The Evolution of Agents

Mohammad Adil Qureshi

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

July 2001

Abstract

Genetic Programming (GP) is a technique that can be used to automatically program computers to perform some required task. The technique is a kind of genetic algorithm in which the representation is a program tree instead of a bit-string and the fitness of each tree is evaluated by executing the computer program that it represents.

The subject of this thesis is to investigate the use of GP to automatically program multiagent systems. To achieve this goal, we consider the general problems in creating multiagent systems, and show how GP can be used to provide solutions to many of them. Our key contributions are as follows:

We show that it is possible to evolve multiagent systems using GP that:

- exhibit coordinated, coherent behaviour
- communicate explicitly, and in doing so decide what to communicate and how
- can resolve conflicts
- can be integrated into an existing society of agents

We also consider the technical scalability issues involved in the use of GP, both generally and in particular as a technique for automatically programming agents, and propose some solutions to these problems.

Acknowledgements

I would like to thank the department of computer science at UCL for awarding me an EPSRC PhD quota award which funded this work. I would also like to thank my supervisor Jon Crowcroft for his encouragement and support. Many thanks to William Langdon for introducing me to GP, providing useful pointers and for reviewing some chapters of this thesis. Thanks also to Thomas Haynes for answering questions relating to his research. My thanks to UCL computer science and Java training company JB International for allowing me to use their machines for the compute intensive task of evolving agents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nature has been very successful at ”programming” sophisticated multiagent systems ranging from ”simple” agents such as ants and bees to sophisticated cognitive agents - ourselves. The mechanism used to create these systems is Darwinian evolution. Our hypothesis is that a technique based on Darwinian evolution called Genetic Programming (GP) can be used to automatically program software multiagent systems.

A multiagent system (MAS) is one in which several interacting, intelligent agents pursue some set of goals or perform some task. The central question in such systems is when and how should which agents interact in order to achieve their goals. Answering this question is difficult and raises a number of issues which are discussed further in chapter 3. We propose that we can use genetic programming to provide a solution to this question and the related issues that it creates, for any given problem domain.

Genetic Programming (GP) [Koz92b] is a relatively new technique that uses the principle of Darwinian evolution [Daw86] to automatically program computers. GP has many advantages over other techniques that have been used in an attempt to automatically program agents such as neural networks and conventional genetic algorithms. The most important is that the representation used by GP is a computer program. It is therefore possible for humans to understand generated solutions and therefore predict their behaviour (compare with neural networks). Furthermore, the generated programs are easily executed on current computer architectures and do not require any specialised environments. A second advantage is that GP has been shown to be very general and has been successfully applied to a wide variety of problems [Koz92b; Koz94b; KDBK99]. Koza claims GP is the most general machine learning technique [Koz92b].

To prove our hypothesis that Genetic Programming can be used to automatically program multiagent systems we attempt to answer the following questions:

- What are the key issues in implementing multiagent systems?

- Is GP capable of providing solutions to them?
- What are the technical problems in scaling GP to real world multiagent applications?
- How can we solve these technical scalability problems?

Answers to some of these questions are supported by results from experiments.

Our key contributions are as follows:

We show that it possible to evolve multiagent systems using GP that:

- exhibit coordinated, coherent behaviour
- communicate explicitly, and in doing so decide what to communicate and how
- can resolve conflicts
- can be integrated into an existing society of agents

We also consider the technical scalability problems involved in the use of GP, both generally and in particular as a technique for automatically programming agents, and propose some solutions to these problems.

Lastly we designed and implemented a state of the art GP system in the Java programming language and made it freely available (complete with source code and documentation) to the GP community in 1997. This system has been supported and kept uptodate and has facilitated GP research by others, which we believe is a useful contribution to the GP community.

The layout of this thesis is as follows:

Chapters 2 and 3 introduce Genetic Programming and Multiagent Systems. Together, they set the technical grounding for this thesis. Research combining the two fields is surveyed in chapter 3. The pursuit problem was designed as a testbed for researching MAS architectures. Haynes and Sandip [HSSW95b; HS96c; HS97] used GP to evolve both heterogeneous and homogeneous agents for this problem. We note however that whilst their work was successful at demonstrating how GP can decompose tasks, conflict resolution was provided automatically by the environment and was not evolved. The main focus of chapter 4 is to show that we can evolve agents that also resolve conflicts using GP. We also further investigate task allocation for the pursuit domain. Communication is an important aspect of multiagent systems, chapter 5 demonstrates that GP can be used to evolve agents that communicate explicitly. We would like to be able add agents to a society of previously-created agents to solve new problems. In chapter 6 we show that GP can be used to evolve agents that must work within a society of existing agents and know how to interact with them. Chapter 7 discusses the comparative performance of GP versus random search. The experience gained in evolving agents using GP is used to define a a methodology for automatically programming multiagent systems. The technical scalability problems arising from applying this technique to larger and more complex problems are considered and some solutions proposed. Finally chapter 8 summarises our work, and critically evaluates it. Further areas of research stemming from our work are proposed. Appendix A provides an overview of the Genetic Programming system (called GPsys) developed for this work and which was used for all of the experiments in this thesis.

Chapter 2

Genetic Programming

Some of the most sophisticated programs in existence are not human coded, they evolved naturally. They are to be found inside the cells of living creatures in the form of DNA. These programs were "coded" by the process of natural selection as described by Charles Darwin and are responsible for the highly optimised "machines" that we see around us. It is not surprising therefore that the AI community has been active in trying to harness the power of natural evolution to build both hardware and software. The purpose of this chapter is to introduce genetic programming as a means of automatically generating computer programs.

2.1 Genetic Algorithms

John Holland [Hol92] devised a class of algorithms inspired by Darwin's theory of evolution by natural selection called *Genetic Algorithms*. These algorithms abstract the key mechanisms used in natural evolution, namely *selection, reproduction and variation*, and use them for optimisation and search [Gol89]. A population of trial solutions to a given problem are first generated randomly. New solutions are bred from the fittest solutions selected from the old population by applying genetic operators to them. These genetic operators vary the solutions, so that new solutions are generated. The results is a new population of solutions. By repeatedly generating subsequent solutions in the same manner we can theoretically generate progressively better solutions. The best solution obtained so far is tracked during this process. When the best solution meets our termination criteria, or we decide the maximum number of generations that we permit has been exceeded we terminate the run.

The representation of solutions used in Genetic Algorithms commonly consists of bit-strings of fixed length. In an engineering optimisation problem for example, these bit-strings might encode a set of design parameters that are used to tweak the performance of some system. The initial population consists of a randomly generated set of such bit-strings. The fitness of the bit-string is evaluated by measuring how useful it is. In our example we would decode each bit-string to

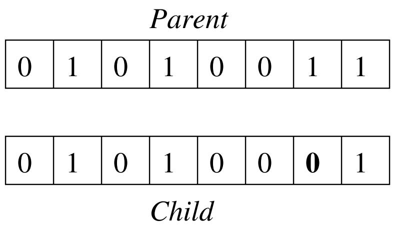


Figure 2.1: GA mutation

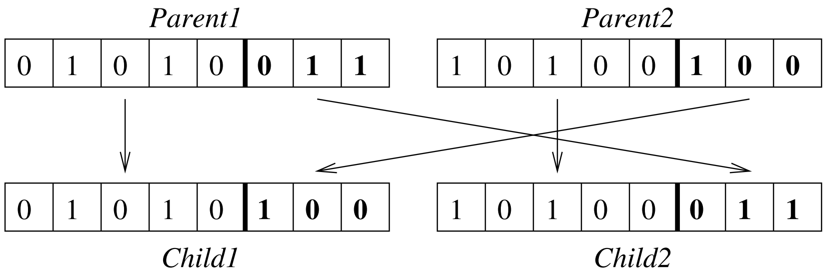


Figure 2.2: GA crossover

obtain our parameters, apply them to our system, and measure the performance.

There are two selection schemes that are commonly used, *fitness proportionate selection* and *tournament selection*. In fitness proportionate selection, individuals are selected with probabilities in proportion to their fitness. This requires that the sum of the fitness values of the entire population is known, and makes parallel/distributed implementations difficult. An alternative to fitness proportionate selection is *tournament selection*. In tournament selection a small group of individuals are chosen randomly from the population (typically 7), and ordered by their fitness values. The fittest of these individuals are selected to take part in creating a new individual. In steady state GAs, the individual to be replaced is often the individual with the lowest fitness in the tournament. An advantage with tournament selection is that it is simple to implement, and secondly that by varying the tournament size, one can vary the fitness pressure.

The genetic operators used to create new individuals include *reproduction, mutation and crossover*. The reproduction operator simply creates an exact copy of the selected individual. The mutation operator copies the selected individual and then randomly changes a portion of the bit-string (figure 2.1). The crossover operator takes two individuals and exchanges a random portion of the bit-strings of each individual (figure 2.2). This operator potentially creates two new individuals. Often however, only a single individual is created by discarding one of the resultant off-springs.

GAs have been successfully applied to a wide variety of problems including engineering, financial, imaging, VLSI circuit layout, gas pipeline control and production scheduling [Dav91], and have been shown to provide near optimal solutions. A mathematical explanation for their success called the *Schema Theorem* was proposed by Holland [Hol73], and has been backed by empirical support [Gol89]. The space of bit-strings of a given length can be partitioned into sets, where each set consists of strings with similarities at certain positions in the string. For example consider the set of all 5 bit-strings which begin with 1 and end with 0. We can describe such a string by the *similarity template* or *schema* 1***0 where * represents a wild-card or "don't care" value. There are two properties of schemas that are used by the schema theorem, *order* and *defining length*. The order of a schema H, denoted $o(H)$ is the number of fixed values in the bit string. For example the string 01**1 has an order of 3. Schemas of higher order are more specific than ones with lower orders. The defining length of a schema H, denoted $\delta(H)$ is the distance between the first and last fixed values. For example, the string 01*0* has a defining length of 3. The schema theorem makes predictions on the processing of schema by the GA which can be expressed mathematically as follows:

$$E\left[m(H,t+1)\right] \geq m(H,t)\frac{f(H,t)}{\overline{f}(t)}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right]$$

where:

- $m(H,t)$ is the number of strings with schema H in the population at time t
- $f(H,t)$ is the average fitness of the strings with schema H in the population at time t
- $\overline{f}(t)$ is the average fitness of the population at time t
- l is the length of the strings
- p_c is the probability of crossover
- p_m is the probability of mutation

The theorem predicts that short defining length, low order, above average schemas called *building blocks* receive exponentially increasing trials in subsequent generations. A possible explanation of how GAs work is therefore by combining building blocks (or partial solutions) to form strings of high fitness (optimal or near optimal solutions).

A key criticism of this schema theorem is that it only provides a lower bound on the expected number of schemas. This is a consequence of focussing only on the destructive effects of crossover and not explicitly accounting for schema reconstruction [SV00]. Stephens and Waelbroeck provide an alternative schema theorem based on the notion of *effective fitness*. The standard notion of fitness used in GAs is also called *reproductive fitness* because it is a measure of the probability that an individual reaches reproductive age. Effective fitness in contrast is a complete measure of the reproductive success of an individual and takes into account the effect of

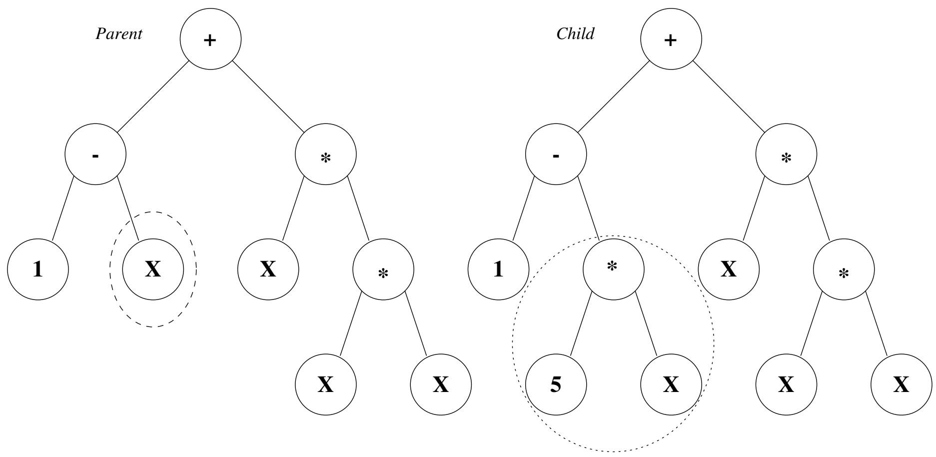


Figure 2.3: GP mutation

all genetic operators, such as mutation and crossover. Their schema theorem predicts exactly the expected number of schemas, and takes into account schema reconstruction.

2.2 Variable Length and Hierarchical Representations

A variable length representation for GAs was first proposed by Smith [Smi80] to create a variant on the classifier systems (evolved production systems) defined by Holland and Reitman [HR78]. The GA was used to evolve complete rule-based programs for playing poker. The crossover operator used by Smith set restrictions on where crossover can be applied in an evolving structure. This inspired Cramer [Cra85] to use a tree based representation to evolve symbolic expressions, using sub-tree crossover. Koza developed a technique for using sub-tree crossover to evolve expressions in the LISP programming language and successfully demonstrated the usefulness of the technique on wide range of problems [Koz92b]. This technique is now known as genetic programming.

2.3 Genetic Programming

Genetic Programming is technique for automatically programming computers. It uses the power of GAs to search the space of possible computer programs to find suitable programs for the desired application [LQ95]. Koza describes it as “the most general machine learning technique”, a claim which is substantiated by the large number of applications in which it has been successfully used [Koz92b; Koz94b; KDBK99]. The representation commonly used in GP is a program parse tree. Program parse trees have the advantage that they represents syntactically correct computer programs, and are easy to manipulate. The branches in the parse tree represent functions which take arguments, the leaves represent zero-argument functions, variables or constants. The fitness of a GP is measured by testing the program that it represents. Fitness is awarded to programs in

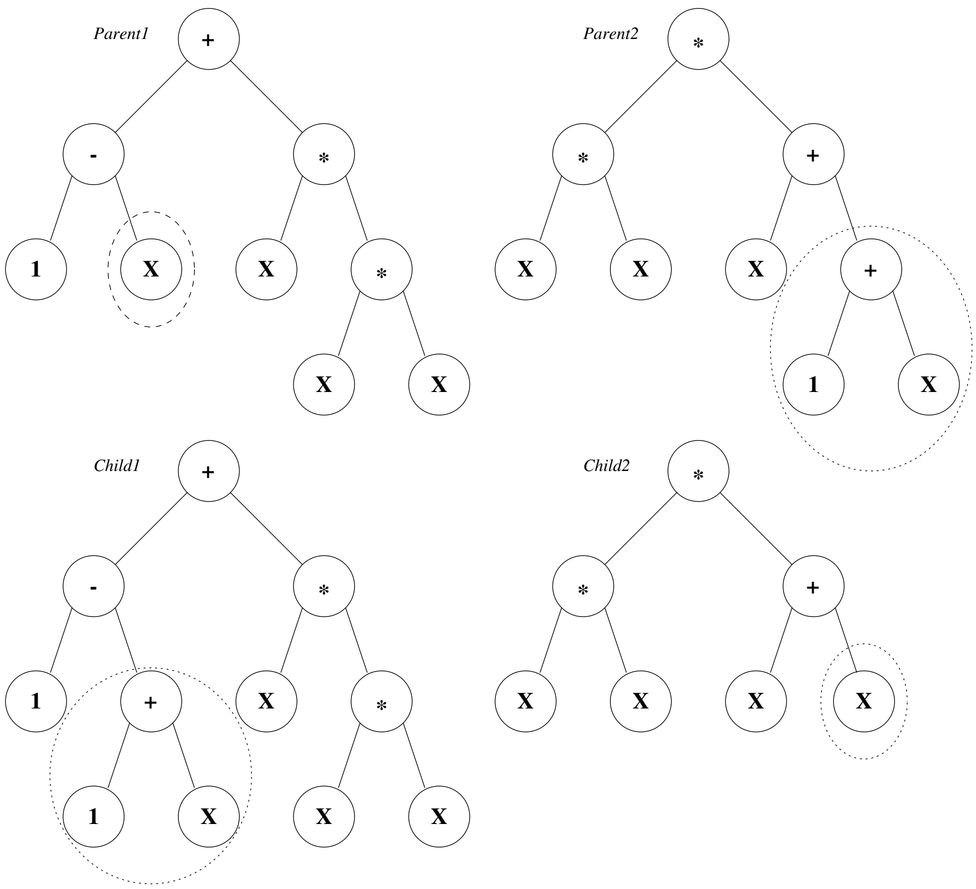


Figure 2.4: GP crossover

proportion to how well they perform.

The reproduction, mutation and crossover operators work with program parse trees instead of bit-strings. Reproduction creates a clone of the selected individual. Mutation works by copying the parent parse tree, selecting a node in the copy at random and then replacing it with a randomly created subtree (figure 2.3). The latter is created using the same process used for generation of the initial population. The crossover operator works by copying the parse trees of the parents, selecting random nodes from the copies and swapping them (figure 2.4). Although two children can potentially be created, most GP implementations discard one of them.

The standard GP algorithm which is described with pseudo code in table 2.1 is very similar to a GA. The main difference is that whereas in a GA mutation follows crossover, in GP mutation is used as an alternative to crossover. The most common selection methods used in GP are fitness proportionate selection and tournament selection, and work in the same way as they do in GAs.

The initial population is created by randomly generating program parse trees. The parse trees are formed by selecting functions and terminals from problem specific sets defined by the user. The specific selection mechanism will be described later in the section on run parameters. To

Table 2.1: The GP Algorithm

```
initialise population with randomly generated individuals
evaluate the fitness of the individuals in the population
while the termination criteria has not been met
     for p = 1 to size of population
         choose a genetic operator
         select parent(s) from current population based on fitness
         create offspring using selected operator and parent(s)
         place offspring in the new population
         evaluate the fitness of the offspring
     endfor
     the new population becomes the current population
endwhile
```

allow arbitrary trees to be generated, the functions in the function set must accept arguments chosen from any of the primitives in the function and terminal sets. This property is called *closure*. During tree generation, whenever a function is selected, subtrees for its arguments are also created recursively using the same process. Tree growth is restricted both naturally when the tree has leaves on each outer node, and artificially when a branch of the tree reaches the maximum allowable depth. Other possibilities include when the tree has the maximum allowable nodes, this is typically used with linear representations. To prevent very short trees the root node is usually always chosen from the function set.

2.3.1 The Five Basic Steps

There are 5 preparatory steps required to code a GP application [Koz92b]. These involve defining the following:

1. *The Terminal Set*

2. *The Functions Set*

3. *The Fitness Function*

4. *The Run Parameters*

5. *The Termination Criteria*

Koza [KDBK99] describes the first two steps as defining the search space for the problem, and the fitness measure as a means of determining the outcome of the search.

2.3.2 The Function and Terminal Sets

The function and terminal sets together form a limited language with which GP can generate programs. The function set contains the primitive functions in the language that take one or more arguments. These functions are selected because they a-priori are believed to be useful for solving the problem. The functions can be typical constructs found in most programming languages such as:

- conditionals (if, if-then-else, or switch statements)
- loops (while, for, do-while)
- recursive function calls
- variable assignments
- indexed memory operations (array manipulation)
- math functions
- logical and bitwise operations

Alternatively they can be user defined functions peculiar to the application domain, such as *If-CellOccupied*, or *sendMessage*. It should be noted that the inclusion of loops and recursion create the possibility that a program loops or recurses indefinitely, thereby making fitness evaluation difficult. Many solutions to the iteration [Koz94b; KDBK99], and recursion [Bra95; Bra96; WL96; YC98; KDBK99] problems have been provided.

The terminal set contains the variables, constants, zero-argument functions, and *random ephemeral constants* for the application. Random ephemeral constants are real numbered or integer constants that are initialised with random values at the start of the run and maintain their values throughout the run.

2.3.3 The Fitness Function

The fitness function measures the success (fitness) of an evolved program. The value is typically obtained by executing the program for a set of test cases, and measuring how closely the programs generates the desired output. The fitness of a program can also contain other measurements such as efficiency and size of code. The measured fitness of a program is used to determine whether it will be selected to take part in the production of the next generation. Good fitness functions are those that are accurate and highly sensitive to differences in performance of the programs.

The fitness measure can be thought of as a means of specifying the high-level requirement specifications for the solution. It describes "what needs to be done but not how to do it" [KDBK99].

2.3.4 The Run Parameters

Two of the most important parameters are the *population size* and the *number of generations*. Very small populations are disadvantaged in that they may not contain sufficient information (genetic material and fitness information) that can be exploited to explore better parts of the search space. A GP run with a small number of generations is disadvantaged in that there is less opportunity to exploit the information in the population. Large populations and generations both have performance implications that limit the values that we may choose. Unfortunately choosing appropriate values for these parameters is largely trial and error.

The reproduction, crossover and mutation probabilities are used to select the genetic operator to be used to create offsprings. They specify the probability with which each of the three genetic operators are chosen. Reproduction and mutation probabilities are usually set to be very small, mutation rates of less than 0.05 are quite common.

The shape and depth of the trees created for the initial population can be specified by the creation method, and the maximum depth parameters respectively. There are two creation methods commonly in use called the *full* and *grow* methods. The full method creates symmetric trees that are of maximum depth along any branch from the root. This is achieved by selecting the root node randomly from the function set, and then continuing recursively to select randomly from the function set for sub-nodes until the maximum allowable depth is reached (at which point a node is selected from the terminal set).

Selecting the grow method creates asymmetric trees. As with full trees, the root is always chosen from the function set, this time however we continue recursively by selecting children from the function and terminal set, until either a terminal is selected or the maximum depth is reached (at which point a terminal is always selected).

A third method called *ramped half-and-half* is used to ensure population diversity in the initial population. The size of trees created is ramped up from 2 to the maximum allowable depth. Half of these trees are created using the full method, and the other half are created using the grow method.

Luke [Luk00] points out that these tree generation techniques suffer from some important deficiencies including the inability to create trees of either a certain fixed size, or average size. Furthermore, there is no way to fine tune the preference of some functions over others. Luke developed two alternative tree creation algorithms to overcome these shortcomings. Variants of these algorithms are designed to work with either standard GP with *closure* or strongly typed GP (described later in this chapter). Probabilistic tree creation algorithm 1 (PTC1) and its variant Strongly-Typed PTC1 can be used to generate trees expected around a user specified size. PTC2 and its variant Strongly-Typed PTC2 is like PTC1 except that it also allows the user to control the *variance* in tree size. This is achieved by allowing the user to specify a probability distribution of requested tree sizes.

2.3.5 The Termination Criteria

The termination criteria determines when the algorithm should stop. Typical termination criteria include:

- the number of evolved generations exceeds some maximum
- the fitness of the best individual reaches the desired value
- the genetic information known to be needed for the solution has been lost

The first and last of these effectively abort a less than successful run.

2.4 GP Extensions

The original GP technique has been advanced with many extensions. Those that have been used or referenced later in this thesis are described in this section. A more complete survey of techniques can be found in [LQ95] and in [BNKF98].

2.4.1 Automatically Defined Functions

If we chart the development of programming languages, we see a steady trend of higher and higher levels of abstraction being applied. The earliest languages were machine code and were quickly replaced by assembly level languages. These in turn lead to the development of high level languages such as FORTRAN. The latter facilitated code reuse first in the form of subroutines and later in the more easily parameterised forms of functions and procedures. Useful functions were made available in the form of libraries which are themselves a higher form of functional abstraction. The introduction of object oriented languages which support data abstraction or abstract data types (ADTs) further increased the amount of code reuse.

The advantages of abstraction are that it allows the control of complexity and facilitates code reuse. Code reuse is particularly useful for GP because it means that a block of useful code need not be independently rediscovered at different places within the program where it is needed. In this section we look at a key technique that has been used to introduce abstraction into GP.

Automatically Defined Functions(ADFs) [Koz93; Koz94b] are evolvable functions (subroutines) within an evolving genetic program, which the main result producing branch (RPB) of the program and possible other ADFs can call. Each ADF is a separate tree; consisting of its own terminal and functions sets. The terminal sets for ADFs that take arguments will also contain terminals for each argument. These terminals get instantiated with the values of the parameters that have been passed when the ADF is called during fitness evaluation. The functions sets for the ADFs can contain primitive functions, as well as other ADFs that have been defined. The result producing branch is just a special ADF which is called by the fitness function when evaluating the program. ADFs effectively act as functional units or building blocks that can be combined to create programs.

The crossover and mutation operations are modified to work with ADFs, so that mutation mutates an ADF at random, and crossover exchanges subtrees between the same ADFs of each individual. Depending on the implementation crossover will either be performed for a single ADF or the RPB, or on all ADFs and the RPB.

ADFs have been successfully used on problems that proved too difficult for genetic programming without ADFs. The addition of ADFs adds an extra step to the preparatory steps for a GP application. The extra step defines the architecture in terms of number of ADFs and the arguments that they take, and the result producing branch.

2.4.2 Strongly Typed GP

Koza's early GP system was restricted in that, functions and terminal were designed to be all of the same type. This property that Koza calls *closure* allowed functions to take any terminals or function calls as arguments, and return any terminals or function calls as results. The advantage with this approach was that it greatly simplified tree generation and their manipulation by the crossover and mutation operators.

Montana [Mon95] relaxed this restriction in his Strongly Typed Genetic Programming (STGP) system thus allowing functions and terminals to have type specifications. STGP also supports generic types, which help reduce the number of different function definitions that are needed. In the implementation, the GP algorithm was modified to ensure that arguments to functions are always of the correct type, and that the generated program (or ADF) returns the correct type. Tree generation was facilitated by type possibility tables (computed at the start of a run) that list functions and terminals available at a given tree depth. Care is taken so that when a tree is mutated the root of the subtree replacing the node that is mutated is of the same type. Similarly crossover is restricted so that only subtrees with roots of the same type may be exchanged. Montana argues that the increase in complexity in the GP algorithm is more than compensated by the considerable reduction in the size of the search space, which in turn reduces the effort required to find solutions.

A concern with this approach, is the effect that it has on the connectivity of the search space. Highly restrictive grammars may make the search space too sparse, making it difficult to move from one point to another. This in turn is likely to cause premature convergence.

2.4.3 Steady State GP

The standard GP evolutionary engine is described as *generational* because it works in stages called generations. After the initial population has been created, each subsequent generation is created by applying genetic operators to individuals selected from the last generation to form an entire new population which replaces the old. New individuals are not made available for reproduction until the entire new population has been created.

In a *Steady State GP (SSGP)* engine [Rey92; Rey94b; Rey94c], there is no notion of new or old generations, instead there is just a continuous change of individuals in the population. When a new individual is created it replaces an existing individual in the population, and hence immediately becomes available for reproduction. The benefits of this approach are that it is memory efficient, and that it is easy to parallelize. It is memory efficient because only one population needs to be in memory, whereas with a standard generational engine, two are required.

2.4.4 Indexed Memory and Data Structures

In his earlier work, Koza used named memory to allow state to be maintained. The function Set-Value in conjunction with a terminal representing a named memory location(variable) can be used to set the value of that memory location. Teller [Tel94] added indexed memory capability to GP. He used an array of memory locations which could be manipulated via read and write functions. The latter take the index of the memory location as an argument.

Langdon [Lan95; Lan96a; Lan96b; Lan98] used GP with indexed memory to successfully evolve queue and stack abstract data types (ADTs) using a chromosome structure with multiple trees. Each tree in the chromosome was used to represent one of the required ADT member functions. The results suggest that it may be possible to extend the level of abstraction in GP from pure functional abstraction as in ADFs to full ADTs. This may help increase the scalability of GP.

2.4.5 Co-Evolution

In nature organisms interact not just with their environment, but also with each other. Sometimes they are in direct competition with each other as happens with predators and prey, or parasites and hosts. The fitness of these organisms becomes relative to each other. A predator is deemed fit if it can catch prey necessary for its survival. Conversely, a fit prey is better able to escape predators. Dawkins [Daw86] describes how such *competitive co-evolution* leads to an arms race, where each organism evolves to counteract any improvement by evolution of its competitor.

This approach has been applied to GP [Koz91; AP93; Jan94; Rey94a; HSSW95b], and involves creating and evolving two or more populations instead of one. The fitness of individuals in one population is measured relative to individuals in the other and vice versa.

We can do this by testing each individual in one population against:

1. each individual in the other
2. a random subset of the other
3. the current best of the other

The first method is likely to be too compute intensive to be practical and hence one of the other two techniques is normally used.

In *cooperative co-evolution* (which is comparable to symbiosis in nature [Sap94]) the programs are tested together, and pairs of programs that work well together are rewarded with higher fitness.

2.5 Theory

2.5.1 Fitness Landscapes

The notion of a fitness landscape is a powerful tool for visualising the difficulty of a problem for evolutionary computing [Jon95]. Fitness landscapes are created by representing individuals as points on a landscape, where points that are higher with respect to their neighbours represent fitter individuals than their neighbours. Peaks on this landscape therefore represent locations of high fitness, and the valleys locations of poor fitness. Points nearest to each other are those that can be reached by a single application of the search operator (crossover or mutation).

When we generate an initial random population of individuals, we are effectively placing a mesh over the fitness landscape, simultaneously sampling the fitness landscape at different places. Genetic operators such as mutation and crossover then provide a means for exploring this landscape. An assumption of the GA is that if we select individuals of higher fitness, then by applying the genetic operators, we should end up on the landscape at a point of high fitness. Thus there should be some correlation between parents and children.

The ruggedness of a fitness landscape defines the search difficulty. A particularly difficult problem for a given representation, fitness function and search operator might result in a fitness landscape which is flat and virtually featureless, with the exception of a few widely distributed spikes (figure 2.5). Conversely a very easy problem might consist of a “mountain” with a single peak (figure 2.6).

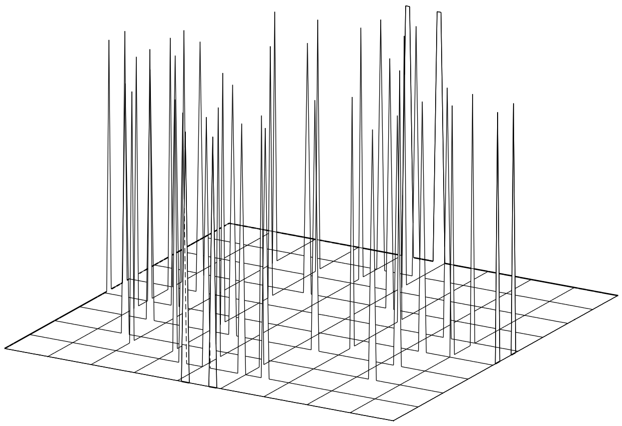


Figure 2.5: A difficult to search fitness landscape.

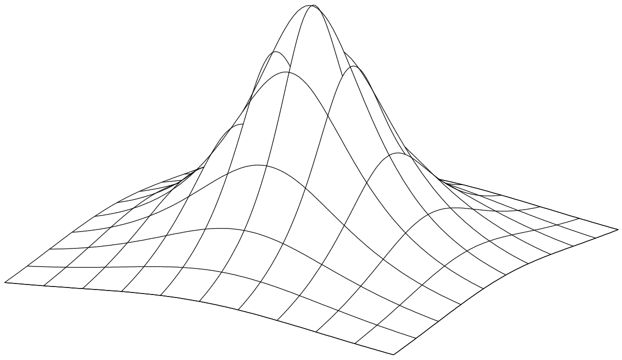


Figure 2.6: An easy to search fitness landscape.

Kinnear [Kin94] examines the structure of the fitness landscape on which GP operates, and analyses the landscapes of a range of problems of known difficulty in order to determine the correlation between landscape measures and problem difficulty. The landscape analysis techniques that he employs include adaptive walks, and the autocorrelation of the fitness values of random walks. His results indicate that the former shows better correlation with the problem difficulty than the latter for GP.

2.5.2 GP Schema Theorem

A number of attempts have been made to provide a schema theorem (ST) for GP. All the early attempts were position-less STs, in which information about the position of the schema components were omitted. Position-less schemas can be instantiated many times in the same program. Koza defined a schema as a set of trees all having one or more subtrees in common [Koz92b]. O'Reilly and Oppacher [OO95] extended this definition to include incompletely defined expressions called *tree fragments*. A tree fragment is a tree with at least one "don't care"(symbol #) leaf. In their definition a schema is multiset of subtrees and tree fragments.

In later attempts positioned schemas were used, in which schemas are represented using rooted trees or tree fragments. Rooted schemas unlike position-less schemas, can be instantiated no more than once in a tree. Rosca [Ros97; RB99] devised a rooted tree schema theorem in which a schema is a tree composed of the same function and terminal sets as used in the run, with the exception that the terminal set includes "#", which can stand for any valid subtree. In Poli and Langdon's schema [PL97], the "don't care" symbol ("=" in their case) is not restricted to appearing in the leaves of schema trees, and furthermore instead of representing any valid sub-tree, it represents a single node. A schema therefore represents the set of trees that are of the same size and shape, and which have the defined nodes of the schema in the same places. Their schema theorem applies only to one-point crossover, and asymptotically converges to Holland's GA schema theorem. More recently Poli in his hyperschema theorem [Pol00] allows terminals in a schema to contain either a wild-card that represents any node("="), or any valid sub-tree.

2.6 GP Applications

GP has been successfully applied to a very wide range of problems including:

image processing	*computer graphics*
process control	*autonomous agents*
robotics	*data mining*
electrical circuit design	*neural networks*
modeling	*natural language processing*
classification	*pattern recognition*
signal processing	*art*

A survey of some these applications can be found in [LQ95; BNKF98; Koz92b; Koz94b; KDBK99]. William Langdon maintains a bibliography of published material at:

"http://www.cs.bham.ac.uk/~wbl/biblio/README.html"

2.7 GPsys

To help advance research in Genetic Programming we designed and developed an extendible GP system in the Java programming language. This environment was used for all the experiments in this thesis. GPsys can be downloaded from:

"http://cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html"

Details of the implementation of GPsys can be found in appendix A of this thesis.

2.8 Conclusion

We can think of GP from two different perspectives. From a *machine learning* perspective GP is a general technique that allows a machine to learn how to solve a given task from training examples (the test cases). ML researchers would describe GP as a kind of *beam search* in which the population of computer programs is the beam, and the evaluation metric is the fitness function [Mit97]. From an *algorithmic complexity theory* perspective [LV97], we can view GP as compression algorithm which creates a compressed representation (the computer program) of a set of input/output tuples (the test cases).

Chapter 3

Intelligent Agents and Multiagent systems

In this chapter we provide a brief overview of the concepts of agents, and multiagent systems and look at the key problems in designing and implementing them. A survey of the application of Genetic Programming to the automatic programming of agents is presented, setting the background for the rest of the thesis.

3.1 Agents

There is no universally agreed definition of the term agent, the definition of agents that we use in this thesis is that used by Jennings and Wooldridge [JSW98; WJ95]:

“an agent is computer system, *situated* in some environment, that is capable of *flexible autonomous* action in order to meet its design objectives.”

Their definition is based on the following concepts:

autonomy agents have the ability to act without the direct intervention of humans or other agents, and can control their own actions and internal state

situatedness agents are situated in some environment which they can at least in part perceive and manipulate

flexibility agents possess the properties of *responsiveness*, *pro-activity* and *social ability*

responsiveness the agents perceive their environment and respond in timely fashion to changes that occur in it

pro-activity agents should not simply act in response to their environment, they are able to exhibit goal-directed behaviour by *taking the initiative*

social ability agents should be able to interact with other agents (and possibly humans) to complete their own problem solving and to help others with their activities

They further define an *agent-based* system as one in which the key abstraction used is an agent. Such a system could be comprised of one or many agents. When they are composed of more than one agent they are referred to as *multiagent systems*. A key question in agent research is how to build agents that have the desirable properties stated above. A number of architectures for building agent-based systems have been proposed. These can be classified into deliberative, reactive and hybrid architectures.

3.1.1 Deliberative Architectures

Deliberative architectures are derived from classical approaches to artificial intelligence, and treat agents as knowledge based systems [JSW98; WJ95]. The emphasis is therefore on knowledge representation techniques and inference methods. Typically a model of the environment, the goals of the agent and its capabilities are represented symbolically, and some form of symbolic reasoning is used to determine which action to perform. A weakness with this approach is that the time taken for such reasoning may invalidate or reduce the utility of the action, so that responsive behaviour may not be possible. Furthermore, symbolic representation of some environmental information may not be easy to accomplish (consider visual information).

BDI (Belief, Desire and Intention) agents are a class of deliberative agents that draw inspiration from the theory of human practical reasoning developed by the philosopher Bratman [Bra87]. Such agents are built from components consisting of data structures representing the *beliefs, desires and intentions* of the agent, and functions that represent its *deliberation and means-end reasoning*. The information that the agent has of its environment is represented by its beliefs. The desires represent options available to the agent, which become intentions if the agent decides to commit resource to achieving them. Deliberation is the process of deciding what goals (desires) we want to achieve, and means-end reasoning determines how to achieve them [Wei99; JSW98].

3.1.2 Reactive Architectures

Reactive architectures pioneered by Brooks [Bro86] take a radically different approach to building agents than that of classical AI, and reject symbolic representation and reasoning. Instead they stress the importance of *situatedness and embodiment*, in the creation of intelligent rational behaviour. Intelligent behaviour is seen as *emerging* from the interaction of an agent with its environment and from the interaction of simpler behaviours [Bro91]. To test these ideas, Brooks developed an an agent control architecture called the *subsumption architecture* with these properties. In the subsumption architecture an agent is composed of a collection of *task accomplishing behaviours* arranged in layers. The task accomplishing behaviours themselves are extremely simple, consisting of finite state machines that map sensory input to some action. To prevent multiple behaviours from being simultaneously activated, the layers have priorities, such that lower layers can inhibit behaviours in upper layers.

3.1.3 Hybrid Architectures

Hybrid architectures attempt to combine both deliberative and reactive approaches. These architectures often use vertical or horizontal software layers arranged in a hierarchy, with different levels of abstraction. The distinction between the two layering methods is that with horizontal layering, all layers have access to sensors and effectors, whereas with vertical layering, sensors and effectors are each dealt with by at most one layer. Typically three layers are used, which starting from the lowest layer include the reactive level, the knowledge level and the social knowledge level. The knowledge level has a symbolic representation of the agent's environment, and the social knowledge level has a symbolic representation of other agents [Wei99; JSW98].

3.2 Multiagent Systems

Distributed Artificial Intelligence (DAI) is a subfield of artificial intelligence and is concerned with situations in which several systems, interact in order to solve a common problem [CC96]. These systems may include both humans and machines. Whereas in classical AI, the metaphor of intelligence is based on an *individual human behaviour*, and the emphasis put on knowledge representation and inference methods, the metaphor used in DAI is based on *social behaviour*, and the emphasis is placed on actions and interactions [SDB92].

Research in DAI is traditionally divided into two subfields, *distributed problem solving (DPS)* and *multiagent systems* [CC96; Gas91]. DPS is concerned with dividing the work involved in solving a particular problem amongst a number of nodes that divide and share knowledge about the problem and developing solution. The emphasis in DPS is on task decomposition and solution synthesis. MAS research is concerned with the behaviour of a collection of autonomous agents aiming to solve a given problem, where the problem being solved is often beyond the knowledge and capabilities of any individual agent [Wei99; CC96]. Weiss uses a modern definition of multiagent systems which is interchangeable with DAI and covers both DAI subfields [Wei99]. This is the definition that is used in this thesis. Potential benefits of MAS are increased reliability, better performance and lower software complexity.

The characteristics of multiagent systems are as follows [JSW98]:

- Each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;

- data is decentralised; and
- computation is asynchronous.

Owing to these characteristics the task of designing such agents is difficult. The following list taken from [JSW98] represent the key problems in the design and implementation of MAS:

1. How to formulate, describe, decompose, and allocate problems and synthesise results among a group of intelligent agents?
2. How to enable agents to communicate and interact? What communication languages and protocols to use? What and when to communicate?
3. How to ensure that agents act coherently in making decisions or taking action, accommodating the non-local effects of local decisions and avoiding harmful interactions?
4. How to enable individual agents to represent and reason about the actions, plans and knowledge of other agents in order to coordinate with them: how to reason about the state of their coordinated process (e.g. initiation and completion)?
5. How to recognize and reconcile disparate viewpoints and conflicting intentions among a collection of agents trying to coordinate their actions?
6. How to effectively balance local computation and communication? More generally how to manage allocation of limited resources?
7. How to avoid or mitigate harmful overall system behaviour, such as chaotic or oscillatory behaviour?
8. How to engineer and constrain practical DAI systems? How to design technology platforms and development methodologies for DAI?

Our general answer to these questions is that by using GP to evolve agents, we can simultaneously solve many of these problems. Furthermore, we are suggesting that GP could be used as a general methodology for developing multiagent systems.

3.3 Evolving Agents with GP

We can apply GP to agent-based computing in the following ways:

- use GP as a learning or planning technique for agents
- evolve complete agents

- combine both (use GP to evolve agents that themselves use GP as a learning technique)

All of the research applying GP to agents to date has focussed on the first two approaches. Much of the research in GP suggests that GP can be used as very generic learning technique, and therefore its use as a learning technique for agents is highly attractive.

3.3.1 GP as a Learning Technique for Agents

Clack *et al* [CFLY97; Cla97] for example use GP to evolve document classifying agents. GP is used to search the space of document classification expressions to find expressions that correctly classify the users information needs. The evolutionary process need not ever end, allowing the classifications to co-evolve with changes in the users information needs.

Since plans are effectively just computer programs, the use of GP for plan generation is tempting. The Genetic Planner devised by Handley [Han93; Han91; Han94b] achieves planning by searching the space of plans. The search space is defined in the usual GP way as the set of functions (operators in planning terminology) and terminals. Plans are evaluated for fitness in a simulation of the world in which they are to be executed. Handley showed that he could generate plans for navigation of a robot on a 2d world without any reasoning mechanism, and thus avoid traditional planning issues such as the Sussman anomaly, and the dynamic and temporal world problems [RK90]. Furthermore since we can stop the GP process at any point and ask for the best evolved solution, Handley argues that the genetic planner is an *anytime algorithm*. A key problem with this approach is that it requires a model of the environment, so that we can evaluate the relative success of a plan without executing it. Furthermore measuring the relative success of a plan requires generation of a fitness function, consider for example a plan that needs to switch off a light in a room. Basing a fitness function on the desired outcome, which is that the state of environment has changed so that the light is switched on, would not work. Partial plans that get close to the switch for example would not be rewarded in this scheme. A second problem is the time and space problems associated with evolving a large population of plans.

3.3.2 Evolving Complete Agents

Work done on applying GP to evolve agents has been focussed on mainly reactive agent architectures. There are three key techniques that have been developed. The first is for creating homogeneous agents, and involves evolving a single GP that is instantiated many times, once for each agent [Koz94a; Koz91]. Fitness evaluation is performed by executing the same program in the context of each agent repeatedly for a fixed number of iterations, or until the task that the agents are trying to solve is completed. The fitness value assigned to the GP is measured in terms of how close the agents were to solving the problem at hand. The second technique is used to create heterogeneous co-operating agents, and involves evolving a GP that consists of multiple trees (similar to ADFs), one for each of the heterogeneous agents [HSSW95a; HS96a; HS97]. During fitness evaluation each agent is instantiated using one of the trees in the GP. The fitness is assigned in the same way as with homogeneous agents, so that the heterogeneous agents are treated as a unit or team. The advantage with these architectures is that they avoid the *credit assignment problem* [Min63], i.e. how to assign credit to each of the participating agents. The third technique is to co-evolve competing agents by using separate populations for each of the competing agents [HSSW95b; HS95a]. During fitness evaluation agents in one population compete against agents in the other, so that the fitness of agents in one population are relative to their performance when tested with agents from the other population. Competing agents therefore do not share fitness unlike the team approach mentioned above. Although this technique could also be used to co-evolve co-operating agents [Iba96], the problem of credit assignment makes this approach more difficult.

Koza used GP to evolve wall following behaviour for a simulated autonomous mobile agent using the subsumption architecture [Koz92a]. The wall following problem was originally devised by Mataric [Mat90], and consists of a robot situated in an irregular room that has the task of following the walls of the room. To guide it, it is provided with 12 sonar sensors, each covering 30 degree sectors, and an additional sensor which could detect if the robot had stopped. The robot is capable of moving forwards and backwards by a constant distance, rotating left and right by 30 degrees, and stopping. Mataric developed a subsumption architecture comprising of 4 programs that control the robot called stroll, avoid, align and correct. This architecture was sufficient to achieve the task without any modeling or planning as is used in the design of non-reactive robotic systems. Koza, showed that GP could be used to evolve a subsumption architecture for this problem automatically. This paved the way for using GP to evolve reactive agents.

Koza evolved homogeneous reactive agents to solve the “painted desert” problem [Koz94a]. In his definition of the problem, there are 10 ants and 30 grains of sand each of three colours. The 10 ants and 30 grains are placed at random positions on a two dimensional torroidal grid of size 10 by 10. The ants are driven by a common program (the ants are homogeneous), and can only sense information about their current grid position. No direct communication between the ants is permitted. The goal is to arrange the grains in vertical bands, where grains in any given band are of the same colour. In addition a band of a given colour must always occupy the same predefined column on the grid (black, grey and striped bands occupy columns 1,2 and 3 respectively). Parallel movement of the ants is simulated by executing the evolved program for each ant sequentially. This process is repeated for 300 iterations. Using simple function and terminal sets and a suitable fitness function, Koza was able to evolve a complete solution to this problem, and hence demonstrated that homogeneous reactive multiagent systems that cooperate to solve a task can be evolved using GP. A similar approach was used to program ants (homogeneous reactive agents) that collectively find and transport food in an efficient manner to their nest [Koz91].

Andre [And94] used GP to evolve programs that are capable of storing a representation of their environment (map-making), and then using that representation to control the actions of a simulated robot. He used GP to effectively co-evolve two programs (each making use of ADFs) for each individual. He split the fitness evaluation of each individual into two phases. In the first phase, the first program is run to extract and store features from the environment. Only the first program is provided with the set of functions to probe the environment. In the second phase, the second program is executed to use the stored representation to control the actions of a robot. The fitness value was awarded only for the success of this second phase. He found that using this approach, he was able to successfully evolve programs that solve simple problems by examining their environment, storing a representation of it, and then using the representation to control action. These programs can be viewed as communicating agents.

Reynolds [Rey94a] used competitive co-evolution to evolve vehicle steering control programs for pursuers and evaders in the game of tag. He obtained near optimal solutions without the use of either expert pursuers or expert evaders.

Haynes [HW95] evolved programs to control an autonomous agent that needs to survive in a hostile environment. The simulated environment consisted of a 2 dimensional grid of cells which can contain other agents, mines and energy. The goal of the agents was to sense and mark the location of the mines and energy by moving through the minefield. Sensing was made possible through tallies available at each cell which count the number of these items available in neighbouring cells (like the game minesweeper). The agent also has access to memory containing information about visited cells. Using a fitness function that uses fluctuating environments he was able to evolve a robust implementation that could handle any environment. Haynes and Sandip [HWS94; HW95; HWS95; HWSS95; HS95b; HSSW95a; HS95a; HSSW95b; HS96c; HS96a; HS97] evolved both homogeneous and heterogeneous agents for the pursuit domain. They also competitively co-evolved predators and prey. Their work used the techniques mentioned at the beginning of this section and is detailed in the next chapter.

Luke and Spector [LS96] compared *cloned* (homogeneous), *free* (heterogeneous with crossover allowed between trees representing different agents) and *restricted* (crossover permitted only between trees representing the same agents) breeding policies for a predator/prey domain called the Serengeti. They also compared a range of coordination mechanisms including *no sensing* (in which the predators were unable to sense each others positions), *deictic sensing* (in which the predators could sense each other in a relative manner such as the position of the nearest agent), and *name-based sensing* (the position of other agents can be sensed referenced by their names). They found that the heterogeneous approaches produced better results than the homogeneous approaches, and furthermore that the restricted breeding method worked best. Name-based sensing was found to have produced better results than deictic sensing, which in turn produced better results than no sensing.

Qureshi [Qur96] was first to show that GP could be used to evolve agents that use *send* and *receive* operators for explicit communication. He evolved homogeneous agents for a co-operative domain that learn both how and what to communicate. This work is detailed in chapter 5.

Iba compared homogeneous, heterogeneous and co-evolutionary (co-operative) breeding of 2 agents for the tile world problem [Iba96]. He used two different tile world problems, and found that for the first, the heterogeneous worked best, and for the second the co-evolutionary approach worked best. In his co-operative co-evolution approach, he used three populations, one population for each agent and a third as common pool for evolving shared building blocks. Individuals from this shared pool were migrated from the common pool to the other populations (10 per generation). The fitness of the individuals from the two agent populations were evaluated in the first generation by picking random partners from the other, and subsequently by teaming with the best of the other population. Iba also performed experiments in which he evolved both homogeneous and heterogeneous agents that use send and receive primitives similar to that used by Qureshi [Qur96] for a robot navigation domain [INU97; Iba98; Iba99]. The *send_i* deictic operators that he used sends the the position vector to the ith nearest agent (there are 4 of these operators, one for each of the four agents), the receive operator returns the first message from a FIFO queue for the receiving agent. In the same experiment he also used specialised *Send_iS, Send_iR, and Send_iY* operators (12 in total one of each of the four nearest agents) to send stop, random and yield commands to the specified agents. These commands cause the recipients to stay in the same place, move randomly, and move to one of the adjacent vacant square respectively. The targeted agents do not execute any explicit receive primitive to achieve these tasks, nor do they evolve any code to perform the intended task, or have in control of whether the code gets executed. Instead the behaviour is precoded in the simulator, and hence these operators are more like direct control operators. He also evolved heterogeneous predatory agents for the pursuit domain that use name-based primitives called *COM1, COM2, COM3, COM4* which request position information from predatory agents 1, 2, 3 and 4 respectively. The effect of this operator is to return the displacement vector from the calling agent to the target if the receiving agent is within some fixed range of the prey, otherwise the only argument to the function is returned. The agent that is the recipient of request however does not evolve any code to respond to the request, instead this is precoded in the simulator. The operator is therefore more like a conditional sensor. More details of this work are provided in the next chapter.

Figure 3.1: Soccerserver screenshot

Iba also evolved heterogeneous agents that use ACL-like communication commands (*propose, accept and reject*) to negotiate cooperation in the tile world domain [Iba99].

The approach used by Haynes to evolve a team, represents each agent as a separate tree, rather like an ADF. The number of agents to be evolved is therefore specified in advance by the number of trees. By using Koza's architecture altering operations [KDBK99], Bennett [Ben96a; Ben96b] developed a method for evolving both the number of agents and their code.

A number of researchers [LHF+97; Luk98; AT99] have investigated the use of GP to evolve agents for robotic soccer played in a simulator (Soccer Server [Its95] as shown in figure 3.1). In the RoboCup competition (the softbot part), soccer teams consisting of 11 players (softbots) play in real-time tournaments against other teams. Luke *et al* [LHF+97; Luk98] evolved players for their team using GP. Each player consisted of a move program and a kick program. Luke *et al* investigated the possibility of evolving both homogeneous and heterogeneous agents for this domain. The evolution of heterogeneous agents would have required that each individual in the GP population consisted of 22 trees (11 move and kick tree pairs). They therefore chose to evolve pure homogeneous agents and hybrid *pseudo-heterogeneous* teams. The latter consist of a smaller number of heterogeneous agents than required, which are instantiated multiple times to form *squads* which together generate a team of 11 (the team therefore consists of sets of homogeneous agents which are heterogeneous with respect to each other). For both homogeneous and pseudo-heterogeneous approaches they disallowed crossover between kick and move trees. Furthermore they used their restricted crossover approach mentioned earlier to restrict crossover in the pseudo-heterogeneous approach so that it occurs only between the same squads. To allow fitness evaluation, they used a competitive fitness approach (they were hence co-evolving teams). The team represented by each individual was made to compete against the team represented by another individual chosen randomly from the population. Their fitness measure was based crudely on the number of goals scored. They found that the number of goals scored by individuals in the initial population were surprisingly high. The best evolved solution which was of homogeneous type, was submitted to the RoboCup-97 competition and won the first two games against human-crafted opponent teams.

Andre and Teller [AT99] used a similar approach to create a robotic soccer team which they call "Darwin United". They created heterogeneous agents (represented as 11 trees per individual) which also share 8 ADFs (8 further trees per individual). A restricted breeding scheme is used so that crossover occurs between trees representing the same player, and between the same numbered ADFs. Unlike the approach used by Luke *et al*, Andre and Teller use indexed memory, so that the evolved agents are no longer just reactive. They also use a much more complex fitness evaluation approach. A fitness value for a team consists of an ordered list of scores with items ranging from the least important (getting close to the ball and kicking it) to the most important (scoring a goal and winning the games). Fitness evaluation involves pitting the team represented by the individual being evaluated by graduations of progressively skillful players. The individual gets to play successive teams if it succeeds against earlier teams, the score is the average across the games played. The teams that the individual is pitted against range from an empty field (no team) to the winning team from the 1997 Robocup competition . If the team represented by this individual succeeds then it is also pitted against the teams represented by three individuals who also got to this stage.

3.4 Conclusion

The elegance of using GP for programming multiagent system comes from being able to search for desired behaviour at both the individual and group level simultaneously. Using GP we can avoid having to deal with the *micro/macro problem*. We generate agents at the micro level, and evaluate them at the macro level. The agents with the best evaluations (fitness) are used to create new agents at the micro level. This process is iterated until desired micro and macro level behaviours emerge. The fitness function does not need to detail how each agent is to behave, or interact with each other or the environment. It just needs to provide a relative measure of how successfully the agents solve the global problem. The *potential* ways in which agents can interact with the environment and other agents is specified by the function and terminals sets. Further advantages are that the GP approach does not rely on extensive domain knowledge, and that it is focussed on performance, which in the end is what matters. Note that performance can be measured in a number of ways, ranging from how effectively a systems meets its goals to how efficiently this is achieved.

Chapter 4

Task Allocation and Conflict Resolution

Task allocation and conflict resolution are both key research areas in multiagent systems. In this chapter we show that GP can be used to evolve agents that automatically allocate tasks amongst themselves both *statically* and *dynamically*, and can resolve conflicts arising from their interaction.

4.1 The Pursuit Problem

The pursuit problem (also called the Predator/Prey problem) is a well studied testbed for DAI research. Benda *et al* [BJD86] formulated the original problem definition as consisting of four *predators* that need to capture a single *prey*. The predators and the prey are situated on a two dimensional world consisting of cells (figure 4.1). Movement in this world, is possible in orthogonal directions only, and takes the agents from one cell into another (figure 4.2a). The goal for the predators is to move so as to *capture* the prey (figure 4.2c). To capture the prey, the predators must occupy all cells immediately adjacent to the prey (which we call *capture positions* - figure 4.2b). The prey moves randomly, and is slower than the predators (usually implemented by ensuring the prey is stationary some percentage of the time). In most of the implementations, only one agent may occupy a cell at a time, and therefore conflicts can occur if one or more agents try to occupy the same cell (in figure 4.2d agents 0 and 1 are in conflict). Many variations to the original definition have been devised by changing key domain parameters, these parameters are described in [SV97] and listed in table 4.1.

Gasser *et al* [GRHL89] proposed a solution based on what they call the *Lieb configuration*. In the Lieb configuration the grid is divided into 4 quadrants by diagonal lines that pass through the cell occupied by the prey, and the predators each occupy a different quadrant. They proposed that the predators first try to achieve the Lieb configuration, and then follow a set of *Lieb Rules* to capture the prey.

Stephens and Merx [SM90] investigated the pursuit domain using three control strategies gov-

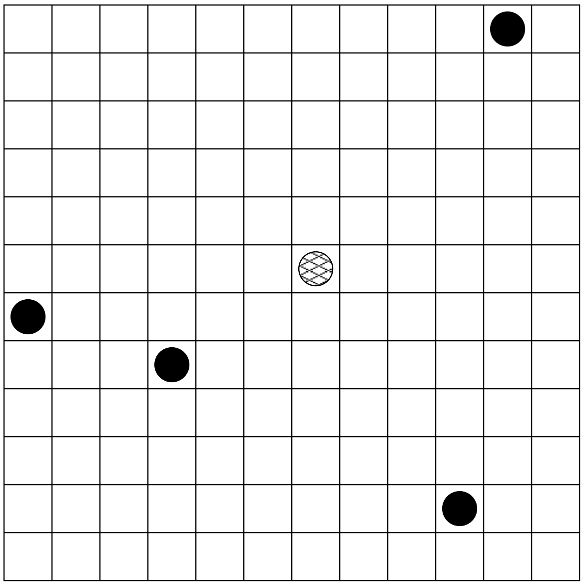


Figure 4.1: The environment of the pursuit domain consists of a two dimensional (usually torroidal) world in which four predators must try to capture the prey by surrounding it.

Table 4.1: Pursuit Domain Parameters

Parameter	Examples
grid size	finite, infinite
grid shape	square, hexagonal, continuous, torroidal, edged
capture definition	prey surrounded, predator on same square as prey
legal moves	orthogonal only, diagonals permitted
movement	agents take turns to move, agents move simultaneously
sensors	what can be sensed, by whom and from how far away
predator communication	permitted, disallowed, range, implicit, explicit
prey movement	random, stationary, deterministic
predator/prey numbers	1/4, 2/6

erning the predator agents. The control strategies included *local control, distributed control and central control*. In their local control strategy, the predator agents pursue their local goals and only communicate when they occupy a capture position by broadcasting that position. This information is picked up by the other predators and is used to update the list of potential local goals. In their distributed control strategy, the predators use both a shared convention, and communication to achieve capture. At the start of each move cycle, each agent computes the straight line distance to each of the capture positions and chooses the closest position as their intention. This intention is then broadcast to the other predators. The shared convention is that the agent

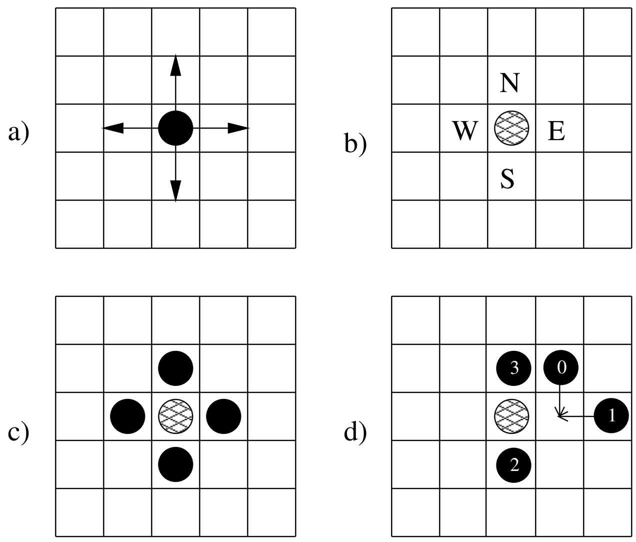


Figure 4.2: In the pursuit domain; a) movement is restricted to orthogonal directions, b) capture positions are cells immediately adjacent to the prey, c) the prey is captured when all capture positions are occupied by predators, d) conflicts occur when one or more agents try to occupy the same cell.

furthest (straight line distance) from their intended capture position is allocated that position. In the central control strategy, one predator (which has complete and accurate information about the position of the other predators) plans and broadcasts commands to other three. In all three control strategies, the predators had accurate information of the position of the prey. They tested each of the control strategies using 30 randomly generated initial placements of the predator agents (the prey was always centered) on 30x30 grid. All agents moved at the same speed, the moves were ordered to avoid collisions, and the prey moved randomly. In their results they report that the central control strategy was the most successful, and the local control strategy least successful. Central control however was less efficient than distributed control, and suffers from having a central point of failure.

Korf [Kor92] proposed that the pursuit problem can be solved without any explicit coordination. He investigated the use of purely greedy strategies for the predators. Korf conducted experiments using 100x100 *orthogonal, diagonal and hexagonal* grid variations, in which the prey actively tries to evade the predators by moving to a neighbouring cell that is the furthest from the nearest predator. In the diagonal grid configuration, the the agents can move diagonally as well as orthogonally, and consequently eight predators instead of four were used. In the hexagonal grid, 6 predators were used to cover each of the exits from a cell. Conflicts were avoided by ordering the movement of the agents, and the prey moved at 90% of the speed of the predators (the prey stays still with a probability of 0.1). The initial positions of the agents were randomly generated. In his simplest solution (which solves problems for all three grid configurations), the predators try to minimise their distance to the prey. The more efficient solution which he employs in the diagonal and hexagonal versions of the game, uses an attractive force towards the prey and a repulsive force between the predators. Korf concludes that these results lend support to the theory that much coordination and cooperation can be viewed as an emergent property of the interaction of greedy agents in the presence of environmental constraints.

Manela and Campbell [MC93] agreed with Korf that the 4 predators, one prey game could be solved without explicit coordination, but argued that the pursuit game could be made more interesting by having more than one prey. They used a GA to optimise parameters of hand coded predators that communicate. They also introduced the concept of *boredom* (similar to timeouts) to prevent predators from repeatedly executing the same actions without getting any closer to the goal.

Haynes et al [HWS94; HWSS95; HS95b; HSSW95b] used GP to evolve homogeneous predator agents for the pursuit domain. They argued that the ordered execution of moves as used by Korf is a form of communication that helps coordinate the predators. The agents in their version of the pursuit game therefore move concurrently. Agents that try to move to the same cell are bumped back to their original position. A predator may push another predator out of the cell that it occupies if it decides not to move. However this push operator is not available in the function set presented to the GP system. This form of conflict resolution therefore is executed by the environment rather than the agents. They use a 30x30 torroidal, orthogonal game board, in which the agents can move orthogonally or stay still, and randomly generated training test cases consisting of the prey in the center, and the predators placed randomly. The prey moves at 90% of the speed of the predators. They used a STGP system to allow functions and terminals of different types to be combined in one tree, which they argue reduces the search space. They demonstrated that the latter is superior to standard GP applied to this domain through comparison experiments. Their trees return the direction to move, which can be any of North, East, South, West or Here (random instances of which are generated as constants in the terminal set). Here indicates that the agent is to stay put. They provide a function called *CellOf* which takes two parameters, the first being an agent, and the second a compass direction (*tack* in their terminology), and return the cell that is positioned in the specified tack relative to the cell occupied by the specified agent. The terminal set includes a reference to the prey and current predator. Other functions that they use include *IfThenElse,* $<$ and *MD*, where MD returns the Manhattan distance between the two cells that are provided as arguments. The fitness function that they employ awards agents for getting close to the prey, with further bonuses for occupying capture positions, and capturing the prey. To ensure *stable captures* as opposed to *shadow captures* they continue the simulation even when the prey has been captured, unlike some of the other approaches mentioned above which stop at this point. They used 100 time steps in their training, however when comparing the evolved predators with the handcrafted greedy algorithms devised by Korf, they use up to 2000 time-steps. They also varied the type of prey, using randomly moving, move away from the nearest predator (MAFNP), and competitively co-evolved preys. They found that GP evolved solutions that are competitive with the hand coded greedy algorithms presented by Korf. Their results with a co-evolved prey however was less successful. They had anticipated that the performance of the predators and prey would oscillate as one populations learns to outwit the latest strategy used by the other. Instead they found that a simple strategy employed by the prey, in which the prey chooses a random direction and continues to move in that direction (referred to as the *Linear Prey*) successfully evaded all of the predator strategies that they considered. Further investigation showed that these strategies also performed badly against a still prey. Their analysis suggested that the act of the prey moving helps the predators to get out of deadlock situations created by their greedy approach. Later they evolved Heterogeneous predator agents [HSSW95a; HS96a; HS96b; HS97] using the same games rules (including the implicit push operator), function and terminal sets used in their homogeneous approach. The heterogeneous agents were evolved as a team by representing each agent as a separate tree in the same individual, as mentioned in the previous chapter. This resulted in very a successful strategy being evolved, and they note that this success is attributed to the ability of heterogeneous agents to avoid potential deadlocks.

Iba [Iba98] also used GP to evolve heterogeneous predators for the pursuit domain. In his game configuration, Iba used a orthogonal grid, and a prey that tries to avoid the predators by moving away from their centre of gravity. 15 training and 10 generalisation test cases were used, in which the agents were placed in randomly generated *Lieb* configurations. Each individual was evaluated in a simulator that runs for a maximum of 30 time steps, so that each agent could move a maximum of 30 times. Simulation was stopped when either the prey was captured or the the maximum simulation time expired. The agents could choose to stay in the current position or to move into one of the four orthogonal cells. The fitness was calculated by summing the total distances of each of the predators from the prey over each time step of the simulation. Lower fitness values therefore correspond to fitter individuals. The fitness function effectively rewards individuals that capture the prey in the shortest simulation time. It is unclear from the description whether the agents move in order or concurrently. The function and terminal set operate over vector types, and the evolved trees return a value of this type. A mapping functions maps the returned vector to one of the five movements mentioned earlier. The terminal set included primitives that return the displacement vector to the prey, and each of the other agents, and hence global information was available to the predators at all time. The function set included various vector operators, and some conditionals. Thirty runs were executed, each with a population of 500 individuals evolved over 100 generations. The best individual of this experiment resulted in a success rate (ranging 0 to 1) of close to 1. Iba repeated variations of this experiment to explore the use of communication. Curiously, in these experiments, he employed a randomly moving prey instead of the evading prey used earlier. The range of the primitives used to sense the location of the prey and the other agents were reduced so that they can only sense agents within a circle of radius 5 around their position. Three experiments were conducted, the first of which was identical to the earlier experiment except for the changes mentioned. The second experiment added what Iba calls communication primitives to the function set. As mentioned in the previous chapter, we view these primitives as conditional name-based sensors. The communication primitives called COM0, COM1, COM2 and COM3 request position information from predatory agents 1, 2, 3 and 4 respectively. The effect of this operator is to return the displacement vector from the calling agent to the target if the receiving agent is within some fixed range of the prey, otherwise the only argument to the function is returned. However, a penalty associated with using these operators is that the predator must wait one time unit before moving. In the third experiment in addition to the communication primitives he added a VIEW primitive which evaluates the one and only argument if and only if the prey is visible to the caller. The motivation for the latter was to enable a reduction in the reliance on communication if the prey is already visible. Iba reports that communication with the VIEW primitive gave the best results and reduced the amount of communication necessary.

4.2 Using the Pursuit Domain to Investigate Task Allocation and Conflict Resolution

In the pursuit domain conflicts arise when one or more agents try to occupy the same cell. If such conflicts are not resolved, then they can result in *deadlock*. Just as in concurrent and distributed systems, there are two ways of handling deadlocks in the pursuit domain; *deadlock avoidance*, and *deadlock detection and resolution*. Deadlock avoidance is the simplest and most effective approach that we can employ. Here the predators will need to coordinate their movement to eliminate or minimise conflicts. Deadlock detection and resolution is far more complex. Agents in conflict must first be able to detect the conflict and then coordinate their behaviour so as to resolve it.

We note that in much of the research above, conflict resolution has been avoided by providing environmental mechanisms for resolving conflicts or avoiding them. Examples include the ordered execution of moves used by Stephens [SM90] and Korf [Kor92], and the implicit push operator used by Haynes et al [HSSW95b; HS96b]. One of the important points that Haynes raised for the homogeneous agents that they evolved was that agents driven by deterministic algorithms can get out of conflict situations by the act of the prey moving. Hence, still preys were very effective at causing deadlocks. Furthermore given a large number of simulation cycles the predators are

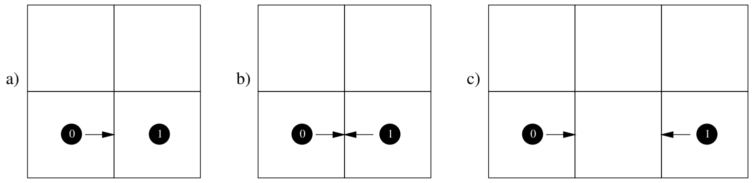


Figure 4.3: Collision conditions: a) agent 0 tries to move to a cell occupied by agent 1, b) agents 0 and 1 try to exchange positions, c) agents 0 and 1 try to move to the same cell.

more likely to get out of deadlock situations as a result of the prey moving randomly and hence later capture the prey. We note that in a separate non-GP study Haynes and Sandip [HLS96; HS96d] employed a case-based reasoning approach to learning conflict situations which are subsequently avoided by the predators. The main purpose of this chapter is to directly evolve conflict resolution strategies.

4.2.1 The Pursuit Rules

Our pursuit games were designed to maximise the probability of conflicts, and reduce inadvertent resolution of conflicts (those that occur due to the movement of the prey). The rules of our pursuit games were as follows:

Goal: To capture the prey by occupying each of the four orthogonal positions around the prey and maintaining these positions for the rest of the simulation.

- A simulation lasts for 50 cycles
- A new simulation is run for each test case
- For each simulation cycle each agent is allowed one move
- The moves can be any one of North, East, South, West or Here
- Here is used by an agent to forfeit its move and remain still
- Each of the agents move concurrently so that there is no ordering of moves
- No two agents can occupy the same cell
- If a collision occurs then the agents involved have their moves cancelled
- Collisions occur when:

  one or more agents try to move to a cell that is already occupied (figure 4.3a)

  two adjacent agents try to exchange cell positions (figure 4.3b)

  two or more agent try to move to the same cell (figure 4.3c)

- The randomly moving prey moves 90% of the time.

Our rules differ from those of Haynes and Sandip [HSSW95b; HS96b] in that we only allow 50 cycles for each test case instead of the 100 or more (upto 2000) reported. This reduces the probability of deadlocks being resolved by the movement of the prey. Furthermore our agents cannot exchange positions which would involving crossing each others paths. It is not clear from the papers by Haynes and Sandip whether or not this was possible. This restriction was introduced to maximise the likelihood of conflicts occurring. Our pursuit game *does not* feature any implicit or environmental mechanisms for resolving conflicts.

4.2.2 The Fitness Function

We used the same fitness function as reported by Haynes and Sandip [HSSW95b; HS96b]. After each cycle of a test we measure the Manhattan distance between each of the agents and the prey, the sum of which is added to the current fitness value. After each simulation we give further rewards for each agent that is still occupying a capture position, and add yet a further bonus if the prey is captured. This fitness function rewards movement close to the prey, with bonuses for occupying capture positions and a further bonus for capturing the prey.

Table 4.2: Fitness Evaluation Pseudo code for the Pursuit Problem

```
initialise the positions of the agents according to the test case
fitness = 0
for cycle = 0 to MaxCycles
 move the predators according to the evolved code
 if the prey is not stationary move the prey (only 90\% of the time)
 check to see if the moves are valid
 undo invalid moves
 commit valid moves
 foreach predator p, fitness += gridWidth / mahattenDistance(p, prey)
endfor
fitness += MaxCycles * gridWidth * NoOfCapturePositionsOccupied
if the prey is captured
  fitness += MaxCycles * gridWidth * 4
```

4.2.3 Fitness Evaluation

For training purposes, we used 30 randomly generated test cases, with the prey in the center and the predators in random positions. For each test case, we deployed both randomly moving and stationary preys. The latter again helps to maximise conflict situations as discussed earlier. We therefore had 60 test cases. The computational costs of evaluation made it infeasible to run every test case for each individual of each generation. There are four agents whose code need to be evaluated 50 times for each test case for each individual for each generation. For just one generation the computational costs are 4 * 50 * 60 * M, where M is the population size. Assuming a population size of 3000 as used in our experiments, this figure evaluates to 36 million evaluations. We could reduce the computational cost by reducing the population size, however this parameter is known to be one of the most important for the successful application of GP. We therefore used a sampling technique whereby for each generation, we select 10 test cases at random (without reselection) from the 60 test cases and use them to evaluate each individual. This allows us to fairly compare the fitness of individuals of the same generation whilst minimising the cost of evaluation. The cost of evaluating a generation is now reduced to 1/6th. However, a penalty with this technique is that we need to run the evolutionary process for a longer number of generations to give a good chance for each test case to be evaluated many times. Hence we doubled the number of generations that we would normally use, bringing a reduction in computation to 1/3rd of the original number.

A key problem with this approach is that as we move from one generation to the next, the best of the generation may outperform than the current best of the run, not because it is generally better, but because the sampled test cases were easier. Ideally we should weight the fitness value according to the difficulty of the test cases that were passed. Unfortunately this is not easy in the pursuit domain, we discuss this issue further in chapter 7. To overcome this problem we devised a technique whereby after each generation, we re-evaluate the best individual of the generation using the full 60 test cases and store that individual in the variable bestRunFull if the fitness value is better than the last value of bestRunFull. This mechanism provides an effective way to to track the best individual of the run.

After each run, we took the best individual of each run, and re-evaluated the individual using 2000 new test cases. These test cases were created using 1000 randomly generated initial positions. As before, two test cases were created for each initial position using stationary and randomly moving prey. For comparison purposes the same two thousand tests were used for each run. The results of this evaluation was used to measure and compare the generality of the evolved solutions.

4.2.4 The Function and Terminal sets

Table 4.3 lists the functions and terminals that were common to all the pursuit experiments that were performed. Additional primitives for experiment variations are described in the appropriate sections. The functions and terminals operate over three types, *Cell, Direction and Agent*. The Cell type represents cells on the playing field, the Direction type allows the evolved programs to return a suitable direction in which to move, and also allows cells orthogonal to a given cell to be accessed. Only four compass directions are permitted (*North, East, South and West*),

Table 4.3: Experiments 1-4 : Common Functions and Terminals

Primitive	Purpose
Direction IfNorth(*Cell c1*,*Cell c2*,*Direction d1*,*Direction d2*)	**if** *c1* is North of *c2* **return** *d1* **else return** *d2*
Direction IfEast(*Cell c1*,*Cell c2*,*Direction d1*,*Direction d2*)	**if** *c1* is East of *c2* **return** *d1* **else return** *d2*
Direction IfCellsEqual(*Cell c1*,*Cell c2*,*Direction d1*,*Direction d2*)	**if** *c1* is equal to *c2* **return** *d1* **else return** *d2*
Cell CellYofX(*Agent x*,*Direction y*)	**return** cell *y* of cell of *x*
Direction North	the *North* direction
Direction East	the *East* direction
Direction South	the *South* direction
Direction West	the *West* direction
Direction Here	no direction
Agent Me	the agent being evaluated
Agent Agent4	the prey
Direction Arg1, Arg2	arguments of current ADF

an additional direction called *Here* represents the null direction. The latter is used as a return value by evolved programs to indicate the agent is to stay in the current cell, and is also used in conjunction with the *CellYofX* function to access the cell of a given agent. The CellYofX function is the only function that operates on an agent and is equivalent to the CellOf function used by Haynes and Sandip. The functions *IfNorth* and *IfEast* allow the relative position of cells to be compared and depending on the result, allow one of two direction resulting expressions to be evaluated and returned. These functions are aware of the torroidal geometry of the playing field, and have been designed to simplify the task of sensing the relative direction of the prey. The terminal *Me* is instantiated at run-time to be the current agent that is being evaluated.

4.2.5 Behavioural Analysis

To enable behavioural analysis of our evolved code, at the end of each run, we re-evaluated the best Individual of the run using the full set of training test cases. During this evaluation, at the start of each test and after each subsequent simulation cycle, we wrote the state of the game board to a file. This file is effectively a *video* in which each game board state is a *frame*. A Java "video player" application was written which allows these videos to be viewed, as shown figure 4.4. The video player has controls that allow each test for each experiment to be selectively viewed and compared. A pause button together with frame advance and reversal allow each test to be carefully analysed. Each frame shows the board state, with agents represented as numbers. The pursuit agents are numbered 0, 1, 2, 3 and the prey is labelled 4. This numbering is useful for analysing who does what and when.

4.2.6 Comparison with Random Search

An important question to ask is whether for this problem GP performs any better than random search. The question is important in that it it tells us whether GP represents a process which helps us locate fit individuals within the search space of possible programs more efficiently than merely picking individuals randomly. As a further benefit it helps provide some insight about the search space in terms of how easy it is to find fit individuals.

For this purpose each experiment was repeated using random search. To allow fair comparison, the same number of individuals created by 10 runs of each GP experiment were generated randomly and evaluated. These individuals were created using a process similar to that used to create the initial population for GP, except that the average size of the individuals created was made comparable to the average size of the best end of run individuals generated by GP. This was achieved by generating individuals using the *grow* method and accepting only individuals that have a size that deviates no more than 30% of the desired average. The test case sampling approach used in the GP experiments cannot usefully be applied to random search and hence the individuals generated by random search were evaluated using the full 60 test cases. The best individual found by random search was re-evaluated using the same 2000 test cases used to measure generality of the GP solutions. The last row in the result tables for each experiment details the performance of the best of these individuals. The run name is replaced by the number of individuals that were generated and evaluated.

4.3 Experiment 1 - Basic Pursuit

Using our version of the pursuit game, and our primitives, we evolved both homogeneous and heterogeneous pursuit agents. To improve scalability we used ADFs, so that the the architecture for each agent consists of one result producing branch (RPB) and one ADF. The homogeneous agents all run the same code, hence the architecture requires only one set of RPB and ADF to be evolved. The heterogeneous agents require 4 sets. The detailed architecture for each approach is shown in table 4.4.

4.3.1 Results

The results of the experiments are detailed in tables 4.5 and 4.6, and shown as graphs in figures 4.5 and 4.6.

The evolved heterogeneous agents show a very high capture rates. The best individual of all the runs (best of run 6) successfully captures the prey in 58 out of the 60 training test cases. This individual also generalises very well, successfully capturing the prey in 1700 out of the 2000 generalisation test cases. Behavioural analysis of these agents show that they use strict task allocation to divide the capture task between them. Each agent assumes a different capture position around the prey. This allows them to easily avoid conflicts and results in a high capture rate. The generalisation of the solution is also very good for the same reason. The specific capture position assignment used by the best individual of run 6 is shown in figure 4.7. Different runs evolved different assignments.

As expected the homogeneous agents performed very badly, with the best individual of all runs succeeding at capture in only 9 out of the 60 test cases. Behavioural analysis of this individual revealed that without the environmental conflict resolution, the small number of permitted moves, and the still prey test cases, the agents frequently got themselves in conflict situations which they were unable to resolve, resulting in deadlock. Homogeneous agents unlike heterogeneous agents must allocate tasks dynamically. Without knowing which tasks have already been allocated it is improbable that they choose a task which does not conflict with that of another agent. One way around this problem would be to assign each agent a unique identity and to allow the agents to access their identity and compare their identities with the set of assignable identities. This should allow homogeneous agents to select tasks based on identity. We were also interested in seeing how tasks would be allocated for a mixture of homogeneous and heterogeneous agents. Both of these issues are investigated in subsequent experiments.

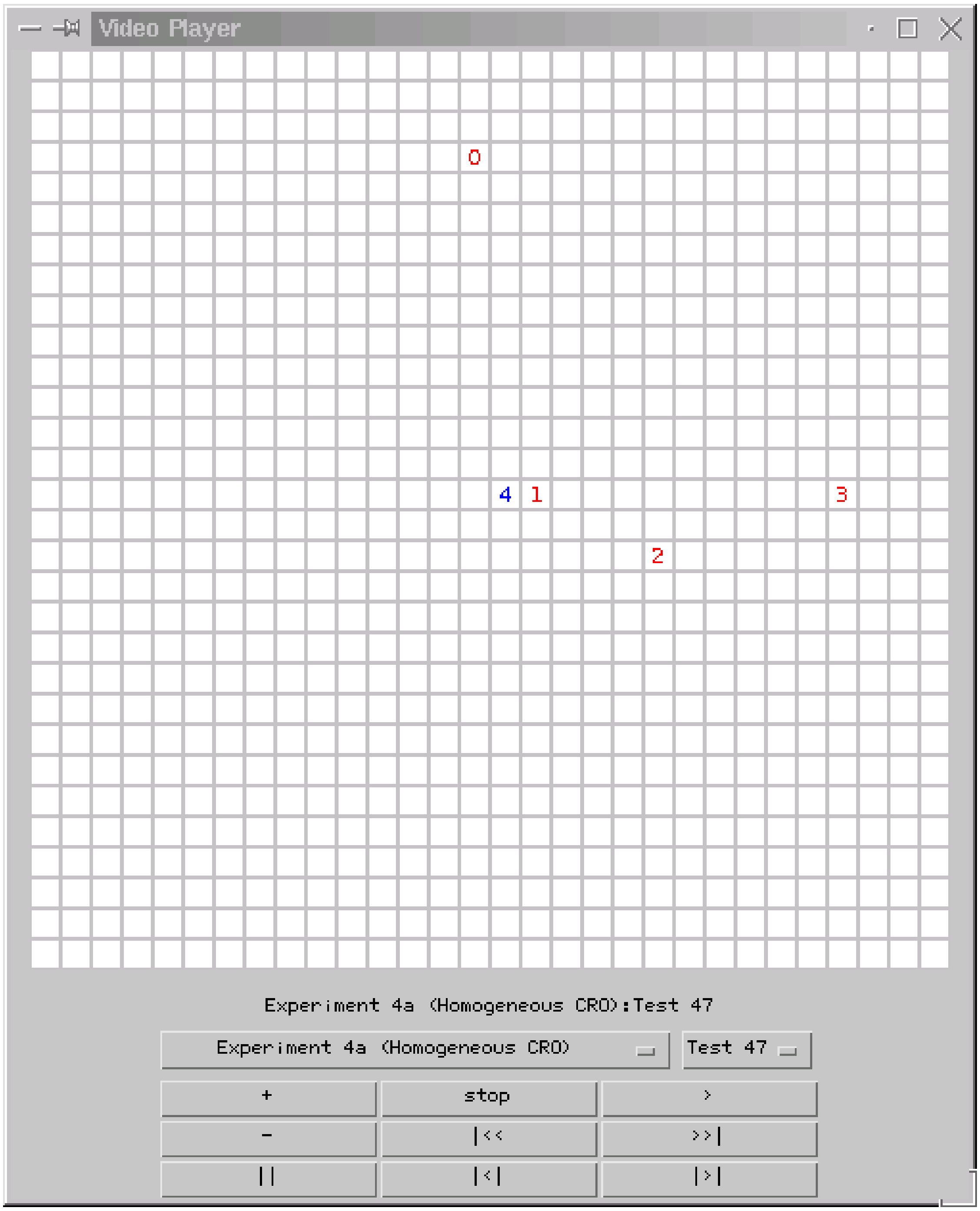


Figure 4.4: Video Player used to Trace Movement of Pursuit Agents

Table 4.4: Experiment 1 : Problem Definition

Objective:	Move the agents in directions that enable capture.	
Homogeneous Agents:	Functions	Terminals
Direction Adf0	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF1)	IfNorth, IfEast IfCellsEqual CellYofX, ADF0	Me, Agent4, North, South, East, West, Here
Heterogeneous Agents:	Functions	Terminals
Direction Adf0	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF1)	IfNorth, IfEast IfCellsEqual CellYofX, ADF0	Me, Agent4, North, South, East, West, Here
Direction Adf2	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF3)	IfNorth, IfEast IfCellsEqual CellYofX, ADF2	Me, Agent4, North, South, East, West, Here
Direction Adf4	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF5)	IfNorth, IfEast IfCellsEqual CellYofX, ADF4	Me, Agent4, North, South, East, West, Here
Direction Adf6	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF7)	IfNorth, IfEast IfCellsEqual CellYofX, ADF6	Me, Agent4, North, South, East, West, Here
Fitness test cases:	10 randomly selected per generation from 60 fixed test cases.	
Fitness:	Average of the raw fitness scores of the 10 test cases.	
Parameters:	Pop = 3000, G = 500, Runs = 10.	
Termination Condition:	Best solution captures prey for all 60 test cases.	

Table 4.5: Experiment 1a : Fitness of best evolved homogeneous agents

Homogeneous - Basic		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
0	460	6616.667	50.000	3	6148.785	50.000	38
4	280	7061.233	47.850	5	6649.372	48.386	122
2	202	7395.900	46.650	8	6742.492	48.535	140
6	262	7447.050	48.067	5	7000.521	48.111	149
7	288	7486.783	46.733	7	6922.912	48.186	141
5	216	7598.467	46.883	8	6748.692	48.313	125
1	284	7798.900	46.467	8	6717.876	48.953	86
8	428	7864.367	46.417	9	6895.240	48.185	148
3	174	7881.050	46.933	7	6864.084	48.132	139
9	312	7906.450	46.083	9	7046.939	48.117	145
Mean	290.600	7505.687	47.208	6.900	6773.691	48.492	123.300
1.5×10^7	362	6188.067	50.000	0	6096.149	49.677	31

Table 4.6: Experiment 1b : Fitness of best evolved Heterogeneous agents

Heterogeneous - Basic		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
3	582	9396.550	44.450	22	8191.741	45.564	500
2	670	10097.233	42.683	31	8736.327	44.376	750
4	574	11003.933	43.750	34	9575.257	43.896	843
0	430	12668.250	39.950	41	11081.221	42.028	1004
5	668	13215.467	38.517	45	12524.272	39.121	1361
9	652	13634.417	38.867	49	12062.410	41.320	1267
8	700	14253.783	35.933	50	12581.798	38.674	1366
7	582	14678.183	37.933	54	13547.649	37.638	1545
1	680	14704.650	34.167	52	13715.610	36.388	1528
6	724	15750.567	28.700	58	14586.280	32.969	1700
Mean	626.200	12940.303	38.495	43.600	11660.257	40.197	1186.400
1.5×10^7	702	3894.000	50.000	0	3581.799	50.000	0

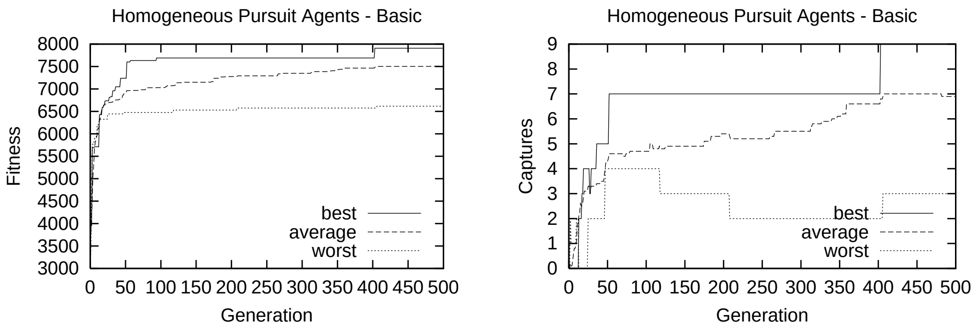


Figure 4.5: Experiment 1a : Performance Graphs of Evolved Homogeneous Agents

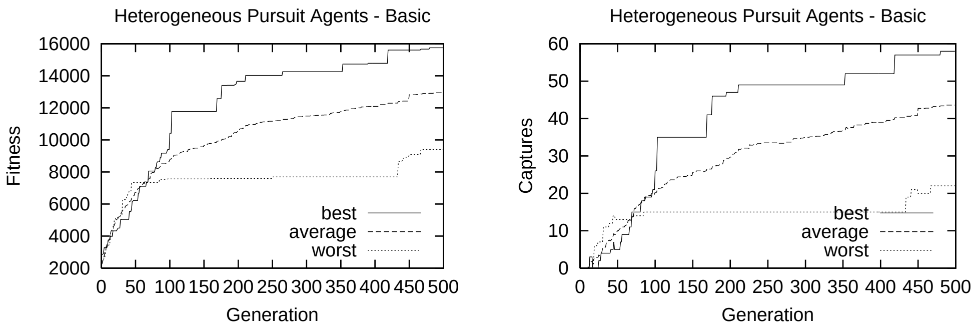


Figure 4.6: Experiment 1b : Performance Graphs of Evolved Heterogeneous Agents

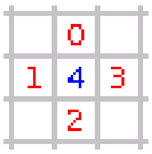


Figure 4.7: Experiment 1b: Capture Position Assignment

4.4 Experiment 2 - Identity

In the previous experiments we evolved heterogeneous agents that were very successful at capturing the prey and noted that their success was as a result of static allocation of tasks between the team members. In this experiment we assigned identities to each of the agents and provided operators that allow agents to enquire their identity and to compare the identity with the set of all identities allocated (see table 4.7). The purpose of this experiment was to see if we can evolve homogeneous agents that use their identity to allocate tasks between them. The experimental setup is described in table 4.8.

Table 4.7: Experiment 2 : Additional Functions and Terminals

Primitive	Purpose
Direction `IfAgentIdEquals`(*Agent agent*,*AgentId id2*,*Direction d1, d2*)	**if** *agent.id* == *id2* **return** *d1* **else return** *d2*
AgentId `id0`	the identity *id0*
AgentId `id1`	the identity *id1*
AgentId `id2`	the identity *id2*
AgentId `id3`	the identity *id3*

Table 4.8: Experiment 2 : Problem Definition

Objective:	Move the agents in directions that enable capture.	
Homogeneous Agents:	Functions	Terminals

Direction `Adf0`	`IfNorth, IfEast`
	`IfCellsEqual`
		`North, South`
		`East, West`
		`Here, arg0, arg1`

Direction `RPB (ADF1)`	`IfNorth, IfEast, ADF0`
	`IfCellsEqual, CellYofX`
	`IfAgentIdEquals`
		`Me, Agent4, North, South,`
		`East, West, Here`
		`id0, id1, id2, id3`

Fitness test cases:	10 randomly selected per generation from 60 fixed test cases.	
Fitness:	Average of the raw fitness scores of the 10 test cases.	
Parameters:	Pop = 3000, G = 500, Runs = 10.	
Termination Condition:	Best solution captures prey for all 60 test cases.	

4.4.1 Results

The results of the experiment are shown in table 4.9 and the graphs in figure 4.8. As can be seen, the evolved homogeneous agents were significantly more successful at capturing the prey

Table 4.9: Experiment 2 : Fitness of best evolved homogeneous agents (with identity)

Homogeneous - Identity		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
0	380	8306.683	45.867	12	7212.681	48.199	226
4	164	8518.450	44.950	12	7354.320	47.915	192
2	334	8676.600	44.233	14	7175.482	48.038	163
7	370	9273.217	44.000	15	7708.118	47.059	223
8	322	9664.500	44.217	16	9285.950	44.345	465
3	358	10261.467	42.200	18	8390.556	46.540	251
6	296	10757.017	40.450	23	9054.132	44.421	427
9	430	11961.617	37.983	39	9541.213	42.336	795
5	186	12462.533	35.667	34	11309.306	39.231	838
1	374	12723.683	38.133	38	10983.227	42.174	884
Mean	321.400	10260.577	41.770	22.100	8801.499	45.026	446.400
1.5×10^7	232	5983.233	50.000	0	5600.924	50.000	1

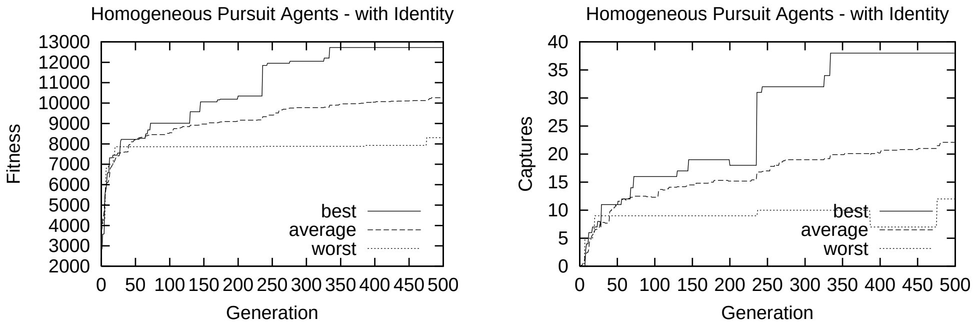


Figure 4.8: Experiment 2 : Performance Graphs of Evolved Homogeneous Agents (with identity)

(percentile comparison) than the homogeneous agents of experiment 1a. Behavioural analysis confirms the allocation of tasks between the agents based on their identities. In fact the best individual of all the runs (best of run 1), employed the same assignment of capture positions as the best evolved GP for the heterogeneous agents of experiment 1b (best of run 6). This is probably the result of a bias created by the initial placement of the agents in the training set. Just as with experiment 1a there were variations on the capture position assignments for the best GPs from the other runs. Whilst the evolved agents were much better than in experiment 1a, they still fall short of the heterogeneous agents of experiment 1b. A possible explanation is that the homogeneous agents have far fewer opportunities for crossover in the single set of ADFs than the heterogeneous agents where crossover occurs over multiple sets of ADFs, one for each agent. Furthermore the separate set of ADFs also allow better functional decomposition. We therefore believe that if we were to repeat the experiments using multiple ADFs and the ability to invoke any of these ADFs from the RPB based on the identity of the agent, we should be able to improve the results. An alternative explanation is that the problem is simply more difficult (the ratio $\frac{number\ of\ solutions}{number\ of\ possible\ programs}$ is much smaller).

4.5 Experiment 3 - Mixed Agents

The purpose of these next set of experiments was to investigate how tasks are allocated to an evolved mixture of homogeneous and heterogeneous agents. There are three possible experiments for our instantiation of the pursuit domain:

1. *experiment 3a(31)* one set of homogeneous agents and one agent that is heterogeneous with respect to them
2. *experiment 3b(22)* two sets of homogeneous agents, where agents in one set are heterogeneous with respect to the other.
3. *experiment 3c(211)* one set of homogeneous agents and two agents that are heterogeneous with respect to each other and the homogeneous set

In experiment 3a we evolved two sets of agent code, the first set is used by the predatory agents labelled 0 to 2, and the second by agent 3. This effectively makes agents 0 to 2 homogeneous and agent 3 heterogeneous with respect to them. A similar setup is used for experiment 3b, where two sets of agent code are also evolved, but this time the first set is used by agents 0 to 1, and the second set by agents 2 to 3. This effectively creates two sets of homogeneous agents, but with the two sets being heterogeneous with respect to each other. Experiment 3b required 3 sets of agents code to be developed, the first set is for homogeneous agents 0 and 1, and the second and third are for heterogeneous agents 2 and 3 respectively.

The problem definitions are similar to that of experiment 1 (see table 4.10, except that the architecture is modified appropriately. Experiments 3a(31) and 3b(22) have the same architecture, only two sets of agents code are evolved, so we need only Adf0, Adf1, Adf2 and Adf3. In experiment 3a(31) Adf0 and Adf1(RPB) are used by all three homogeneous agents, the agent that is heterogeneous with respect to them uses Adf2 and Adf3(RPB). In experiment 3b(22) Adf0 and Adf1(RPB) is used by the homogeneous agents of the first set, Adf2 and Adf3(RPB) are used by the second set of homogeneous agents. In experiment 3c(211) three sets of agent code is evolved, the architecture therefore requires 3 Adf pairs in total. The 2 homogeneous agents share Adf0 and Adf1(RPB), the remaining agents use Adf2, Adf3(RPB) and Adf4, Adf5(RPB) respectively.

4.5.1 Results

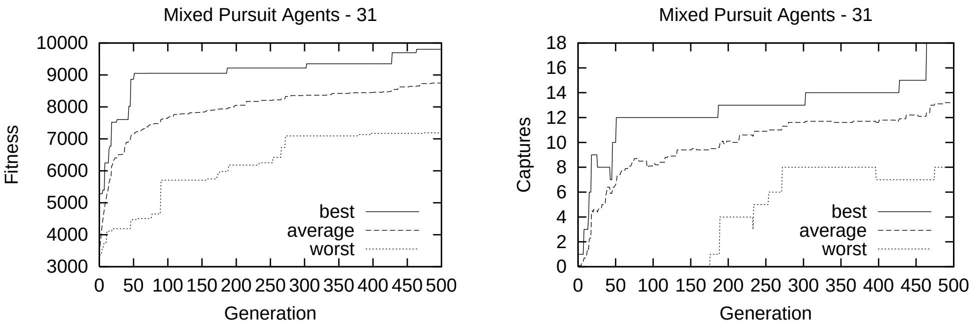


Figure 4.9: Experiment 3a(31) : Performance Graphs of Evolved Mixed Agents

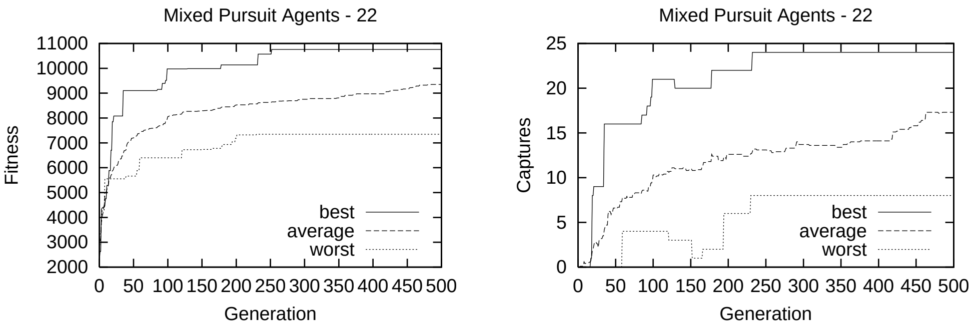


Figure 4.10: Experiment 3b(22) : Performance Graphs of Evolved Mixed Agents

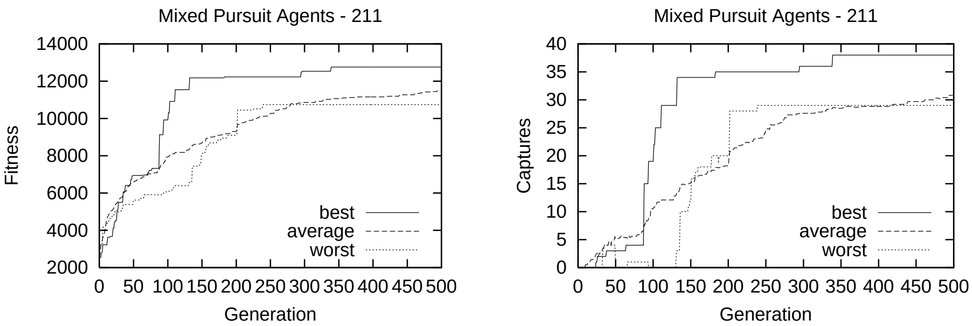


Figure 4.11: Experiment 3c(211) : Performance Graphs of Evolved Mixed Agents

The results of the experiment are listed in tables 4.11, 4.12, and 4.12, and shown by the graphs in figures 4.9, 4.10 and 4.10.

Behavioural analysis of the best solutions for all three experiments, shows that they made use of as much static task allocation as possible. In experiment 3a(31), 3 captures positions were allocated to homogeneous agents 0 to 2 which shared these positions, and the remaining capture position was allocated to agent 3. Most of the conflicts occurred between the 3 homogeneous

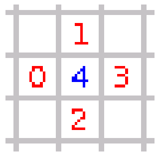


Figure 4.12: Experiment 3a(31): Capture Position Assignment

agents which need to dynamically choose from one of the three capture positions allocated to them. The best individual of all the runs for this experiment (best of run 3) assigned the capture positions North, West and South of the prey to the 3 homogeneous agents, and the capture position East of the prey to agent 3 as shown in Figure 4.12.

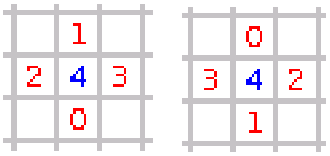


Figure 4.13: Experiment 3b(22) : Capture Position Assignment

The best solution for experiment 3b(22) allocated two sets of capture positions for each of the two sets of agents. Again conflicts were avoided between the two sets of agents, but there were conflicts within each set. The best individual of all the runs (best of run 9) assigned the capture positions North and South of the prey to homogeneous agents 0 and 1, and assigned capture positions East and West of the prey to homogeneous agents 2 and 3 as shown by figure 4.13.

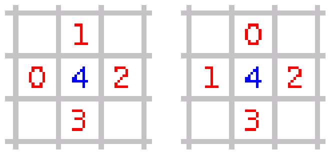


Figure 4.14: Experiment 3c(22) : Capture Position Assignment

The best solution for experiment 3c(211), was also the best result for all the experiments as we might expect. Two capture positions were allocated to the homogeneous agents, and of the remaining captures positions, one was allocated to agent 2, and the other to agent 3. Again conflicts were mainly between the homogeneous agents, but were minimal compared to the other experiments as there was more static task allocation. The best individual (best of run 8) assigned the capture positions North and West of the prey to homogeneous agents 0 and 1. The remaining capture positions East and South of the prey were assigned to agents 2 and 3 respectively as shown on figure 4.14.

These results suggest that GP tries to optimise the allocation of tasks to the agents.

Table 4.10: Experiment 3 : Problem Definition

Objective:	Move the agents in directions that enable capture.	
Mixed Agents(31 and 22):	Functions	Terminals
Direction Adf0	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF1)	IfNorth, IfEast IfCellsEqual CellYofX, ADF0	Me, Agent4, North, South, East, West, Here
Direction Adf2	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF3)	IfNorth, IfEast IfCellsEqual CellYofX, ADF2	Me, Agent4, North, South, East, West, Here
Mixed Agents(211):	Functions	Terminals
Direction Adf0	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF1)	IfNorth, IfEast IfCellsEqual CellYofX, ADF0	Me, Agent4, North, South, East, West, Here
Direction Adf2	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF3)	IfNorth, IfEast IfCellsEqual CellYofX, ADF2	Me, Agent4, North, South, East, West, Here
Direction Adf4	IfNorth, IfEast IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF5)	IfNorth, IfEast IfCellsEqual CellYofX, ADF4	Me, Agent4, North, South, East, West, Here
Fitness test cases:	10 randomly selected per generation from 60 fixed test cases.	
Fitness:	Average of the raw fitness scores of the 10 test cases.	
Parameters:	Pop = 3000, G = 500, Runs = 10.	
Termination Condition:	Best solution captures prey for all 60 test cases.	

Table 4.11: Experiment 3a(31) : Fitness of best evolved mixed agents

Mixed - 31		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
9	496	7187.717	47.617	8	6582.446	48.367	175
8	404	7794.700	47.200	8	6979.263	48.194	215
4	422	8206.400	45.333	12	6531.860	48.444	163
0	566	8717.350	45.467	11	8291.702	46.494	264
6	566	8861.500	44.700	13	7614.285	47.069	215
7	224	8943.217	45.417	15	7801.929	46.564	280
1	290	9108.100	43.850	13	7850.674	46.885	249
2	308	9416.250	43.000	16	8342.736	45.478	354
5	452	9429.150	44.300	18	7810.500	46.904	284
3	432	9802.317	42.550	18	8094.508	46.878	289
Mean	416.000	8746.670	44.943	13.200	7589.990	47.128	248.800
1.5×10^7	530	5308.800	49.483	1	5076.779	50.000	0

Table 4.12: Experiment 3b(22) : Fitness of best evolved mixed agents

Mixed - 22		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
4	362	7346.567	47.617	8	6371.912	48.766	118
3	374	7628.767	46.983	11	6740.513	47.939	217
1	390	9202.883	43.383	16	7806.854	47.132	227
5	392	9358.717	45.200	16	8053.090	46.926	315
0	286	9376.183	43.017	16	8443.133	45.120	363
8	454	9472.033	42.183	17	8422.005	45.156	381
2	596	9798.183	43.016	19	8485.865	45.280	397
7	444	10218.000	41.717	22	8279.560	45.466	381
6	190	10355.533	39.867	24	8601.506	44.723	412
9	304	10760.950	40.183	24	8438.601	45.214	388
Mean	379.200	9351.782	43.317	17.300	7964.304	46.172	319.900
1.5×10^7	276	4461.750	50.000	0	4865.211	50.000	3

Table 4.13: Experiment 3c(211) : Fitness of best evolved mixed agents

Mixed - 211		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
7	422	10739.567	40.700	29	8411.547	45.066	525
1	424	10763.500	44.967	29	8803.552	45.783	578
6	648	10799.917	43.100	28	10041.422	44.066	778
5	628	10961.967	40.667	27	9986.056	42.937	726
3	476	11034.333	41.500	26	9381.893	44.870	531
4	606	11577.667	41.817	32	10388.950	42.896	808
9	606	11741.733	40.300	32	10114.998	42.828	723
0	504	12105.700	38.500	34	10537.085	41.878	836
2	332	12291.117	38.083	33	10162.118	43.302	708
8	642	12757.817	36.917	38	10839.444	41.383	840
Mean	528.800	11477.332	40.655	30.800	9866.707	43.501	705.300
1.5×10^7	504	4188.067	50.000	0	3982.094	50.000	0

4.6 Experiment 4 - Conflict Resolution

The purpose of the next set of experiments (which is also the main focus of this chapter) is to see if we can evolve agents that make use of local information to avoid and/or resolve conflicts. The local information is provided by an operator that allows the agents to detect whether or not a cell neighbouring the cell that they or the prey occupy is also occupied (see table 4.14). We are particularly interested in how the addition of this operator will help homogeneous agents to resolve conflicts, as it is likely that heterogeneous agents will still use static task allocation to avoid conflicts. The experimental setup is detailed in table 4.15.

Table 4.14: Experiment 4 : Additional Terminals and Functions

Primitive	Purpose
Direction `IfCellOccupied`(*Cell cell*,*Direction d1*,*Direction d2*)	**if** *cell* is occupied **return** *d1* **else return** *d2*

4.6.1 Results

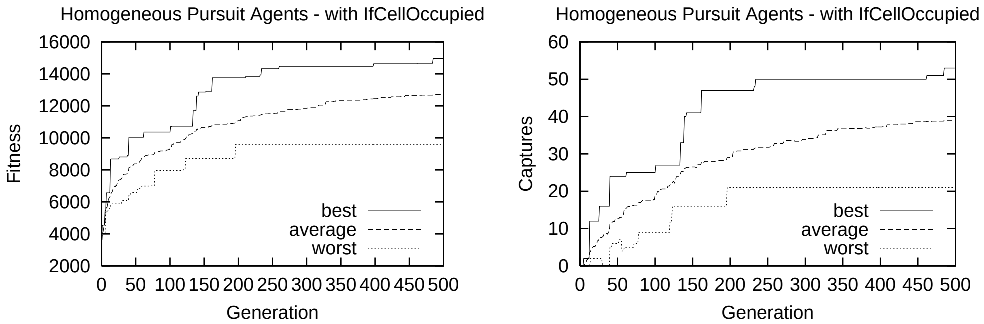


Figure 4.15: Experiment 4a : Performance Graphs of Evolved Homogeneous Agents (with CRO)

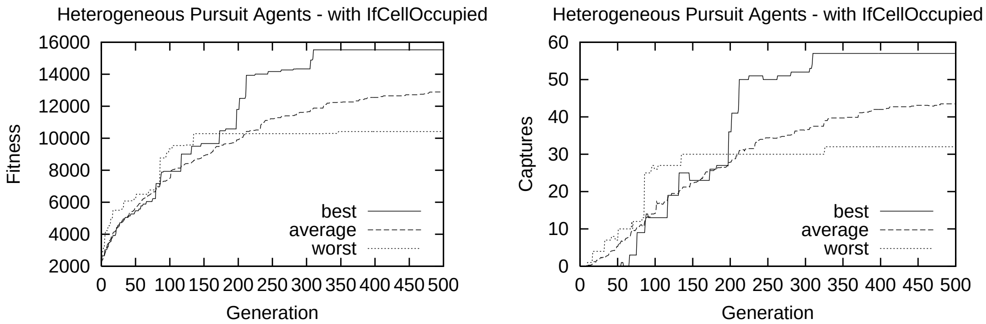
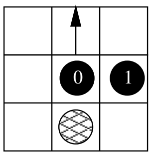


Figure 4.16: Experiment 4b : Performance Graphs of Evolved Heterogeneous Agents (with CRO)

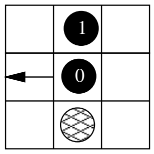
The results of the experiment are listed in tables 4.16 and 4.17 and shown graphically in figures 4.15 and 4.16. As expected there was little difference in performance for the heterogeneous agents, which continue to use static task allocation to avoid conflicts. The results for the homogeneous agents however showed a substantial improvement in capture rate when compared to the homogeneous agents of experiment 1. Behavioural analysis of the best individuals revealed that the mechanism used by the agents to improve their performance was far more sophisticated than expected. We had intended the *IfCellOccupied* operator to be used by the homogeneous agents to avoid movement towards a cell that is already occupied, thus resolving the conflict. This was evident in some test cases, however in many test cases, the agents used what could be viewed as either polite *social conventions* or *stigmergic communication* to resolve conflicts. Careful behavioural analysis of all 60 training test cases, revealed a total of 6 distinct behaviour patterns which when combined yield very sophisticated conflict resolution strategies. These 6 behaviour patterns are described below. In the descriptions N_{prey} is read the cell north of the prey, and $E(N_{prey})$ is read the cell that is east of the cell that is north of the prey (north-east of the prey). Agents named agent 0 and agent 1 were used for illustration, but since the agents are homogeneous the behaviour is not peculiar to these agents.

Behaviour EN



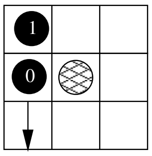
Agent 0 occupies cell N_{prey}, and agent 1 occupies the cell $E(N_{prey})$. Agent 0 responds by moving upwards freeing N_{prey}.

Behaviour NN



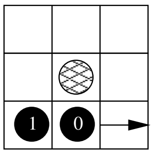
Agent 0 occupies N_{prey}, and agent 1 occupies $N(N_{prey})$. Agent 0 reacts by moving to position $W(N_{prey})$ freeing N_{prey}.

Behaviour NW



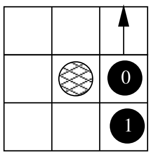
Agent 0 occupies W_{prey}, and agent 1 occupies $N(W_{prey})$. Agent 0 reacts by moving to position $S(W_{prey})$ freeing W_{prey}.

Behaviour WS



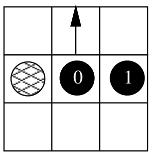
Agent 0 occupies S_{prey}, and agent 1 occupies $W(S_{prey})$. Agent 0 reacts by moving to position $S(E_{prey})$ freeing S_{prey}.

Behaviour SE



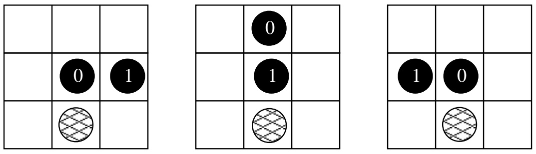
Agent 0 occupies S_{prey}, and agent 1 occupies $S(S_{prey})$. Agent 0 reacts by moving to position $E(N_{prey})$ freeing S_{prey}.

Behaviour EE



Agent 0 occupies E_{prey}, and agent 1 occupies $E(E_{prey})$. Agent 0 reacts by moving to position $E(N_{prey})$ freeing E_{prey}.

The above simple behaviours can be combined to create more complex patterns of behaviour. A good example is the pass through behaviour shown below.



The pass through behaviour starts with agent 0 occupying N_{prey}, and agent 1 occupying $E(N_{prey})$, and hence agent 0 invokes behaviour EN and moves up. Agent 1 simultaneously occupies the vacated position N_{prey}. Next agent 1 invokes behaviour NN, moving to cell $W(N_{prey})$, the freed cell N_{prey} is simultaneously re-occupied by agent 0. The effect is that agent 0 allowed agent 1 to pass through position N_{prey}. This example works because the two agents both play their parts, one agent invokes a conflict resolution behaviour, the other cooperates in the resolution.

Further examples of such behaviours are shown by frames captured from the video player application used for behavioural analysis. To maximise illustration, the examples are chosen from test cases in which the prey was still, although the same behaviours are used by the agents when capturing moving prey. To minimise space, the frames were cropped around the prey and frames were omitted when there was no change in the cropped region. In the examples shown by figures 4.17 and 4.18, the frames are to be read left to right, top to bottom.

In the example shown in figure 4.17, at the fifth frame, agent 3 is setup to follow behaviour EE and move north. This happens in the next frame, and agent 2 takes occupies the now vacant position. This in turn sets up agent 0 to invoke behaviour EN. In the next frame agent 0 therefore moves north, and agent 3 occupies the vacated position. Now agent 3 is setup for behaviour NN, and hence in the next frame moves west, and agent 0 re-occupies its previous position. In the same frame agent 1 moved to the position E_{prey}, and is therefore setup for behaviour NW. Agent 0 moves south in the next frame and agent 3 occupies E_{prey}. Finally the displaced agent 1 occupies the only vacant capture position by moving east. This example uses a total of 4 different behaviours EE, EN, NN and NW to resolve conflicts and capture the prey. The example in figure 4.18 similarly shows the use of behaviours NN, NW and WS. Note how the movement of the predators (driven by the best individual for this experiment) around the prey in both examples is anti-clockwise. The best individuals of some of the other runs used similar behaviours, but moved clockwise instead.

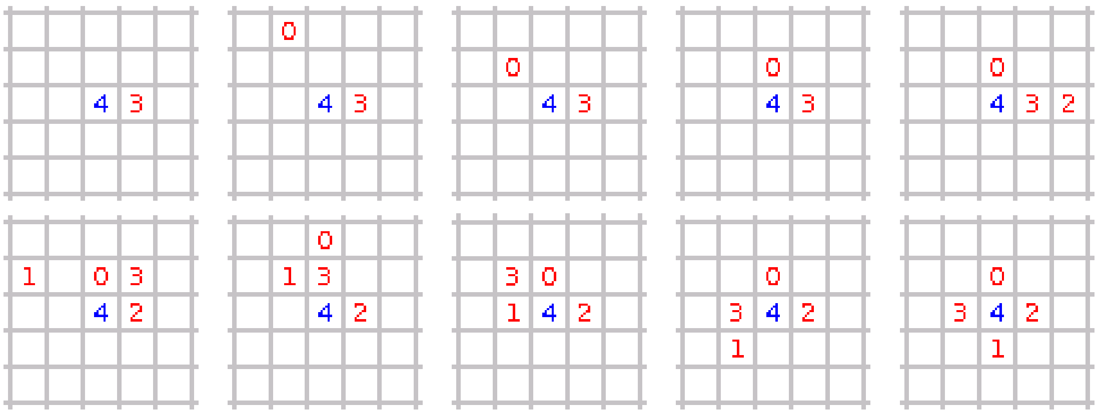


Figure 4.17: Experiment 4a(test 13) : End Game Trace of Agent Movement

4.7 Experiment 4s - Sampling Test Cases

To justify the test case sampling approach used in this thesis we conducted a comparison experiment. We implemented an experiment which was identical to experiment 4a (homogeneous pursuit with conflict resolution operators) except that the fitness evaluation used all 60 test cases for

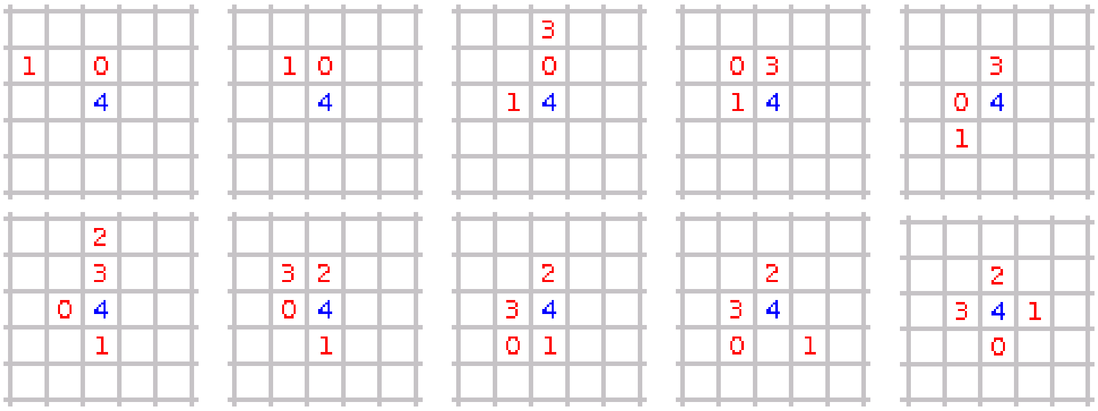


Figure 4.18: Experiment 4a(test 31) : End Game Trace of Agent Movement

each individual instead of samples of size 10. The significantly higher computational costs meant that we could only run the experiment for 250 generations instead of 500. The results of the experiment are shown in table 4.18 and the graphs of figure 4.19. The total number of evaluations of each individual using this approach is estimated at $250*3000*60*50*4 = 9*10^9$. For the original experiment, the total number of evaluations is estimated at $500*3000*10*50*4 = 3*10^9$. The new experiment therefore required 3 times as many evaluations as the original experiment. The results, for the new experiment when compared with the results of the original experiment do not justify the extra computational effort. Furthermore, the results from the original experiment show a greater generality.

It could be argued however that the new experiment was only run for 250 generations and therefore was disadvantaged in that the total number of individuals explored were fewer. To make the comparison fairer, we therefore present results from experiment 4a at generation 249 in table 4.19. Clearly these results show that even after just 250 generations the runs produced on average more general solutions. The best individual of the run was only slightly worse for the 60 test cases (passing 50 instead of 53) but was significantly more general. Furthermore, we effectively halved the number of evaluations and can therefore state that using our approach not only reduces the evaluation cost significantly, it also creates more general solutions. The problem with using a fixed of set test cases seems to be overfitting. These results should not surprise us, using the same fitness test cases for each generation allows successful individuals in early generations to swamp subsequent generations which in turn causes loss in genetic diversity and leads to premature convergence. By contrast in the sampling approach, those individuals successful in one generation may not be so successful in subsequent generations. We believe that sampling effectively creates niches in the population, where each niche contain individuals that are successful for a particular set of test cases. Like demes these niches help preserve population diversity and hence delay convergence. Furthermore as we move from one generation to the next, the individuals which propagate the most are going to be those that can score high fitness regardless of the test cases presented to them, meaning those that are more general. The graphs in figure 4.20 show how the fitness and capture count of the best of the generation varies over generations. The graphs show the averages for the runs, the best run and the worst.

We have one last point that needs testing; the re-evaluation of the best individual of a generation using the full training set of test cases before comparing with the current best of the run. As an alternative we could compare without a full evaluation. The results we would obtain using the latter for experiment 4a are shown in table 4.20. Clearly these results are significantly worse than the original results hence, justifying this approach.

The technique described by Gathercole and Ross [GR94] as Random Subset Selection is similar to the technique described above.

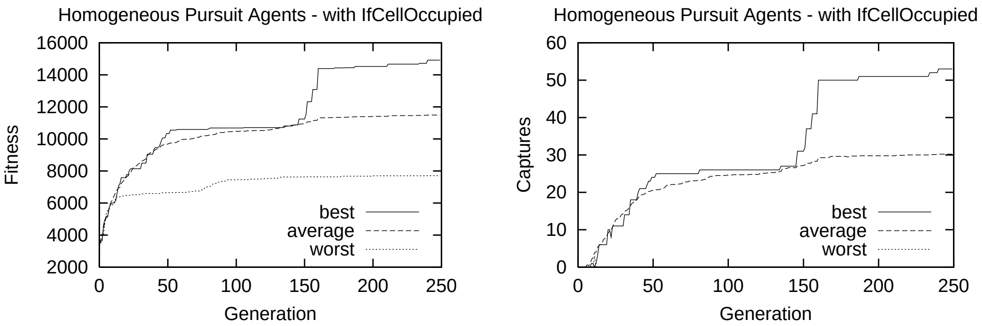


Figure 4.19: Experiment 4s : Performance Graphs of Evolved Homogeneous Agents (Full Evaluation)

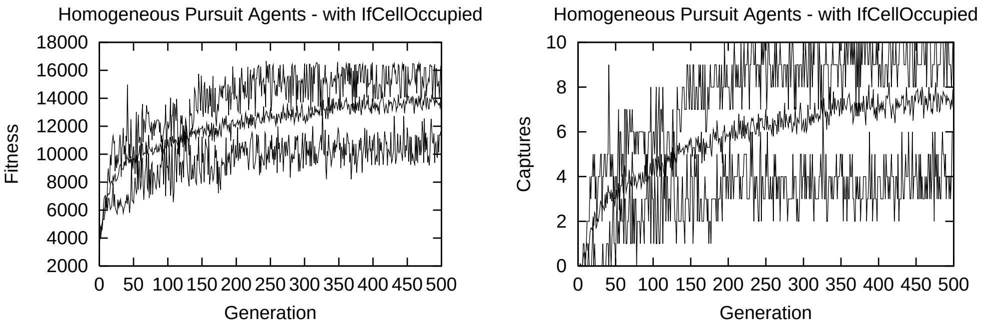


Figure 4.20: Experiment 4a : Performance Graphs of Evolved Homogeneous Agents (Fittest of Generation)

4.8 Comparison with Random Search

As can be seen from tables 4.5, 4.6, 4.9, 4.11, 4.12, 4.13, 4.16 and 4.17, in all of the experiments, the best individual created via random search performed significantly worse than GP. A discussion of the reasons why can be found in chapter 7.

4.9 Conclusion

We have shown that GP can be used to evolve agents that:

- allocate tasks amongst themselves both statically and dynamically
- use their identities for task allocation
- try to optimise the allocation of tasks
- detect and resolve conflicts

Table 4.15: Experiment 4 : Problem Definition

Objective:	Move the agents in directions that enable capture.	
Homogeneous Agents:	Functions	Terminals
Direction Adf0	IfNorth, IfEast, IfCellsEqual	North, South East, West Here, arg0, arg1
Direction RPB (ADF1)	IfNorth, IfEast, IfCellsEqual IfCellOccupied, CellYofX, ADF0	Me, Agent4, North South, East, West Here
Heterogeneous Agents:	Functions	Terminals
Direction Adf0	IfNorth, IfEast, IfCellsEqual IfCellOccupied	North, South East, West Here, arg0, arg1
Direction RPB (ADF1)	IfNorth, IfEast, IfCellsEqual IfCellOccupied, CellYofX, ADF0	Me, Agent4, North South, East, West Here
Direction Adf2	IfNorth, IfEast, IfCellsEqual IfCellOccupied	North, South East, West Here, arg0, arg1
Direction RPB (ADF3)	IfNorth, IfEast, IfCellsEqual IfCellOccupied, CellYofX, ADF2	Me, Agent4, North South, East, West East, West, Here
Direction Adf4	IfNorth, IfEast, IfCellsEqual IfCellOccupied	North, South East, West Here, arg0, arg1
Direction RPB (ADF5)	IfNorth, IfEast, IfCellsEqual IfCellOccupied, CellYofX, ADF4	Me, Agent4, North South, East, West Here
Direction Adf6	IfNorth, IfEast, IfCellsEqual IfCellOccupied	North, South East, West Here, arg0, arg1
Direction RPB (ADF7)	IfNorth, IfEast, IfCellsEqual IfCellOccupied, CellYofX, ADF6	Me, Agent4, North South, East, West Here
Fitness test cases:	10 randomly selected per generation from 60 fixed test cases.	
Fitness:	Average of the raw fitness scores of the 10 test cases.	
Parameters:	Pop = 3000, G = 500, Runs = 10.	
Termination Condition:	Best solution captures prey for all 60 test cases.	

Table 4.16: Experiment 4a : Fitness of best evolved homogeneous agents (with CRO)

Homogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
7	331	9603.850	43.033	21	9401.273	44.210	624
2	297	11745.433	39.017	34	10355.528	42.105	798
4	323	11940.217	38.050	33	11498.195	39.086	1013
0	316	12542.667	38.750	38	10536.450	41.184	880
3	204	12632.417	34.617	39	11901.102	37.699	1114
6	295	12639.467	35.783	38	10860.907	39.818	890
5	299	12940.100	35.600	40	11382.413	38.935	1029
8	235	13730.600	32.700	44	11692.839	37.294	1072
1	412	14344.967	32.217	50	13030.682	35.325	1381
9	489	14969.417	32.400	53	14596.335	32.008	1695
Mean	320.100	12708.913	36.217	39.000	11525.573	38.767	1049.600
1.5×10^7	263	5696.867	49.517	1	5639.412	49.968	2

Table 4.17: Experiment 4b : Fitness of best evolved Heterogeneous agents (with CRO)

Heterogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
4	615	10415.467	43.200	32	8000.069	46.290	607
3	706	10851.017	44.617	30	8355.619	47.090	509
8	556	12122.917	39.767	37	10137.040	42.784	892
5	744	12334.017	41.283	42	9802.103	43.930	933
9	757	12545.467	43.050	44	9377.983	45.388	772
0	532	13069.000	40.300	43	11934.951	41.045	1201
6	617	13235.800	37.500	47	11140.286	41.012	1154
1	846	14016.517	36.617	50	13270.028	37.664	1447
7	574	14850.400	33.800	53	13725.092	37.403	1573
2	730	15528.817	30.217	57	14763.829	32.877	1735
Mean	667.700	12896.942	39.035	43.500	11050.700	41.549	1082.300
1.5×10^7	593	3225.333	50.000	0	3242.517	50.000	0

Table 4.18: Experiment 4s :Fitness of best evolved Homogeneous agents (Full Evaluation)

Homogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
7	295	7704.550	50.000	0	6641.823	50.000	0
6	249	9781.967	42.733	21	8008.843	46.219	368
9	370	9977.433	43.550	23	7651.716	46.477	352
2	382	10138.950	43.283	22	7854.247	46.382	362
3	226	10595.300	41.817	22	9358.700	43.583	539
0	300	11248.617	40.883	29	10211.205	41.986	768
1	257	12107.433	39.750	35	9364.629	44.225	606
5	245	14179.183	38.067	50	11830.044	39.349	1235
8	266	14296.600	32.333	47	13166.297	34.562	1376
4	237	14918.783	32.117	53	12517.947	36.180	1275
Mean	282.700	11494.882	40.453	30.200	9660.545	42.896	688.100

Table 4.19: Experiment 4a : Fitness of best evolved homogeneous agents at generation 249

Homogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
4	258	8121.167	47.217	11	7343.922	48.300	205
7	331	9603.850	43.033	21	9401.273	44.210	624
2	229	10930.250	39.017	30	9478.439	42.790	697
0	281	11275.600	39.583	29	9255.731	44.354	541
5	228	11673.933	38.683	32	10188.894	41.212	768
3	186	11907.067	36.333	34	10473.023	40.379	812
1	338	12140.450	38.467	36	10992.288	39.824	953
6	227	12307.500	36.383	36	10805.262	40.014	878
8	238	12801.167	36.550	39	11967.796	37.739	1077
9	350	14320.900	32.383	50	13877.423	32.915	1565
Mean	266.600	11508.188	38.765	31.800	10378.405	41.174	812

Table 4.20: Experiment 4a : Fitness of best homogeneous agents of run (no re-evaluation)

Homogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
7	362	8641.983	46.050	14	8802.604	44.564	506
6	320	10070.983	40.500	24	10420.545	40.244	803
2	251	10354.500	38.800	28	9487.855	43.035	700
3	262	10766.650	38.217	29	10322.856	40.523	824
0	274	11037.233	39.683	29	10079.818	42.301	730
4	241	11309.000	40.150	29	10424.150	41.860	784
5	281	11807.717	38.533	36	10488.759	40.588	885
8	192	12887.350	34.967	38	12137.954	37.553	1097
1	386	12982.317	34.550	42	12899.421	34.894	1342
9	352	13122.000	35.483	43	13157.326	34.996	1398
Mean	292.100	11297.973	38.693	31.200	10822.129	40.056	906.900

Chapter 5

Communication

Sophisticated multiagent systems will require agents to be able to communicate not just with their environment, but with other agents to achieve their goals. Such communication may be achieved implicitly through changes in the environment or explicitly via direct message passing. In this chapter we show that GP can be used to evolve agents that communicate with each other explicitly.

5.1 Experiment 5 - Communicating Agents

The purpose of this experiment is to see if GP can be used to evolve agents that communicate to solve problems. For this purpose we have defined a simple problem for which it is necessary for communication to take place and furthermore, for which it is easy to measure the effectiveness of any communication. The problem involves two homogeneous agents A and B that steer two vehicles placed on a two dimensional grid. The vehicles are constantly moving in the direction that they are currently pointing. All movement is discrete, so that the vehicles always have integer (x,y) coordinates. Initially the vehicles are pointing in random compass directions. The steering mechanism allows the orientation of the vehicles to be changed to any one of the 8 compass directions. The goal is to evolve the code for agents A and B such that they steer the vehicles to enable them to meet. The vehicle model closely follows that used by Reynolds [Rey94a].

Agents A and B know only their own positions, but both have access to two bi-directional communication channels which can be used to send and receive messages to each other. The communication channels are "wired" so that one end is connected to one vehicle and the other to the other vehicle. A message sent from one end can only be received at the other end, and vice versa. The channels can only be used to send and receive messages in the form of integers. When an attempt is made to read from a channel the last message received is delivered (default value 0).

Clearly what we would like is for agents A and B to evolve behaviours that use the communication channels to exchange position information. The vehicles are given an equal amount of energy at the start of a simulation, and a single energy unit is consumed each time a vehicle moves one step in any direction. The game ends when both vehicles run out of energy or when they meet.

The fitness of the evolved code is measured using a function composed of two parts. The first part measures the degree of *success* and the second measures the *efficiency*. Whenever two fitness values are compared (for example during tournament selection or whilst updating the current best individual), the success measure is given priority so that the efficiency is considered only when the success values are identical. The success is measured as follows:

$$Success(A) = Success(B) = \frac{D_{before}(A,B)}{D_{before}(A,B) + D_{after}(A,B)}$$

Where D_{before} and D_{after} are the Euclidean distances between A and B before and after execution respectively. This function produces a range of values varying between $\left[\frac{D_{before}(A,B)}{D_{before}(A,B)+totalEnergy_{Allocated}(A,B)\times\sqrt{2}}, 1\right]$, where a value of 1 means that they have met. The efficiency is defined as the percentage of energy remaining at the end of a simulation:

$$Efficiency(A) = Efficiency(B) = \frac{totalEnergy_{Remaining}(A,B)}{totalEnergy_{Allocated}(A,B)}$$

Where $totalEnergy_{Allocated}$ is the total amount of energy allocated to the agents before the simulation, and $totalEnergy_{Remaining}$ is the total energy remaining after simulation.

We had started off by providing each vehicle with a constant surplus amount of energy for each test case. The result was that the code evolved made use of *social conventions* to obviate the need for communication. The agents would always evolve to meet at an “agreed” location (often location 0,0). We therefore rationed energy for each test case to a total value of 1.5 times the initial Euclidean distance between the agents. This provides sufficient energy to allow solutions that do not use diagonals to evolve, but at the same time inhibits solutions that merely use social conventions. A further advantage with this approach is that it ensures our fitness function takes into account the difficulty of the test case in the fitness evaluation. Test cases in which the agents are located at closer starting positions are not advantaged in terms of energy over test cases where the distances are greater. This concept of weighting test cases according to their difficulty is discussed in greater detail in Chapter 7.

5.2 Architecture

Tables 5.1 and 5.2 provide details of the architecture used to evolve programs to steer the agents. There is a single result producing branch (ADF0) which returns a compass direction used to steer the vehicle.

For fitness evaluation, we used the same sampling technique mentioned in the previous chapters.

Table 5.1: Experiment 5 : Actions Performed by Terminals and Functions

Primitive	Purpose
Direction `If`(*boolean test*,*Direction d1*,*Direction d2*)	**if** *test* **return** *d1* **else return** *d2*
boolean `GE`(*int v1*,*int v2*)	**if** $v1 >= v2$ **return** *true* **else return** *false*
boolean `LT`(*int v1*,*int v2*)	**if** $v1 < v2$ **return** *true* **else return** *false*
int `Add`(*int v1*,*int v2*) *int* `Sub`(*int v1*,*int v2*) *int* `SendA`(*int m*) *int* `SendB`(*int m*)	**return** $v1 + v2$ **return** $v1 - v2$ send message *m* via channel A; **return** *m* send message *m* via channel B; **return** *m*
int `GetX` *int* `GetY`	get the current *X* coordinate; **return** *X* get the current *Y* coordinate; **return** *Y*
int `RecvA`	get the message received at channel A; **if** no message received yet, **return** 0; **else return** last message received
int `RecvB`	get the message received at channel B; **if** no message received yet, **return** 0; **else return** last message received
Direction `North` *Direction* `NorthEast` *Direction* `East` *Direction* `SouthEast` *Direction* `South` *Direction* `SouthWest` *Direction* `West` *Direction* `NorthWest`	the *North* direction the *NorthEast* direction the *East* direction the *SouthEast* direction the *South* direction the *SouthWest* direction the *West* direction the *NorthWest* direction

One hundred test cases were randomly generated. Each test case consists of random initial positions and orientations for each agent located within the region (-50,-50) to (50,50). For each generation we select a sample of 5 unique test cases from the 100 previously generated ones and use these five to evaluate each individual in the population. The fitness value awarded is the average of the raw fitness for each of the five test cases. The best individual of the generation is then re-evaluated using the complete set of 100 test cases, and becomes the best of the run if it has the highest fitness recorded so far. At the end of a run, the best individual is further re-evaluated to test for the generality of the solution by using 1000 randomly generated test cases.

Table 5.2: Experiment 5 :

Primitives:	Functions	Terminals

`Adf0 (RPB)`	`Add, Sub, If, GE, LT`
	`SendA, SendB`
		`North, NorthEast, East`
		`SouthEast, South, SouthWest`
		`West, NorthWest`
		`RecvA, RecvB`

Test cases:	100 randomly generated fixed test cases.	
Energy:	$1.5 * Distance_{before}(A, B)$	
Fitness test cases:	5 randomly selected per generation from 100 fixed test cases.	
Fitness:	Average of the raw fitness scores of the 5 test cases.	
Parameters:	Pop = 1000, G = 100, grid range (-50,-50) to (50,50).	

To allow fair comparison, the same 1000 generality test cases are used for each run.

Each individual in the population was evaluated by using it to steer both A and B. This is achieved by alternately executing the GP once in the context of A, and then in the context of B. The result of each execution is used to steer the respective vehicle, which is then moved forward one unit. This process is repeated until either the two agents both run out of energy, or they meet. The pseudo code in table 5.3 illustrates this approach.

Table 5.3: Evaluation Pseudo Code

```
agent = A
while ((A.hasEnergy() || B.hasEnergy()) && (!met(A,B))) {
  if (agent.hasEnergy()) {
    Direction d = eval GP in context of agent
    agent.turnto(d)
    agent.move()
    agent.energy--
  }
  if (agent == A) agent = B else agent = A
}
```

The ordered execution of the agents has the important consequence of *delayed communication* between the agents. If the agents were to use communication to exchange position information, the sender will always be at a different position by the time the recipient receives the message. In our problem definition, for the agents to meet, they must be at exactly the same coordinates at the same time. The agents will therefore also need to compensate for the communication delays.

5.3 Comparison with Random Search

As in previous experiments and for the same reason, the same number of individuals that are created during 10 runs of the experiment were created randomly using the same technique described in chapter 4.

5.4 Results

The results over ten runs of the experiment are listed in table 5.4, and summarised by the graphs in figure 5.1. In four of the ten runs, code was evolved which successfully passed all 100 test cases. Each of these solutions used the two channels to exchange X and Y coordinates, and adapted their movement to achieve the goal. In addition, the evolved solutions found a means for overcoming delays in communication.

The best individual found by random search is also listed at the end of the table, and as can be seen performed significantly worse than the best evolved individual (with 32 meetings versus 100), and used significantly more energy (8% of energy saved versus 39%). In fact the best individual found by random search performed worse than the weakest evolved individual. A discussion as to why is provided in chapter 7.

Table 5.4: Experiment 5 : Results for Evolved Communicating Agents

Evolved Code		Training Tests (100)			Generality Tests (1000)		
Run	Size	Success	Efficiency	Meetings	Success	Efficiency	Meetings
0	84	0.92711	0.11856	47	0.92452	0.12282	475
5	69	0.93485	0.14233	61	0.92100	0.13778	546
3	51	0.93925	0.19065	62	0.94333	0.21326	634
9	122	0.99048	0.05868	66	0.99403	0.07720	672
7	79	0.99359	0.26075	86	0.99608	0.26222	899
1	62	0.99523	0.18096	91	0.99513	0.18649	828
6	65	1.00000	0.28359	100	0.99925	0.29343	994
2	131	1.00000	0.37188	100	0.99984	0.38698	998
8	88	1.00000	0.37268	100	0.99968	0.38796	998
4	80	1.00000	0.39126	100	0.99969	0.39508	999
Mean	83.1	0.97805	0.23713	81.3	0.97726	0.24632	804.3
10^6	85	0.84264	0.08245	32	0.84272	0.09478	323

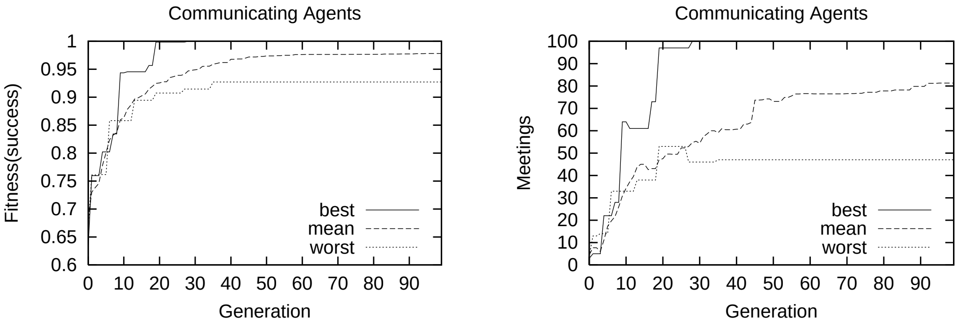


Figure 5.1: Experiment 5 : Performance Graphs of Evolved Communicating Agents

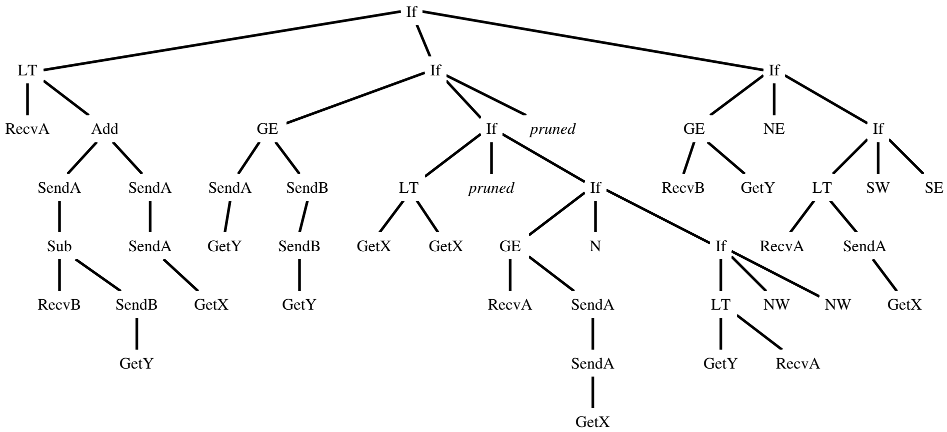


Figure 5.2: Experiment 5 : Evolved Parse Tree for best individual of Run 4

:

Table 5.5: Experiment 5 : Simplified code for Run 4

```
SendA X
SendB Y
If (RecvA - X < RecvB - Y)
    If (RecvA >= X) N else NW
else If (RecvB >= Y) NE
else If (RecvA < X) SW else SE
```

5.4.1 Run 4

Run 4 evolved the best solution for both the training tests and the generality tests. The GP tree representing this individual is shown in figure 5.2, the nodes labelled *pruned* represent branches in the GP which contain redundant code and are hence not shown. We can simplify the code as shown in table 5.5. Clearly Channels A and B are used for exchanging X and Y coordinates

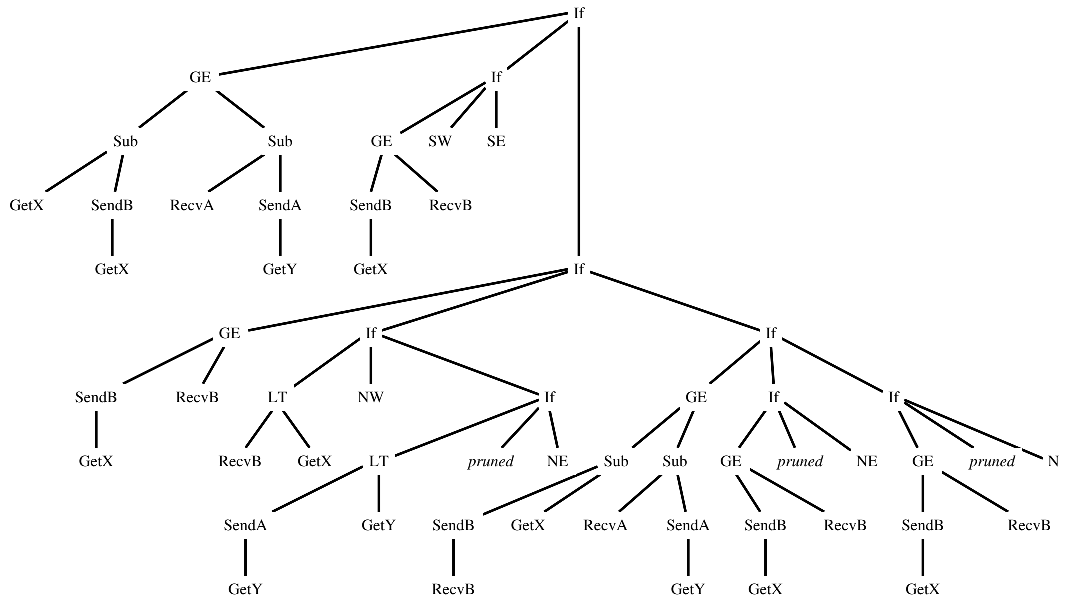


Figure 5.3: Experiment 5 : Evolved Parse Tree for best individual of Run 2

respectively. The information received via these channels is compared with local X and Y coordinates to determine movement. It should be noted that the code used mainly diagonal movement instead of orthogonal movement. This is the result of the restricted amount of energy allocated, and the efficiency pressure of the fitness function.

Sample traces of agents movement are shown in table 5.6. To illustrate the effects of evolution we have compared the best evolved individual for this run for test case 45 at generation 41 with generation 99 (the final generation). One of the key differences is that at generation 41 whilst the evolved code has discovered how to communicate, the code does not take into account the staleness of the information. The solution therefore often misses meetings by one pixel distances. The code for generation 99 not only minimises the energy used, it also uses strategies to overcome communication delay.

Note that the agents are only capable of moving in the directions N, NE, NW, SE, SW. The terminals representing the other directions have been lost during evolution. However despite the absence of the S direction, the agents have evolved to achieve movement in this direction by alternately moving SW and SE as shown by test case 21 in table 5.6.

5.4.2 Run 2

The GP tree representing the best evolved solution for Run 2 is shown in figure 5.3, and the simplified code that it represents is shown in table 5.7. As is shown, this time Channel B is used to communicate X coordinates, and Channel A for Y coordinates.

Table 5.8, show snapshots of the output trace for this individual for some sample test cases. The snapshots were taken at the point when the agents met or when they ran out of energy. Note how the solution involves diagonal movement at angles other than the 45 degree compass directions. This is made possible by alternately moving in different directions. For example by repeatedly moving in sequences of SE, SE, NE, we end up moving in the SE direction at far less steep angle than by just moving in the SE direction.

5.5 Conclusion

We have shown that GP can be used to evolve agents that know:

- what information is to be transmitted and received
- how to transmit/receive the information
- which channel to use to transmit/receive which piece of information
- how to adapt behaviour according to the communicated information to solve a problem
- how to deal with the staleness of data associated with communication delay

We hence conclude that it is possible to use GP to evolve agents that explicitly communicate with each other to solve a global problem. The key limitations of our work is that we have evolved only simple homogeneous agents which carry no state, working on a symmetric problem. An earlier version of this work ([Qur96]) was presented at the GP 96 conference and published in the proceedings. It is the first paper showing that GP can be used to evolve communicating agents.

Table 5.6: Experiment 5 : Trace of Agent movement for Run 4

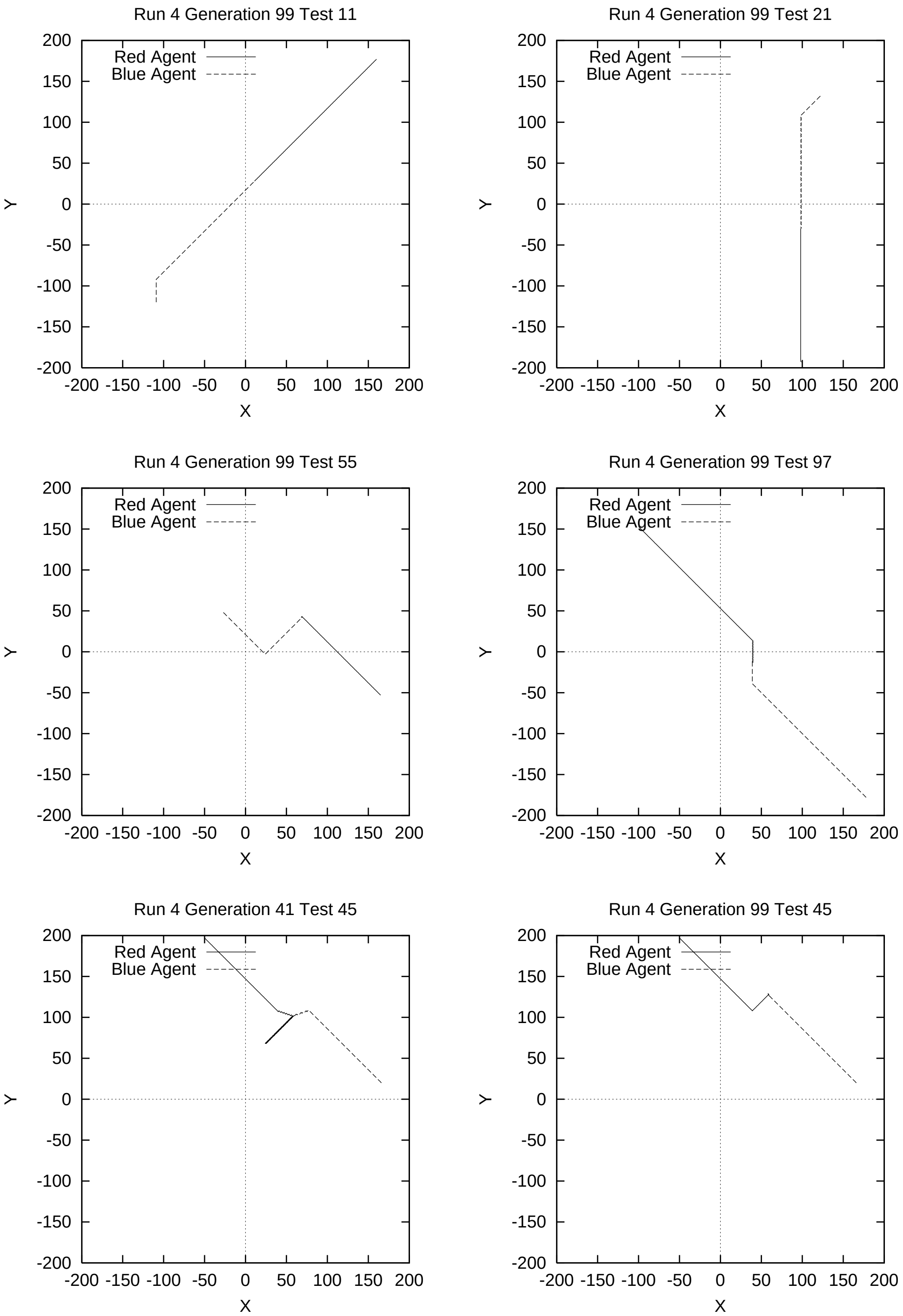
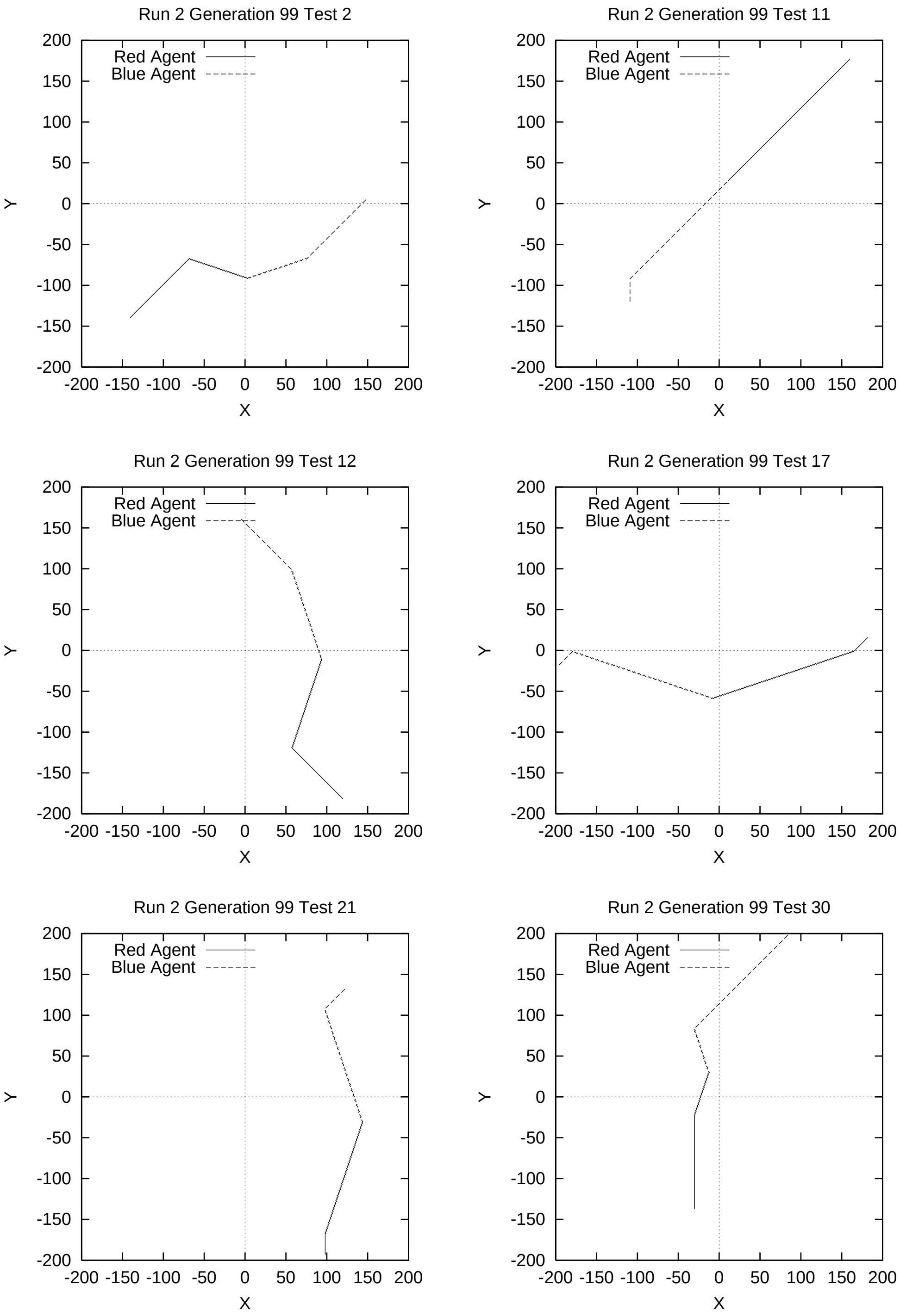


Table 5.7: Experiment 5 : Simplified code for Run 2

```
sendA Y
SendB X
if (Y >= RecvA)
  if (X >= RecvB) SW else SE
else if (X >= RecvB)
      if (RecvB < X) NW else NE
else if ((RecvB - X) >= (RecvA - Y)) NE else N
```

Table 5.8: Experiment 5 : Trace of Agent movement for Run 2



Chapter 6

Socialisation

We would like to be able to design agents that work within an existing society of other agents. The new agent must be capable of interacting both with the environment and with other agents to help achieve individual or group goals. In human programmed MAS, this ability is programmed into the new agents. The focus of this chapter is to show that GP can be used to evolve such agents automatically. In particular we show that the agents evolve to assume a purpose in the team, communicate with team members, resolve conflicts and interact with the environment to achieve goals. We name this process *socialisation*, as it is similar to the sociological term which describes how humans are trained to become integrated members of a society.

In previous chapters we have shown that GP can be used to evolve homogeneous and heterogeneous agents to solve problems. We evolved agents that are capable of decomposing a task, communicating with each other and resolving conflicts. To build an open MAS, we must also be able to evolve agents that work within a society of previously created agents. We therefore have devised three experiments in this chapter to determine if GP can achieve this task. All three build on experiments performed in the previous chapters.

6.1 Communicating Agents

In the first experiment we make use of the best evolved communicating agent of chapter 5 (the best agent from run 2) and instead of using two instances of the same agent, we evolve a new agent that replaces the counterpart. The new agent must evolve to know how to communicate with the other agent to allow them to meet. Unlike the original experiment the new agent cannot arbitrarily exchange information using the channels allocated, but must use the previously assigned purposes of the channels. Success in these experiments would suggest that GP can be used to create agents that know how to communicate with a society of existing agents to solve a problem.

The architecture for the experiment remains identical, just the fitness function is changed so

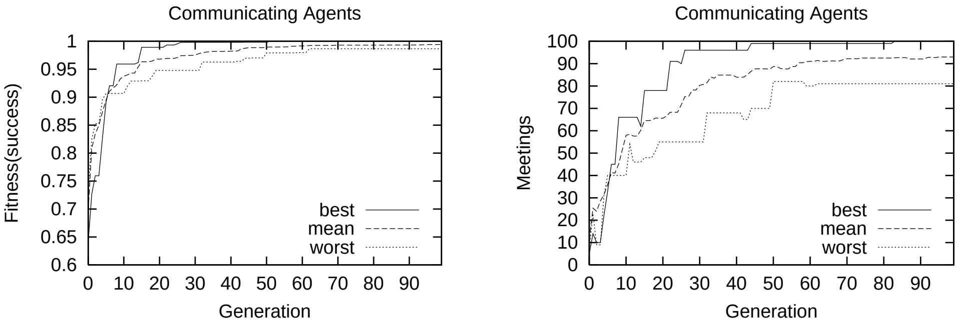


Figure 6.1: Experiment 6 : Performance Graphs of Socialised Communicating Agents

that the first agent is always instantiated using the best evolved agent from run 2 of the original experiment.

6.1.1 Results

Table 6.1: Experiment 6 : Results for Socialised Communicating Agent

Evolved Code		Training Tests (100)			Generality Tests (1000)		
Run	Size	Success	Efficiency	Meetings	Success	Efficiency	Meetings
7	71	0.986758	0.190388	81	0.979776	0.218998	839
5	131	0.986959	0.267658	90	0.987769	0.263201	893
0	220	0.987354	0.264737	84	0.992426	0.271754	861
3	122	0.991897	0.250398	93	0.990781	0.272061	933
2	115	0.995644	0.271641	92	0.998126	0.282659	921
9	80	0.996498	0.227828	97	0.998936	0.244127	991
8	133	0.997794	0.228757	95	0.998234	0.252121	954
6	129	0.998048	0.357939	98	0.999793	0.371336	998
4	94	1.000000	0.293016	100	0.999067	0.308986	993
1	115	1.000000	0.373473	100	0.999760	0.383389	999
Mean	121	0.994095	0.272584	93	0.994467	0.286863	938.2
10^6	112	0.91301	0.10489	34	0.94000	0.15826	501

The results over ten runs of the experiment are listed in table 6.1, and summarised by the graphs in figure 6.1. Two of the 10 runs generated complete solutions, whilst the remaining 8 were very close. Structural analysis of the two solutions shows that both have evolved to to correctly communicate with the previously evolved agent. Furthermore, the agents have also evolved to make use of the communicated information taking into account the effects of communication delay in order to achieve meetings. Note that this task is actually quite complex. The new agent needs to predict how information sent to the other agent is used by the other agent to alter its behaviour. At the same time the new agent needs to know how to make use of the information that is received to govern its own behaviour to ensure successful meetings. A simplified version of the best evolved solution (run 1) is shown in table 6.2 (the GP tree is too large to be displayed). As can be seen the solution correctly communicates X and Y coordinates using channels A and B respectively. There are however some differences in how the information is used, so that the two agents are no longer homogeneous.

As in previous experiments and using the same technique, the same number of individuals that are generated during the GP runs were sampled using random search. The fitness of the best individual found using this method is listed at the end of the table, and as can be seen performed significantly worse than the weakest evolved individual both in terms of number of meetings and efficiency.

Table 6.2: Experiment 6 : Simplified code for Run 2

```
sendA X
sendB Y
If (RecvB + RecvA >= X + Y)
  If (RecvB - Y >= RecvA - X)
      If (X < RecvA) NE
      else If (Y < RecvB) NW
      else E
  else If (Y < 2RecvB - X) E
  else If (Y >= X/2)
      If (Y < RecvB) NW
      else E
  else E
else If (RecvB < Y)
  If (RecvA >= X) S else SW
  else W
```

6.2 Pursuit Agents

The second and third experiments make use of the best evolved solutions of the homogeneous and heterogeneous conflict resolution experiments for the pursuit domain of chapter 4 (experiments 4a and 4b).

In the homogeneous agents case, we instantiate two predator agents using the previously evolved solution and evolve a new program which is instantiated twice to serve the purpose of the remaining two predators. The new agents must evolve to work within the homogeneous society of existing agents which have been previously evolved to dynamically allocate tasks amongst

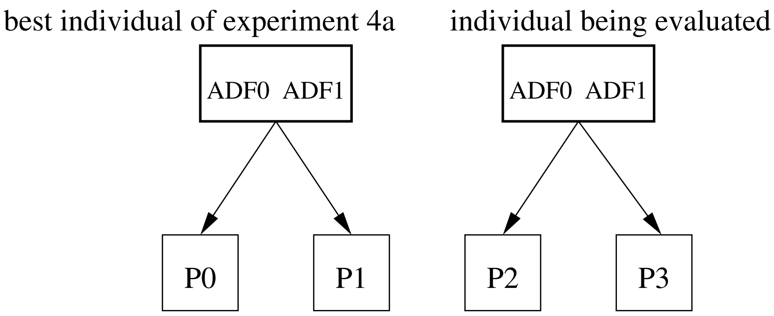


Figure 6.2: Experiment 7a Predator Instantiations

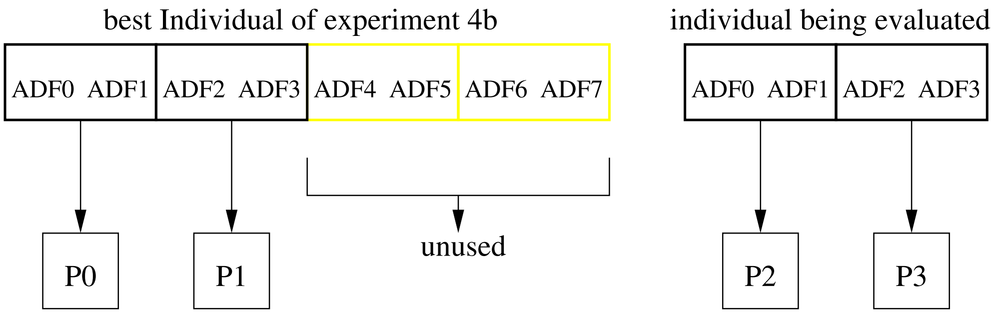


Figure 6.3: Experiment 7b Predator Instantiations

themselves and to resolve any resulting conflicts. We know from our analysis of the best evolved agent of the homogeneous conflict resolution experiment that the agents deal with conflicts by avoidance, and detection/resolution. Conflict resolution was achieved mainly by means of social rules or conventions. The new evolved code is instantiated twice so that it actually forms a homogeneous subgroup of agents. The agents must discover through evolution the social rules used to resolve conflicts and are also free to determine new rules amongst themselves to help resolve conflicts.

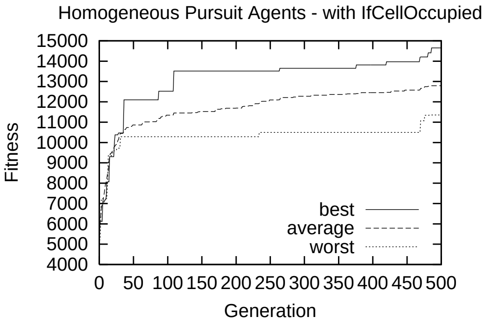
The purpose of the third experiment is to see if a two new heterogeneous agents can be evolved to work within a heterogeneous society of existing agents. We know that the best heterogeneous evolved solutions for the pursuit problem make use of static task allocation to minimise conflicts. We are therefore interested to see if we can evolve new agents that determine their tasks in the team without overlapping with tasks chosen by the previously evolved agents.

Once again the architecture for the two experiments is identical to that used in the conflict resolution experiments of Chapter 4, except that for the heterogeneous predators, instead of evolving trees for four different agents, we need only evolve trees for 2 different agents. The test cases were modified appropriately so that predators p0 and p1 are instantiated from the static previously evolved code, and predators p3 and p4 make use of code from the individual being evaluated (see figures 6.2 and 6.3). For the homogeneous society, predators p3 and p4 would be two instances of the same individual (consisting of Adf0 and Adf1 as shown in figure 6.2). For the heterogeneous agents, Adf0 and Adf1 are used by the predator p2, and Adf2 and Adf3 are used by predator p3 (see figure 6.3). Both experiments were executed with the same run parameters as before, including population size (3000) and number of generations (500). The test case sampling scheme used for fitness evaluation was also the same as that described in the previous experiments.

6.2.1 Results

Table 6.3: Experiment 7a : Fitness of best socialised homogeneous pursuit agents

Homogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
1	286	11374.367	40.317	29	10673.661	41.170	844
4	240	11673.383	39.583	32	10938.404	40.919	900
0	338	12020.867	38.817	36	10105.964	42.013	838
5	350	12478.683	39.400	38	10410.039	42.804	882
3	206	12671.467	39.383	42	10815.092	41.927	1027
9	202	12680.383	37.583	38	11170.342	40.720	955
2	249	13431.983	35.133	43	12845.127	36.790	1282
8	338	13471.733	34.250	44	12138.529	37.760	1164
6	331	13503.317	34.317	44	12308.231	37.719	1188
7	254	14653.083	33.217	52	12305.318	37.233	1237
Mean	279.400	12795.927	37.200	39.800	11371.071	39.906	1031.700
1.5×10^7	317	9688.633	42.967	23	9577.6295	43.775	567



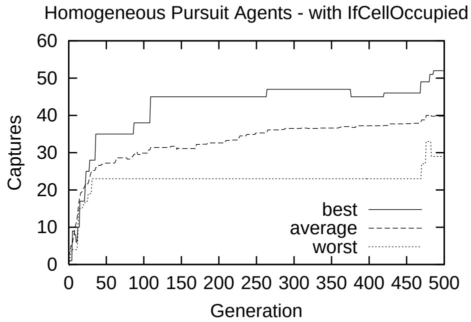
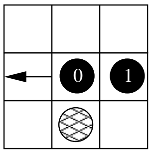
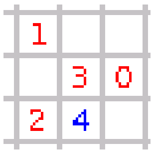


Figure 6.4: Experiment 7a : Performance Graphs of Socialised Homogeneous Pursuit Agents

The results of 10 runs of the homogeneous predator experiments are shown in table 6.3 and the graphs of figure 6.4. The evolved homogeneous subgroup were almost as successful as the original agents, scoring a maximum of 52 captures out of the 60 test cases. Behavioural analysis of the best individual revealed that it had evolved five out of the six behaviours described in experiment 4a (NN, NW, WS, SE and EE). The remaining behaviour (EN) was not discovered, however EN2 an alternative to this behaviour was evolved.



In behaviour EN2 shown above agent 0 instead of moving North as in behaviour EN, instead moves West. Whilst this behaviour continues to resolve the potential conflict, it is different from the behaviour "expected" by the previously evolved agents, furthermore it can change the way that this behaviour can be combined with some of the other behaviours. An example (test 31) is shown below:



In test 31 (in which the prey is still) agent3 is trying to invoke behaviour EN2 but in order to satisfy this behaviour it must move to the same cell that agent1 is trying to occupy (the position North-West of the prey). Hence both are deadlocked. If agent3 had used behaviour EN the deadlock could have been avoided. Behavioural analysis of the best agent suggests that the test cases used for training are less affected by this changed behaviour (in fact test 31 is the only example), however in the large number of tests used for measuring the generality, the effect becomes more pronounced (implying over-fitting). The evolved subgroup however did "know" how to *use* all the conflict resolution capabilities of the previously evolved agents.

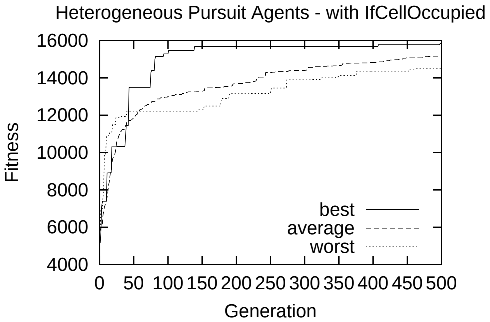
This result is promising, it suggest in principle that using GP we can evolve agents that "know" how to make use of social conventions programmed into an existing society of agents, and furthermore we can actually evolve new agents that behave according to the same social conventions. We believe by using a larger training set (with the same sample size) we could improve the performance of these agents.

Random search for this experiment was relatively successful when compared to experiment 4a. The best individual found by random search still has lower performance than the weakest evolved individual. However, the fitness and capture rate is much closer. A discussion of the possible reasons why is provided in chapter 7.

The results of 10 runs of the heterogeneous predator experiments are shown in table 6.4 and the graphs of figure 6.5. The evolved heterogeneous agents were very successful, nine runs of which achieved a capture rate in excess of 50 out of the 60 test cases. The generality was also very

Table 6.4: Experiment 7b : Fitness of best socialised Heterogeneous pursuit agents

Heterogeneous - CRO		Training Tests (60)			Generality Tests (2000)		
Run	Size	Fitness	Av. Cycles	Captures	Fitness	Av. Cycles	Captures
1	379	13877.667	38.300	49	12614.087	39.447	1380
2	520	14753.600	35.767	54	13804.485	36.684	1616
7	350	14988.450	33.683	54	13905.013	36.204	1600
6	296	15007.533	34.883	55	12746.029	39.148	1367
8	632	15032.117	35.650	55	12770.175	38.638	1389
4	377	15501.967	30.750	57	12910.354	37.478	1383
9	473	15533.000	30.867	57	13696.281	35.121	1545
0	456	15578.133	30.417	57	13788.314	35.917	1580
3	437	15683.733	28.050	57	14307.622	32.195	1595
5	435	15689.983	30.483	58	14322.683	35.189	1681
Mean	435.500	15164.618	32.885	55.300	13486.505	36.602	1513.600
1.5×10^7	350	8111.483	45.067	17	7175.066	46.878	352



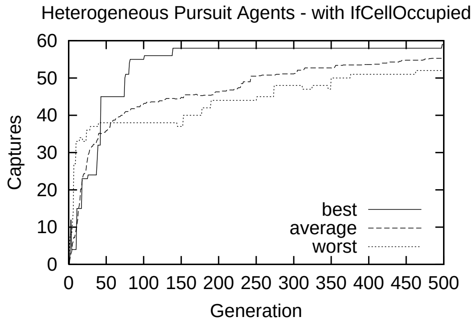


Figure 6.5: Experiment 7b : Performance Graphs of Socialised Heterogeneous Pursuit Agents

good, the best being run 5 which scored a capture rate of 1681 out of 2000. Behavioural analysis of the best evolved code confirmed that the new agents each uniquely selected one of the captures positions which were not allocated to the previously evolved agents. GP determined the roles of the new agents in a society of other agents. This was achieved by using test cases in which the new agents interact with the existing society. Figure 6.6 shows the capture position assignments used by the best individual.

The best individual found by random search was significantly worse than the weakest evolved individual. Its performance however was much better than the individual found by random search for experiment 4b. Possible explanations for the latter are discussed in chapter 7.

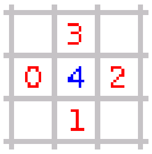


Figure 6.6: Experiment 7b : Capture Position Assignment

6.3 Conclusion

We have shown that GP can be used to create socialised agents. These are agents that have been evolved to work within an existing society of agents. We have also shown that the society can be homogeneous, or heterogeneous, communicating or non communicating. We have demonstrated that the new agents can be evolved to "know" their role in the society and can communicate with existing members both implicitly and explicitly. They can also detect and resolve conflicts between themselves and the existing members. The latter can also involve knowing how to behave in accordance to the the social conventions of the society and make use of them.

An important concept in software engineering is code reuse. Code reuse reduces the effort required in building large complex systems, whilst increasing the robustness of such systems. The ability to use GP to work with previously coded agents provides excellent code reuse and improves the scalability of GP.

Chapter 7

Discussion

In previous chapters we have shown how GP can be used to evolve multiagent systems that are capable of communication, conflict resolution and socialisation. We noted that GP produced superior results compared to random search even though the number of individuals that were sampled from the search space was equivalent. In this chapter we try to explain the poor results obtained using random search.

From our experience gained in evolving agents, we define a methodology for using GP to evolve multiagent systems. We then consider the technical problems that we would encounter in applying this technique to larger problems, and propose some possible solutions.

7.1 Comparing Random Search with GP

Despite limiting our random search experiments to search for programs of average size close to the average size of solutions produced by GP, in most of the experiments random search performed significantly worse. Random search does not exploit any information about individuals found so far to direct its search and hence relies purely on chance. The probability of finding a good program assuming we sample N programs is:

$$N \times \frac{number\ of\ good\, programs}{number\ of\ possible\, programs}$$

Whilst it is difficult to estimate the number of solutions, we can calculate a rough estimate of the size of the search space. Langdon ([Lan00]) defines an equation that can be used to calculate the number of possible trees for a GP of fixed size l and a function set consisting only of functions of arity 2:

$$trees(F,T,l) = |T|^{(l+1)/2}|F|^{(l-1)/2} \times \frac{(l-1)!}{((l+1)/2)!((l-1)/2)!}$$

The terms $|F|$ and $|T|$ represent the size of the function and terminal sets respectively. This equation only provides an upper bound on the number of possible trees if STGP is used as we have done in our experiments. Furthermore it makes no predictions on the number of trees where

ADFs are used, nor does it help in the case where a single GP represents multiple programs as is used to evolve heterogeneous agents. However, we can extend the equation to provide a rough estimate for the latter two cases.

First let us consider the case where we have a GP with a single ADF and a RPB. If we assume that the size of the RPB is fixed at l_{RPB} and the the size of the ADF is fixed at l_{ADF} then knowing $|F_{RPB}|$, $|T_{RPB}|$, $|F_{ADF}|$, $|T_{ADF}|$, we can use the above equation to calculate the number of possible trees for the RPB and ADF. These are trees $trees(F_{RPB}, T_{RPB}, l_{RPB})$ and $trees(F_{ADF}, T_{ADF}, l_{ADF})$ respectively. The total number of possible GPs for this case is given by:

$$programs_{RPB,ADF} = trees(F_{RPB}, T_{RPB}, l_{RPB}) \times trees(F_{ADF}, T_{ADF}, l_{ADF})$$

However for a GP of fixed size l, whilst $l = l_{RPB} + l_{ADF}$, there are many possible combinations of values for l_{RPB} and l_{ADF} that sum to l. Hence we need to sum $programs_{RPB,ADF}$ for each of the possible values of l_{RPB} and l_{ADF} that sum to l. This makes the calculation far more involved, especially where there are more than one ADFs, and hence for simplicity we assume that for a GP of size l, l is divided equally amongst the size of each of the ADFs and RPB. This will only give us a lower bound on the number of GPs. From the point of view of our calculation a RPB is indistinguishable from an ADF and hence we will treat a RPB as just another ADF.

The case for multiple programs within a single GP is similar to the ADF case. Each program within the GP would consist of a one or more ADFs. The fact that each program is used for a different purpose does not affect our counting and hence we can treat a GP with multiple programs (each consisting of one or more ADFs) as a single program with the same number of ADFs. Once again for simplicity we can assume that the size of the GP is divided equally amongst each of the ADFs. This allows us to calculate a lower bound on the number of possible programs. Note that in both cases, the real number of possible programs may still be lower if STGP is used.

Assuming closure, functions with an average arity of two and that the program size was limited to the average size of solutions evolved by GP, we used the above to calculate a rough estimate of the number of possible programs for each of the experiments that we conducted (table 7.1). These search spaces are extremely large, and the number of individuals that have been sampled from this search space for the random search experiments (10^6 for the communicating agents and 10^7 for the pursuit agents) comparatively extremely small.

Comparing the success of random search for each of these experiments it is clear that in general where the programs counts were very large the performance is very poor (table 7.2). In experiments 6, 7a and 7b although the search spaces are still very large compared to the sample size, random search is still relatively successful. We therefore conclude that there must be many fit programs in these search spaces. We discuss below the possible reasons why.

In experiment 6 we are trying to find a GP that can communicate with an existing agent and make use of the communicated information to steer itself towards the agent. The existing agent already transmits its own X and Y coordinates on specific channels, and can make use of X, Y coordinates received on the same channels. Although the most successful new GP would need to correctly transmit its own X and Y coordinates and make use of the received X and Y coordinates of the other agent to steer itself, there are other partial solutions that would still allow many meetings to take place. These include using the received information, but not transmitting or transmitting without using received information.

In experiment 7a we are trying to find a GP which drives two homogeneous agents which must work with other homogeneous agents driven by a previously evolved program. First we note that two of the agents are already evolved to occupy capture positions. This would automatically increase the fitness, but successful capture would still be dependent on the program driving the new agents to also occupy capture positions. To prevent deadlocks, the new program should ideally behave in accordance with the same social conventions as the previous program. However there are other strategies that provide partial solutions. An example is where the program found knows how to exploit the social conventions of the existing society, but does not act in accordance to them. Another example would be a greedy strategy in which the new agents occupy any unoccupied capture position and do not release that position once it is occupied.

In experiment 7b we are trying to find a GP consisting of two programs which drive a pair of heterogeneous agents which must work with two existing heterogeneous agents that have already evolved to choose specific capture positions. Once again, the fact that two of the agents are already driven by programs that cause them to occupy capture positions would mean that the fitness is always going to start off at a higher value. However to effect capture, the programs driving the new agents must each uniquely select one of the two remaining capture positions. This problem is simpler than the problem in experiment 4b. Furthermore even if a GP is found that drives only one of the agents to occupy a remaining capture positions, it is possible that the remaining capture position could be occupied by the other agent through chance alone.

Comparing the success of GP with random search for each of the experiments, it is clear that GP is superior. Despite the fact that GP sampled the same number of individuals as random search, and the fact that the random search was biased to search in the same part of the search space where GP found solutions, GP was still able to significantly outperform random search. We can therefore conclude that GP does not find fit programs by chance alone, and that it must represent a process that uses the information gained during the search to direct the search to explore better parts of the search space.

Table 7.1: Number of possible programs of size equal to the average size of evolved solutions for each experiment

Experiment	GP Size	ADF Size	ADF		RPB		ADF/RPB sets	Programs
			$\|F\|$	$\|T\|$	$\|F\|$	$\|T\|$		
1a	290	145	3	7	5	7	1	10^{309}
1b	626	78	3	7	5	7	4	10^{626}
2	321	160	3	7	6	11	1	10^{360}
3a	416	104	3	7	5	7	2	10^{429}
3b	379	94	3	7	5	7	2	10^{388}
3c	528	88	3	7	5	7	3	10^{536}
4a	320	160	3	7	6	7	1	10^{344}
4b	667	83	3	7	6	7	4	10^{686}
5	83	83	7	10			1	10^{119}
6	121	121	7	10			1	10^{165}
7a	279	139	3	7	6	7	1	10^{302}
7b	435	108	3	7	6	7	2	10^{453}

7.2 Automatically Programming Multiagent Systems Using GP

Based on our experience of evolving multiagent systems for each of the experiments, we can define a general methodology for automatically programming multiagent systems using GP. Our methodology is concerned more with implementation than analysis and design, although it could be used during that design phase for evaluating design options. It should hence be used in conjunction with a methodology for analysing and designing multiagent systems such as that defined by Wooldridge *et al* [WJK00].

Our methodology for automatically programming a MAS requires the following computing environment:

- A Strongly Typed Genetic Programming System Supporting ADFs such as GPsys.
- An environment that is identical or closely simulates the environment in which the agents are to work, and which can safely be used to test the agents in a controlled manner.
- A scalable computer architecture that can support the above.

There are then 8 preparatory steps required to code a GP MAS application. These involve defining the following:

1. The Environment

Table 7.2: Random search results for each experiment

		Training Tests			Generality Tests		
Exp	Programs	Fitness	Cycles or Efficiency	Captures or Meetings	Fitness	Cycles or Efficiency	Captures or Meetings
1a	10^{309}	6188.067	50.000	0	6096.149	49.677	31
1b	10^{626}	3894.000	50.000	0	3581.799	50.000	0
2	10^{360}	5983.233	50.000	0	5600.924	50.000	1
3a	10^{429}	5308.800	49.483	1	5076.779	50.000	0
3b	10^{388}	4461.750	50.000	0	4865.211	50.000	3
3c	10^{536}	4188.067	50.000	0	3982.094	50.000	0
4a	10^{344}	5696.867	49.517	1	5639.412	49.968	2
4b	10^{686}	3225.333	50.000	0	3242.517	50.000	0
5	10^{119}	0.84264	0.08245	32	0.84272	0.09478	323
6	10^{165}	0.91301	0.10489	34	0.94000	0.15826	501
7a	10^{302}	9688.633	42.967	23	9577.629	43.775	567
7b	10^{453}	8111.483	45.067	17	7175.066	46.878	352

2. The Agent Classes
3. The Architecture
4. The Function Sets
5. The Terminal Sets
6. The Fitness Function
7. The Run Parameters
8. The Termination Criteria

7.2.1 The Environment

We need to define the environment in which the agents will operate. The environment can be isolated in that there is no predefined agent society operating within this environment, or populated with other agents.

7.2.2 The Agent Classes

We need to choose whether our agents are to be:

- Cooperative or Competitive

- Homogeneous, Heterogeneous or Mixed Agents
- Reactive or Deliberative (with memory)
- Communicative or non-Communicative

The first choice is to decide whether the agents that we wish to generate are to cooperate or compete. Competitive agents will generally require separate populations to be co-evolved, even when they must sometimes cooperate with each other such as trading agents. Cooperative agents can be evolved as heterogeneous teams or as homogeneous agents. To create purely reactive agents we can omit memory operators, otherwise we must provide a means of storing state information. Communicative agents will require operators that allow communication to take place, either explicitly like the agents of chapter 5 or implicitly like the agents in chapter 4 experiment 4a. It is helpful to know how many agents we require, although it may be possible to discover this number automatically through evolution [Ben96a; Ben96b].

7.2.3 The Architecture

For each agent class that needs to be evolved, we need to define a set of ADFs including the result producing branch. Alternatively we could parameterise the GP system to evolve this architecture using the architecture altering operations reported by Koza [KDBK99].

7.2.4 Function and Terminal Sets

For each class of agent that we need to evolve, we need to define the function and terminal sets. The latter should contain functions and terminals that can be used to compute decisions, communicate with other agents and sense/manipulate the environment.

7.2.5 The Fitness Function

The key to evolving a MAS is to correctly define the fitness function. The fitness function should measure how closely the agents achieve their intended purpose, awarding higher fitness to those individuals that are more successful. We can also add further criteria to the fitness function such as a measure of how efficiently the agents achieve their purpose. We can penalise unwanted behaviour by defining laws which have cost repercussions if they are broken, much like penalties are used in human legal systems. Fitness is evaluated by testing the evolved code against test cases representative of the real problems that the agents must solve. This evaluation involves instantiating the desired number of agents of each class from the evolved code, and iteratively executing them for the given test case until the problem is solved or the maximum number of iterations have expired.

7.2.6 The Termination Criteria and Run Parameters

These are the same as described for GP in chapter 2.

7.3 Technical Issues of Scalability

Even for small sized problems, the memory and processing requirements of GP can be quite significant. Consider for example the pursuit problem, with homogeneous predators. Let us assume a population size of 10,000 individuals, using a GP system that uses 4 32-bit integers for each node in the program tree.

If we assume that at worst each individual consists of 1000 nodes, then our estimate for space requirements would be more than $10,000 * 1000 * 4 * 4 = 160Mbytes$. If we assume the use of a generational engine, then depending on the implementation we can assume memory requirements that are up to double the calculated value.

If we then assume that we execute ten runs of this problem, where each run evolves the population for 500 generations, and that evaluation of each generation involves running the fitness function for each individual in the population, then the total number of fitness evaluations is $10 * 500 * 10,000 = 5 * 10^7$. For the pursuit problem, if the fitness evaluation involves testing the program for over a total of 60 test cases, where for each test case, the program is executed for 4 times (for each predator), for each of the 50 cycles, then the total number of evaluations of the program is $5 * 10^7 * 60 * 50 * 4 = 6 * 10^{11}$ evaluations.

Clearly this brings into question the scalability of GP for solving medium to large problems. This issue is of particular importance when we use GP for evolving complex multiagent systems. The key factors are the space required to store the population, and the time taken to execute the fitness function. The impact of both of these factor can be reduced by improving the performance of the GP system so that it requires a smaller population, running for fewer generations to solve problems.

7.4 Solving Space Problems

We can reduce the memory requirements of the genetic programming algorithm by several ways. These can be categorised into three main groups, improving the implementation of the GP system, improving its performance and tuning the problem definition to optimise performance.

7.4.1 Implementation

We can implement the GP algorithm in ways that reduce the space requirements. The most efficient way of representing a population in a GP system is to use a directed acyclic graph (DAG). Handley [Han94a] used this method to represent the entire population of syntax trees as a single directed acyclic graph (DAG). One advantage with this approach is that eliminates duplication of identical subtrees. Handley reported a 15-28 fold reduction in the number of nodes per generation when using this representation.

An alternative is to use a linear representation, where each tree is represented in polish notation as a byte array [KDBK99]. This is possible since the total number of terminals and functions is unlikely to exceed 256, even when including ephemeral constants. This approach significantly reduces the memory requirements. A program of 1000 nodes can now be represented using only 1KB instead of 16KB as in the above calculation.

There are two widely used GP engines, the Steady State Engine and the Generational Engine. The Generational engine as defined by Koza generates an entire new population from the old, replacing it. This would suggest that the implementation would require space sufficient to hold twice the population of individuals to work. However, Koza [KDBK99] describes a memory efficient crossover algorithm that requires space little more than populations size. This approach has been implemented in a number of GP systems including GPsys. The steady state engine works by placing newly created individuals into the population as they are created, and hence only require a single population to be in memory.

Each of the above techniques will still cause problems as the population size and/or the program size scales up. As the memory requirements increase, the operating system will use more and more virtual memory, causing the process to start paging. A possible solution is to use a database to store the population, a proposal based on this idea is presented later in this chapter.

7.4.2 Tuning the problem definition

We could choose GP parameters that lower the space requirements. The most obvious ones are to restrict the size of the population and the size of individuals in the population.

Restricting the size of individuals in the population is usually a requirement of nearly all GP systems. However, there are two problems with this approach. The first is that it we cannot predict in advance the size of solution programs. Whilst algorithmic complexity theory suggest a lower bound on the size of such a program, the actual size of the evolved program cannot known. A possible solution is the application of *parsimony* pressure in the fitness function. The second problem is that to make GP more scalable we will need to be able to evolve bigger programs. However, the use of ADFs and automatically evolved data structures may help.

The single most important factor for a successful GP application reported by GP practitioners is the population size. Therefore reduction of the population size is unlikely to help make GP more scalable. However, this possibility has been investigated by Gathercole and Ross [GR97a] who report that small populations evolved for a large number of generations can be better than large populations over a few generations. It is noted however that they use a high mutation rate (60%).

Considerable work has been done to demonstrate that Automatically Defined Functions can be used to evolve code re-use [Koz94b]. Code reuse can be used to reduce the size of an evolved solution as well as to reduce the computational effort required to find it, and should consequently be used wherever possible.

7.4.3 Performance

We can optimise the GP algorithm to minimise the required population size. This would involve making better use of a smaller population, the main obstacles of which are premature convergence and bloat.

When a GP run has prematurely converged, the programs start to increase in size until they reach the maximum allowed sized. The code that is added to the solutions is referred to as bloat or introns and has no effect on the fitness of the evolved program. The cause of bloat has been studied by Langdon and others [LP97a; LP97b; Lan97a; LP97c; Lan97b; Ang98]. Suggestions have been made as to how we can reduce bloat and hence the size of evolved solutions. However, this is likely to make little difference as the appearance of bloat marks premature convergence.

Premature convergence leads to the population converging on a less than optimal solution early in the run. One of the explanations for premature convergence is the lack of genetic diversity in the population. This happens when the proliferation of fit individuals earlier in the run, obviate the need for primitives that are needed later. Small populations are more prone to this problem, as convergence happens more rapidly. If however we detect loss of diversity and employ some technique to re-inject it into the population, we may be able to reduce the effective population size. Ways of maintaining population diversity include changing the fitness test cases, so as to prevent individuals that work on easiest of test cases from dominating the population. This approach was used in the experiments in this thesis. Other alternatives include using competitive co-evolution to co-evolve the difficulty of the test cases together with the programs used to solve them. This creates a continuous arms race and prevents simple solutions from dominating the population. This technique was used recently to evolve a checkers program that can win even expert checkers players without ever having played an expert [CF99].

The use of demes present a simple yet effective means of combating premature convergence and is discussed further later in this chapter.

Another common method used to defend against premature convergence is to use mutation to help re-create diversity and re-introduce lost genetic material. Koza initially avoided the use of mutation in GP and relied entirely on crossover, but has since changed his policy.

7.5 Solving Processing Problems

There are two ways that we can reduce the computational requirements for a GP run. We can reduce the cost of evaluation and/or we can reduce the number of evaluations that are required. The former relies on improving the efficiency of the fitness functions, and the latter on improving or optimising the use of the GP algorithm.

In addition we can use various techniques to speed up the evaluation times, including hardware solutions and parallel and distributed processing. The processing requirements can sometimes be aggravated by paging caused by excessive memory utilisation. The solution to the latter has already been covered in the previous section, and therefore will not be examined here.

7.5.1 Speeding Up Fitness Processing

7.5.1.1 Use Hardware

There is considerable work in progress with using evolutionary techniques to evolve hardware. The technique involves evolving connections for Field Programmable Gate Arrays (FPGAs) which can be reconfigured at high speeds, and allows low level signalling to be exploited without simulation. It is doubtful however that this technique is any faster than simulation using Spice [KDBK99] on the fastest general purpose processors. The FPGAs tend to have clock speeds of 500MHz or less whereas the top end Pentium processor have reached almost 2GHz.

7.5.1.2 Machine Code GP

The principle is to get GP to directly evolve machine code. The idea being that the code generated should be extremely fast in execution [NBF99; Nor97]. However, the cost is that we lose the abstraction provided by high level languages resulting in larger code sizes, particularly when we deal with complex problems. This in turn affects the search space; there are fewer permutations and combinations of operators and operands in a 100 node tree than a 10000 node tree.

7.5.1.3 Sub-Machine Code GP

In sub-machine code GP [PL98; PPL99; PP00], the internal parallelism in modern processors is exploited to speed up fitness processing. Poli *et al* report a speed up using this technique of nearly two orders of magnitude for Boolean Classification problems.

7.5.1.4 Use Efficient High Level Languages

It is argued that using efficient high level languages like C and C++ offers performance advantage over languages like LISP, and we note that Koza now uses a C implementation of GP. Languages such has Java however, offer alternatives such as run time optimization and results with GPsys suggest that this may not be necessary.

7.5.1.5 Compiled Code

The code generated by GP is typically interpreted many times during fitness evaluation. We can therefore speed the evaluation time significantly by writing the code to a file and compiling it to generate machine code. This would only make sense if the evaluation time takes many times longer than the compilation time. Harris [HB96a; HB96b] used this approach to speed up the evaluation time of feature detection code generated by GP.

7.5.1.6 Parallel and Distributed Architectures

The use of parallel and distributed architectures provides one of the most significant ways of making GP scalable. The scalability comes from being able to add processing power incrementally as and when required.

Parallel computers minimise the costs associated with having multiple processors by sharing computer hardware, such as the motherboard, disks, memory, VDU and other peripherals between the processors. Their main disadvantage is that they present a single point of failure. There is always a limit to how many processors can be added to a parallel processing system and hence scalability is limited.

In comparison, distributed architectures are more costly per CPU, but offer greater flexibility in that it is easy to add or remove processing power. There is theoretically no limit to how many computers can be added and hence distributed architectures are extremely scalable. Each networked computer can fail without affecting others and hence a distributed system is more fault tolerant. The main disadvantages are cost and management issues.

The most recent trend is to combine both parallel and distributed architectures to produce super computing clusters, which offer the best of each approach. The Linux Beowulf architecture is one example.

There are a number of techniques we could employ to exploit parallel and distributed processing:

- evaluate individuals from a population in parallel
- evolve demes in parallel, allowing communication between demes
- execute a number of runs in parallel

The first technique works very similar to standard GP, except that we farm out evaluation of individuals to a number of processors. These processors may be on the same parallel processing computer hosted by a multithreaded operating system, or distributed across networked computers. The latter is only effective if the time taken to evaluate an individual is significantly larger than the time take to communicate individuals and evaluation results. A disadvantage with this approach is that it is not fault tolerant. If one or more of the processors fail we need to ensure that the evaluation of the individuals allocated to it are restarted elsewhere. As with all distributed systems, complex failure semantics make the latter difficult.

The last technique is extremely simple, and is used by most GP practitioners. Here we execute a separate run on each processor or host. There are no communication overheads with this approach.

The standard GP breeding policy is described as panmictic since it potentially allows any individual in the population to breed with any other individual. An alternative is to divide the population into sub-populations called "demes", or islands. These demes are typically positioned on a two dimensional grid, and breeding is restricted so that any two parents have a much higher probability of being selected from within the same deme or from geographically close demes.

This approach has been investigated both by D'Haeseleer and Bluming in [DB94], and by Tackett and Carmi in [TC94]. Tackett and Carmi use explicitly defined demes of different sizes for a problem with tunable difficulty (the donut problem). Their results suggest that use of demes significantly improves performance. In contrast D'Haeseleer and Bluming instead of using explicit demes, introduce locality in the form of geographic neighbourhoods, and select parents from the same neighbourhood and also place the child into the same neighbourhood. They have detected the spontaneous emergence of deme like structures in their results and report that the introduction of locality significantly improves population diversity which in conjunction with slightly increased generality of individuals, yields a substantial improvement in the generality of the population as a whole. Recalling that loss of population diversity is one of the reasons for premature convergence, demes therefore delay convergence.

One of the advantages of the demic approach is that it is easy to implement using parallel or distributed architectures and has very low communication overheads. A second advantage is that it is extremely fault tolerant, the loss of one or more demes does not halt a demetic run. Koza and Andre [KA95; AK95; AK96a; AK96b] developed a GP system using a network of transputers to exploit the parallelism inherent in the demic model. Each processor was allocated a separate deme. The results of his experiments with EVEN-N-PARITY problems indicated that more than linear speed-up in solving the problem using GP was obtained. Figure 7.1 a) shows the architecture used by Koza. The demes were connected to form a torroidal structure which has the advantage of good communication between neighbouring demes as well the ability to maintain high connectivity in the face of node failure. Bennett and Koza have more recently employed the use of a Beowolf architecture [IKSS99] which effectively combines distributed and parallel architectures to provide scalable processing power. This architecture has been used for evolving programs using GPPS which has 3 orders of magnitude greater processing requirements than

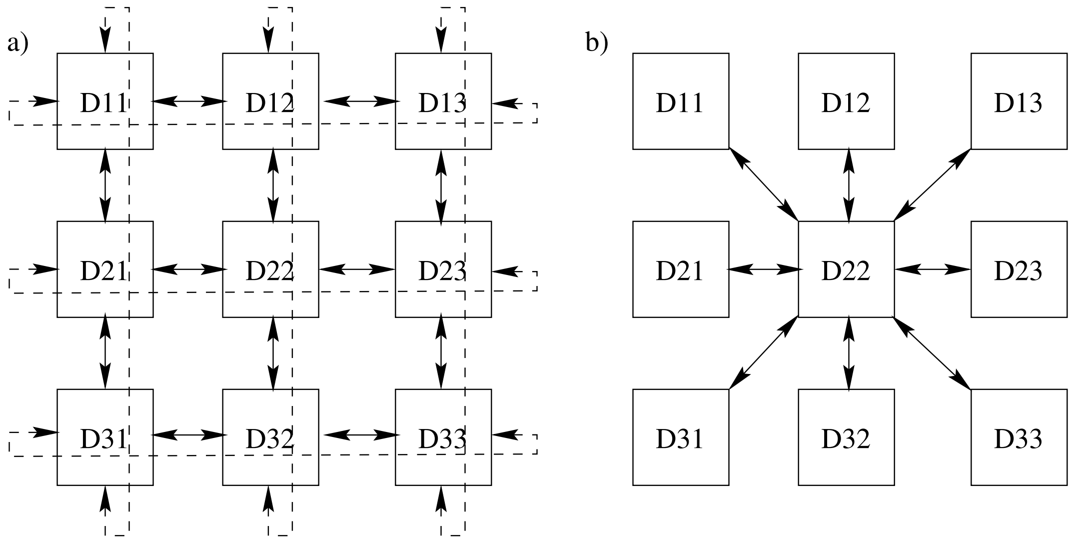


Figure 7.1: Parallel Demetic GP

traditional GP systems.

Fernandez and Punch [FTPS00; Pun98] investigated the use of parallel GP on a number of common GP test problems. Their goal was to understand how parameters such as the the number of sub-populations (demes), the number of individuals in each population, and the migration rate affect the performance of a parallel GP system. They found that a problem dependent optimal range of values for these parameters exists. Using the optimal range of values they were able to reduce the effort in solving the problems.

The Internet with its millions of networked computers offers excellent possibilities for distributing work. This has been exploited by the SETI project which works by means of a specialised screen saver. The latter kicks into operation during long periods of inactivity and connects to a specific computer to download data to be processed. The results are shipped back to the same computer after completion. It would easy to create a demic GP system which works on the same principle. The architecture would probably be shaped as in part b) of figure 7.1. The computer hosting the GP system would form the hub, allowing communication between demes to occur and collection of results. This approach was used by Chong [Cho99a; Cho98; Cho99b; CL99] via a Java based GP system running as an applet.

Poli [Pol96b; Pol96a; Pol97b] devised a new form of GP called Parallel Distributed Genetic Programming(PDGP). PDGP uses a graph-like representation a provides fine-grain parallelism. Poli has shown in experiments with PDGP on standard GP problems that it performs significantly better than standard GP. An additional advantage with PDGP is that it allows symbolic and neural processing elements to be combined freely [Pol96c; Pol97a; Pol97c].

7.5.2 Reducing Fitness Function Evaluation Costs

Reducing the evaluation cost ultimately relies on making the fitness function as efficient as possible. The most effective way of achieving the latter is by using test case sampling. Our proposal of using formal testing techniques is unproven but is promising. The remainder are simple rules of thumb.

7.5.2.1 Deadlock Detection

When evolving MAS programs, we normally repeatedly execute the evolved program for a given test case until the maximum number of iterations permitted have been expended. In situations where deadlock occurs we would be executing the programs needlessly until we have reached the maximum number of iterations. Detection of deadlock would allow the test case to be aborted and hence save processing time.

7.5.2.2 Caching

During fitness evaluation, particularly involving ADFs, it is possible that the evaluation of a branch or ADF gives the same results regardless of how many times it is executed, i.e. it is idempotent. For this case we can cache the results in memory and use this value instead of re-evaluating the code repeatedly.

Handley [Han94a] exploited his DAG representation of the population to implement an extremely efficient method of caching. Duplicated subtrees are represented just once in the DAG representation, and hence by caching the computed value of each subtree he was able to avoid re-evaluation for each instantiation, even when the same sub-tree occurs in a different population. This was made possible by restricting functions so that they have no side effects and by fixing the test cases used by the fitness function. This approach reduced the number of node evaluations by 11-20 fold per run.

7.5.2.3 Reducing Wasted Computation By Aborting Runs

If we can detect premature convergence, we can abort runs which will ultimately fail. However detection of premature convergence is non trivial. One simple check is to ensure that the genetic material known to be part of the solution has not been lost.

Teller and Andre [TA97] describe a method that can be used to reduce wasted computation in fitness evaluation by choosing the optimal number of fitness cases for each individual. Gathercole and Ross [GR97b] use the concept of an error limit to prevent individuals which exceed this limit from being further evaluated. This allows them to reduce the number of fitness test cases applied to an individual.

7.5.2.4 Simplifying the Code

We could copy the individual being evaluated and remove any redundant code (also known as introns) from the copy just before evaluating it. This can be justified as long as the time take to evaluate is significantly larger than the time taken to simplify the code. Care must be taken to prevent code with side effects from being removed. An example is a subtraction resulting in zero, where the operands are calls to functions that are expensive. Koza developed a LISP program that automatically removed redundant code [Koz92b] for evaluation.

7.5.2.5 Sampling Test Cases

Reducing the amount of tests cases against which a program must be tested as part of the fitness function is a simple and highly effective way of reducing the evaluation cost. The mechanism that was used for almost all the experiments in this thesis is variation of test case sampling. In test case sampling instead of applying the full set of test cases for each fitness evaluation, we sample from a set of test cases. Typically a sample size significantly smaller than the test case set is chosen. In our experiments this sample size ranged from 1/6th for the pursuit problem to 1/20th for communicating agents problem. In our approach, for each generation, we randomly sampled a fixed number of test cases from the set of test cases and used them to evaluate each individual of the generation. The fitness value assigned to each individual is the average over the test cases. This approach allows individuals from the same generation to be compared easily. As mentioned in chapter 4, a key problem with this approach is that as we move from one generation to the next, the best of the generation may outperform the current best of the run, not because it is generally better, but because the sampled test cases were easier. Ideally we should weight the fitness values according to the difficulty of the test cases that were passed. Unfortunately this is not easy in the pursuit domain, where given a particular arrangement of agents it is difficult to measure how much effort is required to enable capture. To overcome this problem we devised a technique whereby after each generation, we re-evaluate the best individual of the generation using the full training set of test cases. This individual becomes the best of the run if its full fitness value is greater than the current best. Justification for our approach is provided by experiment 4s in chapter 4. Gathercole and Ross [GR94] describe a technique called Random Subset Selection which is similar to the technique described above.

7.5.2.6 Redundancy in Test Cases

We note that as we approach the end of a run, the number of test cases that are failed become smaller and smaller. It is likely that population would have converged on solutions to the easiest of the test cases early in the run. It seems wasteful therefore that these same test cases are still used later in the run. One solution might be to assign weights to the test cases dynamically, such that test cases most frequently passed are deemed easier and hence have smaller weights than those that are infrequently or never passed. In this scheme, the fitness sampling approach discussed earlier would be augmented to select test cases with probabilities in proportion to their weights. The danger with this approach is that the population may ”forget” how to solve the earlier test cases. However, the dynamic way in which the weights are defined should compensate. This approach has been investigated by Gathercole and Ross and implemented as dynamic subset selection [GR94; Gat98].

7.5.2.7 Software Testing

If we compare the human software development cycle with software generated by GP we note that whereas test cases are used in software engineering to *verify* that the program conforms to the requirement specifications, in GP the test cases along with the fitness function *are* the requirement specifications. The analog of the traditional software testing phase in GP is the testing that we perform at the end of a run for generalization of the evolved code. Once again we note that instead of comparing the results of tests with the specifications, the test cases are themselves most of the specification. This strong dependency of GP on software testing suggests that formal software testing techniques are extremely important for GP. It is surprising therefore that little work has been done on the use of formal software testing in GP. In proposal 2 described later in this chapter, we describe how formal software testing techniques may help improve the accuracy and efficiency of fitness evaluation.

7.5.3 Improving the efficiency of the GP algorithm

We could reduce the cost of evaluation, by reducing the number of evaluations that are required. The number of generations required to find a solution, or one that is closest, is highly dependent on the problem domain. To minimise this number, we could:

- reduce the size of the search space
- improve the quality of the search space (make it less sparse)
- improve the genetic operators used to navigate the search space
- improve the starting population for a run

Important configuration parameters are the population size, the choice and number of functions/terminals and the maximum allowed size of trees. Increasing the population size increases the probability of obtaining a solution in earlier generations, but a significant cost is added for evaluating each generation which is likely to cancel out any useful gains. The maximum allowed size of trees, and the choice and number of functions and terminals directly affects the size of the search space; few high level functions and terminals are likely to be better than many simpler ones. If the number of functions and terminals is given by $|F|$ and $|T|$ respectively, and the maximum size of the trees is limited to l then assuming the arity of functions in the function set is 2, an upper bound for the number of possible program trees that we can generate ([Lan00]) is given by $|T|^{(l+1)/2}|F|^{(l-1)/2} \times \frac{(l-1)!}{((l+1)/2)!((l-1)/2)!}$. Note that this is likely to be an overestimate, as we have assumed *closure*; the use of a strongly typed GP system [Mon95] will reduce the number of possible trees. The latter however may impact the quality of the search space in a negative way, by making it more sparse for example and thus making it more difficult to find solutions and therefore cancelling any gains. Through use of ADFs we can restrict the maximum size of the evolved solution and hence reduce the size of the search space. However, Koza reports that for simple problems it is better not to use ADFs [Koz94b].

Improving the quality of the search space is very difficult. The major factors being the difficulty of the problem itself, the choice of functions and terminals and the fitness function. An ideal fitness function should be sensitive to extremely subtle changes in performance and be more continuous in its measurement than discrete. The challenge is how to describe a problem in a fitness function which has those characteristics.

The standard crossover and mutation genetic operators generate new individuals almost blindly, from their parent(s). The result being that the new individual has a good probability that fit portions of code are disrupted. Creating more intelligent genetic operators that avoid disrupting part of the code marked as useful may help evolution. Teller [TV95; Tel96] developed an intelligent crossover operator that learns to select good crossover points using information such as the execution path of the evolved programs. He reported significant improvements in the rate of production of fitter children using this operator.

Langdon recently analysed how the shape of trees generated for the initial population affect the performance of the a run [Lan99]. He suggested that problems can be divided into two types, those in which the solutions trees are deep and those where they are wide. The creation method parameter is used in GP systems to specify the shape of the initial population, with values including Grow, Full or Ramped half and half [Koz92b]. The Grow method tends to create many short trees, the full method creates deep trees. If we choose the wrong type for a given problem, we may disadvantage the run. An analysis of the fittest individuals of randomly shaped trees may provide insight into the value to be used for this parameter.

7.6 Proposal 1 - Combining GP with a Database

Instead of using memory, we could use a database to store the population. This is possible since, at most we need only two individuals to be in memory at any given time. Fitness evaluation only requires one individual to be in memory, and crossover requires two. Thus we could store the population in a database and retrieve individuals as and when required. An important requirement of this approach is that the time taken to download an individual from the database should be significantly smaller than the evaluation time. Current RDBMS allow very fast storage and retrieval (millisecond access depending on hardware) of a very large number of objects (in the order of Terra-bytes) and hence retrieval of individuals from a database should not impede performance significantly. Furthermore, since there are no concurrent access requirements in our application, we would not require database locking and therefore should be able to significantly increase the performance of database access.

Using a RDBMS or an ODBMS, we should be able to apply GP to particularly difficult domains where the population sizes are very large and/or the size of the evolved programs need to be very large. Using the same calculations used at beginnings of the chapter, we can quite easily increase the population size by 2 to 3 orders of magnitude without running into problems with current databases. Alternatively or in addition, we could also increase the size of the individuals by 3 orders of magnitude. This would allow us to evolve Megabyte size code using populations of millions of individuals. Whilst this seems hopeful, in the next section we shall see that the real constraint is the computational load.

The use of databases brings additional benefits:

- results stored in a relational database allow easy post-processing, and there many tools that integrate RDBMS with spreadsheet calculators
- assuming the use of transactions to write each new generation to the database, we can easily restart runs after a system crash or reboot using the last saved generation
- we could store every generation of a run to allow analysis of evolutionary flows. We would also need to store information about the genetic operator used to create the individual together with the trees/subtrees of the parents involved. This may help us better understand the GP algorithm.
- We can have a population that grows by one generation each generation so that no individuals are ever removed. This could be used to reduce loss of population diversity.

7.7 Proposal 2 - Using Formal Software Testing Techniques for Fitness Evaluation in GP

When evolving programs using GP we measure a programs fitness by testing the program and measuring success. We test programs by initialising the inputs to some predefined parameters, executing the program and then measuring in some way the difference between the desired output and the output obtained. The tuple consisting of input values and desired outcome define a test case. In GP the desired outcome is effectively encoded in the fitness function and hence only the input values for each test case is required. Many test cases are needed to test a program. Ideally we would like to apply all the possible test cases for a given application so that we get an accurate measure of the program correctness or fitness. In practice however this is seldom feasible due to large number of test cases that are possible. Consider for example, our pursuit problem.

There are 4 predators and one prey that can be placed in any of the cells in a 30 by 30 board. If we place the prey first, there are 900 positions to choose from, and for each such positions, we need to choose 4 positions for the predators. If the predators are homogeneous then we can choose from any of the 899 cells in an order independent manner. For heterogeneous predators, the order matters and we need to consider permutations of the different orderings. The number of possible combinations for selecting cells for 4 homogeneous agents and from 899 positions is given by the binomial co-efficient

$$\binom{899}{4}$$

and evaluates to about 27 billion positions. Multiplying by 900 for the prey positions gives approximately 24 trillion test cases. For heterogeneous agents there is an additional factor $K!$ that we must consider, since the ordering of the predators is important. This gives us approximately 583 trillion test cases. To make things worse, we can multiply the above values by the different number of programs that we will be used to drive the prey, since each is effectively a different test case. In a typical GP run consisting of a population of 1000 individuals, evolved for 500 generations, allowing 50 cycles for each test case will require $1000 * 500 * TestCases * 50 * 4$ evaluations of our program.

Clearly exhaustive testing of our program is not going to be possible. We are therefore forced to take a very small subset of the test cases and use them for training purposes. In our experiments for the pursuit problem, we chose 60 such test cases, there were actually 30 different placements for two different types of prey. Furthermore, out of the 60 test cases, to reduce the time taken to evaluate individuals, we exposed each generation to a sample of only ten test cases, which are chosen at random for each generation. Note that for our implementation of the pursuit problem, since the agents are stateless, a new test case is effectively created by each simulation cycle. Therefore the effective number of test cases seen by each individual during an evaluation can be as many as 50 times the number of scheduled test cases. This is still a small number considering the full space of test cases.

Even when testing the generality of any evolved solution at the end of a run, we still can only sample from the full space of possible test cases because of the high cost of evaluations. For our experiments, we generated 1000 unique random positions for the agents and created 2000 test cases out of them using still and randomly moving preys.

Since exhaustive testing is seldom possible, the challenge is to define subset of all possible test cases that can achieve maximum coverage of the test space. Many of the techniques developed for efficiently testing human coded software systems [Mye79; Bei95] help in answering the above question. These techniques can be divided into two sets, *black box testing* and *white box testing*. There are also hybrid techniques that combine the two to maximise test coverage.

7.7.1 Black Box Testing

Black box, behavioural, functional and dynamic testing are all names for testing techniques that test programs by executing them. They treat a program as a black box which has inputs and outputs and are not concerned with the structure or internal behaviour of the underlying code. Tests are performed by applying different inputs and checking the output for conformance with the requirements. Clearly complete testing would therefore require tests for every possible set of inputs, which is is not possible. Consider for example testing an optimizing "C" compiler! *Equivalence partitioning, boundary-value analysis, cause effect graphing and error guessing* are methods that can be used to design black-box test cases that maximise error coverage using the minimum number of test cases. Beizer [Bei95] lists other techniques, but many of these require building a graph based model of the program being tested and rely on assumptions about the likely structure of the code being tested.

7.7.2 White Box Testing

White box, structural logic-driven and static testing are all names for test techniques that work on the source code itself. Tests are created by examining the program logic and selecting inputs that exercise as much of the logic as possible. The analogue of exhaustive input testing for white-box testing is exhaustive path testing. Exhaustive path testing requires that we execute all possible paths of control flow through a program. The latter is again infeasible due to the very large number of flows created via conditionals, loops and function calls. A number of methods are available for white-box testing including *Statement, Decision, Condition, Decision/Condition and Multiple-Condition Coverage*. All of these methods can be automated to enable white-box test cases to be generated by a program. Many test generation tools use white-box testing.

7.7.3 Applying Software Testing to GP

The use of test cases are important for two phases of GP. The first phase is for training purposes and the second is for generalisation.

7.7.3.1 The Training Phase

During the training phase each program in each generation is tested via the fitness function. The fitness function executes the program using a set of test cases and measures success. The test cases could be designed using software testing techniques such as white-box or black-box testing. However, the programs that we are testing in GP are constantly changing, and therefore tests generated via white-box testing techniques cannot be reused, and need to be re-generated for each program during the evolutionary process. Clearly this is likely to be too costly. Black box testing techniques are ideal, as they do not rely on the structure of the code being tested. The implication is that we can design test cases once using black-box techniques and use them throughout for testing each generated program.

Black box-techniques such as equivalence partitioning and boundary-value analysis should help us design the minimum number of test cases to cover a large areas of the test space. This should in theory be much better than the random testing techniques that are commonly used, and hence make our fitness functions more accurate as well as efficient. This in turn should help scale GP. It should be noted that even using black-box techniques, the number of test cases is still likely to be too high for evaluating each individual, and hence the sampling technique detailed used in our experiments is still applicable. The main difference being that the test cases that are sampled have been designed for maximum coverage.

7.7.3.2 Generalisation

At the end of a run GP practitioners usually re-test the program using unseen test cases as check of for the generality of the solution. This is typically done by using a larger sample of randomly generated tests. As a quality control mechanism this is far from optimal. A better strategy would be to use a combination of white-box and black-box test techniques which together would provide a much more accurate measure of the quality of the program. White-box test techniques can be used at the end of run since we only have a single program for which we need to generate test cases.

7.7.3.3 The Pesticide Paradox

In human coded software, information about errors that have been detected via a set of tests is used to edit the programs and remove all occurrences of such errors. The usefulness of the set of test cases that were used therefore decreases. A similar process happens in GP. As evolution progresses more and more test cases are passed by individuals in the population. The effectiveness of such test cases is therefore decreased. Software testing techniques have therefore co-evolved with the program errors that they try to detect. This suggests that maybe research into how we can co-evolve test cases for any given application might be interesting. Hillis [Hil92] co-evolved test cases for a sorting networks application and reports that this approach not only reduced the effort in creating test cases, but also improved the solutions found.

7.8 Conclusion

The relatively poor performance of random search compared with GP was analysed and explained using calculations on the estimated size of the search space. A methodology for evolving multiagent systems using GP, that can be used in conjunction with other methodologies for analysis and design to automatically program multiagent systems was presented. The technical scalability issues arising from the application of this methodology to more complex problems than those considered in this thesis were considered and some solutions proposed.

Chapter 8

Conclusion

In this chapter we review the work that we have done and critically evaluate it. We then propose further research stemming from our work and summarise our contributions.

8.1 Review of the Work

In chapters 4, 5 and 6 we used GP to evolve agents that show well coordinated, coherent behaviour. In chapter 4 we evolved both homogeneous and heterogeneous pursuit agents using game rules and test cases that are biased towards creating more conflicts. We showed that it was easier to evolve successful heterogeneous agents than homogeneous agents. The explanation was that the heterogeneous agents avoided conflicts by statically allocating tasks between them. The homogeneous agents must allocate tasks dynamically which is difficult. We showed that through the use of identities homogeneous agents could perform as well as the heterogeneous agents. We also evolved combinations of homogeneous and heterogeneous agents showed that GP tries to optimise the allocation of tasks by minimising conflicts. We noted that the problem with dynamic task allocation is that the agents are more prone to task overlap. This task overlap in turn results in greater conflicts. We therefore added an operator that we thought would be helpful for resolving conflicts. We then tried to evolve agents for the pursuit problem using this additional operator and showed that homogeneous agents had developed social conventions that help detect and resolve potential conflicts.

In Chapter 5 we showed how GP can be used to create multiagent systems in which the agents communicate explicitly. We showed that GP can create agents that know how to send and receive information across two channels. The programs decided what information was sent across which of the channels, and how to interpret the information received from a channel to achieve the task in hand.

In chapter 6 we evolved agents that know how to work within a society of existing agents. First an agent was evolved which knows how to communicate with the best agent evolved in chapter

5. The agents was evolved through interaction with the environment and the other agent to correctly interpret information received through the communication channels and what information it needs to transmit through the communication channels. Next we showed that we can evolve a pair of heterogeneous agents that know how to work with two other heterogeneous agents (the best evolved heterogeneous agents of experiment 4b). The heterogeneous agents of chapter 4 used strict task allocation, to avoid conflict. The new agents evolved to divide the remaining tasks between them. We then showed that we could also evolve a program that drives a pair of homogeneous agents, which must interact with an existing society of homogeneous agents (the best evolved homogeneous agents of experiment 4a) that have predefined social conventions used to resolve conflicts. We showed that the GP driving the new homogeneous agents evolved to behave according to almost all of the social conventions used by the existing society, and to exploit those conventions.

In chapter 7 we analysed the performance of random search and estimated the size of the search space for each of the experiments in this thesis. We then defined a methodology from evolving MAS using GP and considered the technical scalability issues, and reviewed possible solutions as well as proposed new ones. We concluded that fitness evaluation is the key technical scalability issue.

8.2 Critical Assessment

The following are key limitations of our work:

- We used only symmetric problem domains of limited complexity.
- We did not apply our technique on any real-world problems.
- We focussed our attention on evolving and co-evolving cooperative agents (both homogeneous and heterogeneous) and did not investigate competitive co-evolution of heterogeneous agents.
- We evolved only purely reactive agents.
- We only examined technical scalability issues and did not consider more general scaling issues such as the performance of GP when the number of agents is increased or the problem difficulty is scaled. Nor did we consider issues affecting scalability such as choice of representation or genetic operators
- We proposed that software testing techniques and the use of database may be useful for GP, but did not perform any experiments to support our proposal.

8.3 Future Work

The following lists further research directions stemming from our work:

- Using GP for Real World/Large Scale MAS Applications
- Using GP for Asymmetrical MAS Problems
- Competitive Co-Evolution of Heterogeneous Agents
- Evolving Agents with Memory
- Socialisation in More Diverse Agent Societies
- ADFs and Communicating Agents (modeling communication as function calls)
- Agent Learning and the Baldwin Effect
- Evolving Distributed Algorithms
- Scalability of GP
- Software Testing and GP
- Using Databases to Store Populations
- Automated Software Testing of MAS
- Investigating Evolution of Human Languages by Evolving Communicating Agents

8.4 Summary

We have achieved our objectives of showing that GP can be used to evolve MAS automatically. Our key contributions have been to show that GP can be used to evolve agents that:

- exhibit coordinated, coherent behaviour (chapters 4, 5 and 6)
- can resolve conflicts (chapter 4)
- communicate explicitly, and in doing so decide what to communicate and how (chapter 5)
- can be integrated into an existing society of agents (chapter 6)

We also have defined a methodology for evolving multiagent systems with GP and examined the technical scalability issues involved in the use of GP, both generally and in particular as a technique for automatically programming agents, and proposed solutions to these problems.

Appendix A

GPsys

GPsys is an advanced extendible GP system written in the Java programming language. The system consists of over 16600 lines of code, at least 50% of which are comments and documentation. This environment was used for all the experiments in this thesis and is also used by many GP researchers worldwide (the current release has over 1000 recorded downloads). GPsys can be freely downloaded complete with source code and documentation from:

"http://cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html"

The design goals of GPsys and how they were realised are shown in table A.1.

Design Goal	Implementation
easy of use	Plug and Play model Factory/Observer-Observable design patterns
support for most useful GP extensions	ADFs, STGP
extensibility	OO-Design
flexibility	OO-Design
portability	Java
robustness	Java, Exception Handling
high performance	Java
efficiency	SSGP, Memory Efficient Generational Engine
support for persistence	Java Serialization, GZIP Compression
support for parallel/distributed architectures	Java Networking/RMI

Table A.1: GPsys Design Goals and Implementation

We deemed the most useful GP extensions to be STGP and ADFs, these were consequently in the design. It should be noted that in our ADF implementation, crossover is performed for each ADF. Our original design (GPsys 1) used a SSGP engine to improve memory efficiency,

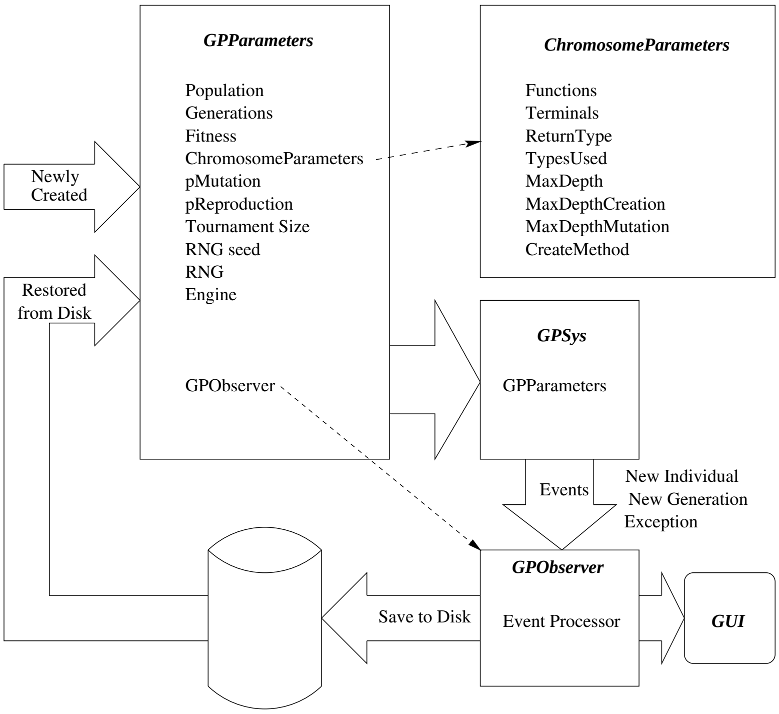


Figure A.1: GPsys a High Performance GP System

the current release has both a SSGP and a memory efficient generational engine. Many of our design goals were realised just by choosing Java as the implementation language.

A.0.1 The Java Programming Language

Java is a robust, powerful and efficient, multi-threaded, distributed object oriented programming language. Java compiles into bytecode, which is a machine code for a virtual machine called the Java Virtual Machine (JVM). The JVM interprets the bytecodes to execute the program, and has been ported to many different platforms (operating systems and processors) and provides Java with excellent portability. Most JVMs contain code to speed up the execution of Java programs including just in time compilers and run-time optimisers. The former dynamically compiles bytecode into the machine code for the target processor, the latter monitors program execution and optimises the executing code. Together these techniques overcome any performance arguments against Java.

The Java language has been designed for robustness, which is achieved by minimising programming errors. Programming errors are minimised by replacing the use of pointers by references. References differ from pointers in that only the JVM can generate references, which cannot be used in arithmetic operations to create new references. This feature alone removes many pointer errors that plague languages like C and C++ such as segmentation violations and bus errors. Arrays access is also checked at runtime for out of bound errors. The inclusion of exception handling forces the programmer to consider and deal with run-time exceptions at compile time. This again reduces the probability of errors.

Java has an extremely rich library of pre-built objects which are included in every JVM. These objects include support for graphics, databases, networking, input/output, advanced data structures, string manipulation, compression algorithms and much more. Thus small, very powerful programs can be written with minimal effort.

There is excellent support for distributed computation, in the form of Sockets, or Remote Method Invocation. JVMs are also included in many web browsers, which allows applets (Java code designed to be run in a browser) to be loaded and executed securely within the browser thus opening the possibility for massively distributed computing.

A.0.2 Application Programmers Interface

One of the key goals of GPsys was that it should be easy to use, requiring no code changes to the GP engine by the user. We therefore created a "plug and play" model as depicted in figure A.1. To use GPsys, one instantiates a GPsys object, and then calls the evolve method, which starts evolution. In order to instantiate GPsys, we must pass to the constructor a *GPParameters* object.

The GPParameters object codes all the parameters that are required by GPsys, and in addition is used by GPsys to store all state information during a run. The GPParameters class definition includes methods that allow a GPParameters object to be saved and restored from disk, hence allowing a GPsys run to be suspended and restarted. The full set of parameters that can be specified include parameters for the run, and parameters for the problem (table A.2). GPsys supports ADFs, and the parameters for each ADF tree is specified using ChromosomeParameters (table A.3). The function and terminal set for each ADF is specified as Function and Terminal arrays in a ChromosomeParameter. Each Function and Terminal in these arrays implements the *factory design pattern* [GHJ$^+$95], so that GPsys can create many instances of these objects by calling factory methods. Many common GP primitives are provided as part of a primitives package in GPsys, including conditionals, arithmetic functions, and indexed memory operators. Some of these primitives support generic types.

The object used to store and measure the fitness of an Individual is specified by the Fitness object in the GPParameters object. Once again, the factory design pattern is used, so that many Fitness objects can be created using this one instance. The Fitness object abstracts the user implementation of a fitness function and fitness representation. Abstract methods define the contract that the user must fulfill to allow the fitness object to be used by GPsys (table A.4). The termination condition is also specified by the fitness function.

GPsys makes use of the *observer/observable design pattern* [GHJ$^+$95] to provide a clean way of monitoring a run. The users need to create and register an *observer* object with GPsys (the *observable* object) in order to be told when something interesting happens. The observer object must be an instance of a class that implements the *GPObserver* interface, and is registered by passing a reference to it in the GPParameters object. The GPObserver interface specifies the call-back methods that are used to communicate information by GPsys to the user (table A.5).

Run Parameters	
Parameter Name	Description
populationSize	The size of the population to be used.
generations	The maximum number of generations.
pMutation	The probability that a mutation operation is chosen.
pReproduction	The probability that a reproduction operation is chosen.
tournamentSize	The size of the tournament to be used for tournament selection.
rng	The random number generator to be used.
rngSeed	The seed to be used for the random number generator.
engine	The GP engine to be used for evolution. Possible selections include a generational engine or a steady state engine.
Problem Definition Parameters	
Parameter Name	Description
adf	An array of ChromosomeParameters which define the parameters for each Chromosome (ADF) to be evolved.
fitness	The fitness definition for the problem.
observer	The observer object to which information during evolution is sent.

Table A.2: GPParameters

A.0.3 Internal Data Structures

The internal data structures used by GPsys are shown in figure A.3. A Population consists of among other things an array of Individuals. Each Individual has associated with it an array of Chromosomes, one for each of the ADFs and a Fitness. The result producing branch(RPB) is normally the first ADF in the array, although this can be changed. Each Chromosome consists of a tree of Genes.

There are two kind of Genes in the tree, GeneFunctions and GeneTerminals. GeneFunctions represent function calls and hence have an array of Gene arguments associated with them. GeneTerminals are the leaves in the Gene tree and represent Terminals. All Genes hold Primitives which contain the Type and code definition for the primitive which they represent.

Parameter Name	Description
createMethod	The method used to create the initial population. Possible values include Full, Grow and Ramped Half-and-Half.
maxDepthAtCreation	The maximum creation depth of the Gene Tree for this chromosome.
maxDepthAtMutation	The maximum depth of trees created for mutation.
maxDepthAt	The maximum depth of trees for this chromosome.
functions	The function set for this chromosome.
terminals	The terminal set for this chromosome.
type	The type to be returned by this chromosome.
types	The set of types used by this chromosome.

Table A.3: ChromosomeParameters

Function	Description
void add(Fitness f)	Add a fitness to this fitness object.
void divide(int divisor)	Divide the fitness by the specified integer.
boolean equals(Fitness f)	Tests if this fitness = f.
boolean lessThan(Fitness f)	Tests if this fitness < f.
boolean greaterThan(Fitness f)	Tests if this fitness > f.
Fitness instance()	Creates a default Fitness object.
Fitness instance(GPParameters p, Individual i)	Creates a Fitness Object by evaluating the fitness of i.

Table A.4: Fitness

There are two kinds of Primitives, Functions and Terminals. Functions represent Primitives which take arguments, and therefore contain Type specifications for each argument that they take. Terminals represent zero-arity primitives, and hence just contain code definitions. ADFunctions and ADTerminals are subtypes of Functions and Terminals which represent automatically defined functions and terminals respectively. They therefore do not contain code definitions, but instead contain an index of the Chromosome that defines them in an Individual's Chromosome array.

Users create their own functions and terminals by extending Function, Terminal, ADFunction or ADTerminal classes. Functions and Terminals are evaluated by GPsys when a GP is executed during fitness evaluation. The return value can be of any type, and depending on the type, the user must override one of the evaluation methods which forms part of the contract of a user defined

Function	Description
void generationUpdate(GPParameters p, int cm)	Invoked after a new generation has been created. cm specifies how generation was created (via creation, from a stream or evolved)
void individualUpdate(GPParameters p, Individual i, int cm)	Invoked after a new Individual has been created. cm specifies how the Individual was created (randomly, via reproduction, mutation or crossover).
void diagnosticUpdate(java.lang.String s)	Invoked whenever unusual event occurs, s indicates what happened.
void exception(GPException e)	Invoked whenever an exception occurs.

Table A.5: GPObserver

Function or Terminal. These evaluation methods have been divided for efficiency reasons into those that return Java primitives (byte, short, int, long, char, boolean, float, double) or those that return an object or an array.

The Population object also contains a reference to a CrossoverBookKeeping data structure. The latter is used to implement memory efficient crossover for the generational evolutionary engine as described by Koza [KDBK99]. Most GP systems maintain two populations during a run, the first is the active population which is used to create the next population. This new population then becomes the active population in the next iteration. The memory efficient crossover process divides the task of generating a new population into two phases. In the first phase we decide which parents from the current population are going to reproduce using which genetic operation. This information is stored in a book keeping data structure. In the second phase we execute the genetic operations to create the new individuals.

The advantage with this approach is that by carefully scheduling the order in which we execute the genetic operations we can greatly reduce the amount of individuals that we need to hold in memory. A pigeon hole argument can be used to calculate that if the population contains M individuals, then we only need space for at most $M+2$ individuals, instead of the $2M$ individuals normally used. The CrossoverBookKeeping data structure holds four lists. These four lists hold information about parents and the genetic operations in which they are involved. Parents that have 0, 1, 2 and more than two genetic operations outstanding are stored in lists 0, 1, 2 and 3 respectively. When executing the genetic operations, the individual to be replaced is chosen from list 0 (which is always initialised to contain information about the two extra slots mentioned earlier). The parent to be used in the next genetic operation is the parent with the least number of outstanding genetic operations.

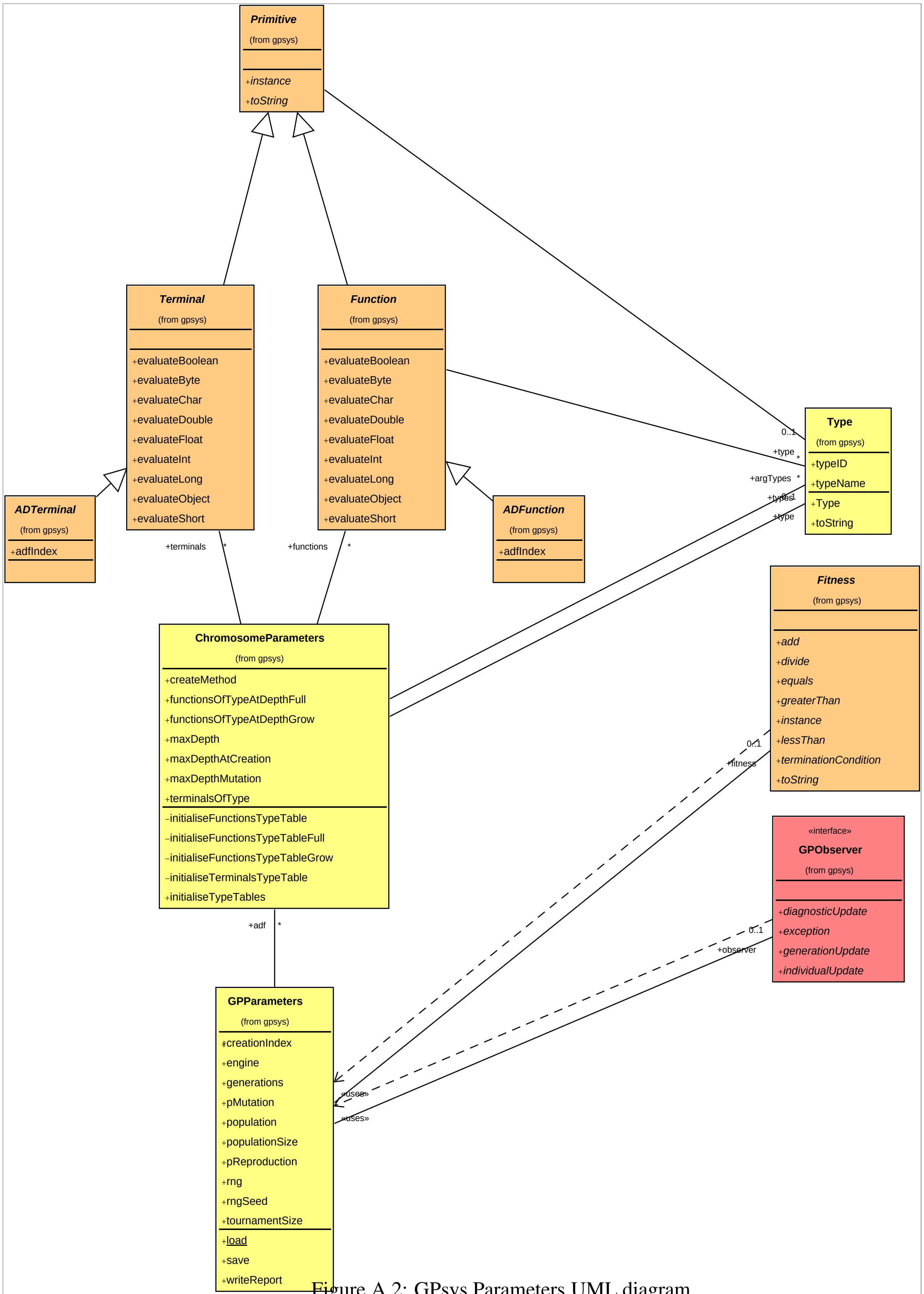


Figure A.2: GPsys Parameters UML diagram

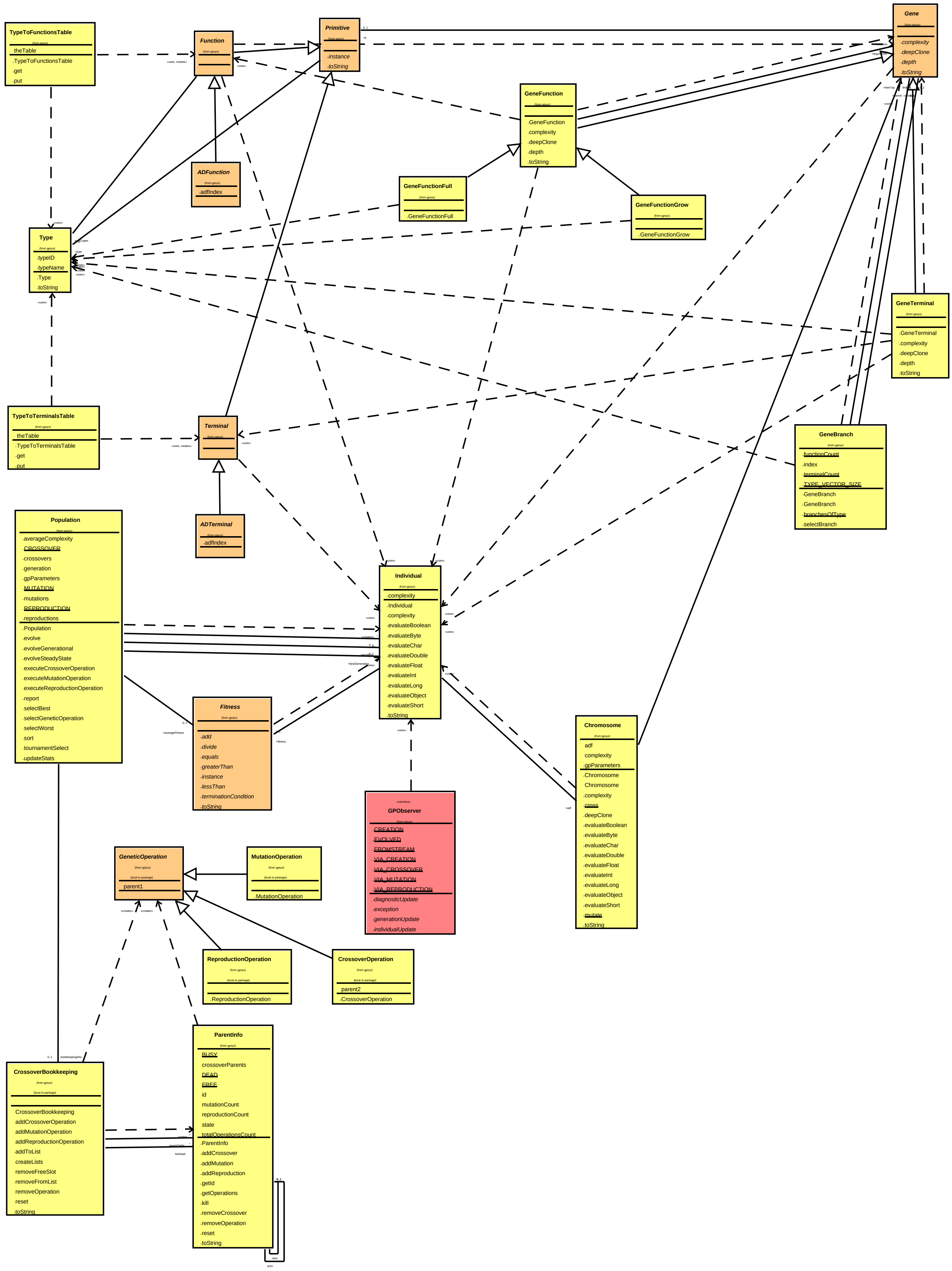


Figure A.3: GPsys overall UML Diagram

Bibliography

[AK95] David Andre and John R. Koza. Parallel genetic programming on a network of transputers. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 111–120, Tahoe City, California, USA, 9 July 1995.

[AK96a] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.

[AK96b] David Andre and John R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1163–1174, Sunnyvale, 9-11 August 1996. CSREA.

[And94] David Andre. Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 250–255, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Ang98] Peter J. Angeline. Subtree crossover causes bloat. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 745–752, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[AP93] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[AT99] D. Andre and A. Teller. Evolving Team Darwin United. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of *LNCS*, pages 346–351, Paris, France, July 1998 1999. Springer Verlag.

[Bei95] Boris Beizer. *Blackbox Testing Techniques for Functional Testing of Software & Systems*, volume 1. John Wiley & Sons, 1995.

[Ben96a] Forrest H Bennett III. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 30–38, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Ben96b] Forrest H Bennett III. Emergence of a multi-agent architecture and new tactics for the ant colony foraging problem using genetic programming. In Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, pages 430–439, Cape Code, USA, 9-13 September 1996. MIT Press.

[BJD86] M. Benda, V. Jagannathan, and R. Dodhiawala. On optimal cooperation of knowledge sources - an empirical investigation. Technical Report BCS–G2010–28, Boeing Advanced Technology Center, Boeing Computing Services, Seattle, Washington, July 1986.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998.

[Bra87] M. E. Bratman. *Intentions, Plans and Practical Reasoning*. Harward University Press, Cambridge, MA, USA, 1987.

[Bra95] Scott Brave. Using genetic programming to evolve recursive programs for tree search. In S. Louis, editor, *Fourth Golden West Conference on Intelligent Systems*, pages 60–65. International Society for Computers and their Applications - ISCA, 12-14 June 1995.

[Bra96] Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, April 1986.

[Bro91] Rodney A. Brooks. Intelligence without reason. In Ray Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, August 1991. Morgan Kaufmann.

[CC96] Cristiano Castelfranchi and R. Conte. DAI and social science: Critical issues. In Greg O'Hare and Nick Jennings, editors, *Foundations of Distributed Artificial Intelligence*, chapter 20. John Wiley and Sons, 1996.

[CF99] K Chellapilla and DB Fogel. Co-evolving checkers playing programs using only win, lose, or draw. In K. Priddy, S. Keller, D.B. Fogel, and J.C. Bezdek, editors, *AeroSense99, Symposium on Applications and Science of Computational Intelligence II*, pages 303–312, SPIE, Bellingham, WA, 1999.

[CFLY97] Chris Clack, Jonny Farringdon, Peter Lidwell, and Tina Yu. Autonomous document classification for business. In W. Lewis Johnson, editor, *The First International Conference on Autonomous Agents (Agents '97)*, pages 201–208, Marina del Rey, California, USA, February 5-8 1997. ACM Press.

[Cho98] Fuey Sian Chong. A java based distributed approach to genetic programming on the internet. Master's thesis, Computer Science, University of Birmingham, 1998.

[Cho99a] Fuey Sian Chong. Java based distributed genetic programming on the internet. Technical Report CSRP-99-7, University of Birmingham, School of Computer Science, April 1999.

[Cho99b] Fuey Sian Chong. Java based distributed genetic programming on the internet. In Erick Cantu-Paz and Bill Punch, editors, *Evolutionary computation and parallel processing*, Orlando, Florida, USA, 13 July 1999.

[CL99] Fuey Sian Chong and W. B. Langdon. Java based distributed genetic programming on the internet. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1229, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Cla97] Chris Clack. Software – the next generation: Evolving document classification. white paper, UCL, Andersen Consulting, University College London, Gower Street, London, April 1997.

[Cra85] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[Dav91] Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[Daw86] Richard Dawkins. *The blind Watchmaker*. Harlow : Longman Scientific and Technical, 1986.

[DB94] Patrik D'haeseleer and Jason Bluming. Effects of locality in individual and population evolution. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 8, pages 177–198. MIT Press, 1994.

[FTPS00] F. Fernandez, M. Tomassini, W. F. Punch III, and J. M. Sanchez. Experimental study of multipopulation parallel genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 283–293, Edinburgh, 15-16 April 2000. Springer-Verlag.

[Gas91] Les Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47(1-3):107–138, 1991.

[Gat98] Chris Gathercole. *An Investigation of Supervised Learning in Genetic Programming*. PhD thesis, University of Edinburgh, 1998.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, January 15, 1995.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison Wesley, 1989.

[GR94] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag.

[GR97a] Chris Gathercole and Peter Ross. Small populations over many generations can beat large populations over few generations in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 111–118, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[GR97b] Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[GRHL89] Les Gasser, Nicolas Rouquette, Randall W. Hill, and John Lieb. Representing and using organizational knowledge in DAI systems. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence*, volume 2 of *Research Notes in Artificial Intelligence*, pages 55–78. Pitman, 1989.

[Han91] S. Handley. The automatic generation of plans for a mobile robot via genetic programming with automatically defined functions. In *Proceedings of the Fifth Workshop on Neural Networks: An International Conference on Computational Intelligence: Neural Networks, Fuzzy Systems, Evolutionary Programming, and Virtual Reality*, 1991.

[Han93] S. Handley. The genetic planner: The automatic generation of plans for a mobile robot via genetic programming. In *Proceedings of the Eighth IEEE International Symposium on Intelligent Control*, 1993.

[Han94a] S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Han94b] Simon G. Handley. The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 18, pages 391–407. MIT Press, 1994.

[HB96a] Christopher Harris and Bernard Buxton. GP-COM: A distributed, component-based genetic programming system in C++. Research Note RN/96/2, UCL, Gower Street, London, WC1E 6BT, UK, January 1996.

[HB96b] Christopher Harris and Bernard Buxton. GP-COM: A distributed, component-based genetic programming system in C++. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 425, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Hil92] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 313–324. Addison-Wesley, Santa Fe Institute, New Mexico, USA, February 1990 1992.

[HLS96] Thomas Haynes, Kit Lau, and Sandip Sen. Learning cases to compliment rules for conflict resolution in multiagent systems. In Sandip Sen, editor, *Working Notes for the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems*, pages 51–56, Stanford University, CA, March 1996.

[Hol73] John H Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computation*, 2:88–105, 1973.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992. First Published by University of Michigan Press 1975.

[HR78] J. Holland and J. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems*, pages 313–329. New York: Academic Press, 1978.

[HS95a] Thomas Haynes and Sandip Sen. Evolving behavioral strategies in predators and prey. In Gerhard Weiß and Sandip Sen, editors, *Adaptation and Learning in Multiagent Systems*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Germany, 1995.

[HS95b] Thomas Haynes and Sandip Sen. Evolving behavioral strategies in predators and prey. In Sandip Sen, editor, *IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems*, pages 32–37, Montreal, Quebec, Canada, 20-25 August 1995. Morgan Kaufmann.

[HS96a] Thomas Haynes and Sandip Sen. Cooperation of the fittest. Technical Report UTULSA-MCS-96-09, The University of Tulsa, April 12, 1996.

[HS96b] Thomas Haynes and Sandip Sen. Cooperation of the fittest. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 47–55, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[HS96c] Thomas Haynes and Sandip Sen. Evolving behavioral strategies in predators and prey. In Gerhard Weiß and Sandip Sen, editors, *Adaptation and Learning in Multi–Agent Systems*, Lecture Notes in Artificial Intelligence, pages 113–126. Springer Verlag, Berlin, Germany, 1996.

[HS96d] Thomas Haynes and Sandip Sen. Learning cases to resolve conflicts and improve group behavior. In Milind Tambe and Piotr Gmytrasiewicz, editors, *Working Notes of the AAAI-96 Workshop on Agent Modeling*, pages 46–52, Portland, OR, August 1996.

[HS97] Thomas Haynes and Sandip Sen. Crossover operators for evolving A team. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 162–167, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[HSSW95a] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright. Evolving a team. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 23–30, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[HSSW95b] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright. Evolving multiagent coordination strategies with genetic programming. Technical Report UTULSA-MCS-95-04, The University of Tulsa, May 31, 1995.

[HW95] Thomas D. Haynes and Roger L. Wainwright. A simulation of adaptive agents in hostile environment. In K. M. George, Janice H. Carroll, Ed Deaton, Dave Oppenheim, and Jim Hightower, editors, *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 318–323, Nashville, USA, 1995. ACM Press.

[HWS94] Thomas Haynes, Roger Wainwright, and Sandip Sen. Evolving cooperation strategies. Technical Report UTULSA-MCS-94-10, The University of Tulsa, Tulsa, OK, USA, 16 December 1994.

[HWS95] Thomas D. Haynes, Roger L. Wainwright, and Sandip Sen. Evolving cooperating strategies. In Victor Lesser, editor, *Proceedings of the first International Conference on Multiple Agent Systems*, page 450, San Francisco, USA, 12–14 June 1995. AAAI Press/MIT Press. Poster.

[HWSS95] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 271–278, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Iba96] Hitoshi Iba. Emergent cooperation for multiple agents using genetic programming. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 32–41, Berlin, Germany, 22-26 September 1996. Springer Verlag.

[Iba98] Hitoshi Iba. Evolutionary learning of communicating agents. *Journal of Information Sciences*, 108(1-4):181–205, 1998.

[Iba99] Hitoshi Iba. Evolving multiple agents by genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 19, pages 447–466. MIT Press, Cambridge, MA, USA, June 1999.

[IKSS99] Forrest H Bennett III, John R. Koza, James Shipman, and Oscar Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1484–1490, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[INU97] Hitoshi Iba, Tishihide Nozoe, and Kanji Ueda. Evolving communicating agents based on genetic programming. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, Indianapolis, 13-16 April 1997. IEEE Press.

[Its95] Noda Itsuki. Soccer server: a simulator for robocup. In *JSAI AI-Symposium: Special Session on RoboCup*, December 1995.

[Jan94] Jan Jannink. Cracking and co-evolving randomizers. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 20, pages 425–443. MIT Press, 1994.

[Jon95] T. C. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, New Mexico, 1995.

[JSW98] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[KA95] John R. Koza and David Andre. Parallel genetic programming on a network of transputers. Technical Report CS-TR-95-1542, Stanford University, Department of Computer Science, January 1995.

[KDBK99] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.

[Kin94] Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Kor92] Richard E. Korf. A simple solution to pursuit games. In *Working Papers of the 11th International Workshop on Distributed Artificial Intelligence*, pages 183–194, February 1992.

[Koz91] John R. Koza. Genetic evolution and co-evolution of computer programs. In Christopher Taylor Charles Langton, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 603–629. Addison-Wesley, Santa Fe Institute, New Mexico, USA, February 1990 1991.

[Koz92a] J. R. Koza. Evolution of subsumption using genetic programming. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life. Towards a Practice of Autonomous Systems*, pages 110–119, Paris, France, 11-13 December 1992. MIT Press.

[Koz92b] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[Koz93] John R. Koza. Simultaneous discovery of reusable detectors and subroutines using genetic programming. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 295–302, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Koz94a] John R. Koza. Evolution of emergent cooperative behavior using genetic programming. In Ray Paton, editor, *Computing with Biological Metaphors*, pages 280–297. London: Chapman and Hall, 1994.

[Koz94b] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[Lan95] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Lan96a] W. B. Langdon. Using data structures within genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 141–148, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Lan96b] William B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[Lan97a] W. B. Langdon. Fitness causes bloat in variable size representations. Technical Report CSRP-97-14, University of Birmingham, School of Computer Science, 14 May 1997. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97.

[Lan97b] W. B. Langdon. Fitness causes bloat: Simulated annealing, hill climbing and populations. Technical Report CSRP-97-22, University of Birmingham, School of Computer Science, 2 September 1997.

[Lan98] William B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[Lan99] W. B. Langdon. Scaling of program tree fitness spaces. *Evolutionary Computation*, 7(4):399–428, Winter 1999.

[Lan00] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming And Evolvable Machines*, 1(1/2):95–119, April 2000.

[LHF+97] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Coevolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[LP97a] W. B. Langdon and R. Poli. Fitness causes bloat. Technical Report CSRP-97-09, University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK, 24 February 1997.

[LP97b] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.

[LP97c] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In John Koza, editor, *Late Breaking Papers at the GP-97 Conference*, pages 132–140, Stanford, CA, USA, 13-16 July 1997. Stanford Bookstore.

[LQ95] William B. Langdon and Adil Qureshi. Genetic programming – computers using “natural selection” to generate programs. Research Note RN/95/76, University College London, Gower Street, London WC1E 6BT, UK, October 1995.

[LS96] Sean Luke and Lee Spector. Evolving teamwork and coordination with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Luk98] Sean Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Luk00] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 2000. Forthcomming.

[LV97] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, 1997.

[Mat90] M. Mataric. A distributed model for mobile robot environment-learning and navigation. Technical Report AITR-1228, MIT AI Lab, Cambridge, MA, 1990.

[MC93] Mauro Manela and J. A. Campbell. Designing good pursuit problems as testbeds for Distributed AI: a novel application of Genetic Algorithms. In *Fifth European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Neuchâtel, Switzerland, August 24-27 1993.

[Min63] Marvin Minsky. Steps toward artificial intelligence. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, USA, 1963.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Mon95] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Mye79] Glen J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[NBF99] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[Nor97] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[OO95] Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.

[PL97] Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[PL98] Riccardo Poli and William B Langdon. Sub-machine-code genetic programming. Technical Report CSRP-98-18, University of Birmingham, School of Computer Science, August 1998.

[Pol96a] R. Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. Technical report, University of Birmingham, UK, August 1996. Presented at 3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97.

[Pol96b] R. Poli. Some steps towards a form of parallel distributed genetic programming. In *The 1st Online Workshop on Soft Computing (WSC1)*, http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/, 19–30 August 1996. Nagoya University, Japan.

[Pol96c] Riccardo Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. Technical Report CSRP-96-14, University of Birmingham, School of Computer Science, August 1996.

[Pol97a] Riccardo Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. In *3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97*, University of East Anglia, Norwich, UK, 1997.

[Pol97b] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[Pol97c] Riccardo Poli. Evolution of recursive transistion networks for natural language recognition with parallel distributed genetic programming. In David Corne and Jonathan L. Shapiro, editors, *Evolutionary Computing*, volume 1305 of *Lecture Notes in Computer Science*, pages 163–177, Manchester, UK, 11-13 April 1997. Springer-Verlag.

[Pol00] Riccardo Poli. Exact schema theorem and effective fitness for GP with one-point crossover. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 469–476, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[PP00] Riccardo Poli and Jonathan Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming And Evolvable Machines*, 1(1/2):37–56, April 2000.

[PPL99] Riccardo Poli, Jonathan Page, and William B. Langdon. Solving even-12, -13, -15, -17, -20 and -22 boolean parity problems using sub-machine code GP with smooth uniform crossover, smooth point mutation and demes. Technical Report CSRP-99-2, University of Birmingham, School of Computer Science, January 1999.

[Pun98] William F. Punch. How effective are multiple populations in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 308–313, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Qur96] Adil Qureshi. Evolving agents. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 369–374, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[RB99] Justinian P. Rosca and Dana H. Ballard. Rooted-tree schemata in genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 11, pages 243–271. MIT Press, Cambridge, MA, USA, June 1999.

[Rey92] Craig W. Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In Meyer and Wilson, editors, *From Animals to Animats (Proceedings of Simulation of Adaptive Behaviour)*. MIT Press, 1992.

[Rey94a] Craig W. Reynolds. Competition, coevolution and the game of tag. In Rodney A. Brooks and Pattie Maes, editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 59–69, MIT, Cambridge, MA, USA, 6-8 July 1994. MIT Press.

[Rey94b] Craig W. Reynolds. Evolution of obstacle avoidance behaviour:using noise to promote robust solutions. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 10, pages 221–241. MIT Press, 1994.

[Rey94c] Craig W. Reynolds. An evolved, vision-based behavioral model of obstacle avoidance behaviour. In Christopher G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 327–346. Addison-Wesley, Santa Fe Institute, New Mexico, USA, 15-19 June 1992 1994.

[RK90] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, 1990.

[Ros97] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Sap94] Jan Sapp. *Evolution by Association: A History of Symbiosis*. Oxford University Press, 1994.

[SDB92] J. Sichman, Y. Demazeau, and O. Boissier. When can knowledge-based systems be called agents. In *Proc. of IX Brazilian Symposium on Artificial Intelligence*, Rio de Janeiro, Brazil, 1992.

[SM90] Larry M. Stephens and Matthias B. Merx. The effect of agent control strategy on the performance of a DAI pursuit problem. In *Proceedings of the 1990 Distributed AI Workshop*, October 1990.

[Smi80] S. F. Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, Pittsburgh PA, 1980.

[SV97] P. Stone and M. Veloso. Multiagent systems: A survey from a machine learning perspective. Report Series no CMU-CS-97-193, Carnegie Mellon University, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1997.

[SV00] C. R. Stephens and J. Mora Vargas. Effective fitness as an alternative paradigm for evolutionary computation 1: General formalism. *Genetic Programming and Evolvable Machines*, 1(1):363–378, 2000.

[TA97] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[TC94] Walter Alden Tackett and Aviram Carmi. The donut problem: Scalability and generalization in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 7, pages 143–176. MIT Press, 1994.

[Tel94] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[Tel96] Astro Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 3, pages 45–68. MIT Press, Cambridge, MA, USA, 1996.

[TV95] Astro Teller and Manuela Veloso. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.

[Wei99] Gerhard Weiss, editor. *Multiagent Systems A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[WJK00] M. Wooldridge, N. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *International Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[WL96] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[YC98] Tina Yu and Chris Clack. Recursion, lambda abstractions and genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 422–431, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.