%. Results from the random search algorithm illustrate that the success objective is nontrivial. Shaded cells correspond to which strategies exhibited the best result for a particular metric. Note there are multiple shaded F_{elite} scores since they are note statistically significantly different from the best result of 371.3 ± 0.9 (based on a 95% confidence interval). The following subsections detail the results of the four ϕ-management strategies.

ϕ-Strategy	$P_{success}(150, 500)$	$P_{success}(501, 1000)$	$P_{success}(150, 1000)$	$F_{Elite}(1000)$
random search	0.000	0.000	0.000 ± 0.000	342.4
pure learning	0.564	0.003	0.234 ± 0.017	371.2
const-ϕ=0.10	0.687	0.021	0.296 ± 0.020	371.3
const-ϕ=0.20	0.719	0.044	0.322 ± 0.025	371.2
const-ϕ=0.30	0.738	0.077	0.350 ± 0.031	371.1
const-ϕ=0.40	0.764	0.110	0.380 ± 0.035	370.6
const-ϕ=0.50	0.766	0.164	0.412 ± 0.042	370.3
const-ϕ=0.60	0.823	0.214	0.465 ± 0.045	369.6
const-ϕ=0.65	0.813	0.246	0.480 ± 0.048	369.0
const-ϕ=0.70	0.818	0.240	0.478 ± 0.048	368.2
const-ϕ=0.75	0.819	0.234	0.476 ± 0.048	367.6
const-ϕ=0.80	0.835	0.216	0.471 ± 0.047	366.6
const-ϕ=0.90	0.791	0.099	0.384 ± 0.038	361.6
const-ϕ=1.00	0.000	0.000	0.000 ± 0.000	108.36
greedy	0.978	0.533	0.716 ± 0.030	364.9
tracking	0.934	0.727	0.812 ± 0.038	369.6

Table 6.4: Mean (N=100) success rates for the ϕ-management strategies. Shaded cells are within the 95% confidence interval of the best result for that measurement. The confidence interval is shown for the whole period performance. The average final elite fitness is also shown.

6.3.1 Pure Learning (Exploration) Strategy

Whether or not the pool fitness ultimately exceeds the desired AAF will strongly influence the success rate. Realistically, if $\lim_{g\to\infty} F_{pool}(g, \phi = 0) > F_{accept}(g)$, then there is a reasonable chance that the problem is somewhat trivial since a maximally learning population can achieve the acceptable performance. Such a situation is reflected by the first level of AAF since all ϕ-management strategies during the period $150 \leqslant g \leqslant g_{change}$. As expected, the pure learning strategy performs the worst of all strategies since knowledge of desired fitness was not exploited to improve the success rate.

A much more likely scenario is where $\lim_{g\to\infty} F_{pool}(g, \phi = 0) << F_{accept}$, which would result in the acceptable fitness never being achieved and a success rate of zero. The second AAF level is somewhat demonstrative of this, yielding a dismal $P_{success}(501, 1000) = 0.003$.

While the pure learning strategy performs poorly against the success rate metric it

does yield an equal highest[3] exploitation potential, indicating, as expected, that this strategy provides the best exploration.

6.3.2 Constant FI Parameter Strategy

The main advantage of a constant-ϕ strategy is that it provides improved average fitness while retaining the ability to adapt and improve through learning. In an ideal scenario where only the rate of learning was impeded by exploitation, the average fitness would ultimately converge to:

$$F_{pool}(g, \phi = x) = F_{pool}(g, \phi = 0) + xF_{\Delta_{ep}}(g, \phi = 0) \tag{6.6}$$

when $0 < x < 1$ and g is large.

From Table 6.5 we see that $F_{pool}(1000, \phi = 0) = 352.3$ and we calculate $F_{\Delta_{ep}}(g, \phi = 0) = 18.9$. By substituting these values into Equation 6.6 we can calculate the achieved $\phi_{actual}(\phi = 0, 1000)$ for the various strategies. If Equation 6.6 was true, it would mean that changing ϕ had no impact on the learning rate and therefore $F_{\Delta_{ep}}(g, \phi = 0)$ could be proportionately exploited by ϕ. As such, then one would expect $\phi_{actual}(1000) = \phi$. However, we can see from Table 6.5 that this is not case for higher values of ϕ which is likely due to the reduction learning rate (manifested by a lower F_{Elite}) caused by the loss of population diversity.

This should in theory improve the chance of meeting the AAF, however with no appreciation of Φ, knowledge of what the AAF is over time is not exploited. As such there will be instances where the AAF could have been met but was not because ϕ was too low, and conversely situations where the AAF was easily met and therefore there was lost learning potential.

This could in turn keep the pool fitness lower than the AAF and generate an accumulating regret (lost performance compounded with time.

[3]within the 95% confidence interval of the highest average

6.3.3 Greedy Strategy

The greedy strategy performed well against the first AAF since in most runs $F_{elite}(150, \phi = 0) > F_{accept}(150)$ and so the success metric could be met for the complete duration up to g_{change}. However, this approach fails to exploit the learning potential during periods where the AAF is being met, in this case during the period $150 \leqslant g < g_{change}$. This results in less exploitation potential at generation g_{change} (344.740 as compared with 351.636 generated by the tracking strategy). This means that learning must occur until the new threshold is reached, which impacts the success rate during the start of the second period. As such $P_{success}(501, 1000)$ for the greedy strategy is much less than that of the tracking strategy. (c.f. Table 6.4).

ϕ-Strategy	$F_{Elite}(500)$	$F_{pool}(500)$	$F_{Elite}(1000)$	$F_{pool}(1000)$	$\phi_{actual}(\phi = 0, 1000)$
random search	342.370	109.703	342.370	109.696	-12.84
const-ϕ=0.00	369.080	350.581	371.240	352.346	0.00 (+0.00)
const-ϕ=0.10	368.870	353.346	371.260	355.741	0.18 (+0.08)
const-ϕ=0.20	368.160	354.642	371.190	357.271	0.26 (+0.06)
const-ϕ=0.30	367.440	355.369	371.060	359.246	0.37 (+0.07)
const-ϕ=0.40	366.110	355.739	370.620	360.079	0.41 (+0.01)
const-ϕ=0.50	364.670	356.202	370.310	361.573	0.49 (-0.01)
const-ϕ=0.60	363.370	357.363	369.570	363.380	0.59 (-0.01)
const-ϕ=0.65	362.380	357.370	368.960	363.552	0.60 (-0.05)
const-ϕ=0.70	360.490	356.271	368.180	363.812	0.61 (-0.09)
const-ϕ=0.75	359.960	356.485	367.620	364.127	0.63 (-0.12)
const-ϕ=0.80	358.530	356.028	366.560	363.931	0.62 (-0.18)
const-ϕ=0.90	353.690	352.787	361.590	360.617	0.44 (-0.46)
greedy	344.740	344.740	364.860	363.340	0.58
tracking	365.090	351.636	369.610	365.057	0.67

Table 6.5: Elite and pool fitnesses (N=100) for the ϕ-management strategies after each AAF target period. The final column shows the average "actual ϕ" observed for each strategy based on the pool fitness achieved of the pure learning strategy at generation 1000 with the variation from the ideal behaviour (shown in parenthesis) where applicable.

6.3.4 FI Tracking Strategy

The $\phi_{tracking}$ strategy demonstrates the best performance for the second AAF threshold and best overall average success rate. It also demonstrates on average a higher final elite

fitness (and higher exploitation potential) suggesting this strategy is more likely to meet increased AAF in the future. The enhanced exploitation potential available to this strategy was in fact exploited at g_{change} which ultimately produced the improvement that resulted in this strategy achieving superior performance.

6.4 Discussion

The purpose of this chapter was to determine whether dynamically managing the performance to minimally meet the AAF (and hence maximise learning rate) would provide a better success at meeting a rise in AAF a later time. Intuitively, it felt that the ϕ-tracking strategy would outperform all other approaches, including the other dynamic strategy of ϕ-greedy, since it was "designed" to minimally meet the AAF and so should therefore maximise learning and improve the potential to meet subsequent increases in AAF.

Interestingly however, ϕ-greedy performed reasonably well, even outperforming the ϕ-tracking strategy during the first AAF period. This makes ϕ-greedy a compelling choice since it is parameterless and simple to implement.

Essentially 3 strategies for manipulating ϕ were compared: a constant ϕ value (including 0), a greedy approach to meeting AAF (greedy strategy) and a dynamic balancing strategy (ϕ-tracking). The experiment was somewhat contrived in that the AAF selected, in particular the second level from $g > g_{change}$ was selected in the knowledge that a pure learning configuration would perform poorly. Surprisingly the simple and somewhat naive 'greedy' strategy performed quite well. Given this approach requires no parameters and is easily implemented, such a strategy may be well suited in various scenarios.

This however does not discount the need for enhanced balancing of exploration and exploitation, particularly when it is possible to have prolonged periods where the AAF can be satisfied.

The exploitation potential, Δ_{ep}, which arises from the difference in fitnesses between the elite and population average (pool fitness), was identified as a key aspect in the control of exploration-exploitation. Other aspects such as the rate of change of the elite and pool fitnesses can also be used to effect the acceptable average fitness, however their impact

is typically far less than that of exploiting Δ_{ep}.

The value of Δ_{ep} and whether it can be exploited to achieve acceptable performance depends on the:

1. Problem type
2. Problem representation
3. Definition of acceptable fitness
4. Evolutionary trajectory

The tracking strategy performance depends critically on the parameters chosen. These will be problem dependant and it would be unlikely that the optimal values could be known *a priori*. Since FI reduces the population size used for exploration, this creates an increase in selection pressure since the number offspring permissible is reduced. Clearly asking the impossible of $F_{accept} > F_{optimal}$ would guarantee acceptable performance is never obtained and so this study restricts the range of acceptable fitness to $0 \leq F_{accept} \leq F_{optimal}$. However, in many real world scenarios $F_{optimal}$ will not be known or even attainable, and so arbitrarily assigning F_{accept} could unknowingly be above what is actually possible to achieve. This is a significant drawback of using absolute fitness values for an acceptability constraint and highlights the attractiveness of the relative acceptability constraint that can be specified with the Fitness Importance hueristic.

6.5 Conclusion

FI is a useful metaheuristic only when there is a significant difference between the best achievable solution (elite fitness) and that of the average solution (pool fitness). In many real world applications, desired acceptable performance will rarely be specified in terms relative to the ongoing system performance. i.e. Asking the system to perform at 50% of its capability, regardless of what the current capability is. More likely, desired performance will be specified in absolute fitness terms or in relative terms to a performance capability of a specific point in time (i.e. Asking to perform at 70% of the Elite performance as

measured at generation 100). For these scenarios, a FI tracking mechanism will be desirable since it has the potential to minimally meet the desired performance when possible, and maximise learning capacity into the future. This has been demonstrated with the dynamic ϕ-management strategies ϕ_{track} and ϕ_{greedy} which were shown to outperform the non-dynamic ϕ-management strategies. The greedy approach, with its simple implementation of exploiting 100% when achieving acceptable performance is possible, performed as well as with the more complex algorithm which attempted to balance FI in a continuous manner. Regardless of which strategy is employed, it is likely that dynamic ϕ-management will be necessary to utilise the Fitness Importance heuristic in real world applications and further research into such strategies across various scenarios is recommended.

7 Conclusion

7.1 Summary and Conclusions

This thesis provided a mechanism for achieving online genetic programming (GP) on highly resourced-constrained, wirelessly networked devices in order to achieve acceptable behaviour. A survey of related research (Chapter 2) revealed that previous research had each only addressed various subsets of this challenge, however it also highlighted the promising benefits of each of the aspects.

Specifically, it highlighted that learning in the target environment (i.e. in situ) could alleviate the need for simulating the real world and avoid the 'transference" problem where logic evolved with simulation breaks when it is deployed in the real world. It also suggested the rich representation offered by GP could potentially automatically generate any logic the devices are capable of since programs are the native logic representation on

such devices. Furthermore, it was identified that GP implementations could effectively implement numerous other learning mechanisms such as recurrent neural networks and rule-based logic.

Approaches that employed distributed evolution demonstrated benefits of higher diversity throughout evolution, typically evolving faster and to better final solutions (e.g. the runs shown in Figure 3.11). Furthermore, as per [79], even though the population locally on an individual node is small (due to device resource/memory constraints), employing the Island Model demonstrated evolutionary trajectories on the individual consistent with much larger population sizes. As such, for networked devices, the limitation on local population sizes need not necessarily significantly hinder the learning rate.

These aspects were considered within the scope (Section 3.1) of the most resource constrained class of embedded systems known as Wireless Sensor Network (WSN) “motes”. Requiring a solution that worked on such highly constrained devices was justified by the argument that if it could be achieved on such a constrained class of device, then it would also be applicable to embedded systems with greater resources.

Chapter 3 presented In situ Distributed Genetic Programming (IDGP) as a framework for achieving distributed evolution using GP on highly resource-constrained devices. The design (Section 3.2) of the framework provided various considerations for embedded systems engineers to develop their own distributed GP implementations on networks of embedded devices. The framework was then implemented (Section 3.3) for the Fleck3b mote - a prototypical highly resource-constrained WSN device.

The mote implementation was evaluated using local time-varying sensing-actuation problem (Section 3.4) and employs distributed evolution for faster learning. A second experiment scenario (Section 3.5) demonstrated multiple devices coordinating by evolving simple communications in order to meet a global objective. The surprise that devices in the initial communications experiment evolved a “denial-of-service” attack to prevent negative fitness feedback served as a cautionary tale of how important the correct specification of the objective function is and that GP can generate novel solutions, often unimagined by the user.

However, while in situ distributed evolution on resource-constrained devices was successfully demonstrated, it was evident that the population-based nature of GP resulted in undesirable behaviour most of the time while evolution was occurring. Ideally the opposite of this, i.e. desirable behaviour most of the time, would be much preferred. This motivated defining an acceptable average fitness (AAF) (Section 4.3.1) as a measure of acceptable performance for population-based learners. However, since AAF is a target expressed in absolute fitness terms, it may or may not be achievable with the current population, specifically if the AAF target is higher than the elite fitness. Therefore, an additional, representation, ϕ, was devised to express the AAF *relative* to the current population characteristics as a percentage value between the pool fitness and the elite fitness. This is a more pragmatic approach since 100% corresponds to the best (elite solution) fitness available within the current population, while 0% is the average (default) fitness of the population.

Fitness Importance (FI) or $\Phi(g)$ (Chapter 4) was then presented as a heuristic to convey the AAF (or desired ϕ) as a function of time or, more specifically in this thesis, as a function of the generation epoch. FI provides contextual information about the importance of demonstrating a level of desired average fitness which is additional to the standard fitness information supplied by a fitness function. FI can be viewed as a satisficing constraint on the average fitness. However, unlike typical use of a satisficing constraint, which would simply focus on meeting the target AAF, FI is used by a population generator metaheuristic (Section 4.4) to attempt to minimally meet the AAF target in order to maximise the learning capacity for continued evolution (given the constraints resulting from meeting the AAF target). This provides the potential benefit of immediately meeting a subsequent increase in AAF and thereby avoiding the period where the new AAF target is not being met while the necessary learning occurs.

FI was integrated into IDGP framework (Section 4.5) by employing a simple implementation of a population generator that biases the average evaluated performance within a generation by evaluating the elite solution enough times so that the desired average fitness is achieved. Furthermore, it redistributes the remaining evaluations as a reduced

learning population. Importantly, the reduced learning population retains a scaled representation of the diversity of "classes" within the population where population classes are defined by the combination of selection and genetic operators used to generate new solutions. Namely, the classes employed were elites, mutations of highly ranked solutions, children (crossover and mutation with fitness proportion-biased parent selection), randoms and "others" (injected programs shared from other devices via the Island Model or potentially human-devised logic).

In Chapter 5, numerous experiments were performed to observe the response of applying a constant values of ϕ at various times during evolution, and gain an intuition of how learning and performance were affected during evolution. It became evident that the benefit of immediate improvement in the average fitness came at the expense of the learning capacity of the system. It was hypothesised that for scenarios where there were increases in the AAF target over time, that dynamically adjusting ϕ to minimally meet the AAF target would be superior to simply applying a constant-ϕ approach for a balanced learning-performing approach.

In Chapter 6, an experiment was devised to empirically determine whether such a scenario exists. A simple, yet complex enough objective function such that GA outperforms standard hill climbing techniques, was employed to benchmark both constant-ϕ and dynamic ϕ-management strategies. Two discrete AAF levels were designed as targets (the second higher than the first) over a finite number of generations with the performance metric as the rate at which the AAF was successfully met over the evolutionary run. It was found that the dynamic ϕ-management strategies, which exploited the additional context of AAF provided by the FI heuristic, did outperform the constant-ϕ strategies for the devised problem.

It was hypothesised that minimally meeting the AAF would perform better than a simple "greedy" approach of employing the elite to meet the AAF when possible and switching to a pure learning strategy when it was not possible. Surprisingly, the "tracking" approach was not shown to outperform the "greedy" approach with any statistical significance for the devised problem, however this does not discount the possibility that in other problems it could.

In conclusion, this thesis has provided a framework to achieve in situ learning using distributed genetic programming on highly resourced-constrained, networked devices. Furthermore, it provided heuristic to describe, and metaheuristic to achieve, acceptable online performance with population-based learners. The additional investigations of the use of the Fitness Importance heuristic demonstrated the challenges and benefits of balancing learning and performance with population-based approaches. The combination of In situ Distributed Genetic Programming with Fitness Importance provides a novel approach towards simultaneously achieving life-long learning and acceptable average performance on resource-constrained embedded systems. As such, this could be used by a variety of networked embedded systems, including robotic swarms, WSN motes and IoT devices, to cooperatively evolve their logic in order to meet dynamic performance requirements in real world environments.

7.2 Future Research

Deliberately this thesis employed the near-simplest scenarios to investigate various aspects of the framework and heuristics. Applying IDGP and FI to “real” problems that are more likely to be on the "near-most complex" side of the problem spectrum is however an obvious next step for this research. In such situations it will probably be pragmatic to seed the evolution with human devised solutions to gain immediate performance by leveraging the domain knowledge of the user. However, seeding populations is also likely to reduce the novelty of solutions found and so striking the right balance of injecting domain knowledge with allowing the system to devise its own novel approaches would be a worthwhile investigation. Furthermore, many “real” problems are likely to have dynamic fitness landscapes, caused by dynamic and unpredictable changes in the environment. In this thesis however, FI was only investigated in context to a dynamic AAF target and not with respect to a dynamic fitness function. Considering this produces 4 distinct problem scenarios, of which only 1 has been explicitly investigated:

1. Constant fitness function with a constant AAF (implicitly studied in this thesis)

2. Constant fitness function with a dynamic AAF (studied in this thesis)
3. Dynamic fitness function with a constant AAF
4. Dynamic fitness function with a dynamic AAF

How the existing ϕ-management strategies perform under these other conditions will be interesting to observe.

There is substantial opportunity for developing significantly more sophisticated ϕ-management strategies. The effectiveness of ϕ-management strategies is greatly affected by the characteristics of the populations generated by the population generator, γ. This thesis provided a basic implemented of γ that ensures the pool fitness is raised to the required level, however it must, by its design, overshoot the target performance and this decreases the learning capability more than neccessary. Ideally, a population generator should generate a population that exactly and minimally meets the AAF target with a complete population that maximises the learning capacity (unlike γ_{simple} which effectively reduces the learning population size). Finding such a population is N-P hard however and so as the population size increases, search metaheuristics will need to be employed. Additionally, one should try to preserve the "anytime-algorithm" nature of supplying a biased population.

Another obvious research direction is investigating the applicability of IDGP and/or FI to global (network-wide) objective functions. In this research, only a two-device problem was investigated for cooperative evolution. How FI and IDGP scale as N increases for global objective function is unknown. Preliminary consideration suggests it would require much higher ϕ per device to allow coordination of cooperation since uncoordinated evaluations from a network of populations with high diversity will result in good solutions from some devices being evaluated (globally) with bad solutions from other nodes. There are also likely to be a number of options for coordinating and/or changing how populations are represented that could be better suited to global network objective functions.

In retrospect, it was realised that a number of the related work approaches could be implemented with, or benefit from employing, FI and its accompanying population generator. As an example, online approaches that use incremental learning, say applying mutation only and sharing elites, are effectively equivalent to employing a ϕ that is close to 1. For

example, with a population size of 100 candidate solutions, $\phi = 0.99$ would evaluate the current elite 99 times plus one other candidate solution generated by applying mutation only to a highly ranked individual (assuming γ_{simple} as the population generator). As such, many strategies could be implemented by simply employing the appropriate population generator and population classes (defined with the combinations of genetic and selection operators). Demonstrating that the equivalent functionality of existing frameworks could be implemented as a specific instance of IDGP and FI would these frameworks to be extended with an additional EE control mechanism. How such frameworks would behave as FI is varied would be intriguing and potentially reinvigorate research on a number of the frameworks.

This thesis provides rich grounds for many avenues of research and, as autonomous systems such as robots, Internet of Things, and even logic on smart phones increase in their ubiquity, the opportunity grows for distributed evolution of logic via frameworks such as In situ Distributed Genetic Programming.

A

Mathematical Nomenclature

Symbol	Description
p_i	Unique population i where $1 \leq i \leq N_P$
N_P	Number of unique populations
P	Set of all possible unique populations $p_1, p_2, ..., p_{N_P}$
g	Generation number where $g \geq 1$
$\mathcal{G}(p_i)$	The set of genetic operations which transforms the current population into the next generation population. $\mathcal{G}(p_i) \mapsto p'_i(2)$, ... $\mathcal{G}(p'_i(g)) \mapsto p'_i(g+1)$
$p'_i(g)$	Optimisation trajectory (as successive populations) given the initial population p_i assuming $\mathcal{G}$ is deterministic and the environment is not dynamic. Note $p_i = p'_i(1)$
N	Problem dimension
$N_{pop}(p_i)$	Number of programs in population p_i
$N_E(p_i)$	Number of elite programs in p_i
$N_H(p_i)$	Number of "high-ranked" (mutated elite) programs in p_i
$N_C(p_i)$	Number of "children" (biased selection + mutation + crossover) in p_i
$N_R(p_i)$	Number of "random" programs in p_i
$N_O(p_i)$	Number of "other" (externally sourced) programs in p_i
N_Q	Number of subpopulation types
$\Phi(t)$	Fitness Importance function. Typically where $0 < \Phi(t) < 1$
ϕ_{accept}	acceptable instantaneous Fitness Importance $0 < \phi_{accept} < 1$
$\phi_{achieved}$	Achieved instantaneous performance improvement ratio. Note $\phi_{achieved}$ may be < 0 or > 1
$F_{pool}(\phi, k, g)$	Pool Fitness at generation g, given ϕ was applied at generation k
$F_{\Delta ep}(\phi, g)$	Fitness Exploitation Potential defined by $F_{elite}(\phi, g) - F_{pool}(\phi, g)$
$L_{pool}(g)$	Pool improvement rate defined by $L_{pool}(\phi, g) = \frac{d}{dg} F_{pool}(\phi, g)$
$L_{elite}(g)$	Elite improvement rate defined by $L_{elite}(\phi, g) = \frac{d}{dg} F_{elite}(\phi, g)$
Q	Set of subpopulation types {E,H,C,R,O,...}
Γ	Set of all unique population structures of size N_P
$\Upsilon(\phi_{accept}, p_i)$	Set of subpopulation types that provide an acceptable performance given a particular population

Table A.1: Symbols

B

Derivation of the Reduction of Learning Rate due to $\phi > 0$

When the Fitness Importance (FI) heuristic is applied at generation k, the new population $pool^*$ is composed of ϕ $elite$ programs and $1 - \phi$ $pool$ programs. This effectively reduces the population size available for learning which in turn reduces the learning rate.

Let us define the new post-FI $elite$ and $pool$ fitnesses in terms of the pre-FI $elite$ and $pool$ fitnesses and their respective post-FI reduced learning rates.

$$F_{elite}(\phi_d, g) = F_{elite}(0, k) + \lambda_{elite} \int_k^g L_{elite}(0, g)$$
$$F_{pool}(\phi_d, g) = F_{pool}(0, k) + \lambda_{pool} \int_k^g L_{pool}(0, g)$$

Thus the new population structure, post-FI, will have a fitness of

$$\begin{aligned}
F_{pool^*}(\phi_d, g) &= \phi F_{elite}(\phi_d, g) + (1-\phi) F_{pool}(\phi_d, g) \\
&= \phi\left[F_{elite}(0,k) + \lambda_{elite}\int_k^g L_{elite}(0,g)\right] + (1-\phi)\left[F_{pool}(0,k) + \lambda_{pool}\int_k^g L_{pool}(0,g)\right] \\
&= \phi F_{elite}(0,k) + \phi\lambda_{elite}\int_k^g L_{elite}(0,g) + (1-\phi)F_{pool}(0,k) + (1-\phi)\lambda_{pool}\int_k^g L_{pool}(0,g)
\end{aligned}$$

$$F_{pool^*}(\phi_d, g) - \phi F_{elite}(0,k) - (1-\phi)F_{pool}(0,k) = \phi\lambda_{elite}\int_k^g L_{elite}(0,g) + (1-\phi)\lambda_{pool}\int_k^g L_{pool}(0,g)$$

if the elite and pool learning rates are effected by the same proportion such that $\lambda_{elite} = \lambda_{pool} = \lambda$, then

$$\begin{aligned}
F_{pool^*}(\phi_d, g) - \phi F_{elite}(0,k) - (1-\phi)F_{pool}(0,k) &= \lambda\left[\phi\int_k^g L_{elite}(0,g) + (1-\phi)\int_k^g L_{pool}(0,g)\right] \\
\frac{F_{pool^*}(\phi_d, g) - [\phi F_{elite}(0,k) + (1-\phi)F_{pool}(0,k)]}{\phi\int_k^g L_{elite}(0,g) + (1-\phi)\int_k^g L_{pool}(0,g)} &= \lambda \qquad \text{(B.1)}
\end{aligned}$$

Equation (B.1) can be interpreted in the following way: The numerator of the LHS describes the fitness improvement since FI was applied because

$$\phi F_{elite}(0,k) + (1-\phi)F_{pool}(0,k) = F_{pool}(\phi_{accept}, k) \qquad \text{(B.2)}$$

The denominator of equation (B.1) represents the amount of learning that would have occurred with the new population structure $pool^*$ if there was no reduction in learning rate. i.e. if $\phi = 0$

$$L_{pool^*}(0,g) = \phi L_{elite}(0,g) + (1-\phi)L_{pool}(0,g) \qquad \text{(B.3)}$$

Substituting equations (B.2) and (B.3) back into equation (B.1), we discover that λ

describes the proportion that the $\phi = 0$ learning rate is diminished by when $\phi = \phi_{accept}$

$$\begin{aligned} \lambda &= \frac{F_{pool^*}(\phi_d, g) - F_{pool^*}(\phi_d, k)}{\int_k^g L_{pool^*}(0, g)} \\ \lambda &= \frac{\int_k^g L_{pool^*}(\phi_{accept}, g)}{\int_k^g L_{pool^*}(0, g)} \end{aligned} \tag{B.4}$$

$Q.E.D.$

C

Estimation of Expected Pool Fitness

We can estimate $F_{pool_0}(g)$ if we know the ϕ_d that was used to generate the observed pool $F_{pool_d}(g)$ and elite $F_{elite_d}(g)$ as :

$$\begin{aligned}
F_{pool}(g) &\approx \phi_d[F_{elite_d}(g-1) - F_{pool_0}(g-1)] + F_{pool_0}(g-1) \\
F_{pool}(g) &\approx \phi_d F_{elite_d}(g-1) + (1-\phi_d)F_{pool_0}(g-1) \\
F_{pool}(g) - \phi_d F_{elite_d}(g-1) &\approx (1-\phi_d)F_{pool_0}(g-1) \\
F_{pool_0}(g-1) &\approx \frac{F_{pool}(g) - \phi_d F_{elite_d}(g-1)}{1-\phi_d} \qquad \text{(C.1)}
\end{aligned}$$

If $F_{pool_0}(g) \approx F_{pool_0}(g-1)$ and $F_{elite_d}(g) \approx F_{elite_d}(g-1)$ then we can express the estimated pool fitness if FI wasn't applied in terms of only the current generation fitnesses as:

$$F_{pool_0}(g) \approx \frac{F_{pool}(g) - \phi_d F_{elite_d}(g)}{1-\phi_d} \qquad \text{(C.2)}$$

Unfortunately, the $1-\phi_d$ term on the denominator of Equation C.1 and Equation C.2 results in a discontinuity and worse it exponentially amplifies any variations in the numerator as ϕ_d approaches 1. Intuitively, this makes sense since we are attempting to predict F_{pool_0} with increasingly less representation of the learning pool (since the typical pool population is being increasingly replaced by the elites).

Therefore the predicted F_{pool_0} as ϕ_d approaches 1 rapidly becomes erroneous with any small variation with the achieved pool fitness compared to the desired pool fitness or error in elite fitness used. Even for a modest variation of the achieved elite or pool fitness from what was expected, means reconstructing the likely pool fitness without ϕ becomes very erroneous as the ϕ used to generate the population approaches 1. See Figure C.1

The loss of accuracy about the pool is exacerbated by the averaging of the fitnesses to obtain the pool fitness. In practice is may be better to keep the learning population separate and monitor that pool for an estimation of the pool, but never-the-less even this estimation will become skewed and erroneous as the learning population size decreases (as ϕ approaches 1).

Note that $F_{pool}(g) \geq \phi_d F_{elite_d}(g-1)$ is necessary to ensure the fitness remains positive. This can readily be verified since $F_{pool}(g)$, ϕ_d and $F_{elite_d}(g-1)$ are all known.

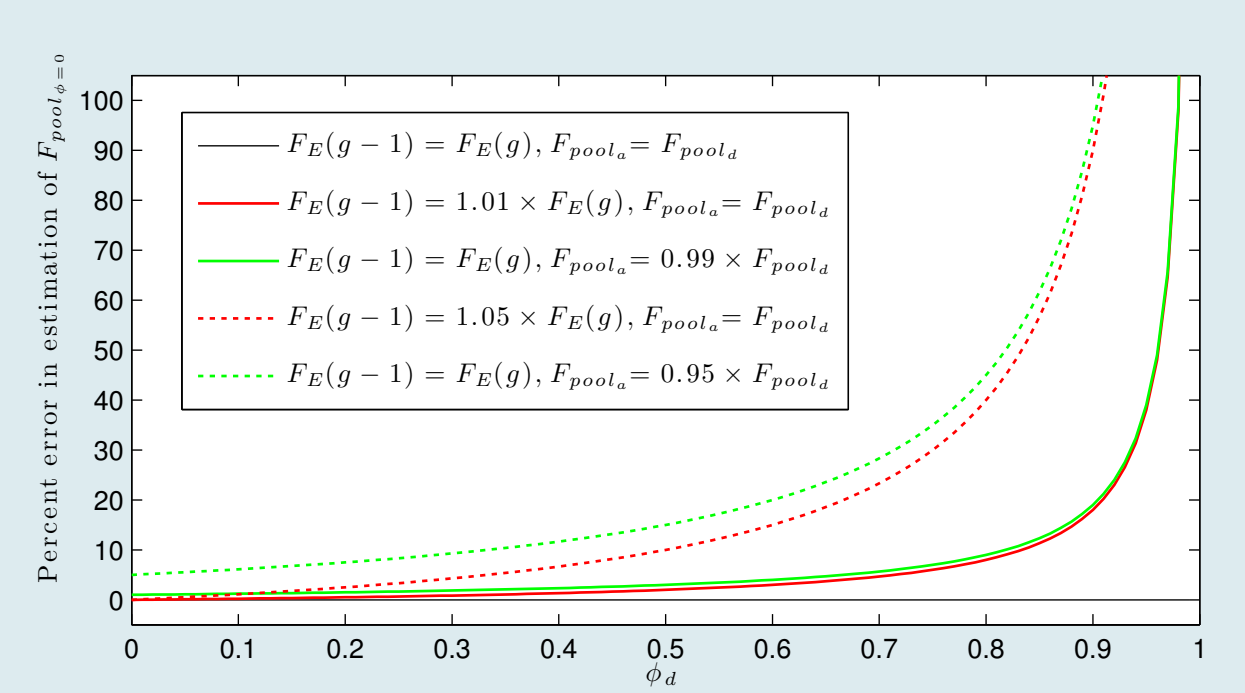

Figure C.1: Variations in the achieved pool fitness, F_{pool_a} from the desired pool fitness, F_{pool_d}, as well as variations in the achieved Elite fitness, $F_{elite}(g)$ from that of the previous generation, can significantly affect the error in estimation of $F_{pool_{\phi=0}}$. Variations of approximately 1% and 5% are shown.

D

Design of the Acceptable Average Fitness (AAF)-Tracking Strategy

The main disadvantage of the ϕ_{greedy} strategy arises from the lack of learning during extended periods where the Acceptable Average Fitness (AAF) is met, but no learning occurs. Unfortunately, the point at which a solution is first discovered that can meet the AAF is typically when $F_{elite} = F_{accept}$.

Assuming $F_{pool}(0, g) << F_{elite}(0, g)$, then based on Equation 4.3.2, ϕ will need to be close or equal to 1 in order to meet the AAF at that point. This in turn will permit little or no diversity in the population for learning and as a consequence $F_{pool}(x \approx 1, g) \approx F_{elite}(0, k)$ where k is the first generation where $F_{elite}(0, k) \geq F_{accept}(0, k)$

Therefore in order to permit some learning, the solution finder must necessarily keep $\phi < 1$. We term this upper limit as ϕ_{max} . While $\phi_{max} < 1$, it should ideally be close to 1

otherwise it will suffer the disadvantage of the constant-ϕ strategy in that the maximum obtainable fitness is limited by ϕ c.f. Equation 6.6. Through implementing the ϕ_{max} parameter, learning can be achieved during long periods and the major shortfall of the offline strategy avoided.

Ideally learning will occur such that after some period $F_{pool}(\phi_{max}, g) >> F_{accept}$. If this occurs however, then it is likely that a lower ϕ value would have sufficed to meet the AAF and hence more learning could have been achieved. The appropriate value of ϕ that should minimally achieve the AAF (ignoring any learning which may or may not occur) is calculable from Equation 4.6 as:

$$\phi_{accept} = \frac{F_{accept}(g+1) - F_{pool}(0,g)}{F_{elite}(g) - F_{pool}(0,g)} \tag{D.1}$$

The AAF, $F_{accept}(g+1)$, and the current elite fitness $F_{elite}(g)$ are known, however unfortunately $F_{pool}(0,g)$ is not measured and only $F_{pool}(x,g)$ is known, where x was the value of ϕ used in generation g. An estimation of $F_{pool}(0,g)$ can be determined however (c.f. Appendix C - note that the $\phi_d = \phi_{accept}$ in this case since the desired value is one which achieves acceptable performance) and used to estimate an appropriate value for ϕ_{accept}.

Naively, one could expect that calculating the ϕ_d that would achieve the desired fitness in the next generation (if possible, i.e. the $F_{elite} > F_{accept}$) and setting $\phi = \phi_d$ would achieve the optimal performance. However, targeting to meet the minimum acceptable performance based on a algorithm with stochastic characteristics generates an expected fitness with a deviation spread around the target fitness. If this spread was gaussian for instance, then roughly half of the time the desired fitness would not be met. Clearly this is unacceptable and so another parameter, ζ, is introduced to bias the expected fitness such that even with stochastic variation, the average fitness achieved will typically be above F_{accept}. Intuitively, the ζ parameter effectively causes the system to overshoot the required AAF and so is referred to as overshoot parameter. The overshoot parameter is designed to add a percentage of available Δ_{ep} since the degree to which it is possible to overshoot the acceptable fitness is not known a priori. For instance you cannot guarantee that $1.2F_{pool}$ will ever be attainable, however $F_{pool} + 1.2\Delta_{ep}$ is obtainable by setting $\phi = 0.2$.

As previously mentioned, the achieved fitness may differ from the desired fitness due to the stochastic characteristics of the solution finder employed (EA). Additionally, there will be some error in the ϕ_d employed since it is based on an estimation of $F_{pool}(0, g)$ (c.f. Appendix C, Equation D.1). Finally, a further error due to the rounding bias arising from the discrete nature of the population size is introduced when constructing the population with $\Gamma_{simple}(\Phi_{strategy})$. These errors can cause the achieved fitness to undershoot and overshoot the desired fitness. As the ϕ applied approaches 1, the error in estimation of $F_{pool}(0, g)$ increases exponentially (as seen in Figure C.1), which could lead to overestimation of the required ϕ_{accept}. High values of ϕ will significantly impact the diversity of the population, which apart from reducing learning potential of the population, also reduces the exploitation potential, Δ_{ep}. Since the estimation of $F_{pool}(0, g)$ is dependant on Δ_{ep}, this can feedback into the error is estimating an appropriate value for ϕ_{accept} and generate an erratic (unstable) control of ϕ. To counter the oscillation generated via this feedback mechanism, a parameter is needed to dampen the rate at which ϕ can be changed. Due to the binary nature of the success rate performance metric, it may be beneficial to bias the controlling mechanism to be able respond faster to the need for increases in ϕ rather than decreases in it. This will be problem specific and largely depend on how important longer term learning is over short term performance losses. In any case, the amount that ϕ can either increase or decrease is limited by α and β respectively.

E

Usefulness of Randoms

One could argue that if the world changed catastrophically, then evolved / niched populations wouldn't provide an optimal search strategy and you'd be better off reinitialising the GP process - i.e. starting with a new random population. Having a few randoms constantly in the population structure could be a sanity check to ensure that the population isn't performing worse than random (e.g. specialisation to a behaviour which is now bad) and at very least it offers the promise of always injecting fresh genetic material.

The situation where randoms become as good (or even better) than niched populations could arise in Nature when resources become abundant. For example, with the Blink3 problem when the nodes are outside of the bag suddenly a random program that doesn't turn the LEDs on at all becomes fitter than previous solutions evolved to achieve good performance in the bag. For Blink3 this is due to the power savings made from no LEDs while still achieving high levels of light into the photodiodes from ambient light. With a

heavily niched population (i.e. low diversity) one could easily imagine that all solutions likely turn a LED on since this was critically important inside the bag where no ambient light is present. Outside of the bag, all solutions perform well, however better solutions won't evolve until a child or mutant 'learn' to switch the LED off early and keep it off. In this situation it would be faster to have a random program present which does nothing with the LEDs to shift the population.

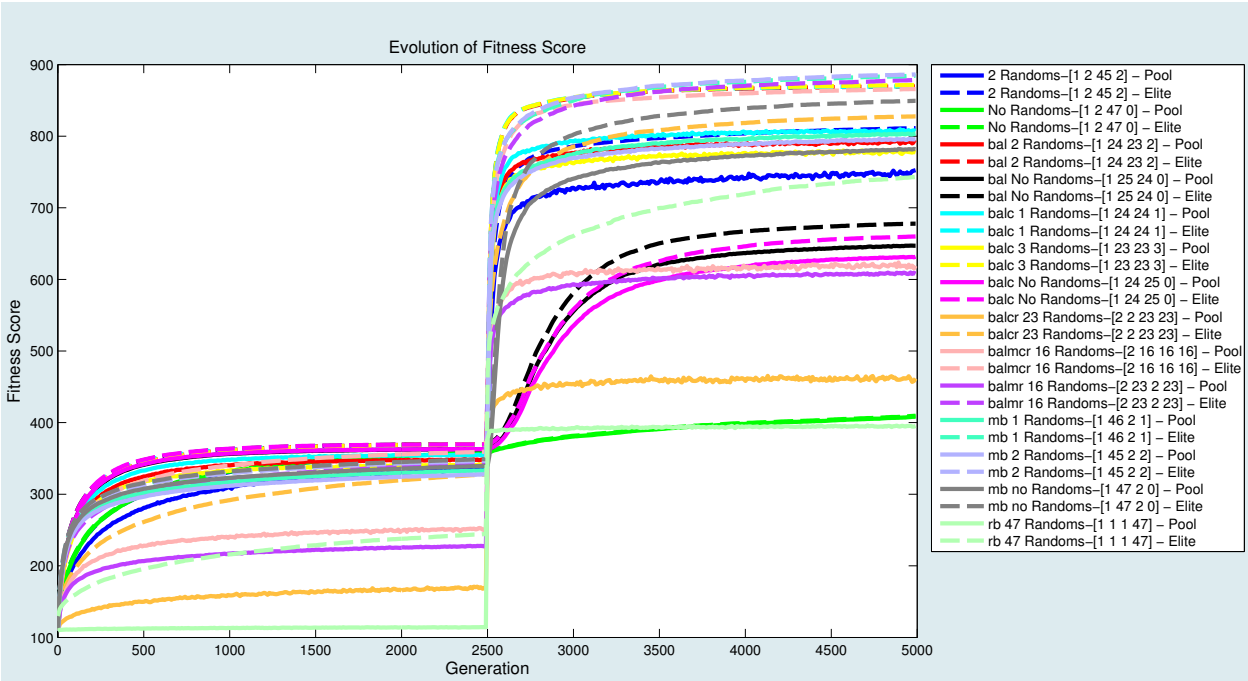

Figure E.1: For the "Concat-V-abundance@2500" objective function, it is beneficial to include "Random" individuals in the population.

The hypothesis that injecting random individuals throughout online evolution was explored empirically by devising an objective function which simulated a dramatic environment / fitness function half way through the lifetime of the system. Figure E.1 shows the averaged (N=100) elite and pool responses of population structures with differing concentrations of random programs when evolved against the "Concat-V-abundance@2500" objective function.

The "Concat-V-abundance@2500" objective function is essentially the standard Concatenated-V (k=7, M=100) problem up to generation 2500 (where M is number of k-sized blocks concatenated together). By generation 2500, many populations have discovered reasonable fitnesses and the diversity of the populations have decreased. After generation 2500, the

block function (k=7) changes to a success function based on whether the number ones within the block equal 3 (the success function is also scaled by a constant). A block in a randomly generated individual has a 27.34375% probability of achieving the optimal block fitness.

In this scenario, having a few randoms clearly provides superior learning and performance to that of having no random individuals in the population. Employing 3 randoms however is not significantly better than not employing random individuals, while employing significant numbers of randoms performs poorly across the entire objective function (as expected).

This experiment demonstrates that scenarios can exist where it is useful to have random individuals generated throughout the life of the system. This does not imply however that such a strategy is necessarily beneficial in real world online problem scenarios. Whether this is true warrants further investigation.

F

ϕ-Tracking Parameter Sweep

$\zeta\alpha\beta\phi_{max}$	$P_{success}(150, 500)$	$P_{success}(501, 1000)$	$P_{success}(150, 1000)$
dt00999999	0.575	0.206	0.359
dt10020290	0.812	0.484	0.620
dt10020295	0.814	0.529	0.646
dt10020296	0.807	0.526	0.642
dt10020297	0.807	0.529	0.643
dt10020298	0.807	0.529	0.643
dt10020299	0.817	0.528	0.647
dt20050199	0.923	0.706	0.795
dt20070199	0.935	0.682	0.786
dt20080199	0.937	0.688	0.791
dt25050199	0.933	0.717	0.806
dt25070199	0.939	0.711	0.805
dt30010390	0.834	0.522	0.651
dt30010395	0.836	0.562	0.675
dt30010396	0.839	0.574	0.683
dt30010397	0.841	0.579	0.687
dt30010398	0.839	0.572	0.682
dt30010399	0.839	0.572	0.682
dt30020190	0.883	0.630	0.735
dt30020195	0.886	0.642	0.743
dt30020196	0.889	0.686	0.770
dt30020197	0.889	0.690	0.772
dt30020198	0.891	0.700	0.779
dt30020199	0.892	0.680	0.768
dt30020290	0.880	0.600	0.715
dt30020295	0.889	0.648	0.748
dt30020296	0.885	0.668	0.758
dt30020297	0.890	0.674	0.763
dt30020298	0.889	0.672	0.761
dt30020299	0.891	0.675	0.764
dt30030190	0.900	0.598	0.723
dt30030195	0.903	0.668	0.765
dt30030196	0.910	0.679	0.774
dt30030197	0.910	0.704	0.789
dt30030198	0.909	0.691	0.781
dt30030199	0.912	0.706	0.791
dt30030297	0.909	0.706	0.790
dt30030298	0.912	0.701	0.788
dt30030299	0.908	0.713	0.793
dt30030398	0.906	0.667	0.765
dt30030399	0.906	0.689	0.779
dt30040198	0.924	0.706	0.796
dt30040199	0.923	0.704	0.794
dt30040298	0.920	0.708	0.796
dt30040299	0.921	0.716	0.800
dt30040398	0.919	0.701	0.791
dt30040399	0.919	0.693	0.787
dt30050197	0.934	0.727	0.812
dt30050198	0.933	0.708	0.801
dt30050199	0.934	0.726	0.812
dt30050298	0.931	0.708	0.800
dt30050299	0.929	0.715	0.803
dt30050399	0.929	0.709	0.800
dt30060199	0.938	0.690	0.792
dt30060299	0.936	0.707	0.802
dt30070198	0.940	0.711	0.806
dt30070199	0.940	0.713	0.807
dt30070298	0.938	0.687	0.791
dt30070299	0.938	0.694	0.794
dt30999999	0.875	0.595	0.711
dt35040199	0.922	0.728	0.808
dt35040299	0.924	0.694	0.789
dt35050199	0.933	0.723	0.810
dt35060299	0.934	0.684	0.787

Table F.1: Parameter sweep of tracking strategy performances

References

[1] Lesson 4: Component composition and radio communication. http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson4.html, 2015. [Online; accessed 1 April, 2015].

[2] The Sensor Network Museum. http://www.snm.ethz.ch, 2015. [Online; accessed 22 April, 2015].

[3] Agre, P. E. *The Dynamic Structure of Everyday Life*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1988.

[4] Akyildiz, I. and Kasimoglu, I. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal*, 2(4):351–367, 2004.

[5] Alander, J. T. On Optimal Population Size of Genetic Algorithms. In *Proc. of Computer Systems and Software Engineering (CompEuro'92)*, pages 65–70. IEEE, 1992.

[6] Alsheikh, M. A., Lin, S., Niyato, D., and Tan, H.-P. Machine Learning in Wireless Sensor Networks: Algorithms, Strategies, and Applications. *IEEE Communications Surveys and Tutorials*, 16(4):1996–2018, 2014.

[7] Arlindo, S., Ana, N., and Ernesto, C. Evolving controllers for autonomous agents using genetically programmed networks. In *Proc. 2nd European Workshop on Genetic Programming*, LNCS 1598, pages 255–269. Springer-Verlag, Göteborg, Sweden, 1999.

[8] Auer, P. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2003.

[9] Auer, P., Cesa-Bianchi, N., and Fischer, P. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[10] Banzhaf, W., Nordin, P., and Olmer, M. Generating Adaptive Behavior using Function Regression within Genetic Programming and a Real Robot. In *2nd Int. Conf. on Genetic Programming*, pages 35–43, Stanford, CA, USA, 1997.

[11] Baram, Y., El-Yaniv, R., and Luz, K. Online choice of active learning algorithms. *Machine Learning Research*, 5:255–291, 2004.

[12] Bengio, Y. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[13] Bhanderi, M. V. and Shah, H. B. Machine Learning for Wireless Sensor Network: A Review, Challenges and Applications. *Advances in Electronic and Electric Engineering*, 4(5):475–486, 2014.

[14] Bosman, P. A. N. and La Poutré, H. Learning and anticipation in online dynamic optimization with evolutionary algorithms: The stochastic case. In *GECCO'07 Proc. 9th Ann. Conf. on Genetic and Evolutionary Computation*, pages 1165–1172, London, England, UK, July, 2007.

[15] Brezzi, M. Optimal learning and experimentation in bandit problems. *Economic Dynamics and Control*, 27(1):87, 2002.

[16] Brooks, R. A. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1–2):3–15, 1990.

[17] Bui, L. T., Essam, D., and Abbass, H. A. The Role of Explicit Niching and Communication Messages in Distributed Evolutionary Multi-objective Optimization. In Vega, F. F. and Cantú-Paz, E., editors, *Parallel and Distributed Computational Intelligence*, pages 181–206. Springer, Berlin, Heidelberg, 2010.

[18] Chinapirom, T., Witkowski, U., and Rueckert, U. Vision Module for Mini-robots Providing Optical Flow Processing for Obstacle Avoidance. In *Learning and Adaption in Multi-Agent Systems*, pages 208–219. Springer, Berlin, Heidelberg, 2009.

[19] Chopra, A., Obsniuk, M., and Jenkin, M. R. The Nomad 200 and the Nomad Super-Scout: Reverse engineered and resurrected. In *3rd Canadian Conf. on Computer and Robot Vision*, page 55, Quebec, Canada, 2006. IEEE.

[20] Corke, P., Wark, T., Jurdak, R., Hu, W., Valencia, P., and Moore, D. Environmental Wireless Sensor Networks. *Proc. IEEE*, 98(11):1903–1917, 2010.

[21] Corke, P. and Sikka, P. Demo abstract: FOS - a new operating system for sensor networks. In Verdone, R., editor, *Proc. 5th European Conference on Wireless Sensor Networks (EWSN 2008)*, Bologna, Italy, 2008.

[22] Cotillon, A., Valencia, P., and Jurdak, R. Android Genetic Programming Framework. In Moraglio, A., Silva, S., Krawiec, K., Machado, P., and Cotta, C., editors, *Proc. of Genetic Programming: 15th European Conference (EuroGP 2012)*, pages 13–24, Málaga, Spain, 2012.

[23] Cramer, N. L. A representation for the adaptive generation of simple sequential programs. In *Proc. 1st Int. Conf. on Genetic Algorithms*, pages 183–187, Carnegie-Mellon University, Pittsburg, PA, USA, 1985.

[24] De Jong, K. A. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Computer and Communications Sciences, University of Michigan, Ann Arbor, MI, USA, 1975.

[25] Deng, L. and Yu, D. Deep Learning: Methods and Applications. *Foundations and Trends in Signal Processing*, 7(3-4):197–387, 2014.

[26] Di, M. and Joo, E. M. A survey of machine learning in Wireless Sensor networks From networking and application perspectives. In Qing, S., Imai, H., and Wang, G., editors, *6th Int. Conf. on Information, Communications and Signal Processing (ICICS)*, pages 1–5. IEEE, Singapore, 2007.

[27] Digalakis, J. G. and Margaritis, K. G. On Benchmarking Functions for Genetic Algorithms. *Computer Mathematics*, 77(4):481–506, 2001.

[28] Duff, M. O. Q-Learning for Bandit Problems. In Prieditis, A. and Russell, S. J., editors, *Proc. 12th Int. Conf. on Machine Learning (ICML'95)*, pages 209–217. Morgan Kaufmann, Tahoe City, CA, USA, 1995.

[29] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *29th Ann. IEEE Int. Conf. on Local Computer Networks (LCN '04)*, pages 455–462, Los Alamitos, CA, USA, 2004.

[30] Dwivedi, A. K. and Vyas, O. P. *Wireless Sensor Network: At a Glance*. Prof. Jia-Chin Lin (Ed.), InTech Open Access Publisher. Available from: http://www.intechopen.com/books/recent-advances-in-wireless-communications-and-networks/wireless-sensor-network-at-a-glance, 2011.

[31] Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. Next century challenges: Scalable coordination in sensor networks. In *Proc. 5th Annual ACM/IEEE Int. Conf. on Mobile Computing and Networking*, MobiCom '99, pages 263–270. Seattle, WA, USA, 1999.

[32] Ficici, S. G., Watson, R. A., and Pollack, J. B. Embodied evolution: A response to challenges in evolutionary robotics. In Wyatt, J. and Demiris, J., editors, *Proc. 8th European Workshop on Learning Robots (EWLR-8)*, pages 14–22. Springer, EPFL, Lausanne, Switzerland, 1999.

[33] Fogel, D. B. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ, USA, 1995.

[34] Förster, A. Machine Learning Techniques Applied to Wireless Ad-Hoc Networks: Guide and Survey. In Palaniswami, M., Marusic, S., and Law, Y. W., editors, *IEEE Proc. of 3rd Int. Conf. on Intelligent Sensors, Sensor Networks and Information (ISSNIP 2007)*, pages 365–370. IEEE, Melbourne, Australia, 2007.

[35] Frommberger, L. Foundations of reinforcement learning. In *Qualitative Spatial Abstraction in Reinforcement Learning*, pages 9–21. Springer Publishing Company, Inc., Berlin, Heidelberg, 1st edition, 2010.

[36] Funes, P. and Pollack, J. Evolutionary Body Building: Adaptive Physical Designs for Robots. *Artificial Life*, 4(4):337–357, 1998.

[37] Garcia-Sanchez, P., Sevilla, J. P., Merelo, J. J., Mora, A. M., Castillo, P. A., Laredo, J. L. J., and Casado, F. Pervasive Evolutionary Algorithms on Mobile Devices. In Omatu, S., Rocha, M. P., Bravo, J., Fernández, F., Corchado, E., Bustillo, A., and Corchado, J. M., editors, *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living: Proc, 10th Int. Work-Conf. on Artificial Neural Networks (IWANN 2009) Workshops, Part II*, pages 163–170, Salamanca, Spain, 2009.

[38] Gittins, J. C. Bandit Processes and Dynamic Allocation Indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, 41(2):pp. 148–177, 1979.

[39] Gobbett, D. L., Handcock, R. N., Zerger, A., Crossman, C., Valencia, P., Wark, T., and Davies, M. Prototyping an Operational System with Multiple Sensors for Pasture Monitoring. *Journal of Sensor and Actuator Networks*, 2(3):388–408, 2013.

[40] Goldberg, D. E. A note on boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. *Complex Systems*, 4(4):445–460, 1990.

[41] Grefenstette, J. Optimization of Control Parameters for Genetic Algorithms. *IEEE Trans. Systems, Man and Cybernetics*, 16(1):122–128, 1986.

[42] Grefenstette, J. and Schultz, A. An Evolutionary Approach to Learning in Robots. In *Proc. of the Machine Learning Workshop on Robot Learning, 11th Int. Conf. on Machine Learning*, New Brunswick, NJ, USA, 1994.

[43] Grefenstette, J. J. The Evolution of Strategies for Multi-agent Environments. *Adaptive Behavior*, 1:65–89, 1992.

[44] Grefenstette, J. J. and Ramsey, C. L. An Approach to Anytime Learning. In *Proc. 9th Int. Conf on Machine Learning*, pages 189–195, Aberdeen, Scotland, UK, 1992.

[45] Grefenstette, J. J., Ramsey, C. L., and Schultz, A. C. Learning Sequential Decision Rules Using Simulation Models and Competition. *Machine Learning*, 5(4):355–381, 1990.

[46] Grefenstette, J. J. The User's Guide to SAMUEL-97: An Evolutionary Learning System. Code 5514, Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Washington, DC, USA, 1997.

[47] Guo, Y., Corke, P., Poulton, G., Wark, T., Bishop-Hurley, G., and Swain, D. Animal Behaviour Understanding using Wireless Sensor Networks. In *31st IEEE Conf. on Local Computer Networks*, pages 607–614, Tampa, FL, USA, 2006.

[48] Handcock, R. N., Swain, D. L., Bishop-Hurley, G. J., Patison, K. P., Wark, T., Valencia, P., Corke, P., and O'Neill, C. J. Monitoring Animal Behaviour and Environmental Interactions Using Wireless Sensor Networks, GPS Collars and Satellite Remote Sensing. *Sensors*, 9(5):3586–3603, 2009.

[49] Hartland, C., Gelly, S., Baskiotis, N., Teytaud, O., and Sebag, M. Multi-armed Bandit, Dynamic Environments and Meta-Bandits. In *12th Ann. Conf. on Neural Information Processing Systems (NIPS-2006) Workshop, Online Trading Between Exploration and Exploitation*. Whistler, Canada, 2006.

[50] Hettiarachchi, S. Distributed online evolution for swarm robotics. *Presented at Autonomous Agents and Multi Agent Systems, Doctoral Mentoring Program*, 2006.

[51] Hettiarachchi, S. An evolutionary approach to swarm adaptation in dense environments. In *Int. Conf. on Control Automation and Systems (ICCAS)*, pages 962–966. IEEE, Gyeonggi-do, Korea, 2010.

[52] Hettiarachchi, S. Improving Swarm Survival Using DAEDALUS. In *Proc. 21st Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2010)*. 2010.

[53] Hettiarachchi, S., Maxim, P., and Spears, W. M. An Architecture for Adaptive Swarms. In *Robotics Research Trends*, pages 121–153. Nova Science Publishers, 2008.

[54] Hettiarachchi, S. and Spears, W. DAEDALUS for Agents with Obstructed Perception. *IEEE Mountain Workshop on Adaptive and Learning Systems*, pages 195–200, 2006.

[55] Hettiarachchi, S., Spears, W. M., Green, D., and Kerr, W. Distributed Agent Evolution with Dynamic Adaptation to Local Unexpected Scenarios. In *Innovative Concepts for Autonomic and Agent-Based Systems*, pages 245–256. Springer, Berlin, Heidelberg, 2006.

[56] Hettiarachchi, S., Maxim, P. M., Spears, W. M., and Spears, D. F. Connectivity of Collaborative Robots in Partially Observable Domains. In *International Conference on Control, Automation and Systems*, 2008.

[57] Hettiarachchi, S. and Spears, W. M. Distributed adaptive swarm for obstacle avoidance. *Int. J. of Intelligent Computing and Cybernetics*, 2(4):644–671, 2009.

[58] Holland, J. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.

[59] Iacca, G. Introducing DOWSN: Distributed optimization in wireless sensor networks. In De Wilde, P., Coghill, G. M., and Kononova, A. V., editors, *Proc. of the IEEE 12th Ann. Workshop on Computational Intelligence (UKCI 2012)*, pages 1–8. Edinburgh, UK, 2012.

[60] Iacca, G. Distributed optimization in wireless sensor networks: an island-model framework. *Soft Computing*, 17(12):2257–2277, 2013.

[61] Iacca, G., Neri, F., Mininno, E., Ong, Y.-S., and Lim, M.-H. Ockham's Razor in memetic computing: Three stage optimal memetic exploration. *Information Sciences*, 188:17–43, 2012.

[62] Johnson, D. M., Teredesai, A. M., and Saltarelli, R. T. Genetic Programming in Wireless Sensor Networks. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., and Tomassini, M., editors, *8th European Conf. on Genetic Programming (EuroGP 2005)*, LNCS 3447, pages 96–107. Springer, Lausanne, Switzerland, 2005.

[63] Keith, M. J. and Martin, M. C. Genetic Programming in C++: Implementation Issues. In Kinnear Jr, K. E., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, 1994.

[64] Kimura, M. and Weiss, G. H. The stepping stone model of population structure and the decrease of genetic correlation with distance. *Genetics*, 49(4):561–576, 1964.

[65] Kinnear Jr, K. E. *Advances in Genetic Programming*, volume 1. MIT Press, Cambridge, MA, USA, 1994.

[66] Koulouriotis, D. E. and Xanthopoulos, A. Reinforcement learning and evolutionary algorithms for non-stationary multi-armed bandit problems. *Applied Mathematics and Computation*, 196(2):913–922, 2008.

[67] Koza, J. R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Tech Report: STAN-CS-90-1314, Stanford University, Department of Computer Science, Stanford, CA, USA, 1990.

[68] Koza, J. R. A hierarchical approach to learning the boolean multiplexer function. In Rawlins, G. J. E., editor, *Proc. Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, pages 171–192. Indiana University, Bloomington, IN, USA, 1990.

[69] Koza, J. R. A Genetic Approach to Finding a Controller to Back Up a Tractor-Trailer Truck. In *American Control Conference*, pages 2307–2311, Chicago, IL, USA, 1992.

[70] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[71] Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., and Lanza, G. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer, MA, USA, 2003.

[72] Kulkarni, R. V., Forster, A., and Venayagamoorthy, G. K. Computational Intelligence in Wireless Sensor Networks: A Survey. *Communications Surveys and Tutorials, IEEE*, 13(1):68–96, 2011.

[73] Lamarck, J.-B. *Philosophie zoologique*, volume 1. Duminil-Lesueur, Paris, 1809.

[74] Latvakoski, J., Livari, A., Vitic, P., Jubeh, B., Alaya, M., Monteil, T., Lopez, Y., Talavera, G., Gonzalez, J., Granqvist, N., Kellil, M., Ganem, H., and Väisänen, T. A Survey on M2M Service Networks. *Computers*, 3(4):130–173, 2014.

[75] Louis, S. Predicting Convergence Time for Genetic Algorithms. *Foundations of Genetic Algorithms*, 2:141–161, 1992.

[76] Mack, C. The Multiple Lives of Moore's Law. *IEEE Spectrum*, 52(4):31, 2015.

[77] Macready, W. G. and Wolpert, D. H. Bandit Problems and the Exploration/Exploitation Tradeoff. *IEEE Trans. Evolutionary Computation*, 2(1):2–22, 1998.

[78] Mahajan, A. and Teneketzis, D. In Hero, A. O., Castañón, D. A., Cochran, D., and Kastella, K., editors, *Foundations and Applications of Sensor Management*, chapter Multi-Armed Bandit Problems, pages 121–151. Springer US, Boston, MA, 2008.

[79] Martin, W. N., Lienig, J., and Cohoon, J. P. Population structures: island (migration) models: evolutionary algorithms based on punctuated equilibria. In Bäck, T., B,

F. D., and Z, M., editors, *Handbook of Evolutionary Computation*. IOP Publishing, New York, 1997.

[80] Mataric, M. Coordination and learning in multirobot systems. *IEEE Intelligent Systems and their Applications*, 13(2):6–8, 1998.

[81] Mataric, M. and Cliff, D. Challenges in evolving controllers for physical robots. *Robotics and Autonomous Systems*, 19(1):67–83, 1996.

[82] Middleton, P., Kjeldsen, P., and Tully, J. *Forecast: The Internet of Things, Worldwide, 2013. (G00259115)*. Gartner Research, 2013.

[83] Miorandi, D., Yamamoto, L., and De Pellegrini, F. A survey of evolutionary and embryogenic approaches to autonomic networking. *Computer Networks*, 54(6):944–959, 2010.

[84] Mitchell, M. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[85] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, L., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[86] Moore, G. E. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[87] Mühlenbein, H. and Schlierkamp-Voosen, D. The Science of Breeding and its Application to the Breeder Genetic Algorithm (BGA). *Evolutionary Computation*, 1(4):335–360, 1993.

[88] Nan, G. and Li, M. Evolutionary Based Approaches in Wireless Sensor Networks: A Survey. In Kellenberger, P., editor, *4th Int. Conf. on Natural Computation (ICNC '08)*, volume 5, pages 217–222. Jinan, China, 2008.

[89] Nelson, A. L., Barlow, G. J., and Doitsidis, L. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, 2009.

[90] Nordin, P. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In Kinnear Jr, K. E., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA, USA, 1994.

[91] Nordin, P. Comparison of a compiling genetic programming system versus a connectionist approach. In Bäck, T., Fogel, D. B., and Michalewicz, Z., editors, *Handbook of Evolutionary Computation*. Oxford University Press, 1997.

[92] Nordin, P. and Banzhaf, W. A genetic programming system learning obstacle avoiding behavior and controlling a miniature robot in real time. Technical Report: SysReport 4/95, University of Dortmund, Fachbereich Informatik, Dortmund, Germany, 1995.

[93] Nordin, P. and Banzhaf, W. Genetic Programming Controlling a Miniature Robot. In Siegel, E. V. and Koza, J. R., editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, USA, 1995.

[94] Nordin, P. and Banzhaf, W. Real Time Evolution of Behavior and a World Model for a Miniature Robot Using Genetic Programming. Technical Report: SysReport 5/95, Dept. of Computer Science, University of Dortmund, Dortmund, Germany, 1995.

[95] Nordin, P. and Banzhaf, W. An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time with Genetic Programming. *Adaptive Behavior*, 5(2):107–140, 1997.

[96] Nordin, P., Banzhaf, W., and Brameier, M. Evolution of a world model for a miniature robot using genetic programming. *Robotics and Autonomous Systems*, 25(1):105–116, 1998.

[97] Nordin, P., Banzhaf, W., and Francone, F. D. Efficient Evolution of Machine Code for CISC Architectures using Instruction Blocks and Homologous Crossover. In

Spector, L., Langdon, W. B., O'Reilly, U., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, pages 275–299. MIT Press, Cambridge, MA, USA, 1999.

[98] Olmer, M., Nordin, P., and Banzhaf, W. Evolving Real-time Behavior Modules for a real Robot with GP. In Jamshidi, M., Pin, F., and Dauchez, P., editors, *Proc. 6th Int. Symp. on Robotics and Manufacturing (ISRAM-96)*, pages 675–680, Montpellier, France, 1996.

[99] Palmer, D., Sikka, P., Valencia, P., and Corke, P. An Optimising Compiler for Generated Tiny Virtual Machines. In *2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, pages 161–162, Sydney, Australia, 2005.

[100] Pintér, J. Global optimization: Software, test problems, and applications. In Pardalos, P. and Romeijn, H., editors, *Handbook of Global Optimization*, volume 2, chapter 15, pages 515–569. Kluwer Academic Publishers, London, UK, 2002.

[101] Platt, G., Wall, J., Valencia, P., and Ward, J. K. The Tiny Agent- Wireless Sensor Networks Controlling Energy Resources. *Journal of Networks*, 3(4):42–50, 2008.

[102] Polastre, J., Szewczyk, R., and Culler, D. Telos: enabling ultra-low power wireless research. In *Proc. 4th Int. Symp. on Information Processing in Sensor Networks (IPSN 2005)*, pages 364–369, Los Angeles, CA, USA, 2005. IEEE.

[103] Poli, R., Langdon, W. B., and McPhee, N. F. *A field guide to genetic programming*. Available Online at `http://www.gp-field-guide.org.uk`, lulu.com, 2008.

[104] Predd, J. B., Kulkarni, S. R., and Poor, H. V. Distributed learning in wireless sensor networks. *IEEE Signal Processing Magazine*, 23(4):56–69, 2006.

[105] Prügel-Bennett, A. Benefits of a population: Five mechanisms that advantage population-based algorithms. *IEEE Trans. Evolutionary Computation*, 14(4):500–517, 2010.

[106] Robbins, H. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[107] Russell, S. J. and Norvig, P. *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall, 2010.

[108] Schaffer, J. D. and Eshelman, L. J. On Crossover as an Evolutionary Viable Strategy. In Belew, R. K. and Booker, L. B., editors, *Proc. 4th International Conference on Genetic Algorithms (ICGA'91)*, volume 91, pages 61–68. Morgan Kaufmann, San Diego, CA, USA, 1991.

[109] Schmidt, M. D. and Lipson, H. Coevolution of Fitness Predictors. *IEEE Transactions on Evolutionary Computation*, 12(6):736–749, 2008.

[110] Schultz, A. C. Using a Genetic Algorithm to Learn Strategies for Collision Avoidance and Local Navigation. In *Proc. 7th Int. Symp. on Unmanned Untethered Submersible Technology*, pages 213–225, University of New Hampshire Marine Systems Engineering Laboratory, 1991.

[111] Schultz, A. C. Learning robot behaviors using genetic algorithms. In *Intelligent Automation and Soft Computing: Trends in Research, Development, and Applications: Proc. 1st World Automation Congress*, volume 1, pages 14–17, Maui, Hawaii, USA, 1994.

[112] Shan, Y., McKay, R. I., Essam, D., and Abbass, H. A. In Pelikan, M., Sastry, K., and CantúPaz, E., editors, *Studies in Computational Intelligence*, chapter A Survey of Probabilistic Model Building Genetic Programming, pages 121–160. Springer, Berlin, Heidelberg, 2006.

[113] Shannon, P. and Nehaniv, C. L. Evolving robot controllers in PDL using genetic programming. In *IEEE Symp. on Artificial Life (ALIFE)*, pages 92–99, Paris, France, 2011.

[114] Sikka, P., Corke, P., Overs, L., Valencia, P., and Wark, T. Fleck - a platform for real-world outdoor sensor networks. In *3rd Int. Conf. on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2007)*, pages 709–714. IEEE, 2007.

[115] Sikka, P., Corke, P., Valencia, P., Crossman, C., Swain, D., and Bishop-Hurley, G. Wireless Adhoc Sensor and Actuator Networks on the Farm. In *Proc. 5th Int. Conf on Information Processing in Sensor Networks*, pages 492–499, Nashville, TN, USA, 2006. ACM.

[116] Sikora, R. T. Meta-learning optimal parameter values in non-stationary environments. *Knowledge-Based Systems*, 21(8):800 – 806, 2008.

[117] Silva, A., Neves, A., and Costa, E. Genetically Programming Networks to Evolve Memory Mechanisms. In *Proc. of Genetic and Evolutionary Computation Conf. (GECCO 1999)*, Orlando, Florida, USA, 1999.

[118] Silva, A., Neves, A., and Costa, E. Polymorphy and Hybridization in Genetically Programmed Networks. In *Parallel Problem Solving from Nature (PPSN VI)*, pages 221–230. Springer, Paris, France, 2000.

[119] Simon, H. A. A Behavioral Model of Rational Choice. *Quarterly Journal of Economics*, 69(1):99–118, 1955.

[120] Smith, D., Dutta, R., Hellicar, A., Bishop-Hurley, G., Rawnsley, R., Henry, D., Hills, J., and Timms, G. Bag of class posteriors, a new multivariate time series classifier applied to animal behaviour identification. *Expert Systems with Applications*, 42(7):3774–3784, 2015.

[121] Smith, R. E. and Smuda, E. Adaptively resizing populations: Algorithm, analysis, and first results. *Complex Systems*, 9:47–72, 1995.

[122] Sofman, B., Bagnell, J. A., and Stentz, A. Bandit-based online candidate selection for adjustable autonomy. In Howard, A., Lagnemma, K., and Kelly, A., editors, *Field and Service Robotics*, volume 62 of *Springer Tracts in Advanced Robotics*, pages 239–248. Springer, Berlin, Heidelberg, 2010.

[123] Steels, L. Emergent functionality in Robotic Agents through on-line Evolution. In Brooks, R. and Maes, P., editors, *Artificial life IV: Proc. 4th Int. Workshop on the*

Synthesis and Simulation of Living Systems, volume 4, pages 8–14, The Massachusetts Institute of Technology, Cambridge, MA, USA, 1994.

[124] Steels, L. Building agents with autonomous behavior systems. In Steels, L. and Brooks, R., editors, *The Artificial Life Route to Artificial Intelligence: Building Embodied Situated Agents*, chapter 3, pages 83–121. Lawrence Erlbaum Associates, New Haven, 1995.

[125] Steels, L. The Artificial Life Roots of Artificial Intelligence. *Artificial Life*, 1(1-2):75–110, 1993.

[126] Stender, J. *Parallel Genetic Algorithms: Theory and Applications*, volume 14. IOS Press, Amsterdam, 1993.

[127] Strehl, A., Langford, J., Li, L., and Kakade, S. M. Learning from logged implicit exploration data. In Lafferty, J., Williams, C., Shawe-taylor, J., Zemel, R. S., and Culotta, A., editors, *Advances in Neural Information Processing Systems*, volume 23, pages 2217–2225. 2010.

[128] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[129] Tan, S. K., Sooriyabandara, M., and Fan, Z. M2M Communications in the Smart Grid: Applications, Standards, Enabling Technologies, and Research Challenges. *Int. J. Digital Multimedia Broadcasting*, 2011(99):1–8, 2011.

[130] Tanev, I. and Yuta, K. Epigenetic programming: Genetic programming incorporating epigenetic learning through modification of histones. *Information Sciences*, 178(23):4469–4481, 2008.

[131] Tekin, C. and Liu, M. Online algorithms for the multi-armed bandit problem with markovian rewards. In *48th Ann. Allerton Conf. on Communication, Control, and Computing*, pages 1675–1682, Monticello, IL, USA, 2010.

[132] Teller, A. and Veloso, M. PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System. Technical Report: No. CMU-CS-95-101, Carnegie-Mellon University Dept. Computer Science, Pittsburgh, PA, USA, Dortmund, Germany, 1995.

[133] Tokic, M. and Palm, G. Value-difference based exploration: Adaptive control between epsilon-greedy and softmax. In Bach, J. and Edelkamp, S., editors, *Advances in Artificial Intelligence: Proc. 34th Ann.German Conf. on AI (KI 2011)*, volume 7006 of *Lecture Notes in Computer Science*, pages 335–346. Springer, Berlin, Heidelberg, Germany, 2011.

[134] Valencia, P. In situ genetic programming for wireless sensor networks. In Hazas, M., Boulis, A., and Girod, L., editors, *SenSys 2007 Doctoral Colloquium*, pages 37–41, Sydney, Australia, 2007.

[135] Valencia, P., Haak, A., Cotillon, A., and Jurdak, R. Genetic programming for smart phone personalisation. *Applied Soft Computing*, 25(2014):86–96, 2014.

[136] Valencia, P., Jurdak, R., and Lindsay, P. Fitness Importance for Online Evolution. In *Proc. 12th Ann. Conf on Genetic and Evolutionary Computation (GECCO10) Companion*, pages 2117–2118. ACM, Portland, OR, USA, 2010.

[137] Valencia, P., Lindsay, P., and Jurdak, R. Distributed genetic evolution in wsn. In *Proc. 9th ACM/IEEE Int. Conf. on Information Processing in Sensor Networks*, IPSN '10, pages 13–23, Stockholm, Sweden, 2010.

[138] Van Veldhuizen, D. A., Zydallis, J. B., and Lamont, G. B. Considerations in engineering parallel multiobjective evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, 7(2):144–173, 2003.

[139] Wark, T., Corke, P., Sikka, P., Klingbeil, L., and Guo, Y. Transforming agriculture through pervasive wireless sensor networks. *IEEE Pervasive Computing*, 2007.

[140] Watson, J. R. and Wiles, J. The rise and fall of learning: a neural network model of the genetic assimilation of acquired traits. In *Proc. of the 2002 Congress on*

Evolutionary Computation (CEC'02), volume 1, pages 600–605, Honolulu, HI, USA, 2002.

[141] Watson, R. A., Ficici, S. G., and Pollack, J. B. Embodied evolution: embodying an evolutionary algorithm in a population of robots. In *Proc. of the 1999 Congress on Evolutionary Computation (CEC 99)*, volume 1, page 342, Washington, DC, USA, 1999. IEEE.

[142] Watson, R. A., Ficici, S. G., and Pollack, J. B. Embodied evolution: Distributing an evolutionary algorithm in a population of robots. *Robotics and Autonomous Systems*, 39(1):1–18, 2002.

[143] Weise, T. *Evolving Distributed Algorithms with Genetic Programming*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Kassel, Germany, 2009.

[144] Weise, T. *Global Optimization Algorithms - Theory and Application (2nd Edition)*. Self-Published. Available at http://www.it-weise.de/, 2009.

[145] Weise, T. and Geihs, K. DGPF - An Adaptable Framework for Distributed Multi-Objective Search Algorithms Applied to the Genetic Programming of Sensor Networks. In Filipič, B. and Šilc, J., editors, *Proc. 2nd Int. Conf. on Bioinspired Optimization Methods and their Application (BIOMA 2006)*, pages 157–166, Ljubljana, Slovenia, 2006.

[146] Weise, T. and Geihs, K. Genetic Programming Techniques for Sensor Networks. In Marron, P. D., editor, *Proc. of 5. GI/ITG KuVS Fachgesprach Drahtlose Sensornetze*, volume 5, pages 21–25. University of Stuttgart, Stuttgart, Germany, 2006.

[147] Weise, T. and Tang, K. Evolving Distributed Algorithms With Genetic Programming. *IEEE Trans. Evolutionary Computation*, 16(2):242–265, 2012.

[148] Weise, T. and Zapf, M. Evolving distributed algorithms with genetic programming: election. In Xu, L., Goodman, E. D., Chen, G., Whitley, D., and Ding, Y., editors,

Proc. 1st ACM/SIGEVO Summit on Genetic and Evolutionary Computation (GEC '09), pages 577–584, Shanghai, China, 2009. ACM.

[149] Weise, T., Zapf, M., and Geihs, K. Evolving Proactive Aggregation Protocols. In O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A. I., Falco, I., Cioppa, A., and Tarantino, E., editors, *Proc. 11th European Conference on Genetic Programming (EuroGP 2008)*, pages 254–265, Naples, Italy, 2008. Springer.

[150] Weise, T. Genetic Programming for Sensor Networks. Technical report, University of Kassel, Distributed Systems Group, Available Online at http://www.it-weise.de/documents/files/W2006DGPFa.pdf [accessed 2016-02-23], 2006.

[151] Whiteson, S. Evolutionary Computation for Reinforcement Learning. In *Reinforcement Learning: State-of-the-Art*, pages 325–355, Berlin, Heidelberg, 2012. Springer.

[152] Whittle, P. Multi-Armed Bandits and the Gittins Index. *Journal of the Royal Statistical Society. Series B (Methodological)*, 42(2):pp. 143–149, 1980.

[153] Wolpert, D. H. and Macready, W. G. No free lunch theorems for search. Technical Report: TR-95-02-010, Santa Fe Institute, Sante Fe, NM, USA, 1995.

[154] Wright, S. Evolution in Mendelian Populations. *Genetics*, 16(2):97–159, 1931.

Index