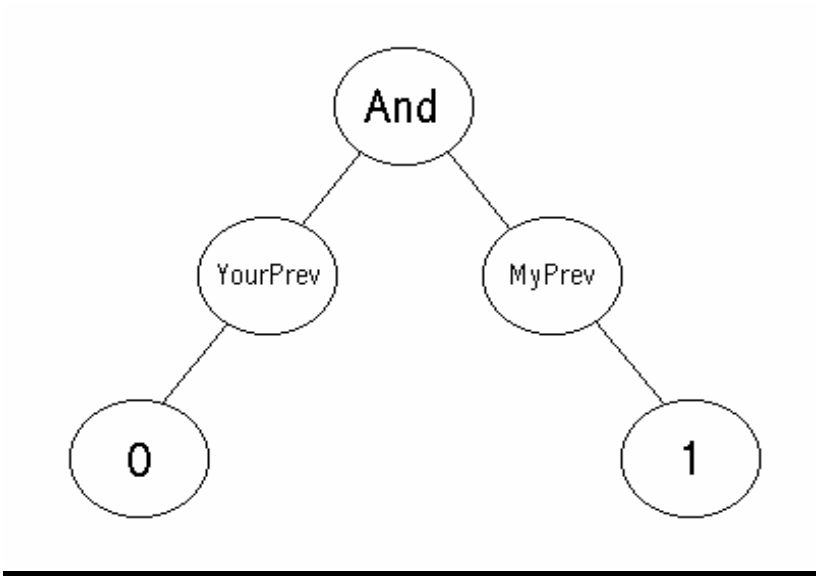
Using Genetic Programming to evolve strategies for the Iterated Prisoner's Dilemma



by Robert De Caux

Supervised by Robin Hirsch

September 2001

This report is submitted as part requirement for the MSc **Degree in** Computer Science **at University College London. It is substantially the result of my own work except where explicitly indicated in the text.**

The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.

Using Genetic Programming to Evolve Strategies for the Iterated Prisoner's Dilemma

by Robert De Caux

Supervised by Robin Hirsch

September 2001

Abstract

The technique of Genetic Programming (GP) uses Darwinian principles of natural selection to evolve simple programs with the aim of finding better or *fitter* solutions to a problem. Based on the technique of Genetic Algorithms (GA), a population of potential solutions stored in tree form are evaluated against a fitness function. The fittest ones are then modified by a genetic operation, and used to form the next generation. This process is repeated until certain criteria have been met. This could be an ultimate solution, or a certain number of generations having been evolved.

Genetic Programming is a fast developing field with potential uses in medicine, finance and artificial intelligence. This project attempts to use the technique to evolve strategies for the game of Prisoner's Dilemma. Although a simple game, the range of possible strategies when the game is iterated is vast, but what makes it particularly interesting is the absence of an ultimate strategy and the possibility of mutual benefit by cooperation.

A system was created to allow strategies to be evolved by either playing against fixed opponents or against each other (coevolution). The strategies are stored as trees, with GP used to form the next generation. The main advantage of GP over GA is that the trees do not need to be of a fixed size, so strategies can be developed which utilise the entire game history as opposed to just the last few moves.

This implementation has advantages over previous investigations, as information about which go is being played can be used, thus allowing cleverer strategies. Work has also been conducted into a hunting phase, where strategies roam a two dimensional grid to find a suitable opponent. By studying the history of potential opponents and using GA, evidence emerged of an increase in cooperative behaviour as strategies sought out suitable opponents, demonstrating parallels with biological models of population dynamics.

The system has been developed to allow a user to alter important parameters, select the evolution method and seed the population with pre-defined strategies by means of a graphical user interface.

Table of Contents

1 Introduction....1

1.1 Aims....1

1.2 Specific objectives....1

1.3 Two player games and strategies....1

1.4 Choice of Game....2

1.5 Why evolve strategies?....2

1.6 Report outline....3

2 Background....4

2.1 Game Theory....4

2.1.1 Background....4

2.1.2 Two-person zero sum games....5

2.1.3 Two-person non-zero sum non-cooperative games....6

2.2 Population Dynamics....7

2.2.1 The Decline of the Rational Player....7

2.2.2 Answers from Nature....7

2.2.3 Evolutionary Stability....7

2.2.4 Nash Equilibria....8

2.2.5 Evolutionary Stable Strategies....8

2.3 Search Techniques....9

2.3.1 Overview....9

2.3.2 Genetic Algorithms....9

2.3.3 Genetic Programming....10

2.4 Details of Genetic Programming....11

2.4.1 Steps to solving a problem with GP....11

2.4.1.1 Terminals and Functions....11

2.4.1.2 Fitness Function....11

2.4.1.3 Control Parameters....11

2.4.1.4 Termination criteria....12

2.4.2 Genetic Operations....12

2.4.3 Fitness Evaluation....13

2.4.3.1 Absolute and Relative Fitness....13

2.4.3.2 Pareto scoring....13

2.4.4 Selection and Breeding....14

2.4.4.1 The Problem of Premature Convergence....14

2.4.4.2 Probabilistic selection....14

2.4.4.3 Tournament Selection....14

2.4.4.4 Steady state and Generational GP....14

2.4.4.5 Demes....14

2.4.5 Strongly Typed Genetic Programs....15

2.5 Prisoner's Dilemma....15

2.5.1 Basic Game Description....15

2.5.2 The Payoff Matrix....15

2.5.3 An Evolutionary Stable Strategy....16

2.5.4 The Iterated Prisoner's Dilemma 16
2.5.5 Simple Strategies for IPD 17
2.5.6 Lack of an Evolutionary Stable Strategy 17
2.5.7 Stochastic Strategies 18
2.5.8 Spatialised Iterated Prisoner's Dilemma 19

3 Analysis 20

3.1 Requirements 20
3.1.1 Functional requiremnts 20
3.1.2 Non-functional requirements 22

3.2 Use Case Diagrams 22
3.2.1 GP engine 23
3.2.2 Whole system 23

4 Design 24

4.1 Choice of Programming Language 24

4.2 Choice of GP engine 24

4.3 Engine details 25
4.3.1 Important classes 25
4.3.2 Realisation of use cases 25
4.3.3 Classes to be extended or implemented 25

4.4 Design of the GP 26
4.4.1 Requirements 26
4.4.2 Choice of Terminals and Functions 26
4.4.3 Examples of Common Strategies as Trees 27
4.4.4 Choice of Fitness function 28
4.4.5 Control Parameters 28
4.4.5.1 Size of Population 28
4.4.5.2 Number of generations 28
4.4.5.3 Tournament Size 29
4.4.5.4 Probability of mutation 29
4.4.5.5 Probability of reproduction 29
4.4.5.6 Chromosome size 29
4.4.5.7 Evolution type 29
4.4.5.8 Other parameters 30
4.4.6 Termination criteria 30

4.5 Design of Computerised Iterated Prisoner's Dilemma 30
4.5.1 Requirements and package gpsys.prisoner 30
4.5.2 Specific classes for playing the game 30
4.5.3 Hunting for an opponent 31

4.6 Design of Hunting Mechanism 31
4.6.1 Requirements and package gpsys.grid 31
4.6.2. Criteria for finding an opponent 31
4.6.3 An algorithm for deciding on an opponent 32
4.6.4 Designing classes 32
4.6.5 Creating and inheriting huntCriteria 33

4.7 Design of the User Interface 34

4.7.1 Editable options for the user 34
4.7.2. Prevention of inconsistent parameters 34
4.7.3 Displaying Information 34
4.7.4 Graphing results 34
4.8 Incorporating Coevolution 34
4.8.1 Standard Coevolution 34
4.8.2 Hunting Coevolution 35
4.8.3 Seeding 35
4.8.4 Fitness evaluation 35
4.9 Extending the engine 36
4.9.1 Additions/changes to original package gpsys 36
4.9.1.1 New classes 36
4.9.1.2 New/edited constructors 36
4.9.1.3 Other changes 36
4.9.2 Extensions to engine 36
4.10 Final Class Diagrams 37
5 Implementation 38
5.1 Problems encountered 38
5.1.1 Lack of diversity 38
5.1.2 Hunting bias 38
5.1.3 Error in Crossover type checking 39
5.1.4 Random numbers 39
5.2 Other program changes 39
5.2.1 Speeding up hunting 39
5.2.2 Adding Dummy function 39
5.3 Examining important methods 40
5.3.1 Editing parameters 40
5.3.2 Implementing the hunt 40
5.3.3 Performing evolution 41
5.3.5 Playing a game between two Individuals 42
5.3.6 Sequence Diagrams 43
6 Testing 44
6.1 Testing Interface 44
6.1.1 Testing illegal selections 44
6.1.2 Testing accept button 44
6.2 Testing Hunting 44
6.3 Testing Crossover 44
6.4 Testing Mutation and Reproduction 45
6.5 Testing Chromosome Functions 45
6.6 Testing seeding 45
6.7 Testing game 45
7 Results 46

7.1 Playing fixed opponent(s) 46
7.1.1 Vs. AllC Player 46
7.1.2 Vs. AllD Player 46
7.1.3 Vs. Tit-For-Tat Player 47
7.1.4 Vs. Tit-For-2-Tat Player 47
7.1.5 Vs. Cooperative and Tit-For-Tat Players 48
7.1.6 Vs. Cooperative, Backstabbing and Tit-For-Tat Players 49
7.1.7 Vs. All players 50
7.2 Standard Coevolution 51
7.2.1 Without functions Go, EQ and If 51
7.2.2 With all functions except Go 52
7.2.3 With all functions 52
7.3 Hunting coevolution 54
7.3.1 Setting the threshold for playability 54
7.3.2 Inducement of cooperation 55
7.4 Results summary 57
8 Conclusions and Evaluation 58
8.1 Conclusions 58
8.2 Success of system 59
8.3 Extending the project 60
A User Manual 61
A.1 Running the Program 61
A.2 The User Interface 61
A.3 Changing the options 62
A.3.1 GPParameters 62
A.3.2 Chromosome Parameters 62
A.3.3 Game Parameters 62
A.3.4 Hunt Parameters 62
A.3.5 Coevolution/Opponent Setup 63
A.3.6 Seed Population 63
A.4 Saving the Population and Generational Reports 63
A.5 Displaying results of Evolution 64
A.6 Displaying the Hunt 64
B System Manual 65
B.1 System requirements 65
B.2 Making changes 65
B.2.1 Extracting the Code and Documentation 65
B.2.2 Javadoc 65
B.2.3 Compiling and Running the System 66
B.3 Extending or Adapting System 67

C Closer examination of GP engine 68
C.1 Storing GP Parameters 68
C.2 Population structure 68
C.3 Evaluation of an Individual 69
C.4 Creating new Population 70
C.5 Evolving Population 71
C.6 Performing the Genetic Operations 71
C.6.1 Reproduction: 71
C.6.2 Mutation: 71
C.6.3 Crossover: 72
C.7 Giving feedback to user 72
C.8 Establishing Fitness 72
C.9 Class diagram 73

D Class and Sequence Diagrams 74
D.1 Class diagrams 74
D.1.1 Package gpsys.prisoner 74
D.1.2 Interface between packages in setting up coevolution 75
D.1.3 Packages gpsys.primitives and gpsys.primitives.prisoner 75
D.1.4 Package gpsys.grid 76
D.1.5 Package gpsys.prisoner.gui 77
D.1.6 Package gpsys after extensions 78
D.1.7 Package structure 78
D.2 Sequence Diagrams 79
D.2.1 Evaluating an Individual 79
D.2.2 Creating a new Population 79
D.2.3 Creating a new Hunt 80

E Source Code 81

F Evolution Logs 106
F.1 Testing genetic operations 106
F.1.1 Crossover 106
F.1.2 Mutation and Reproduction 106
F.2 Standard Coevolution 106
F.3 Hunting Coevolution 107
F.4 Function Set test 108
F.5 Seeding Population 109

G Bibliography – Books 111

H Bibliography – Web links 112

1 Introduction

1.1 Aims

The main aim of this project is to establish the strongest and most robust strategies for playing the Iterated Prisoner's Dilemma, and to determine what makes them effective.

As there is no definitive way to maximise the score against every type of opponent, the strategies being sought are ones which perform best against a variety of other strategies, and which maintain their place in an elitist population. These strategies should maintain a high average score over a period of time. Investigations will also be conducted into the best ways for playing one or a selection of fixed opponents, bearing in mind that the strategies discovered may be opportunistic, rigid and exploitable by other strategies, but ideally suited to this situation. The strategies will have access to which go is currently being played, which is not the case for previous investigations combining Priosner's Dilemma and Genetic Programming, so this will hopefully yield some new results.

An additional aim is to study other means of strategies surviving in a competitive environment, such as the effects of hunting for an opponent before a game takes place, based on criteria established in previous games. This is a relatively unexplored area of research.

1.2 Specific objectives

- To use the technique of genetic programming to discover which strategies are best against given fixed opponents, and which can maintain a place within a population (coevolution)
- To establish how average and extremal scores will vary over time
- To experiment with different evolutionary parameters
- To develop a mechanism for strategies to hunt for suitable opponents using a genetic algorithm
- To discover whether this hunting can help a strategy to survive within a population
- To develop a graphical user interface to implement all alterations to parameters which the user may wish to change

1.3 Two player games and strategies

Most two player games have some strategy available to players which will enable them to improve their performance. In some cases, a definitive strategy can be found which a player can use to guarantee victory. Games such as tic-tac-toe and connect4 fall into this category, as one player can dictate the play so that their opponent has only limited options available on each go, none of which will allow them to win. Of course there are some games in this category, Chess being an example, in which the number of permutations are enormous, but there is still a "best" way of playing. Such games are called two player zero sum games (§2.1.2).

A more interesting type of game from the point of view of studying strategies is one where a good move for one player is not necessarily a bad move for the other, thus incorporating both competition and cooperation. Now there is no ultimate strategy which works best against every type of player, as the effectiveness of any strategy is determined by the decision of the opponent. The poker inspired quote that "you cannot bluff a bad player" highlights this, as a clever bluff against an experienced player is an unnecessary waste of money against an amateur with a good hand[1]. This type of game is known as two player non-zero sum. (§2.1.3).

1.4 Choice of Game

The game of Iterated Prisoner's Dilemma was chosen as it has simple rules and the choices available to a player on any go are very limited, so strategies are relatively easy to understand. It does offer a surprising degree of complexity however, with a plethora of different strategies available, and most importantly it does not have a solution or "best" way of playing against all opponents.

1.5 Why evolve strategies?

The benefit of using evolutionary techniques to evolve the strategies is that the trial and error approach of evolution can produce effective strategies which may learn and adapt to

[1] A bluff in Poker is a deliberately high bet placed to scare the opponent into thinking that a player's hand is stronger than is actually the case. An inexperienced player will only pay attention to the quality of his own hand and not to how their opponent behaves, so will continue to play undaunted. An experienced player may decide that their hand is not sufficiently good enough to play for such high stakes, and concede the money for that hand.

counteract the possible irrationality of opponents. There is little point in approaching the problem from a purely rational viewpoint when the game in question means that a rational[2] player may not score as well as an irrational player. The Darwinian quote "No instinct has been produced for the exclusive good of other animals, but each animal takes advantage of the instincts of others" suggests that the strategies that will evolve are the ones that can compete best within their environment, even if they may not seem to be the most rational. This is exactly what is required.

1.6 Report outline

- *Background:* Details the fundamentals of game theory and population dynamics, before discussing various search techniques for finding solutions, with a heavy emphasis on Genetic Programming (GP). The game of Prisoner's Dilemma is then discussed in depth, including previous research.

- *Analysis:* Details the requirements for the system being built and for the GP engine to be used.

- *Design:* Discusses the GP engine chosen, and how it can be extended to incorporate the game of Prisoner's Dilemma, with emphasis on setting up the GP. Also details design decisions for the Graphical User Interface (GUI), hunting mechanism and coevolution.

- *Implementation:* Discusses problems encountered whilst testing the system and changes that were made. Also details how the use cases are realised.

- *Testing:* Shows tests carried out to verify all of options implemented

- *Results:* Discusses the strategies that have evolved for playing fixed opponents, during coevolution and with the inclusion of hunting, by examination of graphs over a number of generations and generational reports.

- *Conclusions & Evaluation:* Discusses what can be drawn from the results, and how this relates to previous work and to nature, as well as how successful the system is and how it can be extended.

[2] A rational player is discussed in §2.1.1

2 Background

2.1 Game Theory

2.1.1 Background

Game theory can be thought of as a mathematical approach to the study of conflict of interest, and is generally attributed to von Neumann [7]

It describes a game as a set of situations with well specified outcomes, such that were they offered to a player, it could be ascertained which choice he would make. The decision making of players underpins the whole theory, and a player is termed *rational* if they make a decision which will maximise the benefit to themselves.

A game is said to be *normalised* if it satisfies the following:

n players are required to make one choice from a set of choices, without knowledge of what the other players are doing. These choices lead to a resulting outcome for each player. Both players try to maximise the outcome for themselves

The two player version of a normalised game can be characterised by a matrix, termed the *payoff matrix*, where the choices of both players give as the outcome the corresponding matrix entry. As an example, suppose that a goalkeeper is facing a penalty from a striker, and both can choose to go either left or right. If they both choose the same direction, the penalty is saved and the score stays the same (both score 0). If they go in different directions however, the ball goes into the net and the striker scores 1, while the keeper scores –1. The payoff matrix for this situation is given below:

	Goalkeeper		
Striker		Left	Right
	Left	0, 0	1, -1
	Right	1, -1	0, 0

The entries indicate what each player scores. The number of the left is for the striker, the other for the goalkeeper.

A *pure strategy* can be thought of as a pre-conceived method for dealing with every eventuality that may result throughout he game. If both players in a two player game have a pure strategy, this dictates what they would do in every situation, so the game could be played to a conclusion by some umpire with no further input from the players. These are also sometimes referred to as *deterministic* strategies. An extreme example would be two players playing connect4, with one always placing their pieces in the leftmost available column, and the other placing theirs in the rightmost available column.

2.1.2 Two-person zero sum games

Suppose a game is in normalised form, and there is a payoff matrix defined. Each player will have an order of preference for the set of outcomes available. Now if both players have the same preferences, then there is no conflict of interest and the game becomes trivial.

At the opposite extreme, if for all outcomes there is either mutual indifference or one player prefers one and the second player the other, then it is *strictly competitive* and referred to as a zero sum game [3].

If both players are to choose one of their options without knowledge of the other player's choice, they will have a problem deciding, as in all likelihood the best choice that they can make is dependent on the choice of their opponent. To escape from this potentially circular argument, each player should seek to maximise their own *security level*, which is the worst they can do by making any particular choice. If both players choose a strategy to do that, they are effectively making the best choice to counter their opponent, i.e. if one player reveals their choice first that maximises their own security level, the other player can do no better than to choose to maximise their own security level[4]. Readers wishing a more detailed explanation of the above should refer to [8, chap.4], but it can be formalised quite simply:

Suppose player one chooses strategy a_1 from $a_0 \ldots a_n$, and player two chooses b_2 from $b_0 \ldots b_n$. Then if the best outcome $O_{i,2}$ for player 1 is $O_{1,2}$ and the best outcome $O_{1,j}$ for player two is $O_{1,2}$, i, $j \in 0..n$, each player has maximised their own security level, and the pair of choices

[3] Zero sum refers to the sum of the utilities for the two players, which is zero as the name suggests. Utility is discussed in [8, chap.2]

[4] One of the most important principles of game theory however is that it does not say what any player *should* do. Although this decision may give the player the best guaranteed performance, they may be able to improve on it with a different choice. Game theory simply states how to achieve certain outcomes from given situations.

(a_1, b_2) is an *equilibrium pair*, so called because neither player has incentive to change from their decision and so it is repeatedly chosen. These equilibrium pairs may not be unique or may not even exist for a particular matrix[5]

The *minimax* theorem, the central theorem of two-person zero-sum theory, states that there exists a number v, so that player one has a strategy (maximin) guaranteeing at least a return of v, and player two has a strategy (minimax) guaranteeing that player one can get at most v. These strategies are in equilibrium, and any pair of strategies in equilibrium will give maximin and minimax strategies for player one and player two respectively.

This result is basically an extension of the discussion above to take in *any* zero sum game (not just ones with equilibrium pairs), but stipulates that players must be allowed mixed strategies (where each strategy available to a player is given a probability of being chosen).

The solution to a zero sum game is the strategy derived from the minimax theorem, with v being termed the *value* of the game. This will maximise the *expected return* for that player which is the aim of the game.

2.1.3 Two-person non-zero sum non-cooperative games

A two-person non-zero sum game is such that if one player prefers choice x to y, then the other does not necessarily prefer y to x. This introduces the element of agreement, and allows benefit to be gained from cooperation. A non-cooperative game simply means that no communication is allowed before a decision is taken, known as *pre-play communication*.

Unlike zero sum games, there is no solution. This is because the fundamental rules of zero-sum games are violated. In particular, these rules no longer apply:

i) if (x,y) and (x', y') are equilibrium pairs, then (x, y') and (x', y) are also equilibrium pairs.

ii) if x is a maximin strategy and y a minimax strategy, (x,y) is an equilibrium pair.

Analysis of a simple non-zero sum game showing violation of these rules is given in [8, chap.5.3], and a very mathematical analysis of two player games is given in [9].

Now there is no minimax solution, as maximin and minimax strategies do not form an equilibrium. Instead there is always an incentive to "double cross" the opponent if it is felt

[5] Consider the matrix (3 1) – no entry which is min of its row and max of its column,
(2 4) so no equilibrium pair

they are going to stick to the safe strategy, but if both double cross they suffer, so there is a temptation to stick to playing safe, etc. A vicious circle emerges.

A definition was given by Nash for a non-zero sum game to be solvable [8, p106] if all pairs of equilibrium pairs are interchangeable[6], but this is little more than a theoretical solution with no practical use.

Strategies for non-zero sum games can be affected by the number of times the game is played. In a one off game for example, retaliation is not possible so players may feel more inclined to risk trying to maximise their return by double crossing.

2.2 Population Dynamics

2.2.1 The Decline of the Rational Player

The idea of a perfectly rational player providing the best solution was questioned by the so-called "trembling hand" line of reasoning, where by the opponent is thought to occasionally behave irrationally [6, preface xii]. This change in behaviour can force even the most robust rational strategy into trouble. Therefore some dynamical reasoning needed to be injected into the strategy, instead of being totally based on "a priori" reasoning. This of course may not lead to a stable solution.

2.2.2 Answers from Nature

John Maynard Smith, a biologist, drew a comparison between animal conflicts and games, by placing the game in a population-dynamical setting [11, chap.47 209-221]. Each member of the population would have a strategy for playing, and could randomly come across an opponent within the population who they would play against. In this setting the more stable strategies should take control as the strategies adapt to survive. This is similar to self regulation within an animal population – the predator/prey cycle.

2.2.3 Evolutionary Stability

[6] Pairs are interchangeable if for equilibrium pairs (x, y) and (x', y'), (x, y') and (x', y) are also in equilibrium

Behaviour is evolutionary stable if, when adopted by all members of a population, it cannot be invaded under natural selection. This is formalised by Hofbauer and Sigmund [6, chap.6 59-60]

Suppose that we represent the fitness (as discussed in §2.3.5) of an individual by W(I,Q) [7], where I represents the strategy they are adopting, and Q represents the composition of the population. A mixed population would be represented by $Q = x\,J + (1-x)\,I$, $0<x<1$, with x being the frequency of J type individuals, and (1-x) the frequency of I type individuals.

An I type population is evolutionary stable if upon the introduction of a small number of J type individuals, the I types fare better than the J types, i.e. for $J \neq I$,

$W(I, \varepsilon J+(1-\varepsilon)I) > W(J, \varepsilon J+(1-\varepsilon)I)$, $\varepsilon>0$ and small

Provided that W(I,Q) is continuous in Q, As $\varepsilon \rightarrow 0$, $W(I, I) \geq W(J, I)$

Therefore no types can fare better against a population of I types than the I type itself. The only problem is that a strategy I may not actually exist.

2.2.4 Nash Equilibria

A *Nash equilibrium* is a strategy which is a best reply to itself. Any normal form game will give at least one Nash equilibrium [6, chap 13.4]

A strategy **q** is a *strict Nash equilibrium* if it is the unique best reply to itself, i.e.

$\mathbf{q}.M\mathbf{q} > \mathbf{p}.M\mathbf{q}$, $\mathbf{p} \neq \mathbf{q}$, M is the payoff matrix.

2.2.5 Evolutionary Stable Strategies

A strategy p is an evolutionary stable strategy if it satisfies the following two conditions:

i) Equilibrium: $\mathbf{p}.M\mathbf{p} \geq \mathbf{q}.M\mathbf{p}$, $\forall \mathbf{q} \in S_N$, i.e. all possible strategies
ii) Stability: if $\mathbf{q} \neq \mathbf{p}$ and $\mathbf{q}.M\mathbf{p} = \mathbf{p}.M\mathbf{p}$, then $\mathbf{q}.M\mathbf{q} < \mathbf{p}.M\mathbf{q}$

[7] This means that W is a function of I and Q

Just having condition i), the Nash equilibrium, would not be sufficient, as there may be another strategy which is an alternative best reply, and may be able to invade. Clearly the strict Nash equilibrium is sufficient, as any individual not using that strategy will automatically do less well.

2.3 Search Techniques

2.3.1 Overview

The solution or strategy being sought can be thought of as a particular point within a search space of all possible solutions or strategies. There are three main types of search techniques which can be used to find the one desired. They are enumerative techniques (which in principle search every point one at a time, although this search can be restricted only to places that can contain the solution), Calculus based techniques (which treat the search space as a continuous function and search for maxima and minima), and Stochastic techniques (which use information from the search so far to choose the next point probabilistically). The latter techniques include evolutionary algorithms which are the basis for both genetic algorithms and genetic programming.

Evolutionary algorithms use Darwinian natural selection to breed progressively better children from a population by keeping the strong and discarding the weak [4]. The different types of evolutionary algorithm differ in the representation of individuals and the process of evolving new ones. One such technique is genetic algorithms.

2.3.2 Genetic Algorithms

Genetic algorithms (GA) were pioneered by John Holland [12], and take their inspiration from nature. A population of individuals are created, and each of these individuals has a known fitness which has been calculated in some way. This population is then evolved over a number of generations, with new individuals being bred from the fitter individuals in the previous generations. This will hopefully direct the evolution in the required direction, although of course it may take many generations to achieve the required result. Once this result has been reached we can terminate the GA, or if there is no definitive solution we can terminate the algorithm after a certain number of generations.
Each individual is made up of DNA, usually represented by a fixed length vector. Each entry in the vector represents some characteristic of the individual, and the entries are usually

binary. The fitness of the individual is then calculated using the entries in the vector. The DNA does not represent the program itself, but affects how the program will behave.

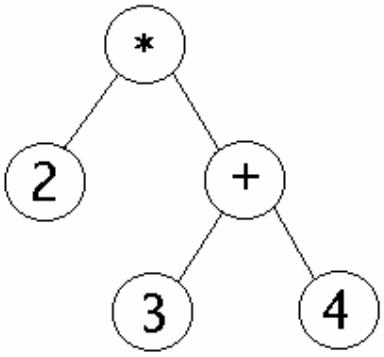
Genetic operations are then applied to the individuals, affecting their DNA entries, and so the vectors change and hopefully become better at solving the required problem. The use of building blocks (a small localised subset of vector entries which are successful) leads to the creation of successful individuals.

2.3.3 Genetic Programming

Genetic programming (GP) is an extension of GA credited to John Koza [13]. Where as vectors are used as the DNA in GA, GP uses a hierarchical representation, usually some form of tree, to store the genetic code. The two main benefits of this over GA is that the size and structure of the tree do not need to be specified in advance making it much more flexible, and also the trees represent the program code itself, rather than just influencing how the program performs. The individual **is** effectively the tree.

Of course there are now difficulties in making sure all trees are valid programs. In fact experiments which tried to evolve trees with no regard for syntax produced very few compilable programs and so were ineffective at approaching the solution [4, p11]. Therefore care must be taken to keep the programs valid when the genetic operations take place, and this is helped vastly by the tree structure, as all operations can be implemented on the branches.

A very simple example is the following tree, representing the program (2+3)*4:



An overview of GP can be found at [19]

2.4 Details of Genetic Programming

2.4.1 Steps to solving a problem with GP

John Koza identified five steps to solving a problem with GP [13]. They are the choice of:

1) Terminals
2) Functions
3) Fitness Function
4) Control Parameters
5) Termination Criteria

2.4.1.1 Terminals and Functions

The terminals and functions are the components of the programs. Terminals act as the leaves of the tree (for example the numbers in the examples above), and functions act as junctions, with their children being their arguments (for example '+' takes two numbers as arguments). Deciding on the function and terminal sets for the problem is a very important part of the design, as these essentially determine what the strategy is able to do. Neglecting to include certain functions or terminals will prevent some strategies from being available.

2.4.1.2 Fitness Function

The fitness function determines how the fitness of an individual is calculated, and therefore how successful it is. The function is designed to be specific to the problem, so this is another important design factor. It is sensible to encourage fitness to increase as solutions improve unless a definite target is known for the solutions to aim for, in which case the fitness should try and be as close to zero as possible, with zero being the perfect solution. Further details on the fitness function and fitness evaluation can be found in §2.4.3.

2.4.1.3 Control Parameters

These are the parameters that affect how the GP is run, such as the size of the population, number of generations, probability of each of the genetic operations, and maximum/minimum sizes for solutions.

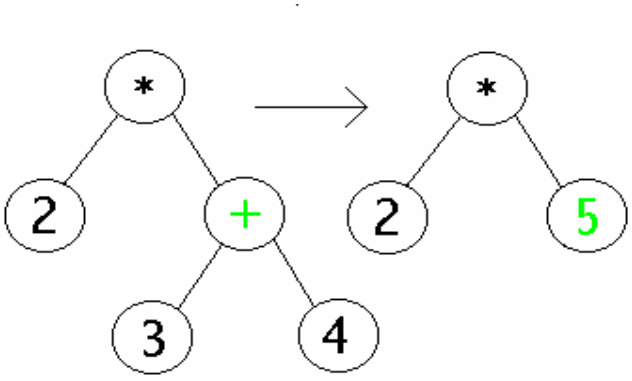
2.4.1.4 Termination criteria

If the termination criteria are met, the GP stops. The rule for stopping could be a perfect solution, or a certain number of generations having elapsed.

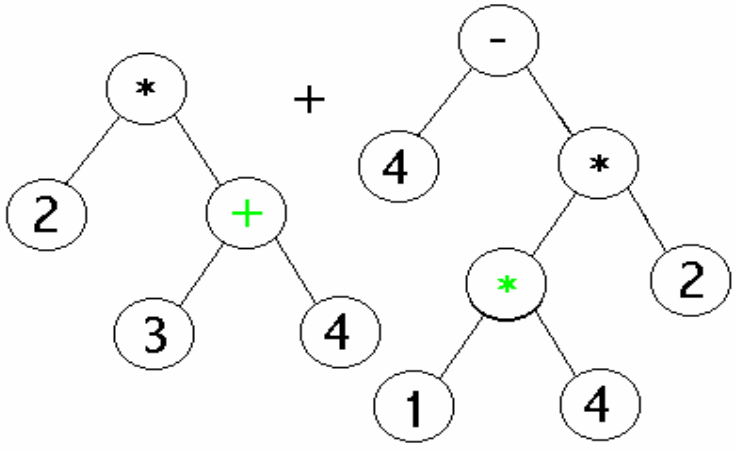
2.4.2 Genetic Operations

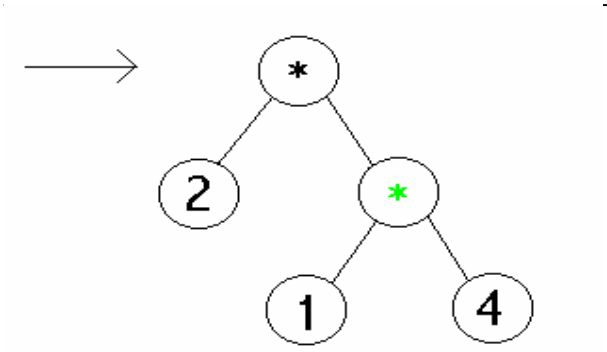
The three most common operations used on the individuals are reproduction, mutation and crossover:
Reproduction is the most straightforward, with the individual being placed unchanged into the next generation.

Mutation randomly changes one of the branches of the tree, then places the new program into the next generation, e.g.



Crossover takes in two parent trees, replaces a random branch of one of the parents with a random branch from the other, and places the new program into the next generation.





There are other operators [4, p26], which will be mentioned but not pursued.

- Hoist – A new individual is created entirely from a subtree of an existing individual
- Self-crossover – Similar to crossover, but the same individual represents both parents
- Context Preserving Crossover – Subtrees for crossover are only allowed if their node positions allow it

2.4.3 Fitness Evaluation

There are a number of issues concerned with the fitness function which are the subject of research. Some of the most relevant are detailed below.

2.4.3.1 Absolute and Relative Fitness

Most GP applications calculate the fitness of an individual by comparing it to a fixed number of test cases or by playing against fixed opponents. This is known as *absolute* fitness, and requires knowledge of an optimal strategy.

To avoid this problem, solutions can be coevolved to give a *relative* fitness. This is achieved by playing the individuals against each other, and so over a period of time, as the test cases are also evolving, hopefully the overall fitness of the individuals should improve. Several experiments have shown co-evolved players to give better and more robust solutions than those derived using absolute fitness [14].

2.4.3.2 Pareto scoring

Pareto scoring allows individuals to be scored on multiple criteria, such as functionality, complexity and efficiency. These could be used to give a single fitness score, or allow different scores to be used in different circumstances, e.g. if functionality is tied, choose individual with lower complexity.

2.4.4 Selection and Breeding

2.4.4.1 The Problem of Premature Convergence

Choosing which individuals are selected from a population and how their offspring are produced is a vital part of any GP. It is crucial that the decision prevents premature convergence, which occurs when a population loses diversity and converges to a non-optimal solution. Much research has been conducted into methods of maintaining diversity, such as Brood selection and Diassortative mating [4, p22-23]. Two of the most common selection methods are discussed below.

2.4.4.2 Probabilistic selection

Individuals are assigned a probability for being selected based on their fitness in relation to the rest of the population. This may be by ranking the individuals and then assigning the probabilities, or by making the probability the individual's fitness divided by the total fitness. This is the standard method for GA systems.

2.4.4.3 Tournament Selection

A small subset of the population is selected at random, and the individual with the best fitness is chosen. Careful consideration must be given to the tournament size – increasing it may stifle diversity, but making it too low introduces more noise.

2.4.4.4 Steady state and Generational GP

When new individuals are created, they can either replace weak individuals already in the population (steady state) or form a completely new generation (generational). Steady state GP is an elitist breeding policy, as individuals with high fitness can never be replaced. Therefore it may not be appropriate for coevolution where the test cases are constantly changing.

2.4.4.5 Demes

Instead of allowing individuals to be allowed to breed with any other individual, the population can be divided into subpopulations or *demes* [4, p23], placed on a 2D grid, and encouraged to breed with individuals in their locality. During coevolution, the idea of demes on a 2D grid could be extended to encourage individuals to use others in their locality as test cases, or to allow individuals to search for suitable test cases.

2.4.5 Strongly Typed Genetic Programs

In most genetic programming systems, the terminals and functions are chosen so that they operate with only a single type. This is known as *closure* of the system. However it is possible to include different types as shown by Montana [16], so long as care is taken to ensure that all functions take in and return the valid types for the branch they are on. Multiple types make the rules of crossover and mutation more complicated, but can allow complicated solutions to be derived faster. It is possible to make functions generic to reduce the number that must be explicitly specified.

2.5 Prisoner's Dilemma

2.5.1 Basic Game Description

The game of Prisoner's Dilemma was first proposed by Merrill Flood in 1951, and arises from the following situation, quoted from [21]:

Two criminals are captured by the police. The police suspect that they are responsible for a murder, but do not have enough evidence to prove it in court, though they are able to convict them of a lesser charge (carrying a concealed weapon, for example). The prisoners are put in separate cells with no way to communicate with one another and each is offered to confess.

If neither prisoner confesses, both will be convicted of the lesser offence and sentenced to a year in prison. If both confess to murder, both will be sentenced to 5 years. If, however, one prisoner confesses while the other does not, then the prisoner who confessed will be granted immunity while the prisoner who did not confess will go to jail for 20 years.

What should each prisoner do?

An alternative analogy is playing a game for coins [16]. If both players cooperate they receive 3 gold coins each, if they both backstab the other they receive 1 gold coin each, but if one backstabs and the other cooperates, the defector gains 5 coins and the cooperator 0. This coin analogy will be used, as high scores are wanted for successful defection, where as the prisoner analogy gives low scores for successful defection (0 years in prison).

2.5.2 The Payoff Matrix

The outcome for each of the participants can be summarised in the following matrix. R represents the reward for cooperation, S represents the sucker's payoff, T represents the temptation to defect, and P represents the punishment for mutual defection [17]

	Cooperate	Defect
Cooperate	R = 3	S = 0
Defect	T = 5	P = 1

The figures being used in the matrix could be altered, so long as they satisfy the following:

1) T>R>P>S and 2) R>(S+T)/2

The first rule ensures the temptation to defect exists. The second ensures that players score worse by taking turns to defect instead of cooperating all the time.

The dilemma arises because it seems to both players that they will better off defecting (5 as opposed to 3, or 1 as opposed to 0), but if they both defect then they will only score 1, less than the 3 they would have got from cooperating. What should a player do?

2.5.3 An Evolutionary Stable Strategy

As T strictly dominates R, and P strictly dominates S, it makes sense that if the game is played just once, a player should defect. This is the rational choice, and satisfies the criteria of an evolutionary stable strategy. However we can see that if two players play irrationally, it is possible to score more than two rational players. This is a worrying observation.

2.5.4 The Iterated Prisoner's Dilemma

The Iterated Prisoner's Dilemma (IPD) involves repeating the Prisoner's Dilemma game a certain number of times, and keeping the overall score. Now the problem of evolutionary stability becomes a problem. If both players are playing cooperatively then they satisfy the Nash equilibrium, but it is not evolutionary stable, as any defector will score T to his opponents S. If both defect, again they satisfy the Nash equilibrium, but they are now scoring less than if they cooperate.

The most disturbing aspect of the IPD was highlighted in the Flood-Dresher experiment [21]. Suppose the game is played 100 times. Both players know that on the final go, they are in fact playing a solitary game of PD, and should therefore both defect. This in effect makes the

99th go the final go, but then this again can be viewed as a solitary game of PD, so both should defect. Iterating this line of thought implies that both players should defect on every go, but they will then only score 100 points, where as cooperators, the seemingly irrational players, will score 300. This is something of an unresolved paradox, and "a demonstration of what's wrong with theory, and indeed the world" [2]. A very theoretical examination of the IPD is given in [8, p.101]

2.5.5 Simple Strategies for IPD

The simplest strategies are to cooperate on every move (AllC), or to defect on every move (AllD). These are the *blind* strategies, relying on no information from the game so far to make their decisions. AllD will perform well if it is in a population of mainly cooperative players as it will be able to exploit them, but will suffer against similar individuals. AllC can only prosper in an environment of cooperative players.

Another simple strategy, Tit-for-Tat, has proved to be one of the most successful, and in fact won a tournament staged by Axelrod of many different IPD strategies [1]. It works by cooperating initially, and then copying its opponent's last move. In this way it can never win, but will score well against all types of opponent. It should be emphasised that the aim of IPD is to score well (maximise expected return), **not** to win. Many other strategies are available for playing on an IPD simulator at [23].

2.5.6 Lack of an Evolutionary Stable Strategy

A strategy is in equilibrium if

$V(S_i|S_i) \geq V(S_j|S_i)$,

where $V(S_j|S_i)$ is the score achieved by strategy S_j playing against S_i.

This is simply condition §2.2.5 i) restated with different notation to represent a game with multiple goes and avoid mentioning the payoff matrix.

If we make the assumption that a strategy has no knowledge of which go is being played, studying the Tit-for-Tat strategy shows it to be in equilibrium according to the above equation, as no strategy playing against Tit-for-Tat can do better than to score an average of 3.0 per go (cooperation), which is what Tit-for-Tat scores against itself.

Unfortunately, §2.2.5 ii) states that (in the new notation)

if $j \neq i$ and $V(S_j|S_i) = V(S_i|S_i)$, then $V(S_j|S_j) < V(S_i|S_j)$ if i is stable.

This condition is violated if we play Tit-for-Tat against the Tit-for-2-Tat (TF2T) strategy, which is very similar to Tit-for-Tat, but won't defect until the opponent has defected twice in a row, thus making it slightly more forgiving. Since neither Tit-for-Tat or Tit-for-2-Tat will ever defect except as retaliation, they will cooperate with each other and score 3.0 per go. Therefore $V(S_j|S_j) = V(S_i|S_j)$, so Tit-for-Tat is not evolutionary stable.

It is quite easy to show how Tit-for-Tat could potentially be invaded by its "twin" strategy[8] Tit-for-2-Tat and a third strategy, Suspicious Tit-for-Tat (STFT), which operates as Tit-for-Tat but defects on the first move. Only a few individuals playing STFT are placed in the population. Now TFT and TF2T both score 3.0 against each other, but where as TF2T can induce cooperation in STFT and score 2.7 [9], TFT will exact retribution and force the game into a series of defections, only scoring 2.5. This will allow the TF2T strategy to slowly take over the population. A more in depth analysis of the above can be found in [17, Deterministic strategies]

If information about which go being played is incorporated into the strategies, it is even easier to displace TFT. A strategy which cooperates on every go except the last [10] will outscore TFT 3.2 to 2.7, and so can take over. It struggles to maintain its superiority however, as it only scores 2.8 against itself, and is easily defeated by an AllD player.

Boyd and Lorberbaum proved that the above thinking about invasion extends to all strategies, and that no deterministic strategy can be evolutionary stable [11, Deterministic strategies].

2.5.7 Stochastic Strategies

A strategy such as Tit-for-Tat will suffer in a noisy environment, where the probability of playing each move according to the strategy is less than 1. An "accidental" defection by the opponent (see §2.2.1) can lead to sequence of recriminatory moves, lowering the average score of both players.

Research by Nowak and Sigmund[3] showed this to be due to the deterministic nature and rationality of the strategies thus far described. A strategy found to due well in this situation was one which defects after S and T, and cooperates after R and P with a very high probability. This can correct occasional mistakes and exploit naïve cooperators, and was

[8] A twin strategy to strategy x would score the same against x as x would against itself
[9] Game lasts for ten goes in all examples, as longer games will not affect outcome.
[10] This will be known as a "sneaky" strategy

dubbed *Pavlov*. Its weakness lies in the fact that it is easily exploitable by AllD, but if the probability of making certain decisions based on experience is reduced from one to slightly less than one, Pavlov can achieve stability against AllD if the game is iterated for long enough. [17, Stochastic strategies]

2.5.8 Spatialised Iterated Prisoner's Dilemma

Some experimentation has been carried out into placing a population of strategies on a 2D grid, and seeing how their distribution evolves by each individual playing its immediate neighbours. Some interesting results can be obtained.

If a small group of AllC strategies are placed together on a grid consisting on predominantly AllD strategies, AllC strategies can spread throughout the population if they only play each other and take over adjacent squares [22]. This would not normally be possible as AllD can easily defeat AllC in a game.

[15] creates a system to randomly place a population on a 2D grid, and hunt for an opponent according to bits selected in a Genetic Algorithm. These bits are set randomly however and no results of note appear to have been obtained, although it provides an excellent mechanism for carrying out the hunting to be investigated in this project

3 Analysis

3.1 Requirements

In order to design a system to meet the desired aims, the requirements must be formalised. Requirements prioritisation is based on the **M**o**SC**o**W** criteria, namely **M**ust have, **S**hould have, **C**ould have and **W**ould like, and divided into functional and non-functional requirements.

3.1.1 Functional requiremnts

	GP Engine	
1	The engine shall support the genetic operations of mutation, reproduction and crossover.	M
2	The engine shall just provide a mechanism for GP to take place	S
3	The engine shall be extendable to whatever scenario is required	M
4	The engine shall support tournament selection	C
5	The engine shall allow a flexible formulation of fitness	M
6	The engine shall store a population of individuals	M
7	The individuals shall be represented by chromosomes, which are in turn made up of genes	M
8	The genes will form a treelike structure, consisting of terminals and functions	M
9	The engine will have editable parameters for the chromosomes	S
10	The engine shall have editable parameters for the genetic program itself, such as number of generations and population size	M
11	The engine shall support different types with appropriate type checking	C
12	The engine shall support co-evolution	M
13	The engine shall be implemented in Java	S

14	The engine shall give regular updates to the user	M
	Prisoner's Dilemma game	
15	The game shall be playable by two computer players against each other	M
16	The game shall be playable by a human against a computer opponent	C
17	The computer player shall operate a deterministic strategy	M
18	The strategy shall have access to a complete history of the game up to that point	S
19	The length of game shall be editable	S
20	The players shall know the length of the game	C
21	The payoff matrix shall be as described earlier, with scores of 5,3,1 or 0.	S
22	The payoff matrix shall be editable	C
23	There shall be a random element possible in a computer player's strategy	C
24	Players shall be able to hunt for their opponents	S
	Hunting for an opponent	
25	Individuals shall be placed on a 2D grid	M
26	Individuals shall move randomly around the grid	S
27	Individuals shall perform a compatibility test if they are adjacent to another Individual	M
28	The compatibility test shall decide whether the Individuals play	M
29	Individuals who have found an opponent shall stop searching	M
30	Individuals shall only have a certain number of goes to find an opponent	M
31	The size of the grid shall be editable	S
32	Hunting shall be visible to the user	C
	Integrating Prisoner's Dilemma into the engine	
33	Each Individual shall represent a strategy	M

34	The fitness of an Individual shall be how well it performs at Prisoner's Dilemma	M
35	The fitness shall be calculated over a number of games	S
36	The functions and terminals shall be relevant to Prisoner's Dilemma	M
37	An Individual shall evaluate to a valid move in Prisoner's Dilemma	M
38	Genetic operations shall alter the strategy that the Individual plays	M
39	Players shall play against fixed opponents or against each other	S
40	Custom built strategies may be developed	S
41	Strategies may be seeded into the Population at the outset	C

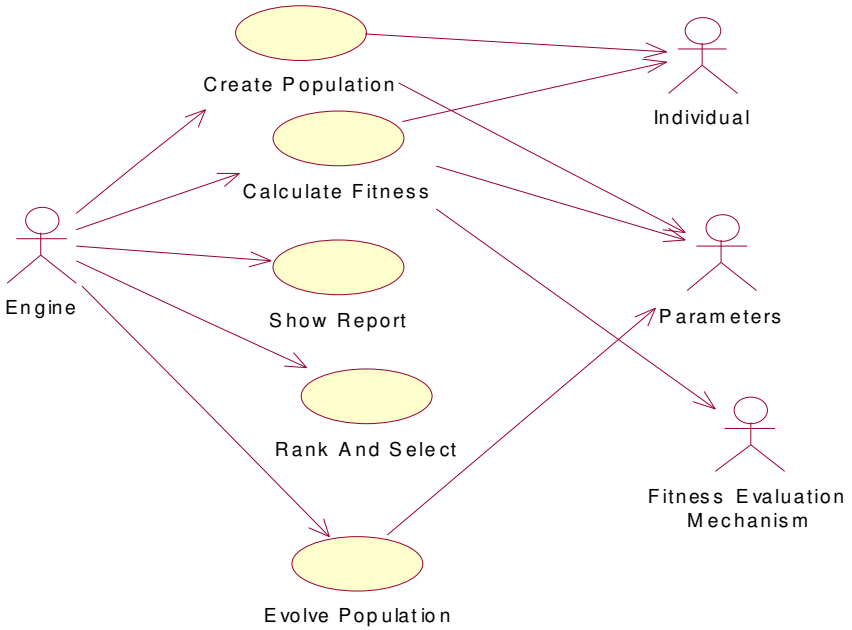
3.1.2 Non-functional requirements

1	The user shall be able to edit all parameters by means of a graphical user interface (GUI)	C
2	Opponents and strategies to be seeded shall be selectable on the GUI	C
3	Generational reports shall be output to the GUI	C
4	A graph detailing fitness information shall be presented on the GUI	C
5	The grid containing representations of individuals shall be displayed on screen	C
6	Individuals shall have different representations according to whether they have found an opponent	C

3.2 Use Case Diagrams

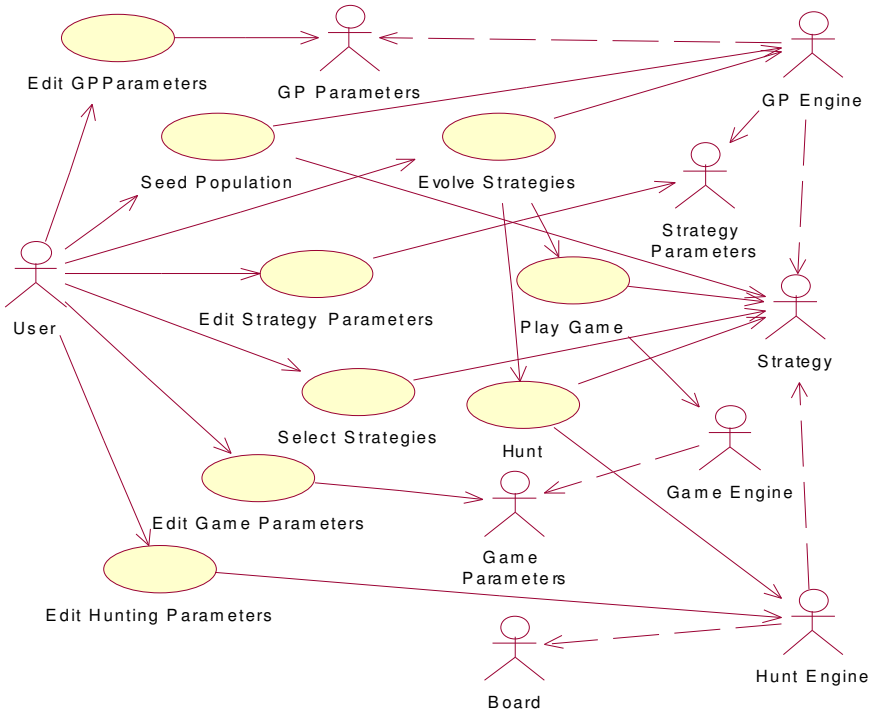
Now the requirements have been formalised, use case diagrams can be drawn to demonstrate what the system should achieve

3.2.1 GP engine



Looking at a higher level, a use case diagram can be used to show what the user is able to do, and what engines and parameters need to be accessed to achieve these choices.

3.2.2 Whole system



4 Design

4.1 Choice of Programming Language

The choice of programming language for the GP engine and extension to incorporate the game will be Java. There are several reasons for this choice. Java allows the use of packages, which will be helpful for separating the engine mechanism from the Prisoner's Dilemma implementation, and Java also allows utilises inheritance, so classes from the engine can be extended as desired.

It will also be straightforward to implement a class hierarchy using Java, which will allow a set of genes to represent a chromosome, several chromosomes an individual, etc.

4.2 Choice of GP engine

Several engines were examined as potential candidates to be extended for the Prisoner's Dilemma. One was Lithos [20], which was turned down due to its stack based mechanism and the fact it was only apparently implemented in C. Although advantages have been reported with a stack based GP [4, p19], it was decided that the standard tree representation would be more suitable.

The engine chosen was written by Adil Quereshi [24]. As well as incorporating all the compulsory features which are stipulated in the requirements, it also allows mixed types within the same chromosome, with appropriate type checking (see §2.4.5). This is a major advantage over most other engines, and should allow more flexibility in the development of strategies.

The engine is also ideal for extension to an appropriate use with its robust framework, including abstract classes and interfaces, and is well annotated to allow this to be implemented relatively easily. The only drawback is that the engine chosen does not currently support coevolution. This must be added, as it is important to see which individuals become dominant in a competitive contained environment.

4.3 Engine details

4.3.1 Important classes

All of the classes below are stored in the gpsys package

GPsys – class which starts the evolution process
GPParameters – stores the parameters for the GP
GPObserver – interface representing the user
Primitive – abstract class specified with the terminal or function required
Gene – abstract class which is specified to be function or a terminal. Stores a primitive.
Chromosome – stores a collection of Genes
Individual – stores a collection of Chromosomes
Population – stores a collection of Individuals
ChromosomeParameters – stores the parameters for creating a Chromosome, including type information
Type – specification for different types available for use

Stored in the package gpsys.primitives are the collection of primitives currently available for use as functions or terminals

4.3.2 Realisation of use cases

Appendix C gives more detail on the methods and classes in the engine, and explains how the use cases of §3.2.1 are realised. The initial class diagram for gpsys is in Appendix C.9

4.3.3 Classes to be extended or implemented

Definite:
GPObserver – *interface*
Fitness – *abstract*

Probable:
GPParameters
ChromosomeParameters

4.4 Design of the GP

4.4.1 Requirements

Each strategy will be represented by an Individual, which will be made up of a single Chromosome storing a tree of Genes, with each Gene representing either a function or a terminal. We only require a single Chromosome as any strategy can be represented as a single tree. The Individual must evaluate to a valid move in Prisoner's Dilemma. As there are only two options available, cooperate and defect, the Boolean nature of this decision can be captured by saying that the Individual must evaluate to either True or False, with True representing defect and False representing cooperate.

4.4.2 Choice of Terminals and Functions

As the Individual must evaluate to a Boolean, it suggests that the Boolean operators And, Or, Xor and Not would be appropriate. Xor represents exclusive Or [11]. The functions If and EQ (which represents equals) will be used for simplification purposes, although both can be represented by complex arrangements of the above functions. The reason is that it will make complex strategies easier to evolve. All these functions take in and return Booleans.

All of the above are available in the gpsys.primitives package of the engine, but some custom functions are also needed.

- YourPrev – takes in an integer argument, and returns a Boolean according to the entry in the opponent's history as indexed by the argument.
- MyPrev – as for YourPrev, but this time checks the history of this player.
- Go – takes in an integer argument, and returns a Boolean according to whether the argument is equal to the current go number.
- Ever – takes in an integer argument, x, and returns a Boolean according to whether the opponent had ever defected up to x goes back in their history, e.g. (Ever 0) would check whether the opponent had ever defected, whilst (Ever 1) would check whether the opponent had defected before his previous go [12].

[11] Xor returns true if x or y is true, but false if x and y are true

[12] This complicated looking function was chosen after an initial terminal, Ever was turned down. This just returned a Boolean according to whether the opponent had ever defected, and it was decided it was not flexible enough. Allowing only some of the goes to be checked improved the flexibility. The function could have been defined to see whether the Individual has defected in the most recent n goes with n the argument, but this seems to similar to the YourPrev function.

The above four have been chosen because they are the only factors that can be considered when deciding on the next move. Combinations of these will be able to create any deterministic strategy for the Iterated Prisoner's Dilemma. The inclusion of Go is potentially problematic, as the players may not realistically know the length of the game when they take part. It shall be assumed that they do.

This just leaves the terminals. Zero and One are obvious choices, and are included in the primitives package. A custom terminal, Last, is also included, which returns an integer corresponding to the length of the game – 1. This is mainly used by the function Go, as (Go Last) would check whether it is the final go of the game, but can be applied to any function that takes an integer argument.

Two more functions need to be added to cover all the integers necessary. They are AddModLength and MinusModLength, which both take in two integers as arguments, and return them added or subtracted, modulo the length of the game. This is because the integers are used for referencing game histories and the current go, so only integers between zero and the game length are relevant.

All the new primitives will be stored in a package gpsys.primitives.prisoner, as they are specific to this system.

4.4.3 Examples of Common Strategies as Trees

Here are some of the most common Iterated Prisoners Dilemma strategies as they would be represented as Individuals using the defined function and terminal sets.

Only cooperate (AllC):	YourPrev Last [13]
Only defect (AllD):	Not (YourPrev Last)
Tit-for-Tat:	YourPrev 0
Tit-for-2-Tat:	If (EQ (YourPrev 0) (YourPrev 1)) (YourPrev 0) (MyPrev 0)
Pavlov:	EQ (MyPrev 0) (YourPrev 0)
Spiteful:	Ever 0 [14]

These six strategies shall be the only ones available for playing as fixed opponents.

[13] YourPrev Last will always return false, as the entire array for the opponents moves is initially set to false. The final array entry (the one being referenced) will not be updated until the entire game history is available, but by this time the game has finished, so it remains false for the duration.
[14] A spiteful strategy will cooperate until the opponent defects. It will defect from then on.

4.4.4 Choice of Fitness function

The fitness of an Individual will be its average score per go of Prisoners Dilemma. It can therefore lie between 0 and 5. The higher the fitness score the better, as that indicates a better performance. This is in contrast to how the engine currently operates, searching for a fitness close to zero (termination criteria is known).

If two fitnesses are equal, the Individual with the lower complexity will be chosen. This is to prevent the Individuals' trees becoming too large.

Other information will be scored and kept in an Individual's Fitness class, but instead of being used to determine who goes through to the next generation, it will be used for selection in the hunting phase. Each Individual will have a score for cooperativeness and whether they are the first to defect during a game (stored as firstToDefect). They will also store an array of hunt criteria which represent the values they wish an opponent's cooperativeness, firstToDefect and fitness to take.

Cooperartiveness, firstToDefect and fitness were chosen as they carry a lot of information about how other Individuals will fare against them. This is further discussed in §4.6.2.

The Fitness will be displayed in a quintuple, with the five entries representing fitness, complexity, cooperativeness, firstToDefect and huntSuccess.

4.4.5 Control Parameters

The control parameters will be editable by the user, but the following are default choices.

4.4.5.1 Size of Population

The default Population size will be 100. This should be large enough to maintain diversity, but should also make computation time reasonable. It also keeps noise to a minimum. If it is discovered that the results produced show a lack of diversity, the Population size could be increased

4.4.5.2 Number of generations

Fifty generations will be the default. Most evolution takes place in the first few generations and then settles down, so this should be enough.

Some theories advocate a greater number of generations, claiming that significant evolution can take place after the 50 point, so if the results are unsatisfactory the number of generations can be increased.

4.4.5.3 Tournament Size

A tournament size of seven will be initially selected. This is the standard size [4], but may need to be altered if the Population size is drastically lowered or there is not enough diversity in the Population. A smaller tournament size will introduce more noise, and larger will stifle diversity.

4.4.5.4 Probability of mutation

The probability of mutation shall be set very low (1%), so that it is rare as in nature. Again there are sources advocating higher and lower rates, so it can be altered if poor results are achieved.

4.4.5.5 Probability of reproduction

This will be set initially to 40%. To test whether crossover has little or no effect, it could be raised to up to 95%.

4.4.5.6 Chromosome size

The maximum depth of a Chromosome is set to 9, as all strategies can be represented within this depth. The maximum depth at creation will be set to 7 to discourage large strategies forming early on, and the maximum depth of mutation will be 3 so that it has a more profound effect. Mutation lower down the tree is unlikely to alter the strategy dramatically.

4.4.5.7 Evolution type

Generational GP will be used to produce the next generation. This is because the elitism that Steady State GP would supply is not wanted here, because coevolution does not search for a best solution. Instead it monitors how the population changes over a certain number of generations.

4.4.5.8 Other parameters

The length of the game will be 10. This seems sufficient to allow the vast majority of strategies to prove themselves, but could be extended if AllD is performing too well.

Twenty opponents will be used to assess fitness. This seems large enough to avoid a strategy being able to survive on a few lucky results.

4.4.6 Termination criteria

If we assume no prior knowledge of a best solution against fixed opponents then we do not want the GP to terminate early, as we have no best fitness to aim for. We could set the GP to terminate if a strategy achieves an average fitness of 5.0, but it is more interesting to graph how the Population develops over time. This is definitely the case with coevolution, where it is vital to let the GP run for the full number of generations.

4.5 Design of Computerised Iterated Prisoner's Dilemma

4.5.1 Requirements and package gpsys.prisoner

The game shall be played by two computer players, represented by Individuals taken from the Population. Each of the players will have a strategy which is represented by their genetic coding and which will tell them how to play on each move. They must also store sufficient information about the state of the game and previous moves made by both them and their opponent in order to implement their strategy.

It must also be possible to have custom made players, both to play against as fixed opposition and to seed into the Population. Seeding design is discussed in §4.8.3.

All classes dealing with the game will be held in the package gpsys.prisoner.

4.5.2 Specific classes for playing the game

Each Individual must store a reference to the game history information which will be held in a PDParameters class. PDParameters will hold arrays for the previous decisions of the Individual and their opponent, as well as the current go number being played. As each new decision is made, it will be added to the front of the array (array index 0), thus forming a game history with all previous goes able to be referenced. All the array entries will initially

be set to false, indicating cooperation. This introduces a slight but unimportant bias into the strategies. All the game information will be available via get and set methods.

The game itself will be handled by a PrisonersDilemma class. This will have a playGame method, which will take in two Individuals, and evaluate their strategies to give either cooperation or defection according to the state of their PDParameters. The scores will be calculated, and the process iterated as many times as required for the complete game.

Creating custom players will be handled by the PlayerGenerator class. This will have methods for creating each of the six possible players specified in §4.4.3

4.5.3 Hunting for an opponent

Instead of simply playing random opposition, a package will be created to allow Individuals to search for desirable opponents before playing them. This hunt package is detailed in §4.6.

4.6 Design of Hunting Mechanism

4.6.1 Requirements and package gpsys.grid

The aim of the hunting mechanism is to allow Individuals to hunt for an opponent according to some criteria, rather than just play random opposition. The Individuals need to be placed on a 2D grid, and then allowed to move around to search for an opponent. If they encounter another Individual on an adjacent grid square, they may perform a test to see whether they make suitable opposition. If so, they then attach themselves to that Individual, and stop searching for an opponent. After a period of time, the search will terminate. Any Individuals with an opponent will engage in a game of Prisoners Dilemma, whilst unattached Individuals will not play. This will be detrimental to their fitness.

All classes associated with hunting and the grid will be stored in the package gpsys.grid.

4.6.2. Criteria for finding an opponent

Three fundamental criteria will be used for the search. They are:

- Cooperativeness of Individual
- How often they are the first to defect (firstToDefect)
- Fitness of the Individual

Almost without exception, the higher the cooperativeness of an Individual, the more desirable they are to play, as they are easily exploitable and consistently friendly. An Individual that regularly defects first is undesirable, as it limits the scoring opportunities for both players. Fitness is slightly more difficult, as an Individual with high fitness may be an excellent opponent who has just scored well by cooperating, and an Individual with a low fitness may be a poor opponent who was involved in a game of mutual defection. However, a fitness under 10 is definitely desirable in an opponent, and over 30 is undesirable.

4.6.3 An algorithm for deciding on an opponent

There needs to be a definitive way of one Individual deciding whether to play another after checking the above criteria. The following algorithm will be used:

i) Store cooperativeness as a double between 0 and 1, where 0 represents always defecting, and 1 always cooperating.

ii) Store firstToDefect as a double between 0 and 1. 0 means they never defect first, 1 means they always do (n.b. if two players both make the first defection simultaneously, they are both guilty and score 1 for that game).

iii) Store fitness as a double between 0 and 5, representing the average score achieved per go of Prisoner's Dilemma.

iv) Let each player store an array of three numbers, with the entries representing the desired values for the three criteria above.

v) Each player instantiates a double, playability, initially set to 3.

vi) Each player tests the three relevant scores of their opponent in turn against their criteria. The difference between the desired and actual value is subtracted from the playability (difference / 5 for fitness as it is 5 times larger).

vii) The playabilities of both players are added together

viii) If the total of step vii) is greater than 5 [15], the two Individuals "agree" to play each other.

The specific figures quoted in the algorithm may be altered after testing.

4.6.4 Designing classes

The framework for the Grid is based on the framework developed by Winder and Roberts for simulating ants [10]. It is not abstract however, as it requires a lot of information specific to Prisoner's Dilemma.

[15] This number is the playability threshold

The hunting will be controlled by a Hunt class, which takes in the grid size and hunt length information, and creates a display frame, an instance of HuntDisplay.

It will have a method, initialiseHunt, which takes in a Population, creates a new Grid, populates it, and then performs the hunting, displaying the Grid after each move.

The Grid itself shall store a collection of Square objects, which are only aware of their neighbours in each direction, called by the joinSquares method. It shall also store an array of Occupants. These are representations of Individuals on the Grid, but store additional information such as which square they are on and who they have chosen as an opponent.

When Occupants come to move, they check whether adjacent Squares are free by referencing their current square. Similarly, when they try to find an opponent, they check adjacent Squares for an unengaged Occupant. If they find one, they perform a compatibility test to decide whether to play them or not.

The Grid showing the state of each square is passed to the huntDisplay, which buffers the information, and then displays the completed Grid on the display.

4.6.5 Creating and inheriting huntCriteria

When a new Individual is created from scratch, values will randomly be assigned to each of the huntCriteria within the appropriate range. If the huntCriteria are treated as a Genetic Algorithm however, they can be inherited from their parents via the same genetic operation by which the new Individual were created. For example, if a new Individual is created as a mutation of the mother, the huntCriteria will be inherited from her with a mutation of one of the values. Similarly crossover will lead to a random crossover of the criteria of the parents, and reproduction will lead to a straight reproduction of the criteria. This allows the huntCriteria to evolve along with the Individuals, although the diversity will decrease rapidly.

4.7 Design of the User Interface

4.7.1 Editable options for the user

The interface needs to allow the user to edit all the options as described in §4.4.5. It also must allow the user to select opponents for non-coevolution, and select players to be seeded [16]. Functions If, EQ and Go can also be excluded from the GP if desired. To aid the interaction, a selection of choice and tick boxes will be set up.

4.7.2. Prevention of inconsistent parameters

The following slections must be avoided.

- Tournament size greater than half Population size
- Grid not large enough to hold Population
- Probabilities of mutation and reproduction totalling more than one
- Too many Individuals seeded
- Max depth at creation or of mutation greater than max depth
- No opponent selected for non coevolution

4.7.3 Displaying Information

All generational reports and diagnostic updates will be written to a text area within a scroll pane on the interface. Population information will be centered.

4.7.4 Graphing results

A graph class will store an image which is updated after every generation with the best, worst and average fitness. A line graph will be created on a special display frame as the generations progress.

4.8 Incorporating Coevolution

An abstract class CoevolutionMechanism will be extended as follows.

4.8.1 Standard Coevolution

[16] These will be limited to the players available in PlayerGenerator

This will be handled by a class StandardPDTournament, which will play games of Prisoner's Dilemma within the population to determine fitnesses. The implementation of this is discussed in §5.3.3.

4.8.2 Hunting Coevolution

This will be handled by a class HuntingPDTournament. The method coevolveFitness will now allow the Individuals to hunt for an opponent as described in §4.6, before all the games of Prisoner's Dilemma are played. Implementation of this is discussed in §5.3.3.

4.8.3 Seeding

Both coevolution classes will define a method seedPopulation, which is called by the Population class. It places pre-defined Individuals passed from the GUI into the Population.

4.8.4 Fitness evaluation

In the case of co-evolution, there is no way of measuring an absolute fitness as there is no point of reference. Instead a relative fitness is calculated. Initially it was thought that a fitness would not need to be assigned until tournament selection was carried out to decide which individuals were to be used to form the next generation. This logic was rejected for several reasons:

i) The Individual would only have a fitness temporarily, and so it could not be used for hunting purposes
ii) The technique limited population diversity, as taking the tournament winner would favour defecting strategies, where as strategies such as Tit-for-Tat which continually score highly but never actually win would never progress through.
iii) Strategies will be selected for different numbers of tournaments, and so their fitness score may become skewed

Point ii) could be solved using a probabilistic selection method, but instead it was decided to assign fitness as follows.

i) Wait until a new generation has been evolved/created (if first one)
ii) Call the required CoevolutionMechanism extension to calculate fitness for the new generation before the generational stats are calculated

Using this method, each individual plays the same number of games of IPD, and so no bias can occur.

4.9 Extending the engine

4.9.1 Additions/changes to original package gpsys

4.9.1.1 New classes

Abstract classes CoevolutionMechanism, TwoPlayerGame and TwoPlayerGameParameters must be added, as well as the exception classes OverseedException and InvalidStrategyException which both extend GPException.

4.9.1.2 New/edited constructors

New constructors must be placed in Individual and Chromosome to allow Individuals to be created with a pre-defined strategy, i.e. a pre-defined Gene tree. The seedPopulation method in CoevolutionMechanism must be called from the Population constructor if coevolution is being employed.

4.9.1.3 Other changes

Public variables for the abstract classes must be added to the GPParameters, as well as integers representing the gameLength and number of opponents required for Prisoner's Dilemma. A public variable for the TwoPlayerGameParameters class must be added to class Individual, a public Individual worstGeneration must be added to the Population class, and a method inherit must be added to the Fitness class. The method coevolveFitness from CoevolutionMechanism must also be called from updateStats in Population if coevolution is being used. The bestRun information can be edited out as it is not required.

4.9.2 Extensions to engine

The following classes shall also be included in the package gpsys.prisoner.

PrisonerGPParameters:
This class is an extension of GPParameters. The ChromosomeParameters will be initialised for the sole Chromosome being used. Generic instances of PrisonerFitness and PDParameters are created for reference by Individuals, and the engine is set to be generational.

PrisonerChromosomeParameters:

This extends ChromosomeParameters. The return type of the Chromosome is set to Boolean, and the function and terminal sets are explicitly defined, with specific type information for the generic primitives.

PrisonerFitness:

This is the most complicated class as it holds so much information about the Individual:

- Fitness information – average fitness per game of Prisoner's Dilemma and complexity of strategy
- Hunt information – cooperativeness, firstToDefect, hunt criteria.

All this information must be updated after the Individual has played a game of Prisoner's Dilemma, and the method updateStats accomplishes this. All the comparative abstract methods from Fitness are implemented. We never wish the GP to terminate early, so the termination criteria are set unreachably high (average fitness per go of 5.1).

The constructor checks whether coevolution is being employed. If so, then the fitness cannot be calculated yet, so all that happens are new hunt criteria are created if required. If there is no coevolution, the fitness is calculated by playing the Individual against each of the opponents that have been specified. The only other important method is inherit, which allows an Individual to inherit huntCriteria from its parent(s). This allows the hunt criteria to evolve in the same way as the strategies, except that it is effectively a GA instead of a GP.

Prisoner:

Prisoner implements the GPObserver interface. It also has the main method for the system. When the system is run, a new GUI is created, information is received back from it, and the evolution process is started.

The generationUpdate method prints out a generational report into the output window on the GUI, and the diagnosticUpdate method prints out the diagnostic information to the same place. The individualUpdate method is declared but not used.

4.10 Final Class Diagrams

These can be found in Appendix D

5 Implementation

5.1 Problems encountered

5.1.1 Lack of diversity

A major problem encountered upon testing the engine was a lack of diversity in the Populations evolved, especially when coevolution was employed. The Population would settle down quickly to incorporate only a few strategies, all of them very simplistic, such as (YourPrev 0). The reason for this is that small strategies are often successful at an early stage of coevolution when complexity is important, and once established in the Population, they are very hard to remove as they have so few Genes available for change.

To solve this problem, two alternatives were considered.

- Employ a single type system with additional operations Reverse and Rest, as used in [5]. This proved efficient at evolving some very clever complex strategies in a prototype version against fixed opponents, but was too slow to use for coevolution as it needed a dynamic data structure which was difficult to clone.
- Introduce a new type, New Boolean. This will be the return type of YourPrev, MyPrev, Ever and Go. The standard Boolean operators must also be adapted to accept either New Booleans or Booleans as arguments and to return either type as well. This means making them generic to a certain extent. Now strategies such as (YourPrev 0) [17] will not be possible as the final return type must be Boolean, not New Boolean, so all strategies will be more robust and open to modification.

The latter option was chosen and successfully increased diversity in the population. Another method was also employed to further improve diversity - instead of choosing the strategy with the lowest complexity if two strategies have the same fitness, a complexity between 5 and 13 was set to be the most desirable, with lower than 5 following, and greater than 13 trailing behind.

5.1.2 Hunting bias

During the hunt method in class Grid, the array of Occupants is looped through, and each tries to find an opponent by checking its neighbours. This introduces a large bias towards the

[17] The simplest representation of Tit-For-Tat is now (Or (YourPrev 0) (YourPrev 0))

Occupants with low indices in the array, as by the time it gets to Occupants with high indices, a lot of potential opponents will already have been taken.

To correct this, a temporary array is created containing every number from zero to the length of the Occupant array in a random order. This array is then used to decide the order in which Occupants get to hunt.

5.1.3 Error in Crossover type checking

A potentially damaging error was discovered in class GeneBranch in the original engine [24]. Branches of the correct type for crossover from the father were not being recognised, and so many trees were returned with the message "Couldn't find compatible branch in dad during crossover" when this was not true. The problem has been rectified, and crossover can now be demonstrated successfully.

5.1.4 Random numbers

Tests with random numbers have shown that they tend to turn up in batches. This has been noted, but hopefully should not affect results significantly - the Occupants in the hunting phase seem to be following a random 2D walk sufficiently well.

5.2 Other program changes

5.2.1 Speeding up hunting

In order to speed up the hunting phase, the display can be disabled, as updating it takes a large amount of time. This change allows a larger number of generations to be checked.

5.2.2 Adding Dummy function

An extra function, Dummy, was added to the function set. It is of type float, and is simply used to fill the function array if we wish to disable some other function. As it has a return type of float, it can never be chosen to appear in a strategy.

5.3 Examining important methods

As there is too much code to analyse in depth, just the parts that implement the use cases from §3.2.2 will be covered.

5.3.1 Editing parameters

The GUI is used to store all the user information, and is constantly checked by the main class Prisoner for the accept button to be pressed [18]. As soon as this happens, class Prisoner can continue. Using get methods from GUI, Prisoner extracts references to a GPParameters object, the graph panel, the output panel and the filenames for saving Population and generational reports, all of which are stored by the GUI. As Prisoner holds the GUI as a static variable, these methods can all be called from anywhere, even from within the main method.

Within the GUI itself, clicking Accept first calls a consistency checker on the parameters. If this passes, a PrisonerGPParameters object is instantiated and set with all the user's choices. All variables in PrisonerGPParameters are public and are therefore easy to access and change.

Care needs to be taken with the coevolution choice. If standard or hunting coevolution is selected, the public variable CoevolutionMechanism in PrisonerGPParameters must be instantiated with the correct extension (and set with hunting parameters if required). Seeding is also controlled in this class, so the GUI must pass information on the Individuals to be seeded as an array to the public CoevolutionMechanism variable.

If no coevolution is selected, an array containing information on which opponents are to be played to determine fitness is passed to the public PrisonerFitness variable in PrisonerGPParameters. This is then called whenever a new PrisonerFitness instance is created for an Individual.

If any of the functions have been deselected, an instance of Dummy is passed to the PrisonerChromosomeParameters object in the correct position in the function array to overwrite the unwanted function.

5.3.2 Implementing the hunt

[18] Checks for a Boolean value in GUI to be true. If not, it sleeps for a second, then tries again

The two most important parts of the hunting phase are performed by the methods *move()* and *findOpponent()* in Occupant.

move checks that the Individual is not engaged, and if this is the case it then chooses a random direction and checks whether the square in that direction is empty[19]. If so, the Occupant moves into it, otherwise the whole process is repeated.

findOpponent is more complicated. Each adjacent square is checked for an occupied status, meaning it holds an unengaged Occupant [20]. If one is found, a private method *compatibilityTest* is called which returns a Boolean. True is returned if

- either player is yet to play a game (they have no stats to test against huntCriteria and so are given the benefit of the doubt)
- the playability score (calculated as in §4.6.3) is above the threshold required

If either is the case, the two Occupants have an engaged variable which they set to be each other, and the squares are set to engaged status.

Class *Grid* creates the order for the Occupants (as specified in §5.1.2), then calls *move* and *findOpponent* in that order for each. Grid is also responsible for initially placing the population on the grid, achieved by choosing random x and y coordinates and testing whether that square is occupied. If it is free the Occupant is placed there, otherwise the process is repeated. A GridException is thrown if the Population is too large for the Grid.

5.3.3 Performing evolution

Although the genetic operations and selection are carried out in the GP engine, the fitness calculation method has to implemented in package gpsys.prisoner.

Calculating the fitness against fixed opponents is relatively easy as this can be done when the fitness of an Individual is created (i.e. in the constructor). The array of opponents to be played should have been passed from the GUI to the PrisonerFitness class, and each player is created as an Individual using PlayerGenerator. The playGame method in class PrisonersDilemma is then called (§5.4) for the Individual in question and each of its opponents, and the scores are averaged to calculate the fitness.

Coevolution is slightly more complicated. The following must first be added to the start of the updateStats() method in class Population.

[19] It can also choose to keep the Occupant on the same square
[20] A Square can be in one of three states – EMPTY, OCCUPIED or ENGAGED

```
if (gpParameters.coevMech!=null)
        gpParameters.coevMech.coevolveFitness(this, gpParameters);
```

This method ranks the Individuals and returns a generational report, so we must assign the Individuals a fitness before this happens (see §4.8.4). In StandardPDTournament, the coevolveFitness method is implemented as follows.

An opponent is chosen at random from the population, making sure that an Individual cannot play itself. The playGame method in PrisonersDilemma is then called, and the scores returned. These scores are passed to the updateStats method in PrisonerFitness which updates the fitness according to the scores achieved [21]. The process is repeated for as many opponents are specified in the GPParameters.

In HuntingPDTournament, the hunt must be performed first – initialiseHunt is called in class Hunt (§4.6.4). This returns an array of all the Occupants on the grid, representing the whole population. Each of these is checked to see if it is engaged (has an opponent), and if so then a game of Prisoner's Dilemma is played between the two Individuals stored within the Occupants. The fitnesses of *both* are then updated [22]. This process (including the hunt) is then repeated as many times as the number of opponents required.

Seeding is also controlled by the CoevolutionMechanism class. A setSeeds method is called by the GUI with the array of players for seeding. Each index in the array represents a different player. This array is stored in the CoevolutionMechanism class until the seedPopulation method is called by Population. The method then works through the array of seeds, and instantiates each one as an Individual using the PlayerGenerator. These then replace existing Individuals in the population.

5.3.5 Playing a game between two Individuals

The actual IPD takes place in the playGame() method of the PrisonersDilemma class.

It creates a temporary PDParameters object for each player as simple reference, and creates an array of size six to store average score, cooperativeness and firstToDefect information for each player. Loop through all the goes, and for each one update the current go, evaluate the Individuals[23] and score accordingly, and then update the PDParameters for each player.

[21] Only the fitness of the Individual being tested is updated, not the fitness of their opponents
[22] updateStats has two variants – one is called if the Individual has played, and all stats are updated. The other is called if the Individual has not played, in which case cooperativeness and firstToDefect are untouched, but huntSuccess and most importantly fitness suffer as a result.
[23] Shown in Appendix D.2.1 as a sequence diagram

An example of scoring is given below:

```
if (playerOneGo & !playerTwoGo)  //Player one defects and player two cooperates
{
        scores[PLAYER_ONE_SCORE] += 5;
        scores[PLAYER_TWO_COOPERATIVE] ++;
        if (scores[PLAYER_TWO_FIRST_DEFECTION] == 0)
                scores[PLAYER_ONE_FIRST_DEFECTION] = 1;
}
```

Once the required number of games has been played, the array of scores is returned.

5.3.6 Sequence Diagrams

Sequence diagrams showing the sequence of method calls required to perform some other interesting functions can be found in Appendix D.2. These are:

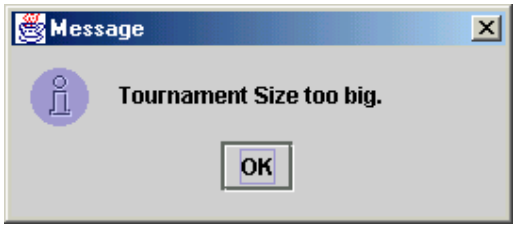
- Evaluating an Individual – shows the use of recursion on the tree structure
- Creating a new Population – shows how the new gpsys.prisoner classes interface with the original engine
- Creating a new Hunt – shows how the hunt is set up

6 Testing

6.1 Testing Interface

6.1.1 Testing illegal selections

A message such as the one below was successfully brought up when the parameters were set to reflect the problems identified in §4.7.2.



All buttons/boxes successfully integrated into program.
Selecting/deselecting functions successfully reflected in strategies.

6.1.2 Testing accept button

The button was tested to make sure it is not active during evolution, but then becomes active again after the run has finished. This proved to be so.

6.2 Testing Hunting

The hunt was tested to ensure that Occupants were moving around the grid successfully, stopping and changing status when they found an opponent, and that this was being correctly displayed on screen. All tests were successful. Lowering the playability threshold caused more Occupants to find opponents as expected.

6.3 Testing Crossover

A Population of 100 was seeded with

- 50 Tit-For-Tat players (Or (YourPrev 0) (YourPrev 0))
- 50 Cooperative players (Or (YourPrev Last) (YourPrev Last))

The probability of reproduction was set to zero, probability of mutation to 0.01. Standard coevolution was selected. As shown in Appendix F.1.1, the Population after one generation consisted of combinations of the two strategies, i.e. the originals and
(Or (YourPrev 0) (YourPrev Last)), (Or (YourPrev Last) (YourPrev 0))

6.4 Testing Mutation and Reproduction

The Population was seeded as above, probability of mutation set to 0.1, probability of reproduction set to zero. As shown in Appendix F.1.2, one of the Population has been mutated after the first generation, and the rest have been reproduced.

6.5 Testing Chromosome Functions

Two runs were implemented using standard coevolution, one using functions If, EQ and Go, and one not. The strategies from the run, shown in Appendix F.4, successfully reflect this.

6.6 Testing seeding

A run was carried out to test seeding. Five Tit-For-Tat players and five AllD players were placed in a population of 10 and coevolved for one generation. As shown in Appendix F.5, after one generation the Tit-For-Tat players had completely taken over, which is to be expected as AllD players score so badly amongst themselves.

6.7 Testing game

A game of length ten was played between a Tit-For-Tat player and an AllD player to test the scoring system. The fitnesses evolved were (2.7, 0.1, 0.0) and (3.2, 0.0, 1.0) [24], which is what the strategies should give. Other games were played between combinations of the six players available for selection, as well as a sneaky player - all of them gave the correct results. This also tested successfully that the six players had been implemented correctly in PlayerGenerator.

[24] (Average score, Cooperativeness, FirstToDefect)

7 Results

All results use the default parameters as specified in §4.4.5 unless otherwise stated.

7.1 Playing fixed opponent(s)

7.1.1 Vs. AllC Player

Figure 1

As expected, an all defecting (AllD) strategy proved to be the best of way of playing an all cooperative player (AllC). Being a simplistic strategy, several versions of AllD turned up in generation 0, notably (Not (YourPrev Last)), and spread so quickly throughout the population that by the second generation, every strategy was playing AllD. This was maintained over 100 generations, apart from a few strategies formed by mutation and crossover which performed less well.

7.1.2 Vs. AllD Player

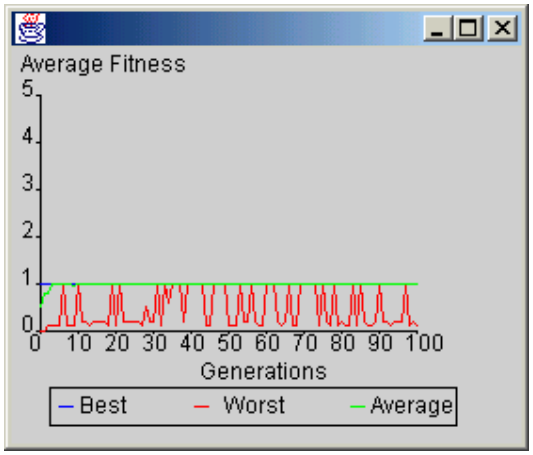


Figure 2

As with the AllC player above, AllD again proved to be the best strategy here, and spread through the Population just as quickly. The fluctuations in the worst strategies are for

new strategies that developed and tried to encourage cooperation unsuccessfully, thus scoring very poorly. An example of one of these is (And (YourPrev Last) (Ever Last)).

7.1.3 Vs. Tit-For-Tat Player

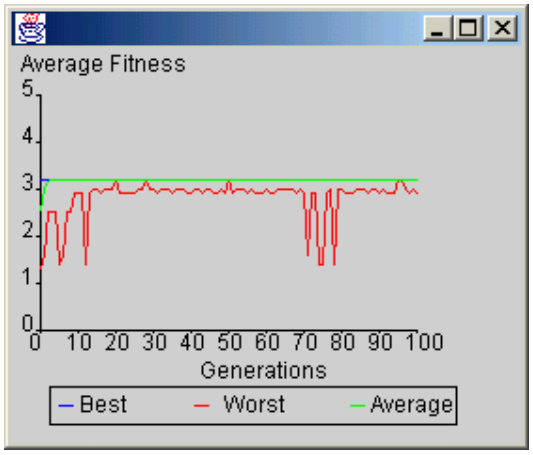


Figure 3

Again it was a simplistic strategy that proved to be the best against a Tit-For-Tat player, namely cooperative on every go until the last, then defect. There are many representations of this, and several turned up in generation 0, including (Or (MyPrev 0) (Go Last)), and spread quickly throughout the Population. Although the average fitness was 3.2, there were always a few strategies that performed worse by trying to defect at an earlier point in the game, thus invoking defection from Tit-For-Tat. Many previous IPD/GP investigations do not incorporate Go information, so they would have failed to find this strategy, ending up with an AllC strategy instead, which scores 3.0.

7.1.4 Vs. Tit-For-2-Tat Player

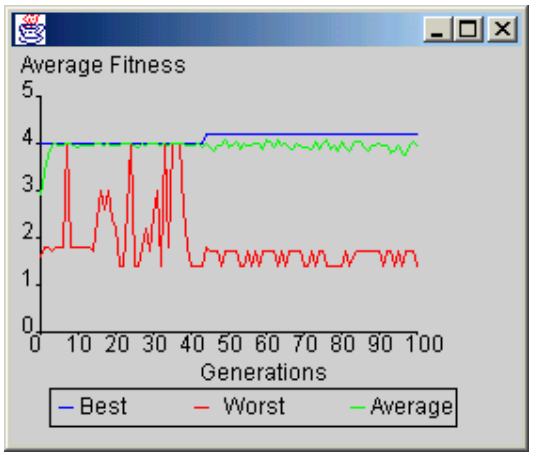


Figure 4

For the first 43 generations, the best strategy discovered was to start by defecting, and alternate between cooperation and defection from then on. This scored 4.0 and spread throughout the Population, before a slightly altered strategy appeared, that was the same apart

from that it defected on the last go regardless. This new strategy scored 4.2. The example which first appeared was

(Not (If (Go (MinusModLength (AddModLength Last Last) (AddModLength 0 Last))) (Ever 0) (MyPrev 0)))

but this soon simplified to

(Not (If (Go Last) (Ever 0) (MyPrev 0))).

This strategy did spread to a certain extent, but because it is slightly larger and more susceptible to change, minor alterations caused major variations in performance. This accounts for the average fitness remaining at around 4.0, and the worst fitness hovering between 1.0 and 2.0.

Figure 4 shows an example of punctured equilibrium [25]

7.1.5 Vs. Cooperative and Tit-For-Tat Players

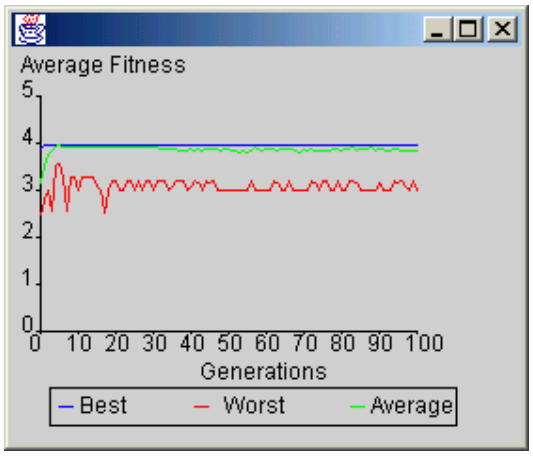


Figure 5

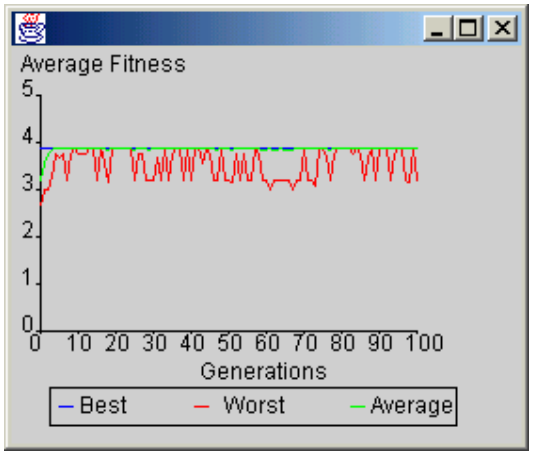


Figure 6

[25] A phenomenon found in nature where one equilibrium is almost instantaneously shifted to a new equilibrium of a different value

This example proved slightly more problematical, as the best solution was not always evolved using the standard parameters – Figures 5 and 6 are from identical runs. In Figure 5, the strategy:
(Not (Or (If (Go Last)(MyPrev Last)(Ever 0))(Ever Last)))
appeared in generation 1. In words, this strategy defects until the opponent defects, and from then on cooperates until the final go when it defects again. This scored 3.95, but couldn't get a major foothold in the Population as a very simplistic strategy, (Not (Ever 0)), scored 3.85 and so was also picked regularly. The average fitness is around this level.

It may seem strange that 3.95 is the best possible score against the two players. Playing AllD against the cooperative player, and cooperating until the last go against the Tit-For-Tat player would score 4.1. The problem is that the opening move is different against either player, and this cannot be possible without knowledge of which strategy is being played. A deterministic strategy can only operate on a set of rules, so will always start with the same move. Therefore the best *deterministic* strategy is the one given above.

Figure 6 shows the best strategy scoring 3.85. This was again (Not (Ever 0)) which spread rapidly and meant that the better but more complicated strategy was never evolved. By increasing the probability of mutation and the Population size, and decreasing the tournament size, greater diversity was encouraged and the best strategy always emerged as in Figure 5.

7.1.6 Vs. Cooperative, Backstabbing and Tit-For-Tat Players

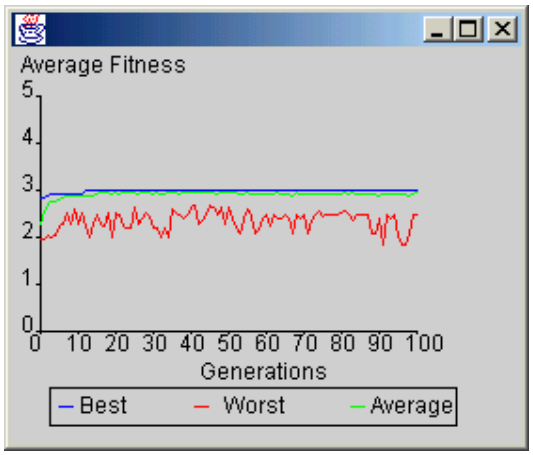


Figure 7

Figure 7 shows two stages of evolution. The best score rose from 2.8666 through 2.9 to 2.9333, where it settled before rising again to 2.9666. The average score stayed at around 2.9333. The best strategy evolved was:
(If (Xor (Or (MyPrev Last) (If (YourPrev 0) (MyPrev 0) (MyPrev Last))) (Or (Ever 0) (Ever 0))) (And (Go Last) (YourPrev (MinusModLength Last (MinusModLength 1 Last)))) (EQ (YourPrev 1) (If (YourPrev 0) (MyPrev 1) (MyPrev Last))))

The simplest way to represent it is:

(EQ (Or (YourPrev 0) (And (Ever 0) (Go Last))) (Xor (Go 1) (Ever 1)))

The way it plays against the three opponents specified can be summarised as defect on go 0 and cooperate on go 1. If the opponent cooperated then defected, continue to cooperate until the last go, and then defect. If they cooperated or defected on both goes, continue to defect on every go. Again, this strategy was not discovered on every run with the default parameters. The best performing simplistic strategy was (EQ (YourPrev 0) (Ever 1)), which scored 2.8333. Changing the parameters as in the previous example yielded the best solution almost every time, and increasing the number of generations allowed the complexity to be reduced.

Again it is hard to appreciate that this is the best strategy, but it highlights the major benefit of using GP – the best solution does not need to be known beforehand, and unlike GA, its structure does not need to be specified. The result gained in [5] that the best strategy would score 3.0 against the three players was only gained by not scoring the first ten goes, thus allowing the first "scored" go to have information about the opponent already. This is not a realistic situation however, as normally there are no practice games beforehand, and scoring starts straight away.

7.1.7 Vs. All players

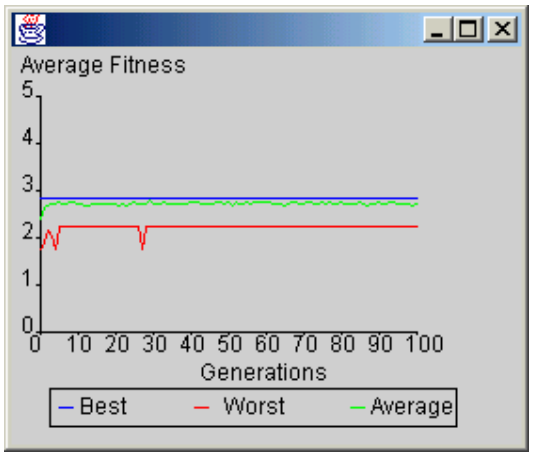


Figure 8

The best strategy for playing all six set players was a very simple one which appeared in generation 0 - (Or (Ever 0) (Go Last)), meaning defect if the opponent has ever defected, otherwise cooperate, but defect on the last go regardless. It scored 2.81666.

7.2 Standard Coevolution

7.2.1 Without functions Go, EQ and If

Figure 9

When all the Individuals play against each other, the fitness level drops off initially as the AllD players exploit the AllC players in the Population, but after a few generations the AllD players start struggling as they are scoring badly playing amongst themselves. This allows the more adaptive players such as Tit-For-Tat to take over as they score much better than AllD when playing each other. The best, average and worst fitnesses rise to level out at 3.0, i.e. cooperate every go. This is the equilibrium position, which is only occasionally disrupted when a few evolved players try to defect. These players are soon forced out however, as there are only a few players in the Population who cannot adapt to deal with an AllD player.

Occasionally when the GP is run, AllD gets such a foothold in the population that no other strategy is able to invade it. This is because the game length is so short, so AllD can get away with some poor scores. If the game length is increased to 100, a game between two cooperative players leaves both 200 points better off than a game between two AllD players, and it is therefore made incredibly unlikely for AllD to take over.

7.2.2 With all functions except Go

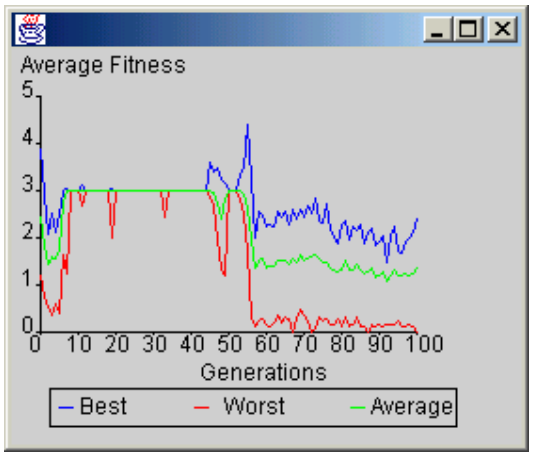


Figure 10

Figure 10 shows how the equilibrium position can be disrupted [26] due to the EQ function, as this can create defective players from cooperative players with only minor alterations to the strategy. For example, (Or (YourPrev 0) (YourPrev 0) operates Tit-For-Tat, where as (EQ (YourPrev 0) (YourPrev 0)) operates AllD. This causes the average fitness to drop off from 3.0 to around 1.5. The equilibrium in §7.2.1 is so strong because most alterations to the strategies caused by evolution do not introduce defection, where as that is not the case here.

7.2.3 With all functions

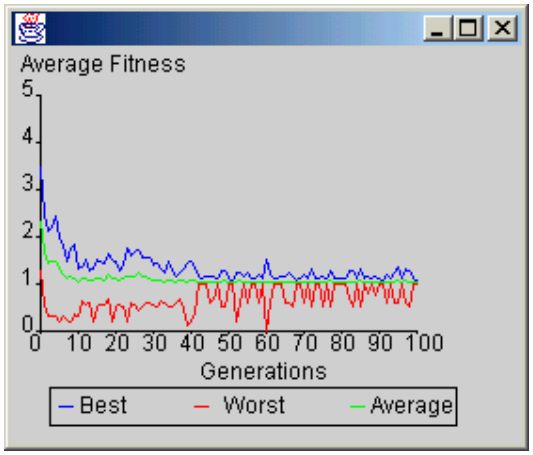


Figure 11

[26] may be disrupted, but often is not

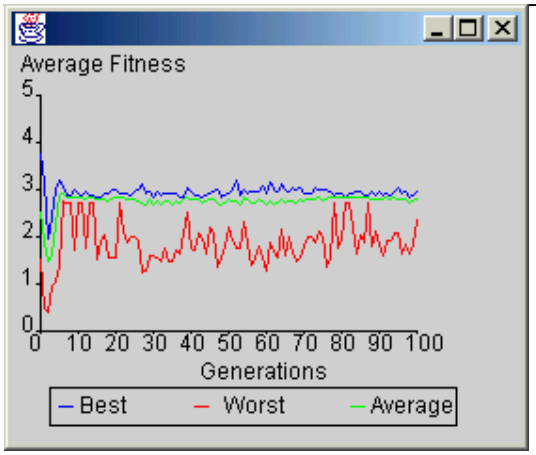


Figure 12

When the function Go is introduced, the situation changes drastically. The equilibrium position achieved in the previous section is no longer stable, as any sneaky strategy that cooperatives until the final go and then defects will be able to beat a strategy that cooperates unless provoked, including Tit-For-Tat. It does not cause a new slightly higher equilibrium however, as when a sneaky strategy plays itself, instead of scoring 3.2 as it would against an AllC player, it now scores 2.8. This is *lower* than the previous equilibrium position. The other problem is that a lot of the strategies which defect solely on the final go are opportunistic and can be easily beaten by invading AllD players. This can cause the average fitness to drop off.

Figures 11 and 12 were obtained on identical runs. Figure 11 shows a case when no strategy can get a foothold in the Population to fend off AllD, so the average fitness drops off to around 1.0. From that point, the equilibrium can almost never be shifted, as Tit-For-Tat players can only invade in large numbers. This situation is more likely than in §7.2.1, but can still be nearly eradicated by increasing the game length.

Figure 12 shows a "quasi-equilibrium" where a robust sneaky strategy such as (Or (YourPrev 0) (Go Last)) forms the majority of the Population, causing the best fitness to fluctuate between 2.8 and 3.2, and the average fitness to stay just below 3.0. It is strong enough to fend off the threats of invading AllC, AllD and Tit-For-Tat players, has all the benefits of Tit-For-Tat, but suffers most by playing itself. Although it is in fact a Nash equilibrium (see §2.2.4), cooperative players could invade by playing amongst themselves and scoring 3.0, provided that there are enough of them. That is why the equilibrium is only semi-stable.

A stable equilibrium could form at 2.8 if the entire population was made up of sneaky players, but this will not happen for two reasons.

- There is always likely to be at least one strategy in the population which tries to cooperate at all times and can be exploited on the final go

- Unlike §7.2.1, minor changes to a sneaky strategy dramatically affect the performance, e.g. if the (Go Last) is changed to (Go 0), it could trigger a series of reprisals with a Tit-For-Tat player, causing the fitnesses of both to drop.

It should be emphasised that Figure 12 is not reached on every run, but it can be encouraged by increasing the game length and population size, and lowering the tournament size.

The last three generational reports of a run with all functions can be seen in Appendix F.2.

7.3 Hunting coevolution

For every run, the length of the hunt was 40, and the size of the grid 20x20 (to best accommodate the Population of size 100). These figures were also used in [15]

7.3.1 Setting the threshold for playability

Early runs involving hunting seemed to be encouraging cooperation, with a high cooperativity generally emerging as the crucial factor in the hunt criteria, but on some runs the hunt criteria would settle to seemingly random levels, and the strategies would adapt themselves to meet these criteria specifically. This suggested that being able to find an opponent was becoming more important than scoring well at the game, and this does not fit well into the GP ideal. By testing how many Individuals were able to find a game on each hunt, it was discovered that only 5-20% were playing in the first few generations. Even increasing the hunt length had no effect [27]. This percentage is clearly too low, as the whole population becomes dictated by the Individuals who do manage to get a game. In order to improve this, the threshold for playability was lowered from 5.0.

At 4.5, approximately 40-60% of Individuals were getting a game, whilst at 4.0, 70-80% were playing, which is much more suitable.

Even though the threshold of 5 was too high, it did yield interesting results. As the strategies adapted to meet the dominant hunt criteria, an unstable equilibrium formed. However if the hunt criteria were altered slightly, the strategies that had been dominant suddenly became redundant as they were so opportunistic and inflexible. Completely new strategies then had to form around the new criteria. The equilibrium only became stable once the cooperativeness

[27] Some tests were carried out on the best hunt length. After 40 moves, most Occupants who could find an opponent had done so. Raising this to 80 slighlty increased the number of Occupants getting a game, but any higher had no effect.

level required became sufficiently high for cooperative players to take over. This demonstrates the ability of GP to create strategies which adjust to their environment, even if this environment is dynamic.

7.3.2 Inducement of cooperation

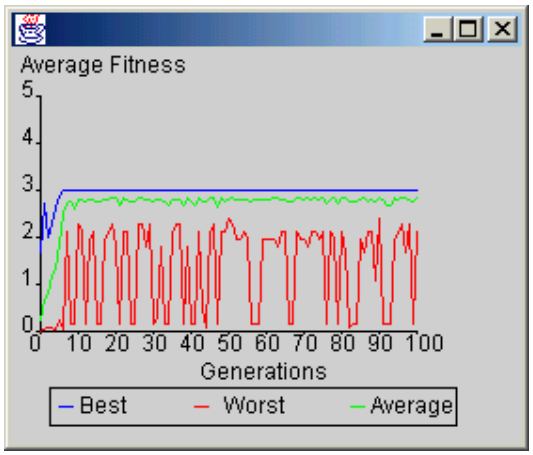


Figure 13 - Playability of 4.5

Runs of the GP with the playability threshold set to 4.5 showed that if an Individual is instantiated with hunt criteria that encourage a cooperative opponent, it tends to do very well as there are so many AllC players in the first generation. Not only does it get a lot of games, but also scores highly, no matter what its genetic make-up.

This means that many evolved Individuals will inherit the hunt criteria which tell them to search for a cooperative opponent. If the Individual which performed well in generation 0 was a defective player exploiting the cooperators, it will no longer be able to find many games and so will not be selected for the next generation. Consequently, cooperative players take over rapidly, as can be seen from the Figure 13. This seems reasonable, as hunt criteria which search for a player with low cooperativity are detrimental to both parties.

As the generations pass, the criteria gradually begin to resemble the type of player everyone wants play – maximum cooperativity, minimum firstToDefect and fitness of 3.0. These "ultimate" statistics cannot quite be reached as diversity within the hunt criteria becomes small, so it takes a lucky mutation to make an advance.

The average fitness remains below 3.0 as there are often strategies that fail to get many games and so score badly. Notably the best fitness is at 3.0 despite all of the functions being included. This seems to be because the cooperativeness required to get a game is so high (0.95 – 1.0) and the firstToDefect sufficiently low, that even sneaky players who cooperate on every go bar one cannot play enough to gain their slight advantage over Tit-For-Tat, so cooperative strategies dominate.

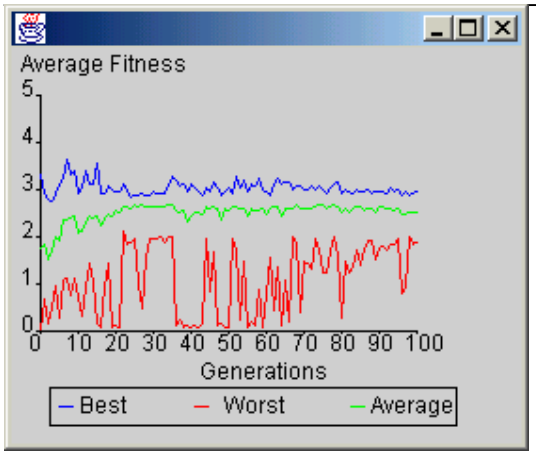


Figure 14 - Playability of 4.0

With the playability threshold lowered to 4.0, finding an opponent becomes less of a challenge and so similar results are obtained to §7.2.3. Sneaky players now start to emerge, and the hunt criteria settle around 0.9 for cooperativeness and 0.8 for firstToDefect.

In both Figure 13 and Figure 14, the equilibrium or quasi-equilibrium is much stronger than in §7.2.1/§7.2.3, because in order for AllD players to invade, they must first get a game, and as their fitness statistics bear no relation to the hunt criteria required, this is almost impossible.

The last three generational reports of a run with playability set to 4.5 can be found in Appendix F.3.

7.4 Results summary

These were the results gained on the majority of runs. For the coevolution fields, these are the best possible strategies that could evolve if an equilibrium is reached. In each case, assume stable equilibrium unless stated otherwise in red, and assume standard parameters as in §4.4.5 unless otherwise stated. The hunting results were taken from actual runs.

GP run	Best/regular strategy	Fitness	Cooperative	FTDefect
vs AllC	Not (YourPrev 0)	5.0	0.0	1.0
vs AllD	Not (YourPrev 0)	5.0	0.0	1.0
vs TFT	Or (Go Last) (Go Last)	3.2	0.9	1.0
vs TF2T	(Not (If (Go Last) (Ever 0) (MyPrev 0)))	4.2	0.4	1.0
vs AllC, TFT	(Not (Or (If (GoLast) (MyPrev Last)(Ever 0))(Ever Last)))	3.95	0.35	1.0
vs AllC, AllD, TFT (tournament size 4, p.mut =0.2)	(EQ (Or (YourPrev 0) (And (Ever 0) (Go Last))) (Xor (Go 1) (Ever 1)))	2.9666	0.2333	1.0
vs All 6 players	(Or (Ever 0) (Go Last))	2.8166	0.7666	0.8333
Standard coev.	(Or (YourPrev 0) (YourPrev 0))	3.0	1.0	0.0
Standard coev. inc. Go, EQ, If (semi-stable)	(Not (YourPrev Last))	1.0	0.0	1.0
Standard coev. inc. Go, EQ, If, Game length 50 (semi-stable)	(Or (Go Last) (Go Last)	3.2	0.9	1.0
Hunting coev, Threshold 5.0 (unstable)	And (EQ (YourPrev 0) (Ever 0)) (Not (MyPrev Last))) Hunt Crit. (0.62, 0.92, 1.71)	2.75	0.6	0.9
Hunting coev. Threshold 4.5	(Or (YourPrev 0) (YourPrev 0)) Hunt Crit. (0.62, 0.96, 2.21)	3.0	1.0	0.0
Hunting coev. Threshold 4.0 (semi-stable)	(Or (Go Last) (Go Last) Hunt Crit. (0.92, 0.13, 3.08)	3.2	0.9	1.0

8 Conclusions and Evaluation

8.1 Conclusions

The use of genetic programming allows optimal strategies to be generated very successfully against a selection of opponents. The random principles of evolution mean that it is sometimes difficult to evolve the more complex strategies, but by adjusting the GP parameters to increase diversity, they are usually discovered. The fact that the whole game history is available is used by some of the more complex strategies, and this is where GP has the advantage over GA.

The experiments with coevolution show that without the If, EQ and Go functions, cooperative behaviour emerges as the equilibrium from a random selection of Individuals. This is because although defective Individuals can gain short-term benefits by exploitation of cooperative players, they struggle over the long term as they perform badly against players similar to themselves. On the other hand, cooperative players perform well amongst themselves, and so do well over the long term. The difficulty is in establishing the cooperative behaviour in the first place, as this requires enough non-defecting strategies to establish a foothold. These strategies must be robust enough to perform well against defective players to have a chance of selection, but then cooperate with each other. This is what makes Tit-For-Tat so successful.

Once in a cooperative situation, any cooperating player will score well, but unless enough of the Population are robust enough to prevent invasion, defective strategies can alter this equilibrium.

When Individuals are allowed to take into account which go of the game they are on, opportunistic strategies are able to disrupt the cooperative equilibrium if they defect on the final go when retribution is not possible. This can allow defective players to take over, but usually more robust versions of the (Go Last) strategy take over and form a quasi-equilbrium around 3.0. This still means that there is a very large amount of cooperation in the Population.

Allowing Individuals to hunt for opponents tends to encourage cooperation, as cooperative players make the best opponents. Once established, this cooperative equilibrium is much harder to disrupt, as defecting players cannot find opponents to play and so score badly. There is a very delicate threshold for setting the playability however, above which finding an opponent becomes the most important factor.

These results are consistent with previous work. The main difference is the choice of parameters – previous investigations [5] have used a larger population size and a much longer game (100-150 goes), but this system has managed to get the same results using lower numbers, and therefore runs a lot faster. The Iterated Prisoner's Dilemma is the standard metaphor for the conflict between mutual support and selfish exploitation in nature [17, The game], and much research has been conducted into why cooperative behaviour emerges from the competitive setting - reciprocal altruism [28]. This project has managed to answer a lot of those questions. Actually finding the IPD in nature is rare however as the parameters can rarely be discovered, although some cases are claimed [17, Where is Prisoner's Dilemma found in nature].

The hunting phase is much more appreciable in nature, even within human culture. Cooperative people will tend to find partners easily, where as backstabbing people, who may benefit well in the short term, will soon become mistrusted and shunned. Of course, the individuals who are mainly cooperative but can exploit others without losing their trust will do very well, and this is mirrored here by the sneaky players. Testing human interaction with Prisoner's Dilemma does not always give decisive results however, as shown by [18].

In summary, the stronger equilibrium of cooperation derived through hunting is hopefully a useful addition to the growing amount of research into the complicated field of modelling biological societies. The effectiveness of sneaky strategies may also be applied to these societies to show how seemingly comfortable equilibria can be dramatically disrupted by Individuals who do not follow the trend in pursuit of personal gain.

8.2 Success of system

The system has given excellent results in all areas tested, so the choice of GP engine was successful. The only query is whether allowing multiple types stifles diversity. As the Boolean operators are given a specific type which can't be altered once initialised, certain functions can only be included in certain places if they are of the required type. This leads to localised evolution, where different parts of the tree are evolved according to their type. At the top of the tree are the Boolean functions, followed by NewBoolean functions, and then integer functions. As the most dramatic changes in strategy occur with the changes in the Boolean functions near the treetop, a lot of evolution is effectively wasted by changing integer functions which have little effect. This doesn't seem to have caused major problems

[28] Reciprocal altruism is the phenomenon of unrelated individuals cooperating, even when it appears that this is not advantageous in terms of inclusive genetic fitness [5]

however, especially if the parameters are adjusted to allow for it, e.g. lower the max depth of mutation.

8.3 Extending the project

Further investigation could be done into hunting for opponents, as this seems to be a potentially exciting area which hasn't had a great deal of research. The GUI is fairly robust to allow for varying parameters, so that will not require much alteration.

The newly developed engine is well suited to the investigation of any other simple two player game where a fitness can be established. The abstract CoevolutionMechanism, TwoPlayerGame and TwoPlayerGameParameters will have to be implemented as required for the game. All Java documentation to assist in this can be found using Appendix B.2.2

Other possible developments would be a status bar for the hunting phase, displaying information such as how many Individuals have found an opponent, as well as the options of stopping and stepping through the hunt and clicking on different Occupants on the display to show an Individual's strategy and fitness. Seeding has not being explored in depth either, and if a hunt is seeded with just a few AllC individuals and the rest AllD, it may be possible to get the AllC individuals to survive by careful selecting their opponents. This would involve pre setting their huntCriteria as well.

A way of extending the number of strategies available for seeding and as fixed opponents would also be an idea. A database could be formed to hold successful strategies, and this could be searched and updated as required.

There is also more potential for experimenting with different parameters, as few combinations were tested. This may be an especially good idea for the hunting parameters – length and grid size.

A User Manual

A.1 Running the Program

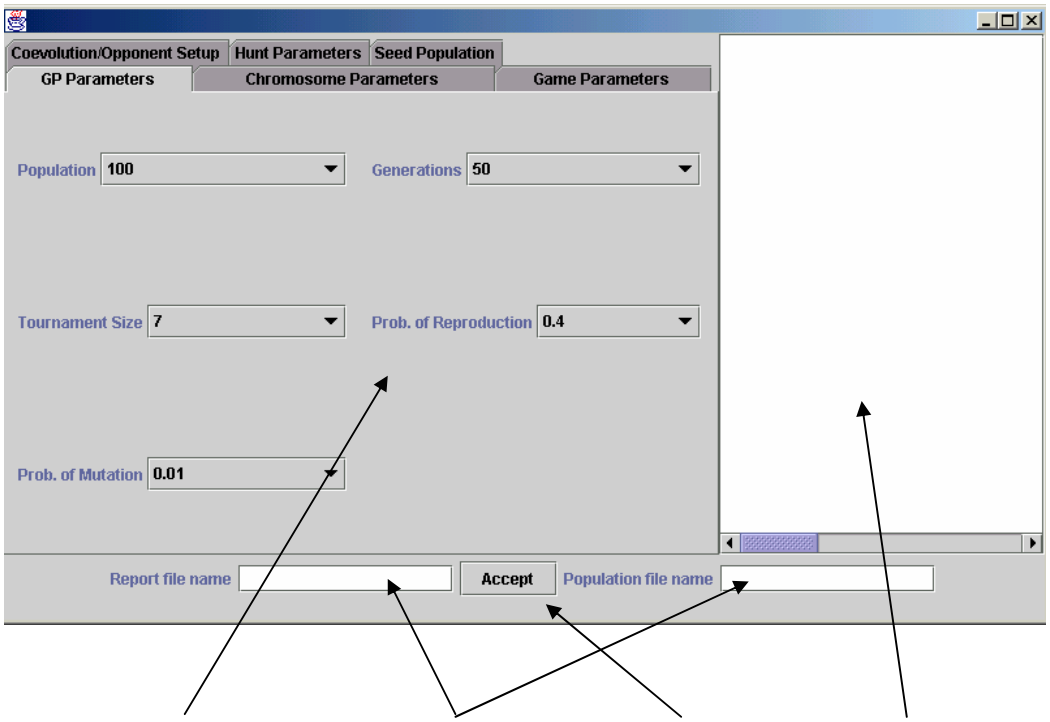
The computer being used must have a version of Java installed. To run the program on a Windows based machine, run the *prisoner.jar* file on the disk by simply double-clicking.

To run on a UNIX based machine, type in
java –jar prisoner.jar
at the prompt.

This will bring up the user interface

A.2 The User Interface

Below is the interface presented to the user on running the program

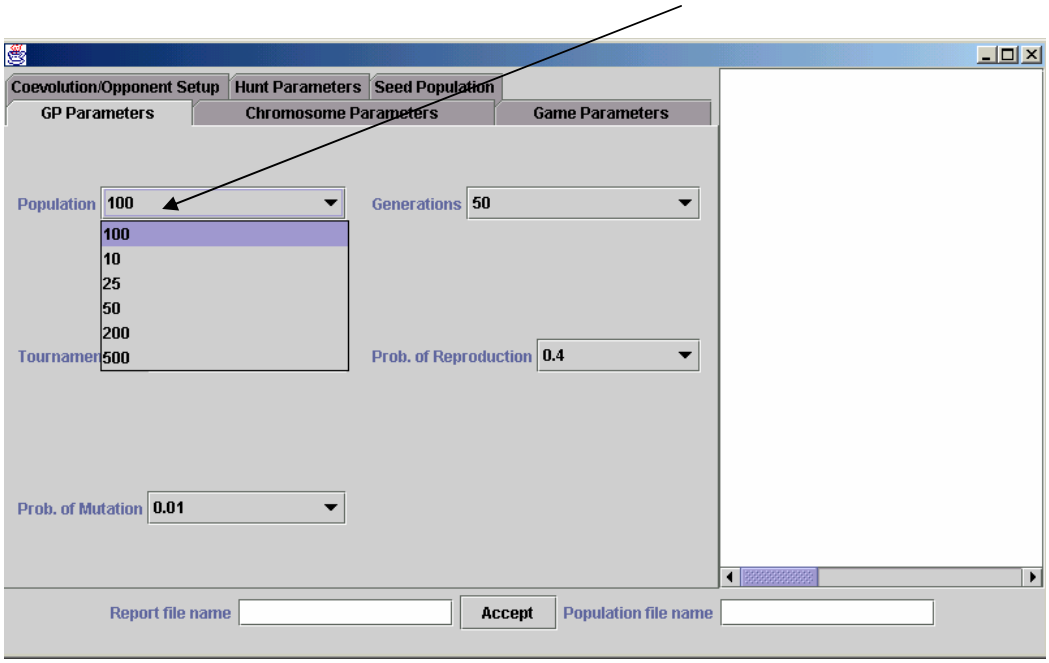


A.3 Changing the options

Click the appropriate tab to select different options panels. The panels are as follows.

A.3.1 GPParameters

Select Population size, number of Generations, size of tournament for Tournament Selection, and the probability of selecting mutation and reproduction when choosing a genetic operation. To select an option, click on the bar shown to bring up a list of possible values.



A.3.2 Chromosome Parameters

Select the maximum depth of strategies, as well as the maximum depth they can be created to, and the maximum depth they can be mutated at. Also choose whether to use the functions If, EQ and Go by ticking the appropriate check boxes.

A.3.3 Game Parameters

Select the game length for Prisoner's Dilemma, and the number of opponents to be played against for determining fitness in coevolution.

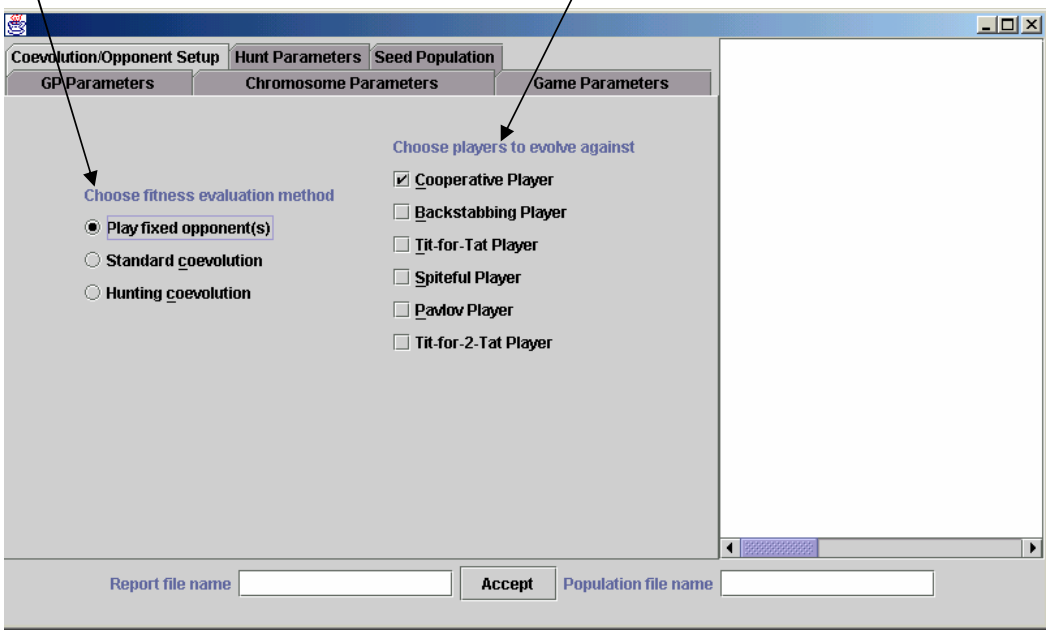
A.3.4 Hunt Parameters

Select the size of the grid for hunting, and how long the Occupants have to find an opponent (in terms of number of moves).

A.3.5 Coevolution/Opponent Setup

Click a radio button to select fitness evaluation method

Click selection of check boxes to choose opponents for fixed opponents method



A.3.6 Seed Population

Select the number of each different type of player to be seeded into the Population.

A.4 Saving the Population and Generational Reports

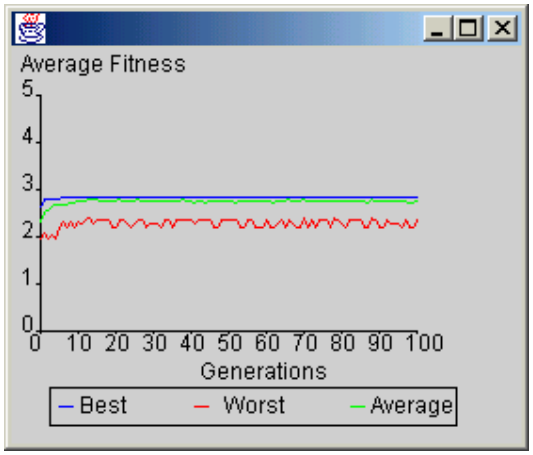
To save generational reports to a text file, enter a file prefix in the Report file name bar, and the reports will be saved as <prefix>.txt in the current directory.

Similarly, to save the final Population, enter a file prefix in the Population file name bar.

A.5 Displaying results of Evolution

Generational reports showing the generation number, average fitness and best and worst Individuals of the generation are displayed in the output pane - this pane can be scrolled. Diagnostic reports are also displayed. The fitness is displayed in the following form: *(average score, complexity, cooperativeness, firstToDefect, huntSuccess* [29]*)*

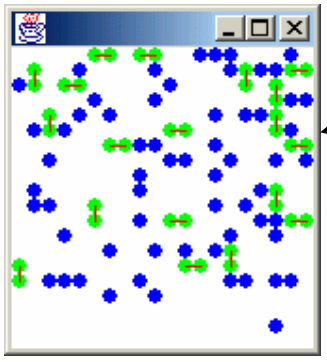
During evolution, a graph is created showing the best, worst and average fitness for the generation.



The graph cannot be obscured or the window minimised once the run has finished, or the picture is lost.

A.6 Displaying the Hunt

During hunting coevolution, the following grid is displayed:



Blue circles represent unengaged Occupants. Green circles represent engaged Occupants, and the line joins them to their opponent.

The grid should be moved to avoid obscuring the graph (above).

[29] Hunt success is out of 1, with 1 being always gets a game.

B System Manual

B.1 System requirements

The computer must have Java 1.3 or better installed in order to run the program. Due to the nature of genetic programming, the program is computationally intensive and so a powerful system is strongly recommended for optimum performance.

B.2 Making changes

B.2.1 Extracting the Code and Documentation

Stored on the disk is the jar file *prisoner.jar*. Copy this into the chosen directory for extraction, and type:

jar xvf prisoner.jar

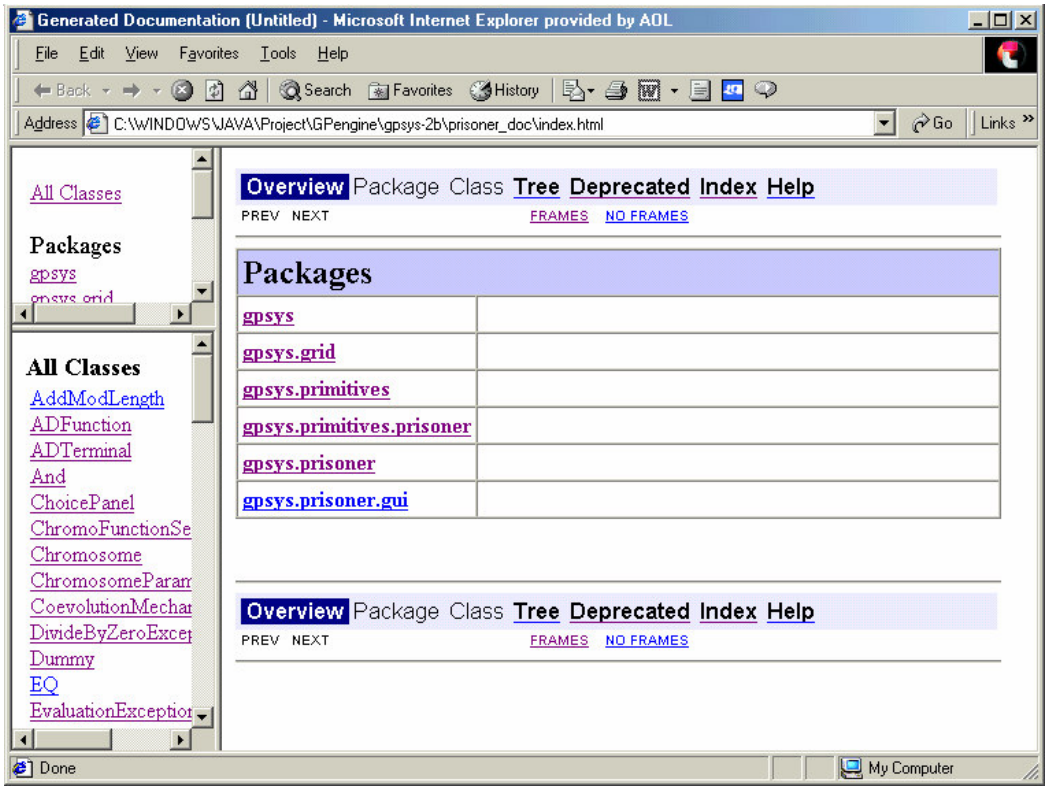
at the command line. This will create the following in the current directory:

- directory *gpsys*, which contains all the *.java* and *.class* files for the system as described.
- directory *prisoner_doc*, which contains all the java documentation for the system
- directory *META-INF*, which contains the manifest file for running the jar file.

All the java files can be viewed on any text editor.

B.2.2 Javadoc

All Java documentation is in the directory prisoner_doc. Using a viewer run the index file using frames, and the screen below will be presented. Click on packages and classes as required.



All classes detail attributes and methods, and the related parameters, return objects and exceptions.

B.2.3 Compiling and Running the System

Firstly, the classpath must be set to the directory in which the jar file was extracted, as this is the base of the package system.

Type *set CLASSPATH = <enter directory here>* in the autoexec.bat file (Windows)
or *setenv CLASSPATH = :./<enter directory here>* in the options file (UNIX)

To compile a java file, the command *javac <filename>* must be typed in the directory containing it.

The main method is in class *Prisoner*, so to run the program type:
java gpsys.prisoner.Prisoner
to bring up the user interface.

To exit, click the cross at the top right of the interface.

B.3 Extending or Adapting System

The Javadoc can be used to adapt the methods and classes as required. If the game is to be changed, packages gpsys.prisoner, gpsys.prisoner.gui and gpsys.primitives.prisoner would need to be altered accordingly.

The following classes must be implemented/extended for package gpsys:

- *Fitness (abstract)*
- *GPObserver (interface)*
- *TwoPlayerGame (abstract)*
- *TwoPlayerGameParameters (abstract)*
- *CoevolutionMechanism (abstract)*

Suitable methods for alteration are:

- *findOpponent* in *gpsys.grid.Occupant* – controls how potential opponents are found and tested for suitability during the hunting phase
- *displayGraph* in *gpsys.prisoner.gui.Graph* – outputs graph to screen. Currently cannot be recovered if covered or minimised
- the constructor of *gpsys.prisoner.gui.GUI* – controls editable options for the user
- the constructor for *gpsys.prisoner.PrisonerChromosomeParameters* – controls functions and terminals to be used in the GP
- *displayPopulation* in *gpsys.grid.Grid* – determines how output of hunt is displayed on screen
- all of *gpsys.prisoner.PrisonerFitness* - controls huntCriteria and fitness information
- all of *gpsys.prisoner.PlayerGenerator* – controls the players available for creation

C Closer examination of GP engine

C.1 Storing GP Parameters

Probably the most important class is GPParameters. This contains all the information related to the GP that is required by the various classes, and is passed between them as a parameter. Its public variables include:

- observer – an object of the GPObserver class representing an observer looking in on the GP process, which will take input from the user, start and control the GP, and produce reports for the user
- engine – whether the GP uses the steady state or generational method
- pMutation and pReproduction – probabilities of mutation and reproduction when a genetic operation is being chosen
- tournamentSize – the size of the group used for tournament selection
- populationSize – the number of individuals in the population
- generations – the number of generations for the GP
- adf – an object of the ChromosomeParameters class, used to determine how new Chromosomes are created
- fitness – a general object of type Fitness, not linked to any Individual. This is used to allow an instance of a concrete fitness class to be constructed, even though the calling class may not know what the concrete class is.
- population – an object of type Population, which is discussed below.

These variables are changed as the GP progresses, so it must always be accessible to classes as they perform their function.

C.2 Population structure

Class Population is where most of the GP work takes place. It stores the following important public variables.

- p – an array of the Individuals in the population.
- generation – the number of current generation
- bestGeneration and worstGeneration – the best and worst Individuals of the current generation in terms of Fitness.
- averageFitness and averageComplexity – statistics for the current generation

- bookkeepingInfo – an object of class CrossoverBookkeeping, used for bookkeeping when creating the next generation. This shall not be detailed any further.

Individuals within the Population are made up of Chromosomes, which in turn are made up of Genes, forming a treelike structure.

The class Individual stores an array of Chromosome objects, a Fitness object which contains its fitness information, and its complexity as public variables.

The Chromosome class stores one Gene, treetop, which can be used to access the rest of the tree by traversing through the children. It also stores its complexity (total number of nodes), the GPParameters that were used to create it, and an index to itself within the array in GPParameters, so thath its ChromosomeParameters can be found.

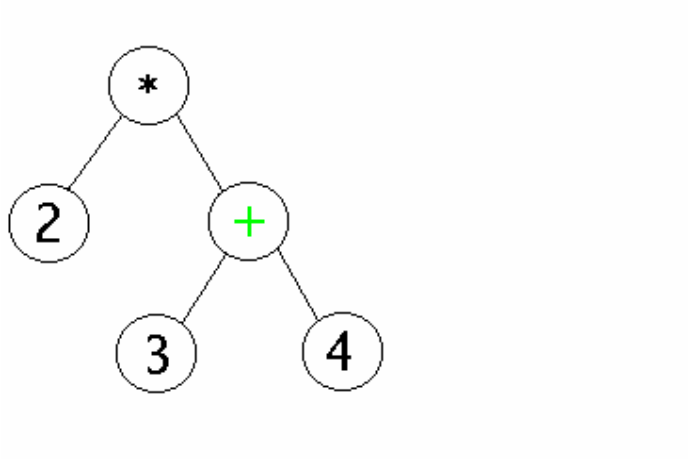
The Gene class itself is abstract, as it needs to be specified as a terminal or function. There are classes GeneTerminal and GeneFunction which extend it, although GeneFunction is itself abstract and must be further extended to specify its method of creation, either GeneFunctionFull or GeneFunctionGrow. Gene stores methods to calculate the depth and complexity of the Gene tree below it (by recursively calling the method on each of the children), an array of its children, and a Primitive object.

Primitive is an abstract class, extended by Function and Terminal, and is used to define the precise nature of the Gene. A separate package, primitives, contains a set of classes extending Function and Terminal, and which are used to represent the actual functions and terminals required.

C.3 Evaluation of an Individual

When an Individual needs to be evaluated, one of a number of methods is called according to the return type required. This recursively calls methods through Chromosome, Gene, Primitive, Function/Type, until the specific class required is reached. This gives precise details on how the individual is evaluated at that particular point, either by evaluating the children if it is a function, or by returning an answer if it is a terminal.

For example, suppose that an Individual consists of a single Chromosome referencing the top of the following Gene tree.



Suppose that the return type of the Individual after evaluation is int, and that the method evaluateInt() is called. This will call evaluateInt(Individual i) in the Chromosome object it is holding, passing itself as a reference. This then calls evaluateInt(Individual i) in the Gene object it is holding as a treeTop, and then finally the Primitive object stored by Gene. In the case above, a class Mul is called which represents the node multiply. This recursively calls the evaluateInt method on both of its children, and returns the result multiplied together. The tree is traversed until terminals are reached, e.g. the class Four above, which would evaluate to 4. Eventually the answer of 14 is returned.

Each class in the primitive package details the number of arguments it takes (if any), their type, and what its return type should be. Appropriate methods are called accordingly.

C.4 Creating new Population

The class GPsys creates a new Population and table of Types available for use in the individuals, and has an evolve method which starts the evolution process.

The Population constructor is called, and passed the GPParameters. This will then call the main constructor in Individual, passing it the ChromosomeParameters specified in GPParameters. Then the Chromosome constructor is called, which will construct the Gene at the top of the tree. This Gene will then recursively form the rest of the tree.

Each tree *must* start with a function. The Gene at the top will then create either a terminal or another function to fill its arguments, according to type checking and random behaviour. Any new function will then create Genes to fill its arguments. This process continues until there are no more functions without arguments – all the leaves are terminals.

Each Individual is then given a new Fitness object to store fitness information.

C.5 Evolving Population

There are two methods of evolution supported – steady state and generational. The latter will be used and is described below:

i) Create a CrossoverBookkeeping object
ii) Randomly select a genetic operation
iii) Call the selectBest method, which will perform tournament selection on a subset of the Population, and return the fittest Individual. (Tournament selection simply ranks the Individuals in order of fitness which has already been established).
iv) If Crossover has been selected as the operation, repeat stage iii)
v) Add the genetic operation and Individuals(s) to the bookkeeping object.
vi) Repeat stages ii)-v) for each new Individual to be created.
vii) Create a new Population using the information in the bookkeeping object
viii) Update the Population statistics (Method updateStats finds the best Individual in the generation, and calculates the average fitness and complexity).
ix) Increase the generation number

C.6 Performing the Genetic Operations

C.6.1 Reproduction:

A special constructor is called in Individual, which creates a new Individual by taking a deep clone of the mother. A deep clone will clone all the objects *and* associations.

C.6.2 Mutation:

The same constructor is called as for reproduction, but a Boolean indicator calls the mutate method in the Chromosome class. The Chromosome is originally the same as the mother, before a branch from the tree is chosen by calling GeneBranch, which checks branches available for selection, and picks one at random. Finally, a new branch is created recursively in the same way as when a completely new Individual is created. The return type of this new

branch is made to be the same as the branch to be replaced. This new branch then replaces the old branch.

C.6.3 Crossover:

Another constructor is called within Individual, which calls the cross method in Chromosome. As with mutate above, a branch is chosen randomly from the mother. The father is then scanned for branches which return the same type, and one of these is randomly chosen. This is handled by GeneBranch. Again, the new branch replaces the old branch.

C.7 Giving feedback to user

An interface GPObserver must be implemented as required by the user, but include the following methods.

- generationUpdate, which has the GPParameters as a parameter. This is called after each generation has been evolved, so the user can view a generational report. All the information required can be accessed via the public variables in GPParameters.
- individualUpdate, which is called after an Individual has been created, to show the user both its tree structure and its creation method.
- diagnosticUpdate, which is called whenever something interesting happens, such an incestuous crossover, or when a tree is discarded for being too large.

This interface, when implemented, will contain the main method for the program.

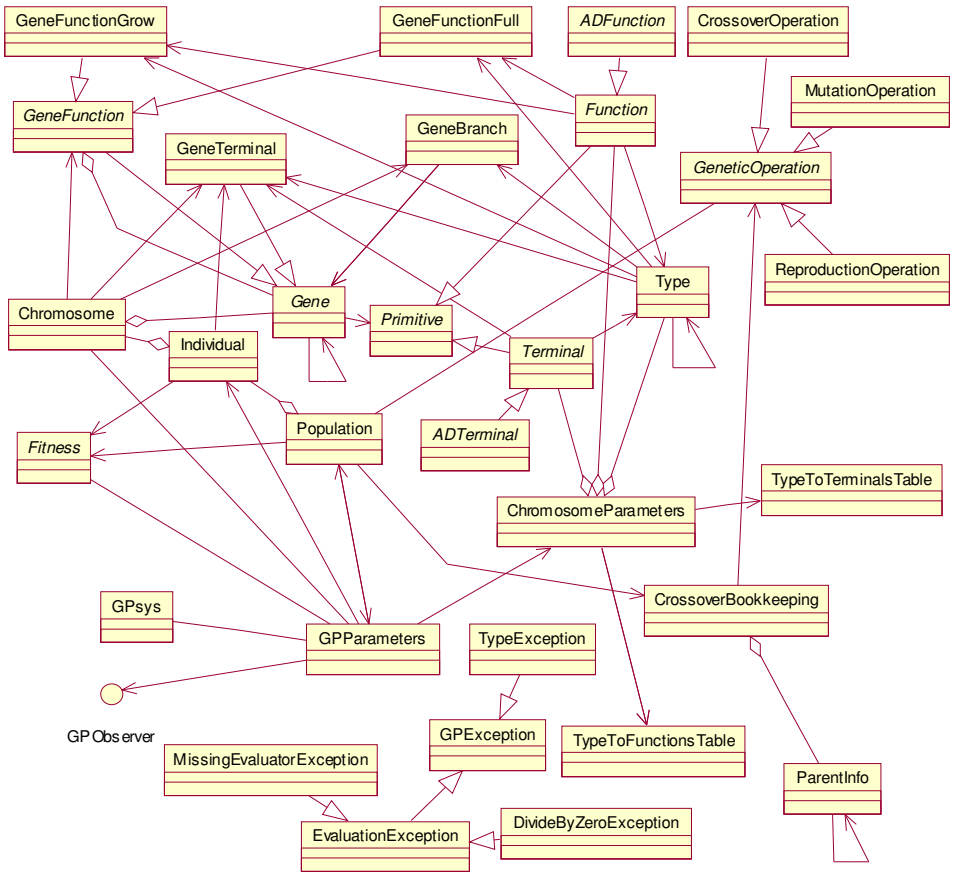
C.8 Establishing Fitness

The class Fitness is abstract, so must be extended as required for the application. Methods that must be declared are add, equals, greaterThan, lessThan and divide. These are all used for comparing different Fitness objects. The other compulsory method is instance, which will call the constructor of the concrete Fitness object. A generic fitness object is held by GPParameters, and this is referenced to create concrete objects belonging to Individuals. There is also a termination condition, which will stop the GP if satisfied, e.g. the fitness is over some threshold.

The actual method of calculating fitness is not specified as it depends on the specific application. As coevolution is not currently supported, the calculation of fitness should be

written into the Fitness constructor, as this means that a new Individual will have a calculated fitness as soon as it is created.

C.9 Class diagram

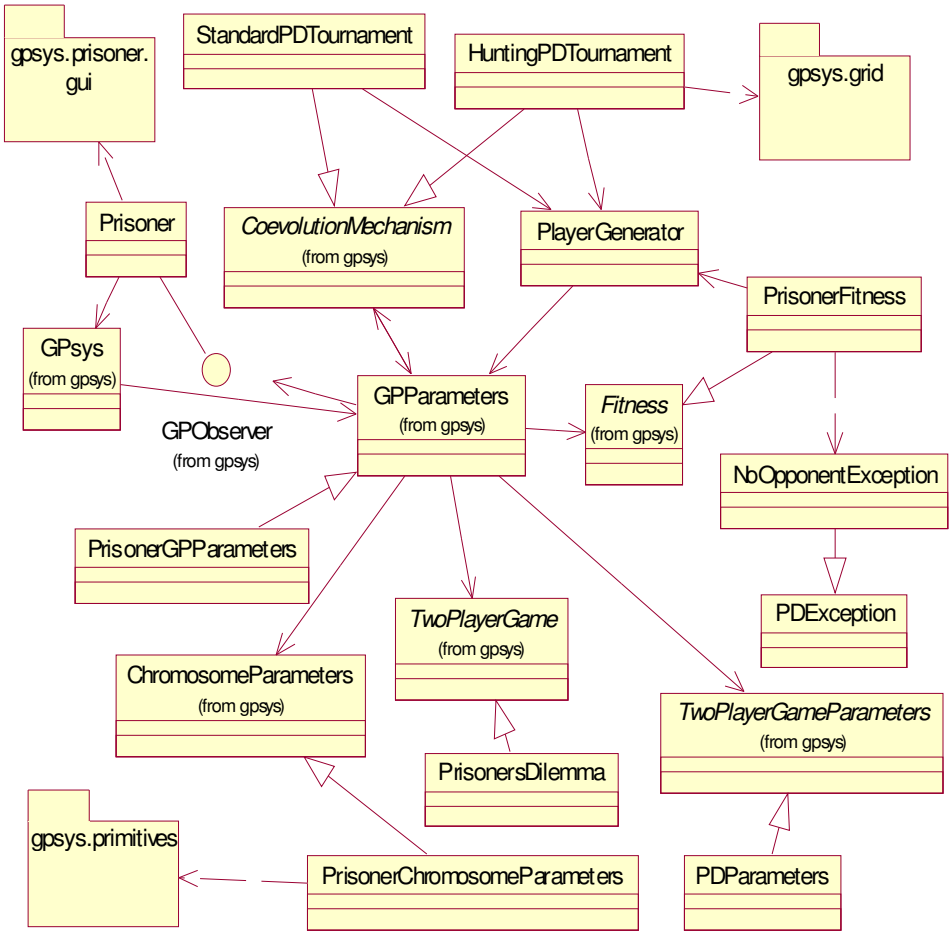


D Class and Sequence Diagrams

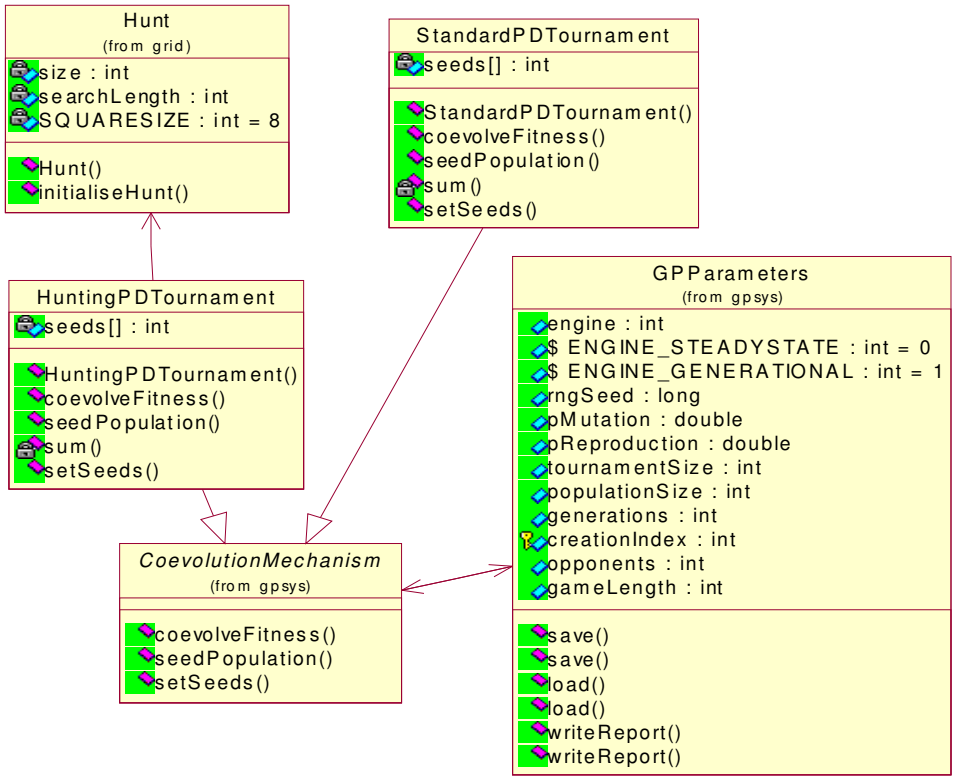
D.1 Class diagrams

Here are the final class diagrams for the new packages and package gpsys

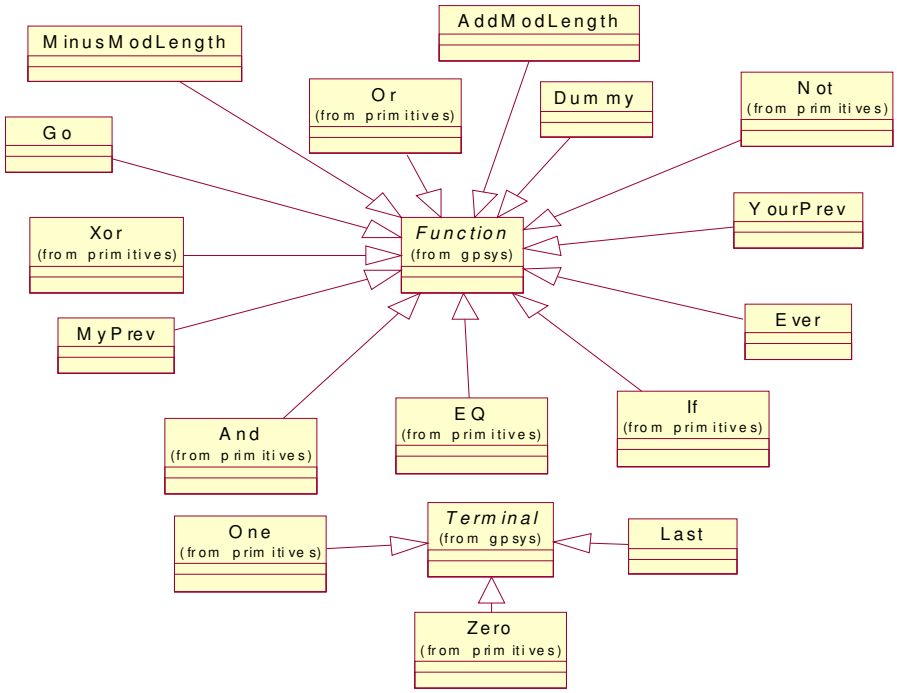
D.1.1 Package gpsys.prisoner



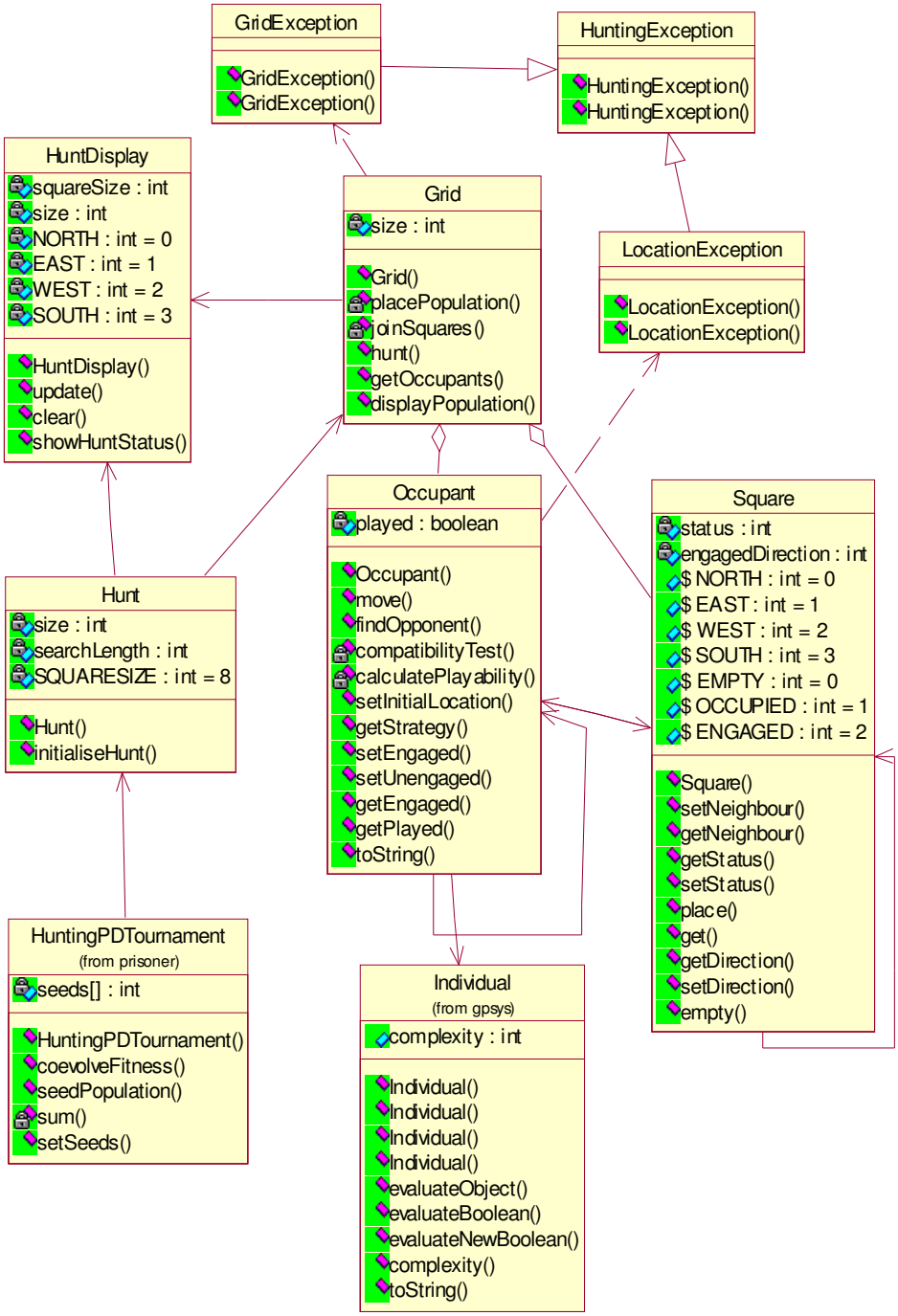
D.1.2 Interface between packages in setting up coevolution



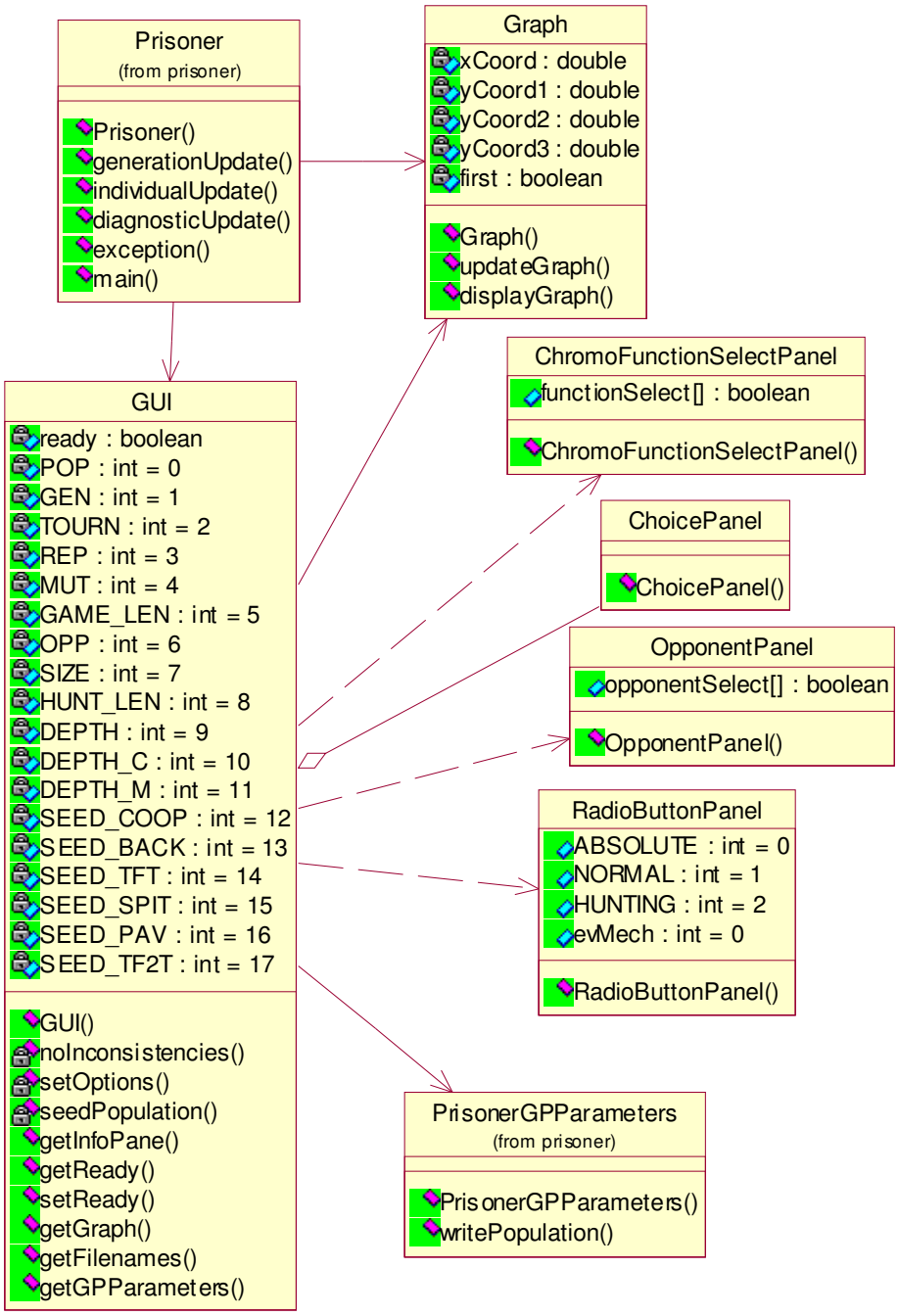
D.1.3 Packages gpsys.primitives and gpsys.primitives.prisoner



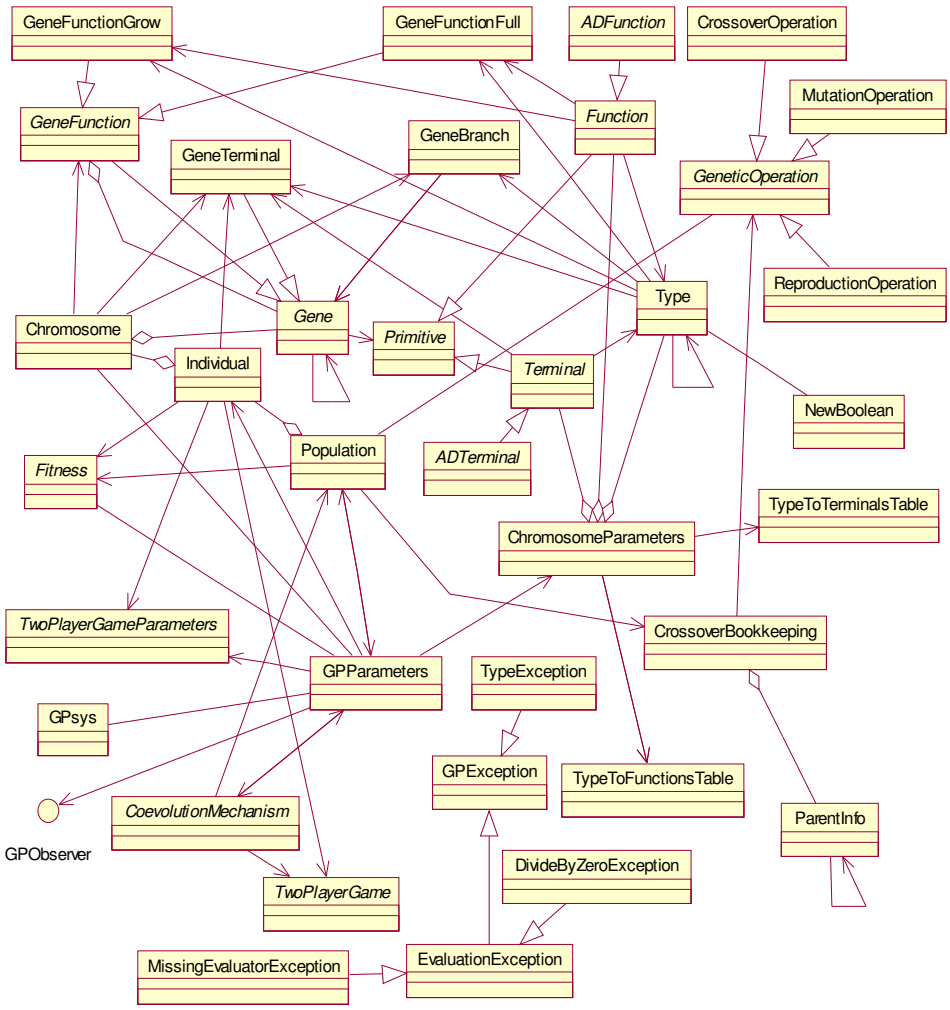
D.1.4 Package gpsys.grid



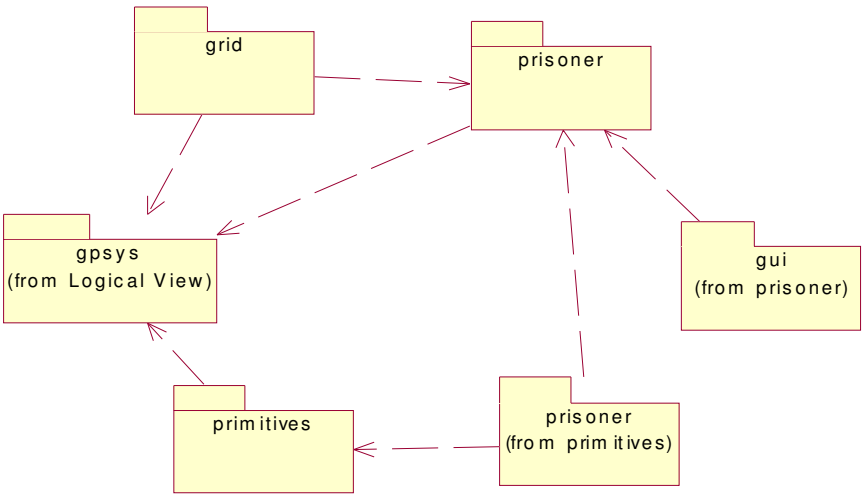
D.1.5 Package gpsys.prisoner.gui



D.1.6 Package gpsys after extensions



D.1.7 Package structure

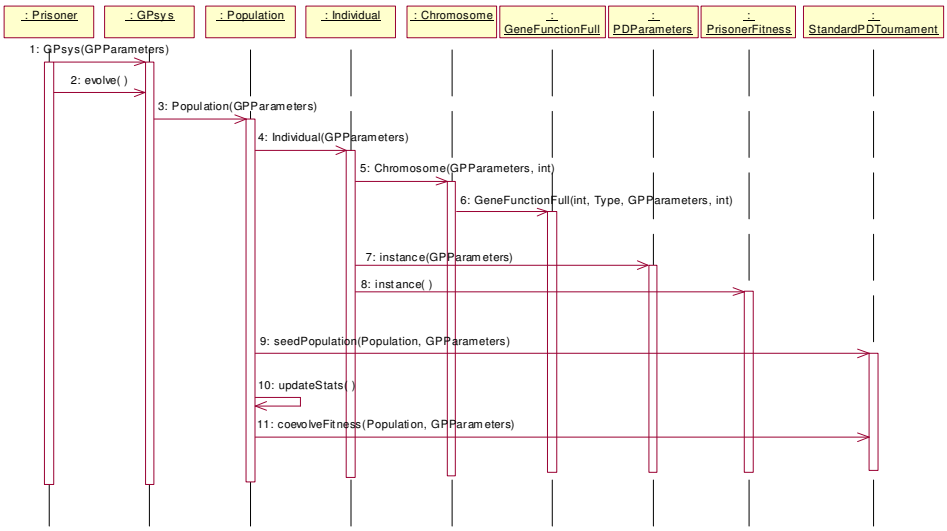


D.2 Sequence Diagrams

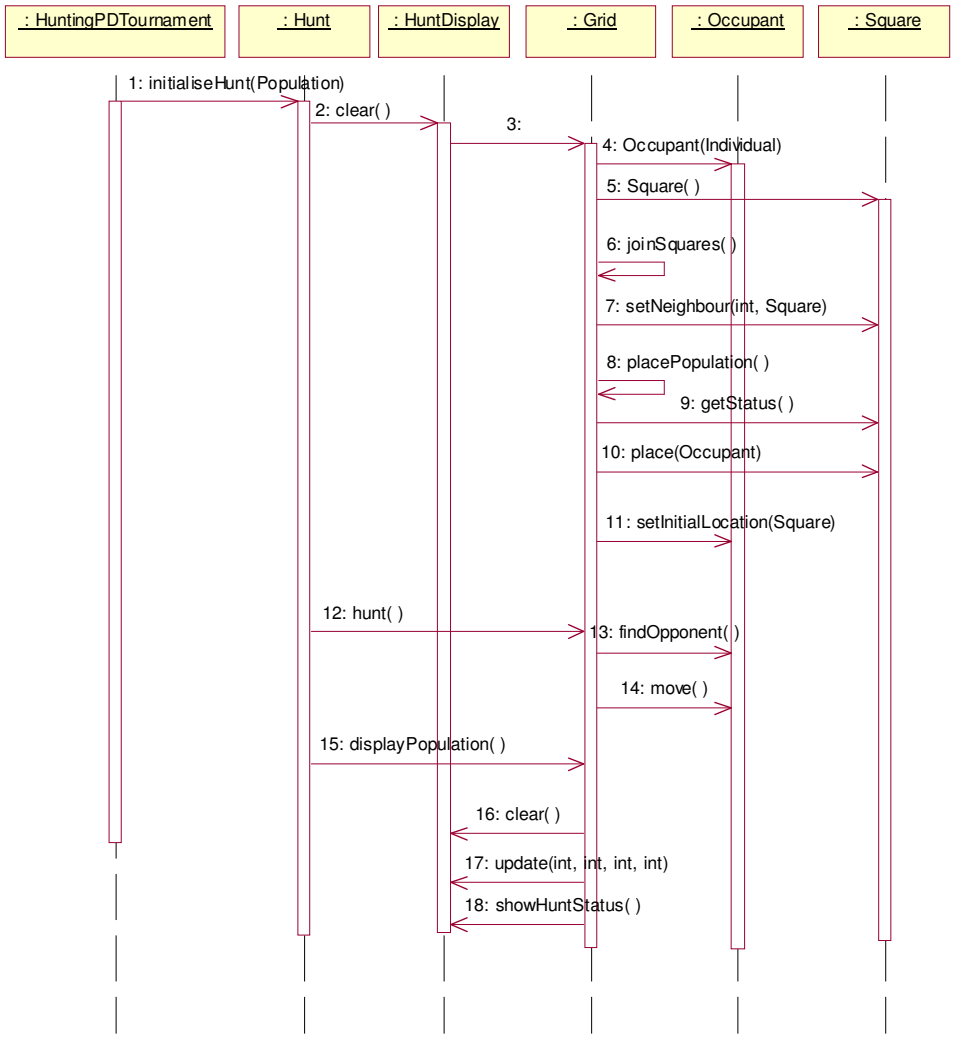
D.2.1 Evaluating an Individual

The Individual in question is (Or (YourPrev 0) (YourPrev 0)), i.e. Tit-For-Tat

D.2.2 Creating a new Population



D.2.3 Creating a new Hunt



E Source Code

Source code for the following classes have been included in this section:

From *gpsys.prisoner*

- HuntingPDTournament
- PDParameters
- Prisoner
- PrisonerChromosomeParameters
- PrisonerFitness
- PrisonerGPParameters
- PrisonersDilemma
- StandardPDTournament

From *gpsys.grid*

- Grid
- Occupant

From *gpsys.primitives.prisoner*

- Last
- MyPrev

It was felt that these classes are the most important ones for viewing purposes. Class GUI, whilst important, is long and has already been explained in §5

The source code of the remaining classes can be viewed on the disc by extracting the jar file (see Appendix B.2.1).

F Evolution Logs

F.1 Testing genetic operations

F.1.1 Crossover

Population 10 – Generation 1

(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev 0) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev 0) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev 0) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)

F.1.2 Mutation and Reproduction

Population 10 – Generation 1

(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev (AddModLength (MinusModLength Last 1) (AddModLength 1 Last))) (YourPrev Last)) Fitness(3.0,11,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)
(Or (YourPrev Last) (YourPrev Last)) Fitness(3.0,5,1.0,0.0,1.0) Hunt criteria(0.0,0.0,0.0)

F.2 Standard Coevolution

Population 100 – Generations 50 – Game length 50 - Last 3 generational reports

Generation 48 completed...
Date :- 05-Sep-01 00:04:16
Average fitness Fitness(2.7109300000000007,12,0.8382599999999997,0.6160000000000002,1.0)
Average complexity 12.4
best individual of generation =
(Xor (Ever 0) (Go (AddModLength (MinusModLength Last (MinusModLength 0 Last))
(MinusModLength Last 1))))
Fitness(3.1409999999999996,13,0.8639999999999999,0.9,1.0)
worst individual of generation =
(Xor (Go (AddModLength (MinusModLength Last 1) (MinusModLength Last 0))) (Go 1))
Fitness(0.535,11,0.9600000000000005,0.85,1.0)

Generation 49 completed...
Date :- 05-Sep-01 00:04:17
Average fitness Fitness(2.8204999999999996,12,0.8846899999999996,0.6349999999999999,1.0)
Average complexity 12.7
best individual of generation =
(Xor (Ever 0) (Go (AddModLength (MinusModLength Last 0) (MinusModLength Last 1))))
Fitness(3.511999999999999,11,0.6869999999999999,0.6,1.0)
worst individual of generation =
(Xor (Go (AddModLength (MinusModLength 1 1) (MinusModLength Last Last))) (Go Last))
Fitness(0.7389999999999997,11,0.9600000000000005,1.0,1.0)

Generation 50 completed...
Date :- 05-Sep-01 00:04:18
Average fitness Fitness(2.7132599999999987,12,0.8441499999999995,0.6184999999999996,1.0)
Average complexity 12.22
best individual of generation =
(Xor (Ever 0) (Go (AddModLength (MinusModLength Last 0) (MinusModLength Last 1))))
Fitness(3.049,11,0.9109999999999999,0.6,1.0)
worst individual of generation =
(Xor (Go (AddModLength (MinusModLength 1 0) (MinusModLength Last 0))) (Go (AddModLength
(MinusModLength Last 0) (MinusModLength Last 0))))
Fitness(0.7279999999999999,17,0.9600000000000005,1.0,1.0)

F.3 Hunting Coevolution

Population 100 – Generations 10 – Game Length 50 – Playability threshold 4.5
Last 3 generational reports

Generation 8 completed...
Date :- 05-Sep-01 00:11:49
Average fitness Fitness(2.796,17,1.0,0.0,0.9320000000000005)
Hunt criteria(0.7786999999999992,0.10879999999999988,2.450699999999996)
Average complexity 17.46
best individual of generation =
(Xor (Ever 0) (MyPrev (MinusModLength 0 0)))
Fitness(3.0,7,1.0,0.0,1.0) Hunt criteria(0.73,0.12,1.04)
worst individual of generation =
(Xor (Ever 0) (MyPrev 0))
Fitness(1.0500000000000003,5,1.0,0.0,0.35) Hunt criteria(0.25,0.12,2.48)

Generation 9 completed...
Date :- 05-Sep-01 00:12:06
Average fitness Fitness(2.714999999999999,14,1.0,0.0,0.9050000000000007)
Hunt criteria(0.7951999999999996,0.1658999999999998,2.422899999999997)
Average complexity 14.88
best individual of generation =
(Xor (Ever 1) (MyPrev (MinusModLength 0 Last)))
Fitness(3.0,7,1.0,0.0,1.0) Hunt criteria(0.97,0.12,2.48)
worst individual of generation =
(Xor (Ever 0) (MyPrev 1))
Fitness(0.15,5,1.0,0.0,0.05) Hunt criteria(0.73,0.92,2.48)

Generation 10 completed...
Date :- 05-Sep-01 00:12:22
Average fitness Fitness(2.784,9,1.0,0.0,0.9280000000000005)
Hunt criteria(0.8448999999999995,0.13549999999999984,2.5739999999999967)
Average complexity 9.23
best individual of generation =
(Xor (Ever 0) (MyPrev (MinusModLength Last 1)))
Fitness(3.0,7,1.0,0.0,1.0) Hunt criteria(0.97,0.25,3.27)
worst individual of generation =
(Xor (MyPrev (MinusModLength Last Last)) (MyPrev 1))
Fitness(2.2500000000000004,7,1.0,0.0,0.75) Hunt criteria(0.73,0.12,2.48)

F.4 Function Set test

Standard coevolution – Generation 0 -All functions included:

(EQ (YourPrev 0) (Go 1)) Fitness(2.5149999999999997,5,0.425,1.0,1.0)
(Not (YourPrev 0)) Fitness(2.8649999999999993,3,0.38499999999999995,1.0,1.0)
(Xor (MyPrev 1) (YourPrev Last)) Fitness(0.765,5,1.0,0.0,1.0)

(EQ (MyPrev 1) (YourPrev 1)) Fitness(2.9449999999999994,5,0.3,1.0,1.0)

(Not (Go 1)) Fitness(3.814999999999999,3,0.10000000000000005,1.0,1.0)

(Not (If (YourPrev Last) (Ever Last) (Go Last))) Fitness(3.55,8,0.10000000000000005,1.0,1.0)

(If (EQ (Or (MyPrev 1) (MyPrev 1)) (EQ (MyPrev Last) (MyPrev Last))) (Or (And (MyPrev 0) (Go 0)) (And (Go 0) (YourPrev 0))) (EQ (Ever (AddModLength Last 1)) (And (YourPrev Last) (Go 0))))

Fitness(1.9900000000000002,33,0.7300000000000002,1.0,1.0)

(Xor (And (EQ (YourPrev 1) (Go 1)) (Not (Go 0))) (Not (Or (YourPrev 1) (Go 0))))

Fitness(1.2650000000000001,16,0.9,0.15,1.0)

(And (Not (If (Ever (AddModLength Last 0)) (Not (Go 0)) (Or (YourPrev 1) (Go 0)))) (Xor (Not (Or (Go 0) (MyPrev 1))) (EQ (EQ (Ever 1) (MyPrev 1)) (Or (Ever Last) (YourPrev 0)))))

Fitness(1.78,33,0.7449999999999999,0.3,1.0)

(Xor (Xor (Xor (And (Xor (YourPrev Last) (MyPrev 0)) (EQ (YourPrev 1) (YourPrev Last))) (Or (Or (YourPrev 1) (Go Last)) (And (Go 1) (Go Last)))) (And (Xor (Go (MinusModLength 1 1)) (If (Go Last) (MyPrev 1) (Go 1))) (Xor (And (Go 1) (YourPrev Last)) (If (MyPrev 1) (Go Last) (Ever 0))))) (EQ (And (EQ (MyPrev (MinusModLength 1 Last)) (Not (MyPrev 1))) (EQ (Or (MyPrev Last) (Go Last)) (And (Go Last) (Go Last)))) (Xor (Or (Or (Ever 0) (Go 0)) (And (MyPrev 0) (Ever Last))) (Or (YourPrev (AddModLength 1 1)) (Or (YourPrev 1) (YourPrev 0))))))

Fitness(2.0650000000000004,94,0.725,0.35,1.0)

Standard coevolution – Generation 0 – If, EQ, Go not included:

(And (YourPrev 0) (MyPrev Last)) Fitness(2.4,5,1.0,0.0,1.0)

(And (YourPrev 1) (YourPrev 0)) Fitness(2.5499999999999994,5,1.0,0.0,1.0)

(Or (YourPrev Last) (MyPrev 1)) Fitness(2.6249999999999996,5,1.0,0.0,1.0)

(Or (Ever 1) (MyPrev 0)) Fitness(2.845,5,0.7999999999999999,0.0,1.0)

(Xor (Ever 1) (MyPrev Last)) Fitness(2.7950000000000004,5,0.8800000000000001,0.0,1.0)

(Xor (Or (MyPrev 0) (YourPrev Last)) (MyPrev (MinusModLength 0 0))) Fitness(2.7,10,1.0,0.0,1.0)

(Xor (MyPrev Last) (And (MyPrev (AddModLength Last Last)) (MyPrev 0)))

Fitness(2.625,10,1.0,0.0,1.0)

(Or (And (Or (YourPrev 1) (Ever 1)) (Not (MyPrev 1))) (Not (Or (MyPrev 0) (Ever 0))))

Fitness(2.7350000000000003,16,0.5,1.0,1.0)

(Not (Xor (Not (Not (YourPrev 0))) (Xor (Xor (MyPrev Last) (MyPrev Last)) (Or (MyPrev 0) (YourPrev 1))))) Fitness(3.499999999999999,17,0.41,1.0,1.0)

(Not (Not (And (MyPrev (AddModLength (AddModLength 0 Last) (MinusModLength 0 Last))) (Not (And (Ever Last) (YourPrev 0)))))) Fitness(2.7,17,1.0,0.0,1.0)

F.5 Seeding Population

Population 10 – Generation 1 – 5 TFT and 5 Backstabbing players seeded

Generation 0

(Not (YourPrev Last)) Fitness(1.2999999999999998,3,0.0,1.0,1.0)

(Not (YourPrev Last)) Fitness(1.2599999999999996,3,0.0,1.0,1.0)

(Not (YourPrev Last)) Fitness(1.22,3,0.0,1.0,1.0)

(Not (YourPrev Last)) Fitness(1.2599999999999998,3,0.0,1.0,1.0)

(Not (YourPrev Last)) Fitness(1.2599999999999993,3,0.0,1.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(1.4249999999999994,5,0.32499999999999984,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(1.7399999999999995,5,0.4599999999999998,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(2.055,5,0.5949999999999998,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(1.9499999999999997,5,0.5499999999999998,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(2.055,5,0.595,0.0,1.0)

Generation 1

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

(Or (YourPrev 0) (YourPrev 0)) Fitness(3.0,5,1.0,0.0,1.0)

G Bibliography – Books

[1] *Evolution of Cooperation* – Robert Axelrod – Basic Books, NY – 1984

[2] *Prisoner's Dilemma* – W.Poundstone – Anchor Books, Doubleday, NY – 1993

[3] *The Arithmetics of Mutual Help* – M.Nowak, R.M.May, K.Sigmund – Scientific American, p76-81 – June 1995

[4] *Genetic Programming – Computers using "Natural Selection" to generate programs (paper)* – William B.Langdon, Adil Qureshi – Dept. of Computer Science, University College London

[5] *Modelling Exchange Using the Prisoner's Dilemma and Genetic Programming (paper)* – Laurie Hirsch, Masoud Saeedi – Dept. of Computer Science, Sheffield Hallam University

[6] *Evolutionary Games and Population Dynamics* – Josef Hofbauer, Karl Sigmund - 1998 - Cambridge University Press

[7] *Theory of Games and Economic Behaviour* – John von Neumann, Oskar Morgenstern – 1944 – Princeton University Press

[8] *Games and Decisions* – R. Duncan Luce, Howard Raiffa – 1957 – John Wiley & Sons, Inc.

[9] *Some Topics in Two-Person Games* – T. Parthasarathy, T.E.S. Raghavan – 1971 – American Elsevier Publishing Company, Inc.

[10] *Developing Java Software, Second Edition* – R.Winder, G.Roberts – 2000 – John Wiley & Sons, Ltd.

[11] *The theory of games and the evolution of animal conflicts* – J.Maynard Smith – 1974 – Journal of Theoretical Biology

[12] *Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence* – J.Holland – 1992 – MIT press

[13] *Genetic Programming: On the Programming of Computers by Natural Selection* – J.R.Koza – 1992 – MIT press

[14] *Competitive environments evolve better solutions for complex tasks* – P.J.Angeline, J.B.Pollack – 1993 – Taken form proceedings of 5th International Conference on Genetic Algorithms, ICGA-93, p264-270 - Morgan Kaufmann

H Bibliography – Web links

[15] www.csse.monash.edu.au/~tonyj/GM3/gentic.html -Tony Jansen - Hunting implemented

[16] http://serendip.brynmawr.edu/playground/pd.html - On-line PD game

[17] www.brembs.net/ipd - Scholarly discussion of IPD

[18] www.patweb.com/game - Real social experiment with version of IPD

[19] www.genetic-programming.com - Home of GP

[20] www.esatclear.ie/~rwallace/lithos.html - Lithos GP engine

[21] www.stanford.edu/~jjchen/game.html - Very good PD discussion

[22] www.mk.dmu.ac.uk/~jmarshall/sipd/sipd1.htm - Spatialised IPD

[23] http://netrunners.mur.csu.edu.au/~osprey/prisoner.html - Loads of strategies

[24] www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html - GP engine used