Evolving Strategies for the Prisoner's Dilemma

Technical Manual

Andrew Errity
CACS4 99086921
Faculty of Computing and Mathematical Sciences
Dublin City University
aerrit-cacs4@computing.dcu.ie
http://www.computing.dcu.ie/~aerrit-cacs4/prisoner/

Supervisor: Dr. Alistair Sutherland

Contents

1 Introduction

The Prisoner's Dilemma is a traditional game model for studying decision making and self-interest. The objective of this project is to provide an environment in which strategies for the Prisoner's Dilemma can be evolved, analysed and interacted with. Moreover this project should provide a means of varying the evolutionary parameters and game rules and facilitate analysis of the results. The aim is to identify optimal strategies and the characteristics they share.

This project should allow the user to, via a graphical interface, view the evolution of strategies and study how a well performing strategy can be generated. This project should also provide a means of closely analysing the behaviour of strategies and comparing them with other strategies.

This project should be of interest to users from many fields. Game theorists may be able to utilise this project to analyse and further understand the Prisoner's Dilemma game. Social scientists may be interested in studying decision making, self-interest and the emergence of cooperation using this project. The evolutionary algorithms used in this project and their results may be of interest to those involved in Artificial Intelligence or Machine Learning. This project should also be a useful starting point for the casual user wishing to experiment with any of the above topics.

2 Background

2.1 Prisoner's Dilemma game

In the Prisoner's Dilemma game two players are faced with a choice, they can either cooperate or defect. Each player is awarded points (called payoff) depending on the choice they made compared to the choice of the opponent. Each player's decision must be made without knowledge of the other player's next move. The player's have no means of communication other than the game choices and there can be no prior agreement between the players concerning the game.

If both players cooperate they both receive a reward, R. If both players defect they both receive a punishment, P. If one player defects and the other cooperates, the defector receives a reward, T the temptation to defect, whilst the player who cooperated is punished with the *sucker's* payoff, S. These game rules (and their typical values [1, p.8]) are presented in the payoff matrix below [Table 1]

	Player B		
Player A		Cooperate	Defect
	Cooperate	R=3,R=3	S=0,T=5
	Defect	T=5,S=0	P=1,P=1

Table 1: Payoff Matrix

The dilemma here is that if both players defect, they both score worse than if both had cooperated.

Assume a rational player is faced with playing a single game (known as one-shot) of the Prisoner's Dilemma described above and that the player is trying to maximise their reward. If the player thinks his/her opponent will cooperate, the player will defect to receive a reward of 5 as opposed to cooperation which would have earned him/her only 3 points. However if the player thinks his/her opponent will defect, the rational choice is to also defect and receive 1 point rather than cooperate and receive the sucker's payoff of 0 points. Therefore the rational decision is to always defect.

But assuming the other player is also rational he/she will come to the same conclusion as the first player. Thus both players will always defect, earning rewards of 1 point rather than the 3 points that mutual cooperation could have yielded. Therein lays the dilemma. In game theory the Prisoner's Dilemma can be viewed as a two-player, non zero-sum and simultaneous game. Game theory has proved that always defecting is the dominant strategy for this game (the Nash Equilibrium) [2, p.147]. This holds true as long as the payoffs follow the relationship $T > R > P > S$, and the gain from mutual cooperation is greater than the average score for defecting and cooperating, $R > (S + T)/ 2$.

While this game may seem simple it can be applied to a multitude of real world scenarios. Problems ranging from businesses interacting in a market, personal relationships, super power negotiations and the trench warfare "live and let live" system of World War I have all been studied using some form of the Prisoner's Dilemma.

2.2 Iterated Prisoner's Dilemma

The Iterated Prisoner's Dilemma (IPD) is an interesting variant of the above game in which two players play repeated games of the Prisoner's Dilemma against each other. In the above discussion of the Prisoner's Dilemma the dominant mutual defection strategy relies on the fact that it is a one-shot game, with no future. The key to the IPD is that the two players may play each other again; this allows the players to develop strategies based on previous game interactions. Therefore a player's move now may affect how his/her opponent behaves in the future and thus affect the player's future payoffs. This removes the single dominant strategy of mutual defection as players use more complex strategies dependant on game histories in order to maximise the payoffs they receive. In fact, under the correct circumstances mutual cooperation can emerge.

The length of the IPD (i.e. the number of repetitions of the Prisoner's Dilemma played) must not be known to either player, if it was the last iteration would become a one-shot play of the Prisoner's Dilemma and as the players know they would not play each other again, both players would defect. Thus the second to last game would be a one-shot game (not influencing any future) and incur mutual defection, and so on till all games are one-shot plays of the Prisoner's Dilemma.

This project is concerned with modelling the IPD described above and devising strategies to play it. The fundamental Prisoner's Dilemma will be used without alteration. This assumes a player may interact with many others but is assumed to be interacting with them one at a time. The players will have a memory of the previous three games only (memory-3 IPD).

2.3 Research into the Prisoner's Dilemma

The political scientist Robert Axelrod pioneered research in this field in the late 1970's [1]. Axelrod was principally concerned with analysing the emergence of cooperation in populations of self-seeking egotists and used the Prisoner's Dilemma to represent his studies. As part of this research Axelrod held a series of computer tournaments in which IPD strategy entries from around the world played games of the IPD against one another in a round robin fashion. This tournament aimed to identify optimal strategies for the Prisoner's Dilemma (there is no *best* strategy; the success of a strategy depends on the other strategies present). The Tit For Tat strategy (TFT) won both computer tournaments conducted by Axelrod indicating that it is an optimal strategy. This strategy simply cooperates on the first move and then only defects if the other player defected on the previous move.

It is interesting to note that the TFT strategy can never obtain a higher score than its opponent. However this is unimportant as the Prisoner's Dilemma is a non-zero sum game, you do not have to do better than your opponent; you have to use your opponent to get a high score yourself.

Axelrod found that a strategy normally has four different characteristics to be successful in a Prisoner's Dilemma tournament:

- Nice – Cooperates on first move
- Retaliatory – will defect if defected against
- Forgiving – can be made to cooperate after starting to defect
- Clarity – don't be too complex

TFT possesses all of these characteristics. The strategies evolved by this project shall be examined to see if they exhibit these characteristics.

2.4 Genetic Algorithms

Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They were originally developed by John Holland at the University of Michigan.

They usually work by beginning with an initial population of random solutions to a given problem. The success of these solutions is then evaluated according to a specially designed fitness function. A form of 'natural selection' is then performed whereby solutions with higher fitness scores have a greater probability of being selected. The selected solutions are then 'mated' using genetic operators such as crossover and mutation. The children produced from this mating go on to form the next generation. The theory is that as fitter genetic material is propagated from generation to generation the solutions will converge towards an optimal solution. This project utilises Genetic Algorithms to evolve successful strategies for playing the Prisoner's Dilemma.

3 System Design

This section identifies the steps taken to construct the initial system design. This is not intended as a complete design document but rather a summary of the design process.

The Object Oriented Model was chosen for this system as it provides flexibility, modularity and maintainability all of which are vital when implementing a system with a limited timescale.

The Unified Modelling Language™ (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artefacts of software systems [3]. It simplifies the complex process of software design, making a "blueprint" for construction. UML was used throughout the design process to guarantee a clear and rigid application design.

3.1 System Requirements

The system requirements were clearly specified in the functional specification document. This document identified the following requirements of this project:

- Provide a means of evolving strategies to play variants of the IPD (possibly using Genetic Algorithms)
- Provide a means of testing and analysing the evolution process and the resulting strategies
- Provide a graphical interface allowing a user to interact with the above systems

Having identified these requirements the initial system design was created. It was proposed that the system have two distinct modes – strategy evolution and strategy analysis.

The strategy evolution mode was planned to allow the user to evolve strategies for two variants of the IPD – a round robin tournament (as used in Axelrod's computer tournament) and a more complex spatial society. The GUI for this mode would need to display statistics regarding the population percentages of the different strategy types and statistics regarding the values of payoffs being received. The GUI would also need to allow the user to control the evolution process.

The strategy analysis mode was envisaged as an area in which the user could play the Prisoner's Dilemma against evolved strategies and compare and contrast these strategies. The GUI for this area would require payoff statistics and game history information as well as controls to vary the analysis.

3.2 Graphical User Interface Design

The above system requirements highlight the importance of the GUI. This will provide the user's view of the system. The GUI design philosophies [4] of this project were:

- Simplicity – Keep the interface simple and straightforward. Ensure the interface is well organised and uncluttered.
- Obviousness – Make controls easy to find and intuitive. The effect of each control should be apparent from looking at the control.
- Safety – Protect users from making errors. Provide default values and prevent users from picking invalid commands or settings.

The system was proposed to have a master view point, with a menu bar from which rules settings and saved strategies could be configured [Figure 3-1]. These dialogs launched from the menu bar were not explicitly designed but the design philosophies were to be taken into consideration when implementing them. The menu bar would also control which type of GUI would be displayed in the master view – a strategy evolution window or a strategy analysis window. This menu bar would be accessible in either mode.

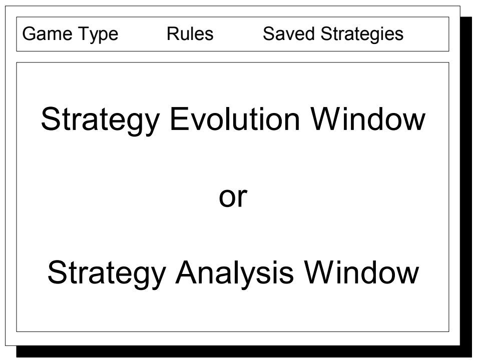


Figure 3-1: Master Window

3.2.1 Strategy Evolution

This window must display the strategy evolution of a tournament or spatial society (chosen from the aforementioned menu bar). To meet the described requirements this interface needed to display statistics regarding the population percentages of the different strategy types and statistics regarding the values of payoffs being received. The interface also needed to allow the user to control the evolution process and view/save strategies for later analysis. The proposed interface was as follows [Figure 3-2]:

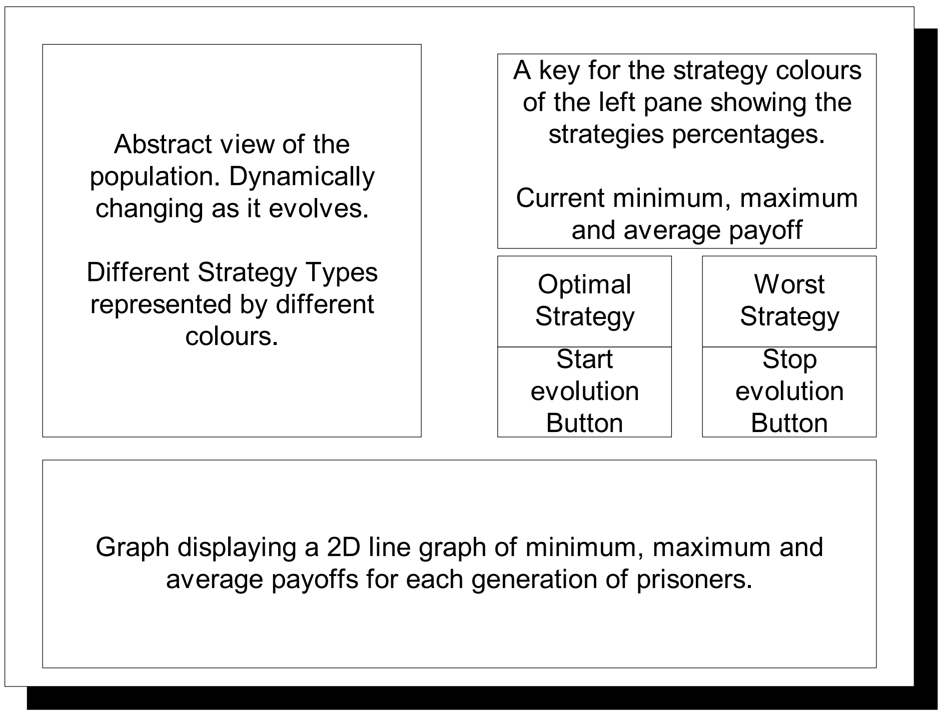

Abstract view of the population. Dynamically changing as it evolves.
Different Strategy Types represented by different colours.
A key for the strategy colours of the left pane showing the strategies percentages.
Current minimum, maximum and average payoff
Optimal Strategy
Worst Strategy
Start evolution Button
Stop evolution Button
Graph displaying a 2D line graph of minimum, maximum and average payoffs for each generation of prisoners.

Figure 3-2: Evolution Window

This interface aimed to provide the user with all of the data required without having to switch between displays. It provided very simple action controls –

- Starting a new population evolution
- Stopping a running evolution
- View/Save the best and worst strategies

3.2.2 Strategy Analysis

This window had to provide an environment in which the user could test strategies and analyse the results. To meet the requirements this window was required to show payoff statistics and game history information as well as controls to vary the analysis. A payoff matrix was added to this window in an attempt to aid the users understanding of the current Prisoner's Dilemma game rules. The proposed window design was as follows [Figure 3-3]:

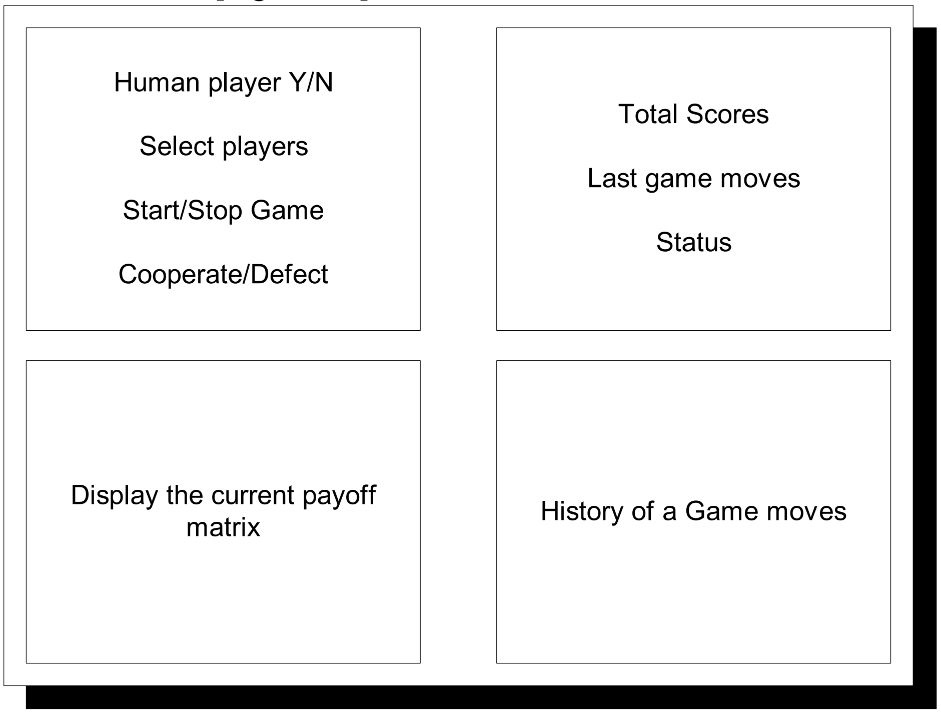


Figure 3-3: Strategy Analysis Window

3.3 System Analysis

Having recognized the system requirements above, the next task was to specify the logical components which would make up the system. The Object Oriented Model facilitated this and allowed the project to be broken into manageable, reusable modules.

The project was divided into three logical subsections.

- Prisoner's Dilemma components
- Genetic Algorithm components
- Graphical User Interface components

This would allow the system to be built in three distinct stages.

The basic components needed for each section were then identified. Prior to implementation the relationships of these objects was designed using a basic form of UML. The possible implementation details were omitted and only simple component relationships designed. This provided a simple top-level overview of the objects which needed to be written.

3.3.1 Prisoner's Dilemma – Object Design

This project subsystem has to provide the basic components to model the Prisoner's Dilemma. This section forms the projects foundation and its main objective is to provide a working IPD tournament with players of varying strategies competing. This subsystem can be further decomposed into several objects.
These objects include:

- Prisoner – A player with a strategy capable of playing the Iterated Prisoner's Dilemma
- Moves – Used to decode the players strategy to obtain next move
- Game – An interaction of the IPD between two players
- Rules – The payoffs and other game parameters
- Tournament – A round robin tournament in which every player plays every other player.

A basic UML overview of these objects' relationships is given below [Figure 3-4]

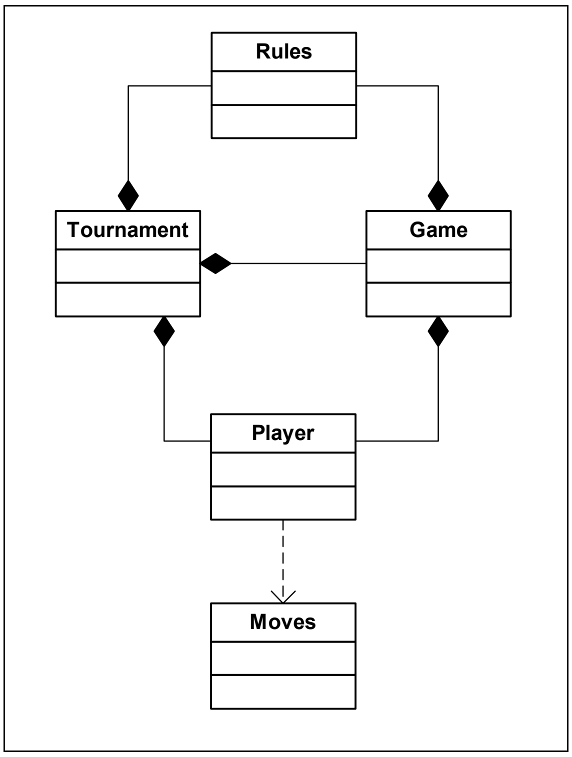


Figure 3-4: Prisoner's Dilemma UML

3.3.2 Genetic Algorithm - Object Design

This subsection has to provide functionality to allow evolution of the Prisoner's Dilemma objects. Its aim is to, coupled with the above subsystem, evolve optimal strategies for the IPD and report statistics regarding this evolution.
The three objects identified in this section are:

- Genetic – Provide Genetic algorithm utilities such as genetic operators
- Spatial – Provide functionality for evolving a spatial society
- Breeder – Provide a genetic algorithm to evolve a population

The relationship of these objects to the project is shown in the basic UML overview below [Figure 3-5]

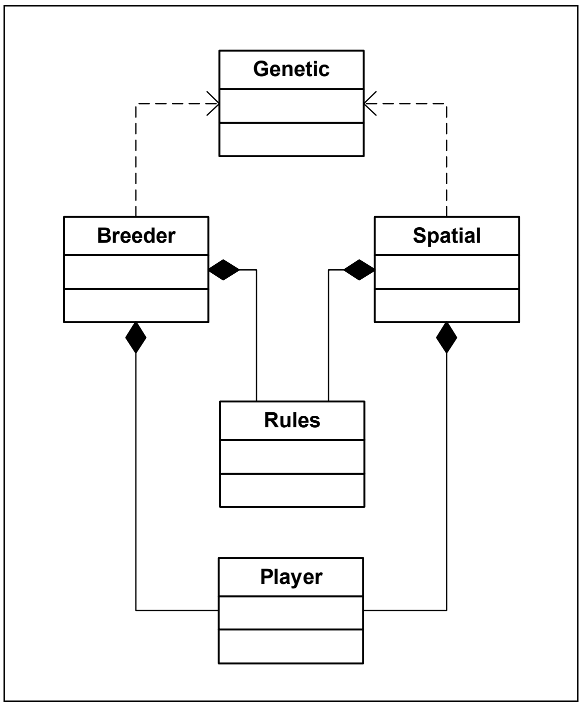


Figure 3-5: Genetic UML

3.3.3 Graphical User Interface - Object Design

Having designed the graphical user interface above it was necessary to identify the logical subcomponents of this GUI. This subsystem acts as a middle man between the system internals and the user. Its purpose is to present the system to the user and allow the user to perform system operations.
The components of this subsystem include:

- Menu Frame – a master frame which will hold all of the different views. This should provide a menu bar allowing the user to switch between game modes, configure parameters and manage strategies.
- Tournament Panel – The tournament game mode GUI described above.
- Spatial Panel – The spatial game mode GUI described above.
- Interactive Panel – The interactive game mode GUI detailed above.
- Rules Dialog – a dialog box allowing the user to set game parameters.
- Strategy Dialog - a dialog box allowing the user to manage strategies.
- Graph Panel – an object which displays information as a line graph
- Pay Off Panel – an object displaying the current payoff matrix

The UML overview below [Figure 3-6] displays a top-level view of the objects which make up the GUI system and how these objects inter-relate and relate to the other subsystem components. Several smaller objects from the GUI subsystem have been omitted for clarity.

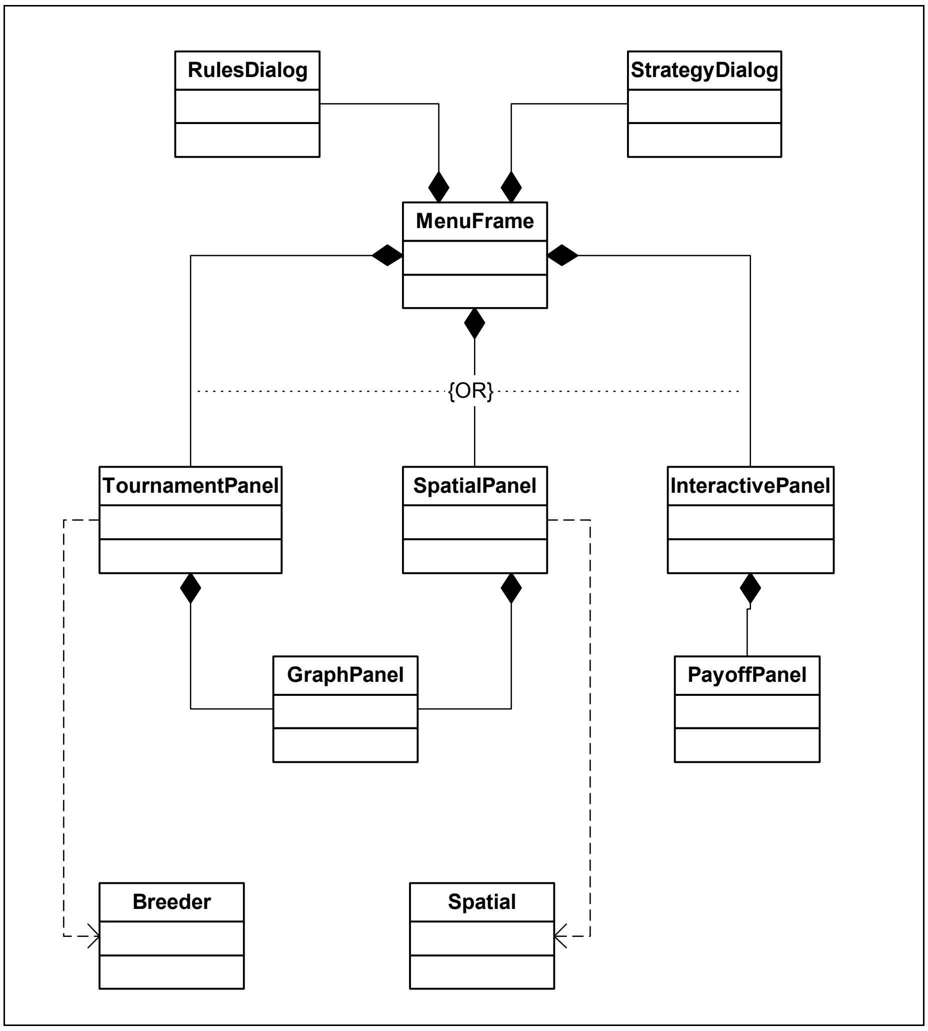


Figure 3-6: GUI UML

3.3.4 System Overview

This analysis provides a clear overview of the objects which are necessary to fulfil the system requirements. The following UML diagram describes the full system design (several smaller objects from the GUI subsystem have been omitted for clarity) [Figure 3-7].

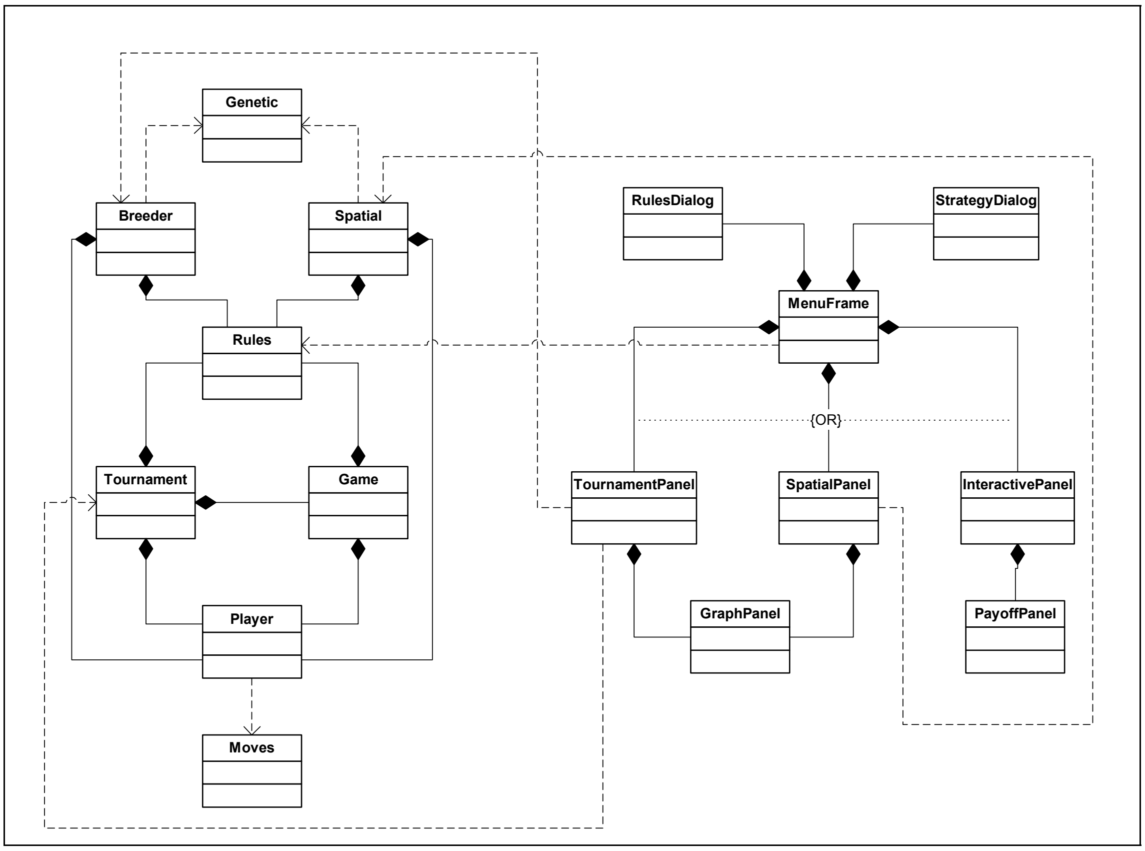


Figure 3-7: System UML Overview

4 System Implementation

This section will outline how the system design described above was implemented. It is not intended as a complete description of the project code; however it will detail the more important algorithms and techniques used to realize the functional specification. For more detail on any of the implementation specifics please see the full source listing and API documentation available at http://www.computing.dcu.ie/~aerrit-cacs4/prisoner/.

4.1 Implementation Language

Java version 1.4.1 [5] was chosen as the implementation language for this project. Java has a number of properties which suited this project:

- Familiar to the programmer – thus faster and less error prone implementation
- Object Oriented – necessary to implement the system design
- Platform independent – Java code will run on any platform that supports Java without modifying the code
- Swing [6] – built-in GUI libraries which guarantee the same look and feel on all platforms

The obvious disadvantage with Java is that it is slower than other standard Object Oriented languages such as C++. However the advantages of using Java were deemed to outweigh this consideration.

The project is organized in a package hierarchy (based on the subsystems identified during the system design) to improve readability and maintainability.

4.2 Genetic Algorithm

Genetic Algorithms provide the means by which strategies for the Prisoner's Dilemma are evolved in this project. As this is the principal objective of the project, naturally the genetic algorithm used is one of the systems major components. The other system components have been designed to suit the Genetic Algorithm. As such, in describing the genetic algorithms implementation most of the other components will also be described. What follows is a description of how a genetic algorithm was implemented to evolve strategies to play the Iterated Prisoner's Dilemma.

4.2.1 Strategy Encoding

One of the key issues in genetic algorithms is how to represent the problem. This issue is very important as genetic algorithms work by directly manipulating a coded representation of the problem. Conventionally the candidate problem solutions are encoded as fixed length strings which may be manipulated using standard genetic operators. A solution's representation within a Genetic Algorithm is called a chromosome.

In this project each chromosome needed to be a strategy for playing the memory-3 IPD. For each play of the Prisoner's Dilemma there are four possible outcomes (CC, CD, DC or DD), so in a memory-3 game this indicates 64 (4^3) possible histories. Thus a string of length 64 could be used to represent the action to take (C or D) following each of the game histories. This just leaves the first three moves, those without a three game history, to encode. Axelrod [7], who first suggested this scheme, encoded an assumption of the pre-game history of each player as 6 extra bits in the chromosome, to deal with the first three moves.

The chromosome used in this project uses a slight variation of the Axelrod encoding scheme. A 64 bit string is used to represent the action to take (1 meaning cooperate [C], 0 meaning defect [D]) following each possible game history. However a different means of encoding the initial 3 games is used. Rather than encode an assumption of the pre-game history, 7 additional bits are used to encode the actions to take for the first three moves relative to the opponents move. The first bit representing the first move to play (C or D), the next two bits indicating what second move to play depending on whether the opponent cooperated or defected, respectively, on the first move. Bits 4, 5, 6 and 7 of the pre-game information indicate what third move to play based on the opponents first two moves.

This 7 bit pre-game encoding is pre-pended to the 64 bit encoding scheme to give a 71 bit chromosome. The full encoding scheme is detailed below [Table 2]. The history in Table 2 is stated in the order Your first move, Opponents first Move, Your second move, Opponents second Move, Your third move, Opponents third Move. The move column indicates what move to play for the given history. Table 2 also shows how the TFT strategy is encoded using this scheme. The resulting TFT chromosome is:
CCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD
CDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDC
DCDCDCDCDCDCDCDCD
This is actually stored as a `BitSet` in Java as follows:
110101010101010101010101010101010101010101010101010101010101010101010101010101010
10101010101010101010101010101010101010101010101010

Bit	History	Move
0	First Move	C
1	Opponent C	C
2	Opponent D	D
3	Opponent CC	C
4	Opponent CD	D
5	Opponent DC	C
6	Opponent DD	D
7	C C C C C C	C
8	C C C C C D	D
9	C C C C D C	C
10	C C C C D D	D
11	C C C D C C	C
12	C C C D C D	D
13	C C C D D C	C
14	C C C D D D	D
15	C C D C C C	C
16	C C D C C D	D
17	C C D C D C	C
18	C C D C D D	D
19	C C D D C C	C
20	C C D D C D	D
21	C C D D D C	C
22	C C D D D D	D
23	C D C C C C	C
24	C D C C C D	D
25	C D C C D C	C
26	C D C C D D	D
27	C D C D C C	C
28	C D C D C D	D
29	C D C D D C	C
30	C D C D D D	D
31	C D D C C C	C
32	C D D C C D	D
33	C D D C D C	C
34	C D D C D D	D
35	C D D D C C	C
36	C D D D C D	D
37	C D D D D C	C
38	C D D D D D	D
39	D C C C C C	C
40	D C C C C D	D
41	D C C C D C	C
42	D C C C D D	D
43	D C C D C C	C
44	D C C D C D	D
45	D C C D D C	C
46	D C C D D D	D
47	D C D C C C	C
48	D C D C C D	D
49	D C D C D C	C
50	D C D C D D	D
51	D C D D C C	C
52	D C D D C D	D
53	D C D D D C	C
54	D C D D D D	D
55	D D C C C C	C
56	D D C C C D	D
57	D D C C D C	C
58	D D C C D D	D
59	D D C D C C	C
60	D D C D C D	D
61	D D C D D C	C
62	D D C D D D	D
63	D D D C C C	C
64	D D D C C D	D
65	D D D C D C	C
66	D D D C D D	D
67	D D D D C C	C
68	D D D D C D	D
69	D D D D D C	C
70	D D D D D D	D

Table 2: Prisoner Chromosome

Each `Prisoner` object within the system contains one of these `chromosome` strings. When given a game history and asked to make a game move the `Prisoner` object decodes the chromosome (using a `Moves` object) and replies with the appropriate move.

4.2.2 Fitness Function

The next problem faced when designing a Genetic Algorithm is how to evaluate the success of each candidate solution. The Prisoner's Dilemma provides a natural means of evaluating the success, or fitness, of each solution – the game payoffs. These payoffs are stored in `Rules` objects within the system. We can state that the strategy which earns the highest payoff score according to the rules of the IPD is the fittest, while the lowest scoring strategy is the weakest. Thus fitness can be evaluated by playing the `Prisoner` objects in some form of IPD. These IPD competitions (and evolution) are organized in two different ways in this project, Tournament and Spatial. This section shall discuss the more straightforward IPD tournaments leaving spatial societies to a later section.

The `Game` object was implemented to play a game of IPD for a specified number of rounds between two players. This object simply keeps track of two `Prisoner`'s scores and game histories while asking them for new moves until the rounds limit is met. The `tournament` object uses this class to organise a round robin IPD tournament, akin to Axelrod's [1] computer tournaments. In such a `tournament` an array of `Prisoner`'s is supplied as the population and every `Prisoner` plays an IPD `Game` against every other `Prisoner` and themselves. Each player's payoff after these interactions have completed is deemed to be the player's fitness score.

4.2.3 Fitness Scaling

The fitness scores calculated above will serve to determine which players go on to reproduce and which players 'die off'. However these 'raw' fitness values present some problems. The initial populations are likely to have a small number of very high scoring individuals in a population of ordinary colleagues. If using fitness proportional selection, these high scorers will take over the population rapidly and cause the population to converge on one strategy. This strategy will be a mixture of the high scorers' strategies, however as the population did not get time to develop these strategies may be sub-optimal, the population will have converged prematurely.

In the later generations of evolution the individuals should have begun to converge on a strategy. Thus they will all share very similar chromosomes and the populations average fitness will likely be very close to the population's best fitness. In this situation average members and above average members will have a similar probability of reproduction. In this situation the natural selection process has ended and the algorithm is merely performing a random search among the strategies.

It is useful to scale the 'raw' fitness scores to help avoid the above situations. This project uses linear scaling as described by Goldberg [8, p.76]. Linear scaling [9, p.72] produces a linear relationship between raw fitness, *f*, and scaled fitness, *f*`, as follows:

$$f' = af + b$$

Coefficients *a* and *b* may be calculated as follows:

$$a = (c-1) * \frac{f_{avg}}{f_{\max} - f_{avg}}$$

$$b = f_{avg} * \frac{\left(f_{\max} - (c * f_{avg})\right)}{f_{\max} - f_{avg}}$$

Where *c* is the number of times the fittest individual should be allowed to reproduce. A value of 2 was found to produce accurate scaling in this system. The effect of this fitness scaling is shown in Figure 4-1.

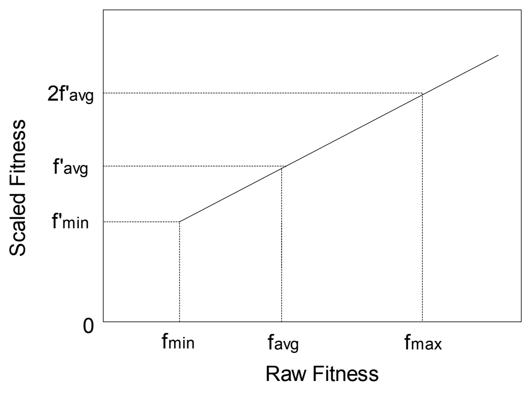


Figure 4-1: Linear Fitness Scaling

This scaling works fine for most situations, however in the later stages of evolution when there are relatively few 'low' scoring strategies problems may arise. The average and best scoring strategies have very close raw fitness and extreme scaling is required to separate them. Applying this scaling to the few low scorers may result in them becoming negative [Figure 4-2].

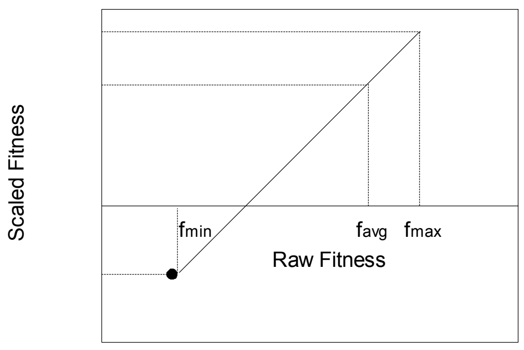


Figure 4-2: Linear Fitness Scaling Negative Values

This can be overcome by adjusting the scaling coefficients to scale the weak strategies to zero and scale the other strategies as much as is possible. In the case of negative scaled fitness values the coefficients may be calculated as follows:

$$a = \frac{f_{avg}}{f_{avg} - f_{\min}}$$

$$b = -f_{\min} * \left(\frac{f_{avg}}{f_{avg} - f_{\min}} \right)$$

This scaling helps prevent the early dominance of high scorers, while later on distinguishes between mediocre and above average strategies. It is implemented in the `Genetic` object and applied to all raw fitness scores (i.e. the IPD payoffs) before performing genetic algorithm selection.

4.2.4 Fitness Proportional Selection

At this point a means of encoding strategies, playing them against one another and finding their fitness has been described. This section will detail the fundamental theorem of Genetic Algorithms – selection. The Genetic Algorithm (`Breeder` and `Spatial` class) starts with a population of random chromosomes, evaluates their fitness, scales this fitness and then must choose two strategies (`Prisoners`) to reproduce. This selection process is based on the Darwinian principle of natural selection – the survival of the fittest. The higher the fitness score of a strategy the greater the probability of it being selected and permitted to reproduce. Thus the genetic material of the lower scoring strategies will be gradually weeded out of the population and replaced with the genetic material of the fitter strategies.

Fitness proportional selection was implemented in this project using roulette-wheel selection [10, p.43]. The principle of this is a search through a roulette wheel with each slot representing a member of the population and each slot weighted according to the player's fitness. This was implemented by first setting a target value, a random proportion of the sum of all population fitness's. The population is then stepped through summing the individual fitness's until the target value is exceeded. The individual who caused the target to be exceeded is chosen for selection. The next selection then proceeds from the position of the last selection with a new target to select a mate for the first individual. The last individual chosen is excluded from the search so an individual cannot mate with themselves.

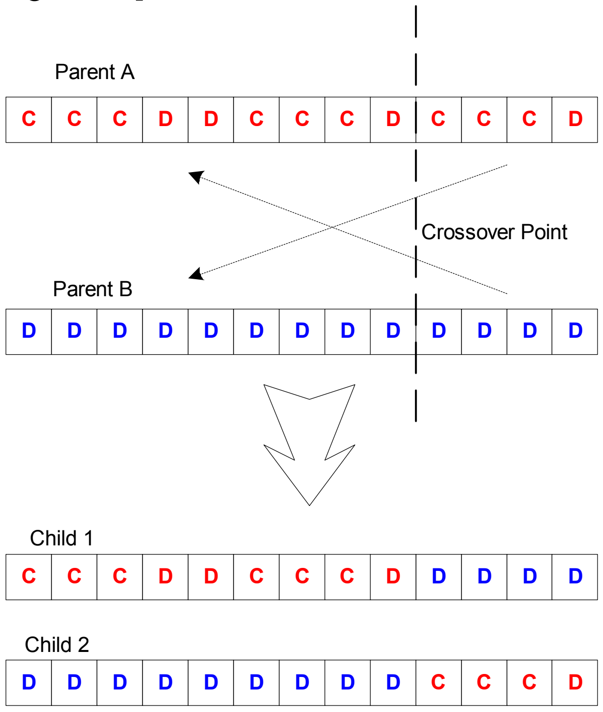
While this scheme is fitness proportional and will mostly select individuals with above average fitness it is not guaranteed to exclusively select fit individuals. In fact it will select 'weak' individuals some of the time. This is an extremely important facet of the project. If the selection scheme were to only select fit individuals the genetic material in the population would rapidly converge on a solution. This can be defined as exploitation, making the most of the knowledge gained. However if the original, fit individuals did not possess the genetic material (chromosome segments) necessary to reach the optimal solution they would converge upon a local optima. Using a heavily 'fit'-biased selection technique the genetic algorithm would be relying on mutation (discussed below) to introduce new genetic material and break the population from its local optima. The rarity of mutations and small effect which they have may cause the population to get trapped on the local optima.

To avoid this premature convergence exploration of the search space must be encouraged. This involves searching with new chromosomes (new strategies), the fitness of which are, as yet, unknown. One source of new genetic material used in this program is allowing weak strategies to reproduce occasionally. This mixes their genetic material with the fit strategies preventing premature convergence. In the later stages of evolution this exploration naturally lessens as strategies converge and exploration gives way to exploitation. This balance of exploration and exploitation is one of the key problems that must be addressed in many forms of Artificial Intelligence and Machine Learning systems.

4.2.5 Reproduction

Having selected two strategies from the population the Genetic Algorithm proceeds to mate these two *parents* and produce their two *children*. This reproduction is a mirror of sexual reproduction in which the genetic material of the parents is combined to produce the children. In this project reproduction allows exploration of the search space and provides a means of reaching new and hopefully better strategies. Reproduction is accomplished using two simple yet effective genetic operators – crossover and mutation, both implemented in the `Genetic` class.

Crossover is an artificial implementation of the exchange of genetic information that occurs in real-life reproduction. It was implemented in this system by breaking both the parent chromosomes at the same randomly chosen point and then rejoining the parts [Figure 4-3]



Parent A
C C C D D C C C D C C C D
Crossover Point
Parent B
D D D D D D D D D D D D D
Child 1
C C C D D C C C D D D D D
Child 2
D D D D D D D D D C C C D

Figure 4-3: Crossover

This crossover action, when applied to strategies selected proportional to their fitness, constructs new ideas from high scoring building blocks. The genetic algorithm implemented in this project performs crossover a large percentage of the time, however occasionally (5% of the time by default) crossover will not be performed and simple natural selection will occur.

In nature small mutations of the genetic material exchanged during reproduction occurs a very small percentage of the time. However if these mutations produce an advantageous result they will be propagated throughout the population by means of natural selection. The possibility of small mutations occurring was included in this system. A very small percentage of the time (0.1% of the time by default) a bit copied between the parent and the child will be flipped, representing a mutation. These mutations provide a means of exploration to the search.

4.2.6 Replacement

The genetic algorithm is run across the population until it has produced enough children to build a new generation. The children then replace all of the original population. More complicated replacement techniques such as fit-weak and child-parent replaced were researched but they were unsuitable for the round robin tournament nature of the system.

4.2.7 Search Termination

The only termination criteria implemented is a limit to the maximum number of generations that will run; this may be set by the user. Other termination criteria were investigated, for example detecting when a population has converged and strategies are receiving equal payoffs, however these criteria resulted in many false positives and it was decided better to allow the user to judge when the algorithm had reached the end of useful evolution.

4.2.8 Genetic Algorithm Flowchart

The following flowchart describes the completed genetic algorithm [11 p.29] [Figure 4-4].

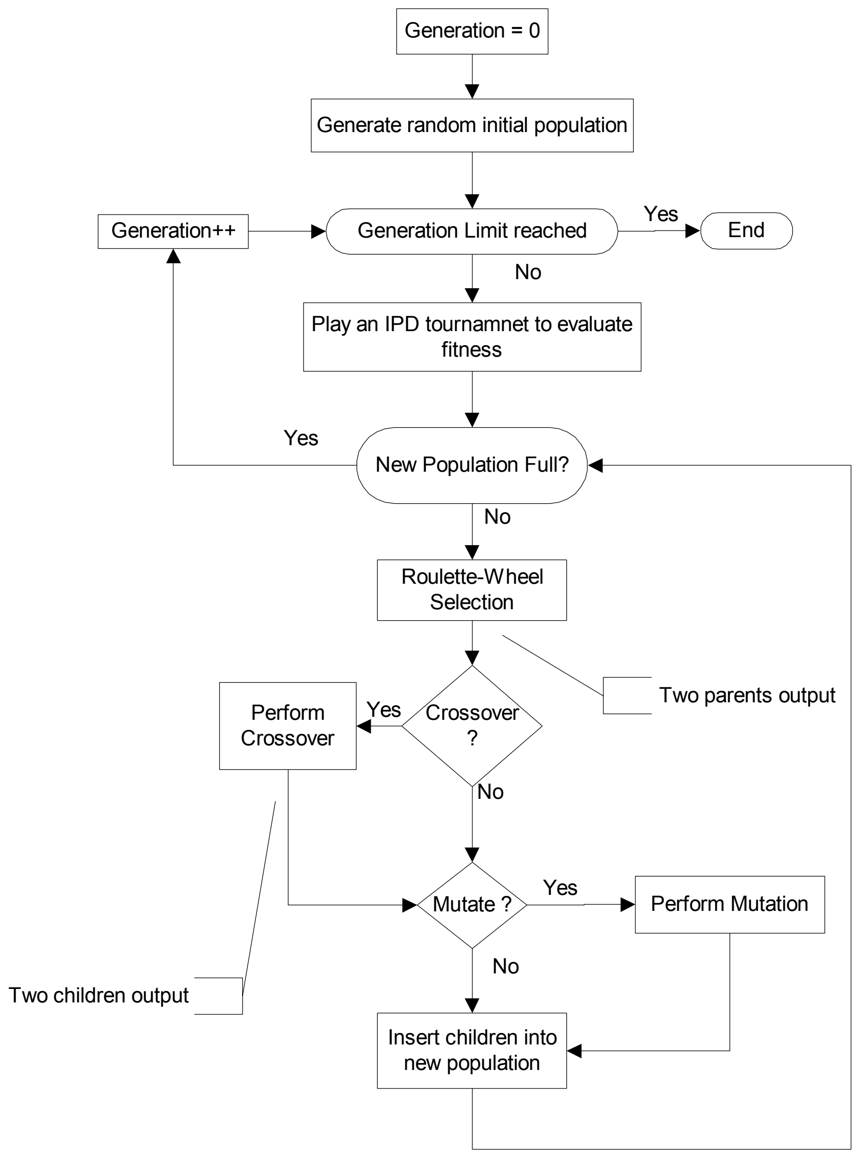


Figure 4-4: Genetic Algorithm flowchart

4.3 Spatial Society

4.3.1 Concept and Implementation

As a secondary experiment, the concept of a population of agents distributed on a space was considered. This space is a grid with agents only interacting with their eight immediate neighbours [Figure 4-5] in games of the IPD. Each agent only occupies one grid cell. This would allow study of the Prisoner's Dilemma and Genetic Algorithms with geographical implications (namely distance) taken into consideration.

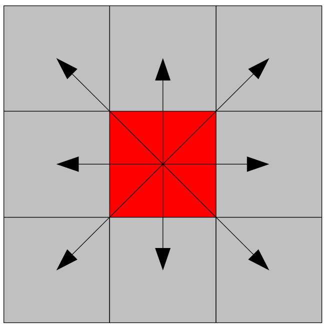


Figure 4-5: Spatial Interactions

The grid was programmed as a two dimensional array with overlapping edges (the `Spatial` object); this meant that every agent on the grid had eight immediate neighbours, even those at the extreme edges of the grid. This grid layout was used to form a society of Prisoners interacting in games of the IPD. Each Prisoner would play there eight immediate neighbours and the payoffs they received would represent their fitness.

The calculation of these payoffs could be quite slow for large populations of Prisoners playing long rounds of the IPD. For example a grid of 100 rows and columns would hold 10,000 prisoners. If each prisoner played its eight neighbours in 100 iterations of the Prisoner's Dilemma 8 million games of Prisoner's Dilemma would have to be played. In order to reduce the time taken for such a grid to initialise a memory system was introduced to the grid whereby if two players had already played the IPD against one another they would not play again but rather there previous score would be recalled from memory. This drastically improved performance, halving the number of Prisoner's Dilemma games required.

4.3.2 Spatial society and the Genetic Algorithm

A number of small alterations to the above Genetic Algorithm were required to apply it to the spatial grid. The fitness proportional selection previously used simply selected two 'fit' parents from anywhere in the population. This is panmictic mating [12, p.58] where partners are chosen solely on the basis of performance. This would violate the isolation by distance created by the spatial grid; that Prisoners can only interact with their immediate neighbours. To enforce the concept of distance within the grid the Genetic Algorithm selection scheme was modified so that the first parent is selected using roulette wheel selection and that Prisoner then chooses its fittest neighbour to mate with. Reproduction then proceeds as normal.

The spatial grid requires a different replacement procedure than the non-overlapping populations of the tournament model. Rather than rebuild a completely new population, the children generated from each Genetic Algorithm run are placed back into the population (an overlapping population). Each parent identifies their weakest neighbour and the parent's child then replaces this weak neighbour. Fitness scores are then recalculated for the children and the surrounding Prisoners (rather than the time consuming recalculation of all fitness scores).

Thus the genetic algorithm can be applied to a spatial environment. The genetic algorithm should be able to maintain several possible solutions in the population. Individuals containing the genes of each population will tend to congregate together with individuals on the edge providing a smooth transition of genotype. These congregations may flourish and dissipate or may achieve stability [12, p.63].

The components used to construct the Genetic Algorithm were also used to construct a simpler, Evolutionary Algorithm for the spatial grid. This differs from the Genetic Algorithm in that it does not use the genetic operator of crossover; its only means of introducing new genetic material is using mutation. Also its means of selection and replacement vary from the above Genetic Algorithm implementation. A random Prisoner is chosen at random from the population; this Prisoner finds its fittest neighbour and is replaced by it (with a small possibility of mutation occurring). This represents a more extreme view of Darwinian survival of the fittest than the Genetic Algorithm and should provide an interesting comparison to it.

Some experimentation [13] [14] with spatially distributed agents playing the IPD has seen the same interesting spatio-temporal patterns that many cellular automata display emerge. Depending on the parameters used various patterns have been found to emerge, including frozen evolution, frozen areas of defection and cooperation and more. The populations evolved using the Genetic and Evolutionary Algorithm will be examined to see if they display any of these patterns.

4.4 Graphical User Interface

The graphical user interface was implemented using the Java Foundation Class SWING [6]. The implementation of this GUI closely followed the design described previously [see above 3.2] this section will discuss how some of the important components were implemented but will omit the numerous smaller components which make up the complete application GUI. This section is not intended as a user manual, for that please consult Appendix A.

4.4.1 Frame and Menu bar

An object (`MenuFrame`) was implemented as the master frame for the application. This object contains the `Tournament` Evolution object, `Spatial` Evolution object, game `Rules` object and saved strategies. These objects all exist within the `MenuFrame` object; this gives the `MenuFrame` the ability to switch between the different object's views without losing the data held in each view.

The menu bar [Figure 4-6] provides the user with a set of master options with which to manipulate the system:

- Rules > Rules Settings [Figure 4-7]: Opens a dialog box which allows the user to configure game rules such as payoffs and generation limits. The parameters in this box are validated so illegal entries will not be permitted. The user is provided with options to save their changes, cancel them or load the default parameters. The rules set here will apply across all game types. This menu is also accessible from the individual game screens.
- Strategy > Strategy Manager [Figure 4-8]: The `MenuFrame` object contains an array of saved Prisoner's (initially the classic strategies TFT, ALLC, ALLD and TFTT). This provides the user with the ability to save strategies for later analysis. This dialog box allows a user to view and delete the saved strategies.
- Game type [Figure 4-6]: Allows a user to switch between the different game types –
  - Tournament – Evolving strategies for a round robin tournament

- Spatial Society – Evolving Strategies for a spatial grid.
- Interactive – Play against strategies and compare strategies.

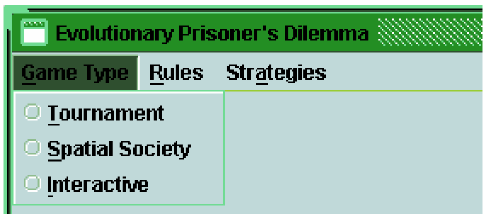
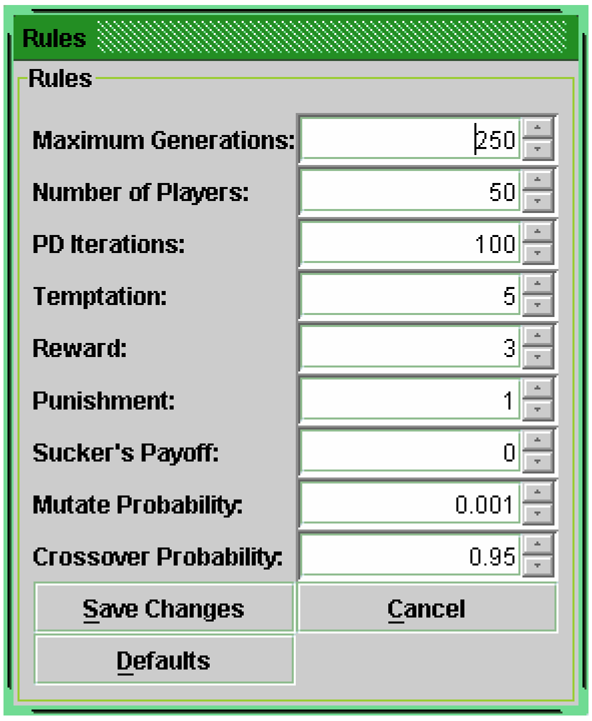


Figure 4-6: Menu Bar


Rules

Rules

Maximum Generations: 250
Number of Players: 50
PD Iterations: 100
Temptation: 5
Reward: 3
Punishment: 1
Sucker's Payoff: 0
Mutate Probability: 0.001
Crossover Probability: 0.95

Figure 4-7: Rules Dialog

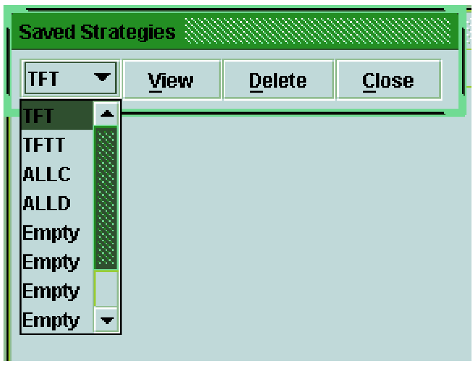


Figure 4-8: Strategy Dialog

4.4.2 Tournament Evolution GUI

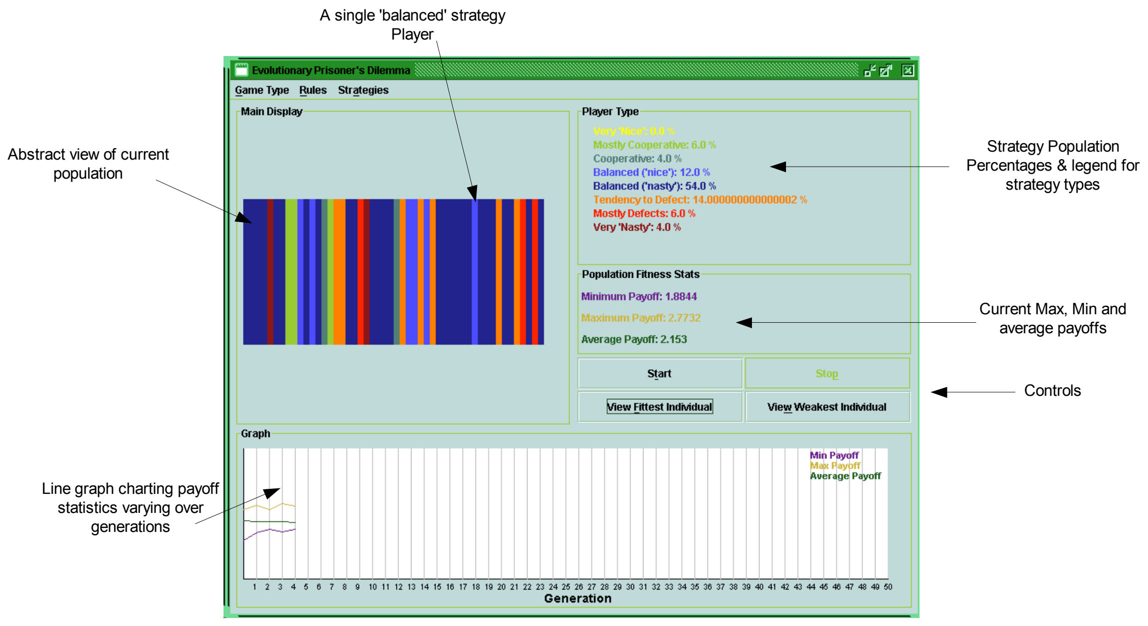
The `TournamentPanel` object implements the GUI [Figure 4-9] to evolve strategies for the IPD round robin tournament. The major problem encountered when implementing this was the heavy computation required to produce each generation.

This computation would cause the GUI to become unresponsive after an evolution had been started. To overcome this multi-threading was introduced to the system. The Genetic Algorithm computation was placed in a separate thread from the GUI. This thread was also responsible for updating the statistics displayed on the GUI. This allowed the application to remain responsive and dynamically update while the Genetic Algorithm was executing.

This module contains a `Breeder` object, the Genetic Algorithm engine, which is responsible for evolution. This `Breeder` object is also capable of presenting the current population to screen. The current population is presented as a series of rectangles. Each rectangle represents a Prisoner within the population. Each Prisoner is rendered in a different colour depending on the percentage of C's and D's within its chromosome, that is, the player's tendency to cooperate or defect. Thus while evolution is running an abstract view of the types of strategy in the current population is visible.

This module also uses the `GraphPanel` object to graph the changing payoffs from generation to generation.

The controls in this 'game type' include buttons to start/stop evolution and buttons to view the current fittest/weakest player strategies. When a strategy is viewed the user is also given the option to save this strategy for later use. If the user saves the strategy it is added to the saved Prisoner array in the `MenuFrame` object and is then accessible in the other windows.



Very 'Nice': 0.0 %
Mostly Cooperative: 6.0 %
Cooperative: 4.0 %
Balanced ('nice'): 12.0 %
Balanced ('nasty'): 54.0 %
Tendency to Defect: 14.000000000000002 %
Mostly Defects: 6.0 %
Very 'Nasty': 4.0 %

Minimum Payoff: 1.8844
Maximum Payoff: 2.7732
Average Payoff: 2.153

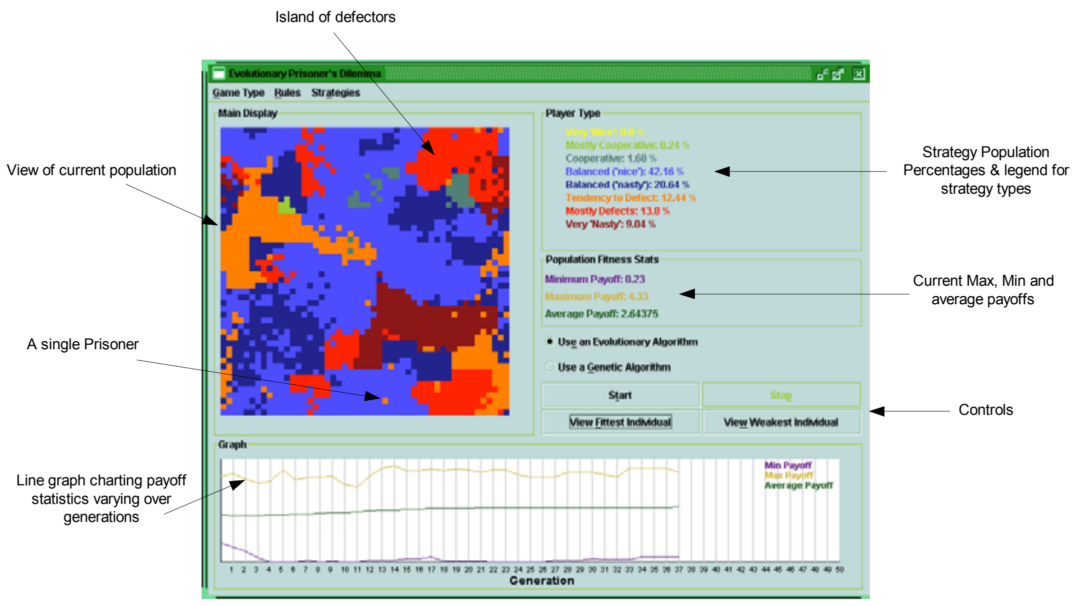
Figure 4-9: Tournament Screen

4.4.3 Spatial Society GUI

A view of spatial grid evolution [Figure 4-10] may also be viewed in the main frame by selecting it from the menu bar. Switching between the spatial 'game type' and tournament 'game type' will cause the currently displayed evolution to stop. If this did not occur than the system would possibly be overwhelmed trying to run two separate computationally intensive Genetic Algorithms concurrently.

This window is implemented by the `SpatialPanel` object which is implemented in the same manner as above, but rather than use a `Breeder` object it uses the spatial variation of the Genetic Algorithm implemented in the `Spatial` object. An additional control is also added to allow the user to choose between a Genetic Algorithm and Evolutionary Algorithm for evolution.

The view of the current population presented in this window (facilitated by the `Spatial` objects ability to draw itself) represents the Prisoners in the population as small rectangles whose positions are based on their positions on the grid. Thus the geographical nature of the grid is conveyed.


Island of defectors
View of current population
A single Prisoner
Line graph charting payoff statistics varying over generations
Strategy Population Percentages & legend for strategy types
Current Max, Min and average payoffs
Controls
Evolutionary Prisoner's Dilemma
Game Type
Rules
Strategies
Main Display
Graph
Generation
Min Payoff
Max Payoff
Average Payoff
Start
Stop
View Fittest Individual
View Weakest Individual

Player Type

Very 'Nice': 0.0 %
Mostly Cooperative: 0.24 %
Cooperative: 1.68 %
Balanced ('nice'): 42.16 %
Balanced ('nasty'): 20.64 %
Tendency to Defect: 12.44 %
Mostly Defects: 13.8 %
Very 'Nasty': 9.04 %

Population Fitness Stats

Minimum Payoff: 0.23
Maximum Payoff: 4.33
Average Payoff: 2.64375

[✓ Use an Evolutionary Algorithm]
Use a Genetic Algorithm

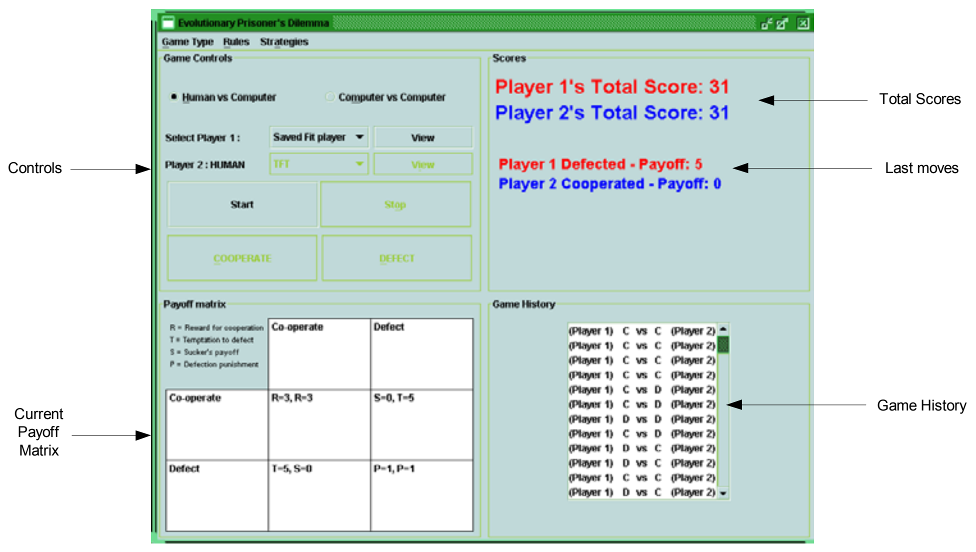
Figure 4-10: Spatial Screen

4.4.4 Interactive GUI

The Interactive 'game type' allows to user to play an interactive version of the IPD against saved strategies or pit two saved strategies against one another. This window is implemented in the `InteractivePanel` object. The implementation is relatively straightforward utilising the `Prisoner`, `Game` and `Rules` objects. The saved Prisoner's are loaded from an array in `MenuFrame` in which the other views may have saved Prisoners.

This window [Figure 4-11] provides controls to select "human vs. computer" or "computer vs. computer" play. In both cases the players play each other in the IPD. In the case of a human player the user must supply the "Cooperate" or "Defect" decisions by clicking on the appropriate buttons. In both cases the game history is stored and displayed along with the current and total payoffs. A visual representation of the current payoff matrix is also provided to aid in the analysis process.

Thus this window allows a user to experiment with evolved strategies to evaluate their traits and relative performance.



Player 1's Total Score: 31
Player 2's Total Score: 31

Player 1 Defected - Payoff: 5
Player 2 Cooperated - Payoff: 0

R = Reward for cooperation
T = Temptation to defect
S = Sucker's payoff
P = Defection punishment

	Co-operate	Defect
Co-operate	R=3, R=3	S=0, T=5
Defect	T=5, S=0	P=1, P=1

```
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  D  (Player 2)
(Player 1)  C  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  C  vs  D  (Player 2)
(Player 1)  D  vs  C  (Player 2)
(Player 1)  D  vs  C  (Player 2)
(Player 1)  C  vs  C  (Player 2)
(Player 1)  D  vs  C  (Player 2)
```

Figure 4-11: Interactive Screen

4.5 Problems Encountered

Several technical issues had to be overcome to complete this system implementation:

- Multi-threading: Had to be implemented to separate the genetic algorithm computation from the GUI computation.
- Speed: Initially the amount of computation required to run the Genetic Algorithm over a large spatial population took a long time (30s+) to complete. This made the application cumbersome and impractical. However this was overcome by improving the efficiency of the algorithm which calculates spatial fitness scores. The application now runs quickly for populations of up to ~6000 prisoners. These times are dependant on system hardware specifications. A PIII 500 MHz with 128MB of RAM is recommended.
- Memory: With very large populations Java may return an "out of memory" error. This occurs quite rarely and was not seen as a major problem as most of the interesting results of this project can be found with relatively small populations. This problem is due the amount memory available and thus varies from computer to computer.

4.6 Possible Improvements

The time constraints of this project caused several minor implementation details to be omitted. These were mainly aesthetical and preference was given to more important system components. Having completed the project it was also clear that some areas could be improved upon or extended further. Some of the possible additions which could be made to this system include:

- Saving - Saved Strategies and Rule settings are currently lost when the application is closed. A means of saving these to disk would be useful
- Genetic Algorithm – It may be interesting to provide several alternative selection, fitness scaling and replacement techniques to allow comparison between them. Also some research [15] has been done where initial populations are seeded with a number of pre-built strategies (TFT…); it may be interesting to add this functionality to this system.
- Prisoner Analysis – Currently the user can only analyse the fittest and weakest members of a population. It would be useful to allow the user to click on any population member and view their strategy and payoffs.

- GUI improvements – At present the GUI aims to provide clarity and ease of use. However it could be improved to provide more informative feedback (such as dialog boxes when performing 'dangerous' actions). Also the GUI current utilises a custom 'green' look and feel, this could be extended to give the user a choice of skins/themes to change the GUI appearance.
- Strategy Generation – There is currently no means for the user to create their own custom strategies, this may be a useful addition.
- Randomness – This system depends heavily upon random number generation; random numbers are used throughout the Genetic Algorithm. Java's implementation of a pseudo random number generator (PRNG) was used to generate these random numbers. This source (seeded with the current time) provides reasonably random numbers. However the system may be improved by utilising the extremely random numbers generated from a cryptographically secure PRNG or a True PRNG.

5 Results and Discussion

The aim of this project was to produce a system which would enable strategies for the Prisoner's Dilemma to be evolved. This section is not intended as presentation of the findings of extensive experimentation, rather it endeavours to describe some of the interesting results found while performing limited experimentation with the system.

5.1 Tournament

5.1.1 Tournament evolution

Experimentation was performed using a population size of 100, an IPD length of 100 iterations and the standard payoffs [Table 1]. Interestingly, despite the game rules making it difficult, mutual cooperation was seen to emerge on most runs of the Genetic Algorithm. Occasionally the Genetic Algorithm would prematurely converge upon a local optima, such as mutual defection, but the majority of the time mutual cooperation (or a slightly sub-optimal version of it) would emerge.

The evolutions generally followed the same pattern. At first, defecting strategies do well by taking advantage of weak strategies, causing the average payoff to fall. However as defecting strategies grow in the population they run out of victims to exploit and begin to score poorly. This allows cooperative strategies to dominate raising the population average. The genetic algorithm eventually converges upon one strategy, usually a 'balanced' strategy or one with a slight tendency to defect.

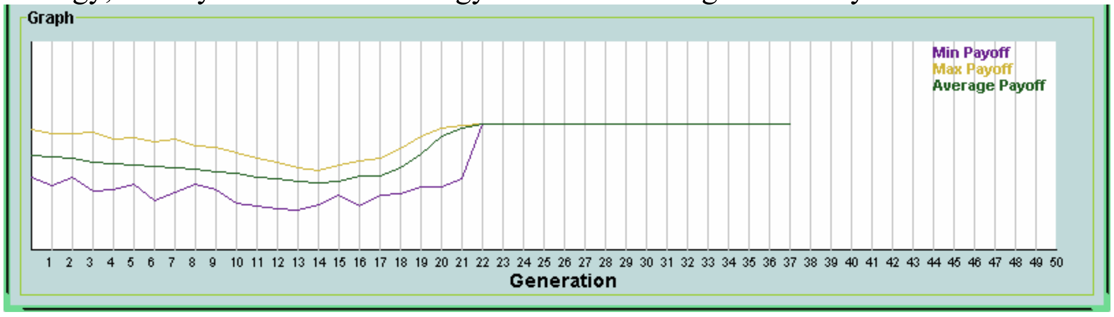


Figure 5-1: Tournament Payoffs

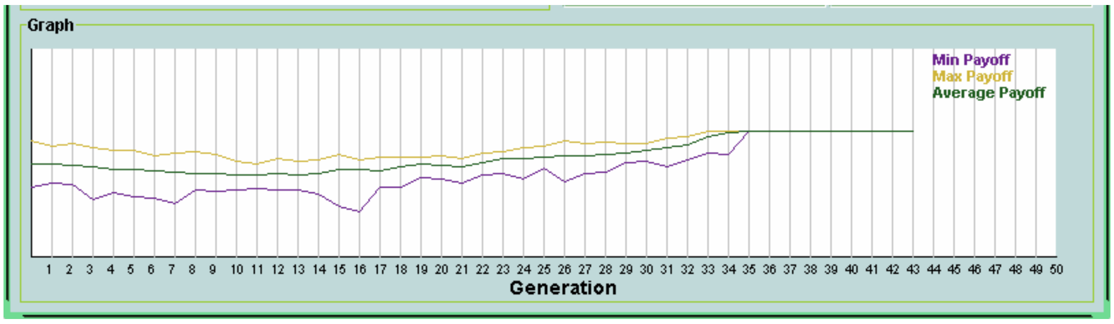


Figure 5-2: Tournament Payoffs

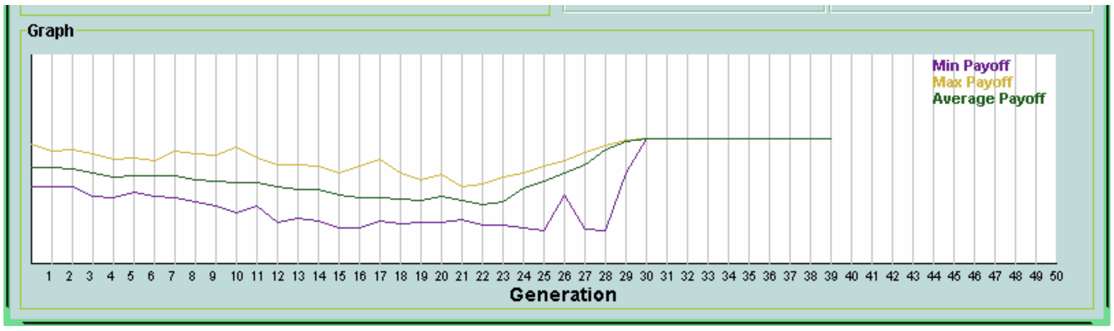


Figure 5-3: Tournament Payoffs

This supports research into strategy evolution and the emergence of cooperation by Axelrod [1] and Yao [15].

5.1.2 Tournament Strategy Analysis

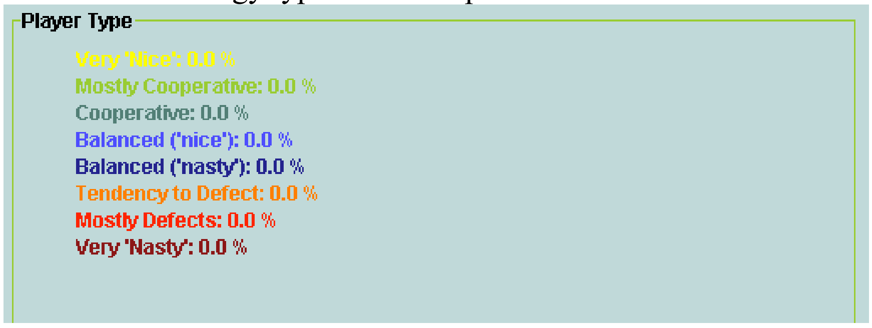
A number of the strategies evolved for the Tournament competition were analysed in the Interactive 'game mode'. This allowed the characteristics of fit individuals to be identified. The fit individuals all displayed extremely similar characteristics – those identified by Axelrod [see above 2.3]:

- Nice - Almost all of the optimal strategies involved would always cooperate with an always cooperating individual. The only deviation from this was occasionally the strategies would defect on their first move but cooperate thereafter.
- Retaliatory - The evolved strategies all exhibited some form of defection pattern after they other player defected.
- Forgiving - This is where the evolved strategies differed slightly from Axelrod's findings [1] concerning TFT. Having started defecting the strategies could be persuaded to begin cooperating again; however this generally took several game iterations. Thus the evolved strategies were not as forgiving as TFT.
- Clarity - the nature of the chromosome encoding [Table 2] breeds simple strategies.

It is also worth noting that these strategies were co-evolved, they were evolved to perform well in there local environment. When removed from this environment and played against a foreign strategy these strategies may perform badly. The strategies evolved from the tournament were tested against the ALLD (always defect) strategy and generally performed poorly, supporting this theory.

5.2 Spatial Society

Some trials of evolution in the spatial society using both the Genetic Algorithm and Evolutionary Algorithm were conducted. These used the standard payoff parameters [Table 1], 100 round IPD and 2500 players (50 rows and 50 columns). The legend used for the strategy types in this experiment is shown below


Player Type

Very 'Nice': 0.0 %
Mostly Cooperative: 0.0 %
Cooperative: 0.0 %
Balanced ('nice'): 0.0 %
Balanced ('nasty'): 0.0 %
Tendency to Defect: 0.0 %
Mostly Defects: 0.0 %
Very 'Nasty': 0.0 %

Figure 5-4: Strategy Legend

5.2.1 Evolutionary Algorithm Results

The payoff statistics were a less informative in this environment as the genetic algorithm is maintaining many different solution populations. It is more interesting to examine the strategy populations on the spatial grid. In the early stages of evolution the populations are randomly distributed [Figure 5-5]. As evolution continues the different strategy types begin to congregate and form distinct strategy islands [Figure 5-6]. The defectors generally form larger groups in the beginning as they consume weak strategies. However as the evolution continues and weak strategies die off the cooperative strategies begin to score more heavily. Thus large populations of defectors and co operators emerge and fight for dominance. This occasionally results in one strategy wiping out the other but usually the strategies form stable groups unchanged by further evolution [Figure 5-7].

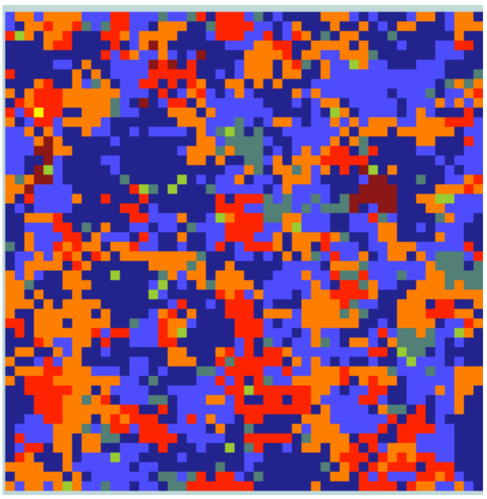


Figure 5-5: Initial Spatial population

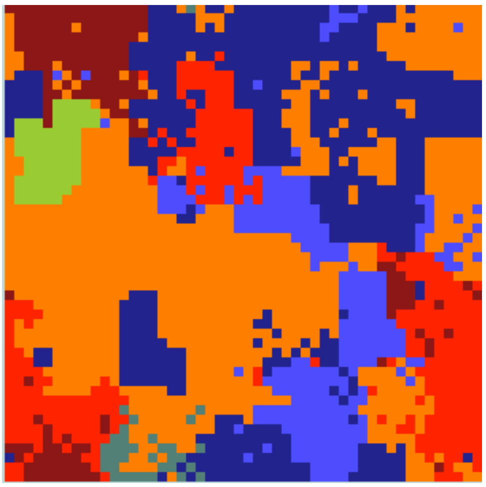


Figure 5-6: Spatial population mid-evolution

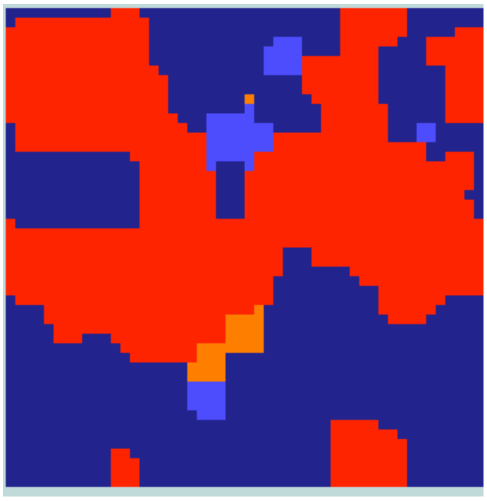


Figure 5-7: Stable Spatial Population s

Due the inherent randomness of the evolutionary algorithm's selection scheme and the initial populations this process is quite different each time. Many fascinating patterns can arise, for example strategies can often be seen invading other strategies until they have completely consumed them.

5.2.2 Genetic Algorithm applied to Spatial model

The Genetic Algorithm was less successful when applied to the spatial model. While patterns could be seen to arise they were not as clear cut as those above [Figure 5-9]. This may be due to the fitness proportional selection algorithm focusing on one area of the grid. It may also be due to the large exchanges of genetic material which take place in the Genetic Algorithm. This would cause different strategy types to be scattered about the grid [Figure 5-8]. However it is clear that the genetic algorithm is in fact optimising the solutions within the grid as the weak strategies, those who cooperate heavily (green and yellow) die off very quickly.

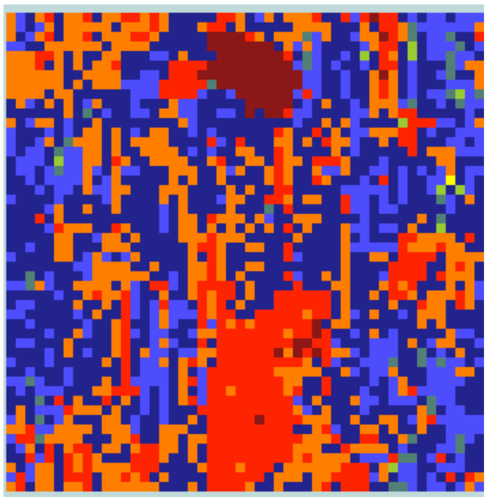


Figure 5-8: Spatial model genetic evolution

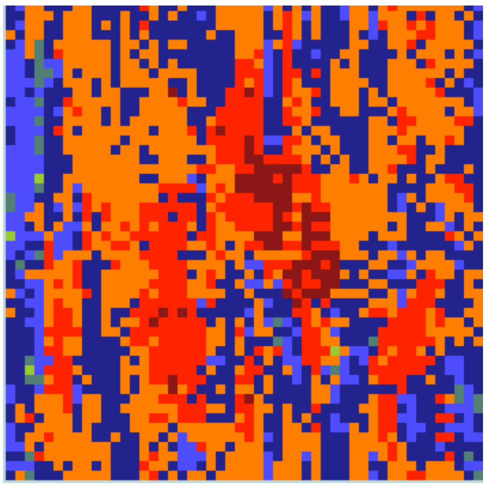


Figure 5-9: Spatial model late genetic evolution

In order to test if the dispersion of genetic material was preventing the strategy islands from forming the probability of crossover was reduced to zero. This meant that the parent individuals would be copied onto their weakest neighbours (a variation of the

Evolutionary Algorithm scheme). Having done this strategy islands could be seen to form as above, supporting the theory.

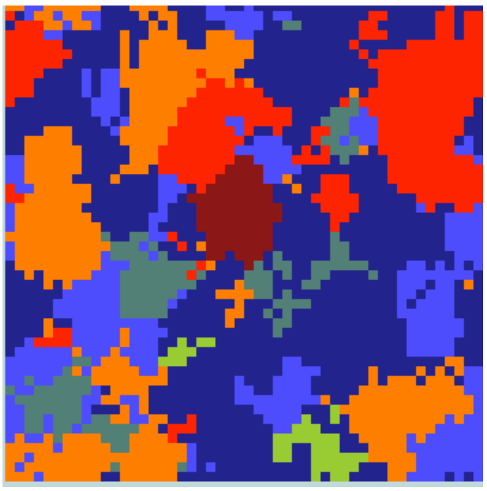


Figure 5-10: Modified GA applied to Spatial Model

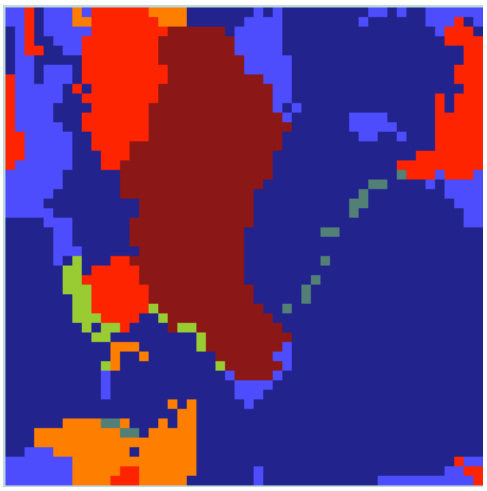


Figure 5-11: Modified GA applied to Spatial Model

6 Conclusion

The completed project has accomplished its goals. The Prisoner's Dilemma problem has been examined and strategies to successfully play the game have been evolved. In the chapters above these strategies have been analysed and proven to display the characteristics necessary to play the Prisoner's Dilemma successfully. This project has also examined a fascinating spatial variant of the Prisoner's Dilemma and evolved strategies in a geographical context.

Moreover this project has accomplished its objective of producing a simulation environment with a graphical user interface. This interface is detailed in the above chapters and was successfully used to run simulations and examine the results.

Furthermore the results of simulations run using the implemented system are consistent with the results of various previous studies.

7 References

1. Robert Axelrod, *The Evolution of Cooperation*, Penguin Books, 1990
2. Shaun P. Hargreaves Heap and Yanis Varoufakis, *Game Theory a Critical Introduction*, Routledge, 1995
3. IBM, UML Resource Centre, http://www.rational.com/uml/
4. IBM, Design Basics, http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/6
5. SUN, Java 1.4.1 Overview, http://java.sun.com/j2se/1.4.1/
6. SUN, Java Foundation Classes, http://java.sun.com/products/jfc/
7. Robert Axelrod, *The evolution of strategies in the iterated prisoner's dilemma*, In L. Davis (ed.), *Genetic algorithms and simulated annealing*. Pitman, 1987
8. David E. Goldberg, *Genetic Algorithms in search, optimization, and machine learning*, Addison-Wesley Publishing, 1989
9. Peter J.B. Hancock, *Selection Methods for Evolutionary Algorithms*, In L. Chambers (ed.), *Practical Handbook of Genetic Algorithms Applications Volume II*, CRC Press, 1995
10. Geoff Bartlett, *Genie: A First GA*, In L. Chambers (ed.), *Practical Handbook of Genetic Algorithms Applications Volume I*, CRC Press, 1995
11. John R. Koza, *Genetic Programming On the programming of computers by means of natural selection*, MIT Press, 1992
12. Conor Ryan, *Niche Species Formation in Genetic Algorithms*, In L. Chambers (ed.), *Practical Handbook of Genetic Algorithms Applications Volume I*, CRC Press, 1995
13. B. Routledge, *Co-Evolution and Spatial Interaction*, mimeo, University of British Columbia, 1993
14. Frank Schweitzer, Laxmidhar Behera, Heinz Mühlenbein, *Evolution of Cooperation in a Spatial Prisoner's Dilemma*, Advances in Complex Systems, vol. 5, no. 2-3, pp. 269-299, 2002
15. P.J. Darwen and X. Yao, *On Evolving Robust Strategies for the Iterated Prisoner's Dilemma*, 1993

Appendix A - User Manual

1. About

Evolutionary Prisoner's Dilemma
This program provides a simulation environment in which strategies for the Iterated Prisoner's Dilemma can be evolved and analysed.

Author:
Andrew Errity
[CACS4 99086921]
Faculty of Computing and Mathematical Sciences
Dublin City University
aerrit-cacs4@computing.dcu.ie

A website accompanying this project may be found at http://www.computing.dcu.ie/~aerrit-cacs4/prisoner/. This includes the runnable program, source code, API and other documentation.

2. About the Prisoner's Dilemma

In the Prisoner's Dilemma game two players are faced with a choice, they can either cooperate or defect. Each player is awarded points (called payoff) depending on the choice they made compared to the choice of the opponent. Each player's decision must be made without knowledge of the other player's next move. The player's have no means of communication other than the game choices and there can be no prior agreement between the players concerning the game.

If both players cooperate they both receive a reward, R. If both players defect they both receive a punishment, P. If one player defects and the other cooperates, the defector receives a reward, T the temptation to defect, whilst the player who cooperated is punished with the *sucker's* payoff, S. These game rules (and their typical values) are presented in the payoff matrix below

		Player B	
Player A		Cooperate	Defect
	Cooperate	R=3,R=3	S=0,T=5
	Defect	T=5,S=0	P=1,P=1

The dilemma here is that the only choice for rational players is to defect.

The name Prisoner's Dilemma comes from the following description of the problem:
The story is that of two prisoners placed in separate cells, with the aim of getting one prisoner to implicate the other. Each prisoner is given the option to defect against the other, by giving evidence against them. If both prisoners defect (give evidence) then the judge, in no doubt over their guilt, will send them both to prison for 3 years. If both prisoners cooperate (don't give evidence), then the judge, with less clear indication of guilt, will send them both to prison for only 1 year. However if one prisoner defects and the other does not, the judge will take this as a clear sign of

guilt, allowing the defector (evidence giver) to walk free whilst sentencing the other prisoner to 5 years.

3. System Requirements

In order to successfully run this program your system will require:

- An Operating System capable of running Java
- Java version 1.4.1 or greater, available for download at http://java.sun.com/j2se/1.4.1/download.html
- A screen resolution of at least 800x600

The recommended hardware specifications are:

- Pentium III 500Mhz
- 128MB RAM

4. Installation Instructions

These instructions assume Java version 1.4.1 or greater is installed on your system.

a. Ensure your system meets the system requirements
b. Obtain the file `epd.jar`. This is available for download at http://www.computing.dcu.ie/~aerrit-cacs4/prisoner/
c. Save this file to a location on your computer, for example `c:\`

The `epd.jar` file is the only file necessary to run the Evolutionary Prisoner's Dilemma program. To run the program, open a command prompt and run the following command,

```
java -jar <file location>\epd.jar
```

For example under Windows this might be:

```
java -jar c:\epd.jar
```

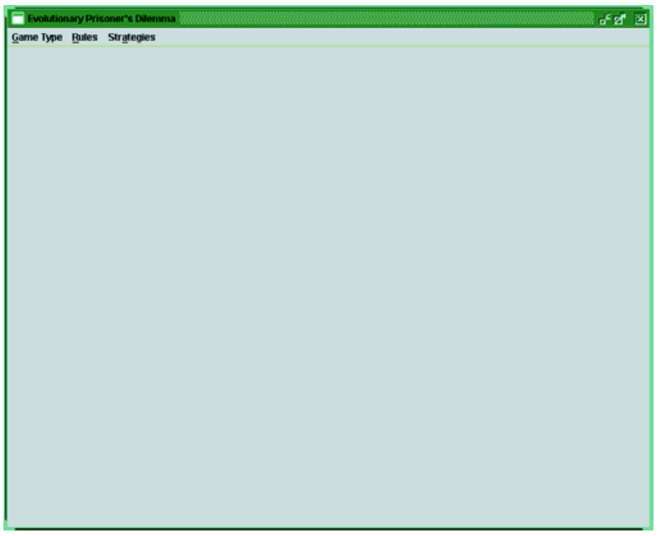
Whereas under a Unix based OS it might be:

```
java -jar /home/userid/epd.jar
```

If `jar` files have been correctly associated with Java in your Operating System settings you may be able to run the program by simply double clicking the `epd.jar` file.

5. Getting Started

When the Evolutionary Prisoner's Dilemma program is first run you will be presented with the blank application screen.



Main Window

The menu bar at the top of this window remains in place in all program modes and always provides the following options:

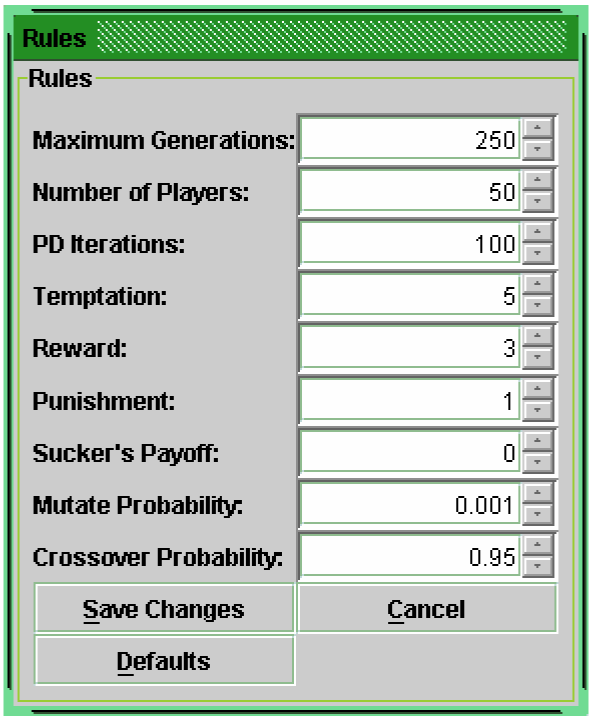
- Game Type > Tournament: Opens the Iterated Prisoner's Dilemma Tournament simulation window [described below] [Warning: Selecting this option while running a Spatial Simulation will cause the Spatial Simulation to end]
- Game Type > Spatial Society: Opens the Spatial Society simulation window [described below] [Warning: Selecting this option while running a Tournament Simulation will cause the Tournament Simulation to end]
- Game Type > Interactive: Opens the interactive strategy analysis window [described below]
- Rules > Rules Settings…: Opens a dialog box which allows you to configure the program settings [described below]
- Strategies > Strategy Manager…: Opens a dialog box which allows you to view or delete the saved strategies [described below]

6. Rules Settings Dialog

This dialog is accessed by clicking Rules in the menu bar and selecting Rules Settings.

This dialog allows you to configure the program parameters. Any changes made to the rules settings will affect all program windows. Changes made to rules during a simulation will not take effect until the next simulation. Changes made to the rules from the Interactive window will take effect immediately.

The Rules settings dialog box for the Interactive and Tournament windows is shown below. The following parameters may be set using the spinners or text entry:


Rules

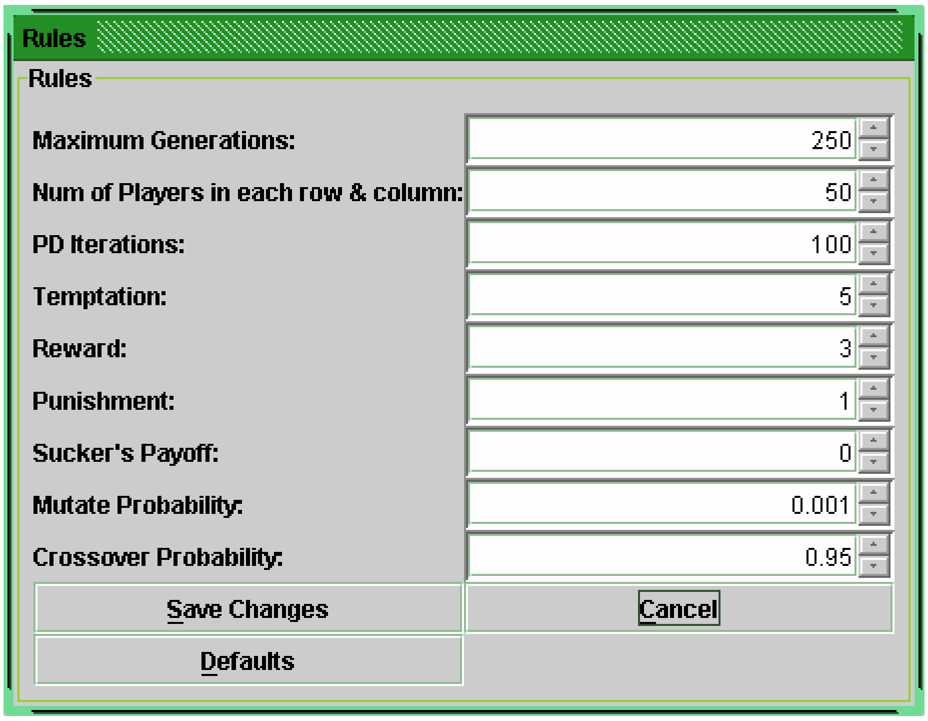
Rules

Maximum Generations:	250
Number of Players:	50
PD Iterations:	100
Temptation:	5
Reward:	3
Punishment:	1
Sucker's Payoff:	0
Mutate Probability:	0.001
Crossover Probability:	0.95

Rules Dialog

- Maximum generations: The upper limit on the number of generations the Genetic Algorithm will produce [not applicable to the Interactive window].
- Number of Players: The number of players in the simulation population [not applicable to the Interactive window].
- PD Iterations: The number of rounds of the Prisoner's Dilemma played in each Iterated Prisoner's Dilemma game.
- Temptation, Reward, Punishment, and Sucker's Payoff: The T, R, P and S values for the payoff matrix described above.
- Mutate Probability: The probability of genetic mutation occurring during the Genetic Algorithm reproduction operations [not applicable to the Interactive window].
- Crossover Probability: The probability of genetic crossover occurring during the Genetic Algorithm reproduction operations [not applicable to the Interactive window].
- **Save Changes**: Saves the currently displayed values
- **Cancel**: Discard these values and return to the previous values
- **Defaults**: Loads the program's preset/recommended rules.

The Rules dialog displayed from the Spatial Society window is shown below:



Rules

Rules

Maximum Generations:	250
Num of Players in each row & column:	50
PD Iterations:	100
Temptation:	5
Reward:	3
Punishment:	1
Sucker's Payoff:	0
Mutate Probability:	0.001
Crossover Probability:	0.95

Rules Dialog - Spatial

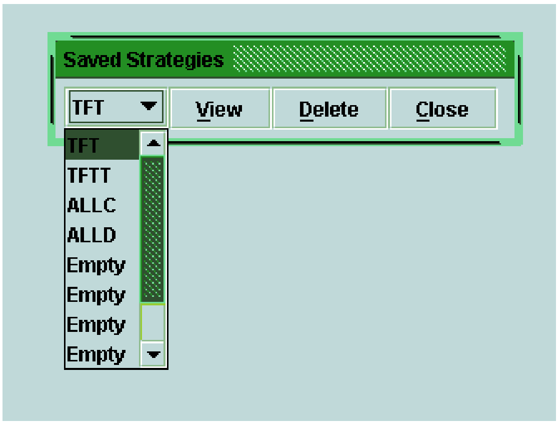
This is identical to the previous dialog except for:

- Num of Players in each row and column: In the previous dialog this specified the total number of Prisoner's in the simulation population. Here this value specifies the number of Prisoner's in each row and column, thus the actual number of prisoners in the population will be the square of this value. This field is limited to 70.

7. Strategy Manager

This dialog is accessed by clicking Strategies in the menu bar and selecting Strategy Manager.

This dialog allows you to manage the currently saved strategies.



Strategy Manager Dialog

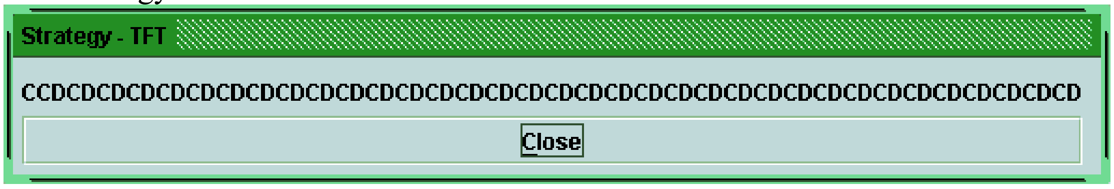
At start-up, several preset strategies are stored. These are:

- TFT – Tit for Tat, cooperates on first move and will only defect if the opponent defected on the previous move

- TFTT – Tit for Two Tats - cooperates on first move and will only defect if the opponent defected on the previous two moves
- ALLC – Always cooperates
- ALLD – Always defects

You can select a saved strategy from the drop down-list on the left and perform one of the following actions:

- View – Displays the strategy string in separate dialog box. For example the TFT strategy shown below.



CCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD

- Delete – Removes the saved strategy

- Close - Closes the dialog box

[Warning: When the application is closed any strategies saved in the strategy manager will be lost.]

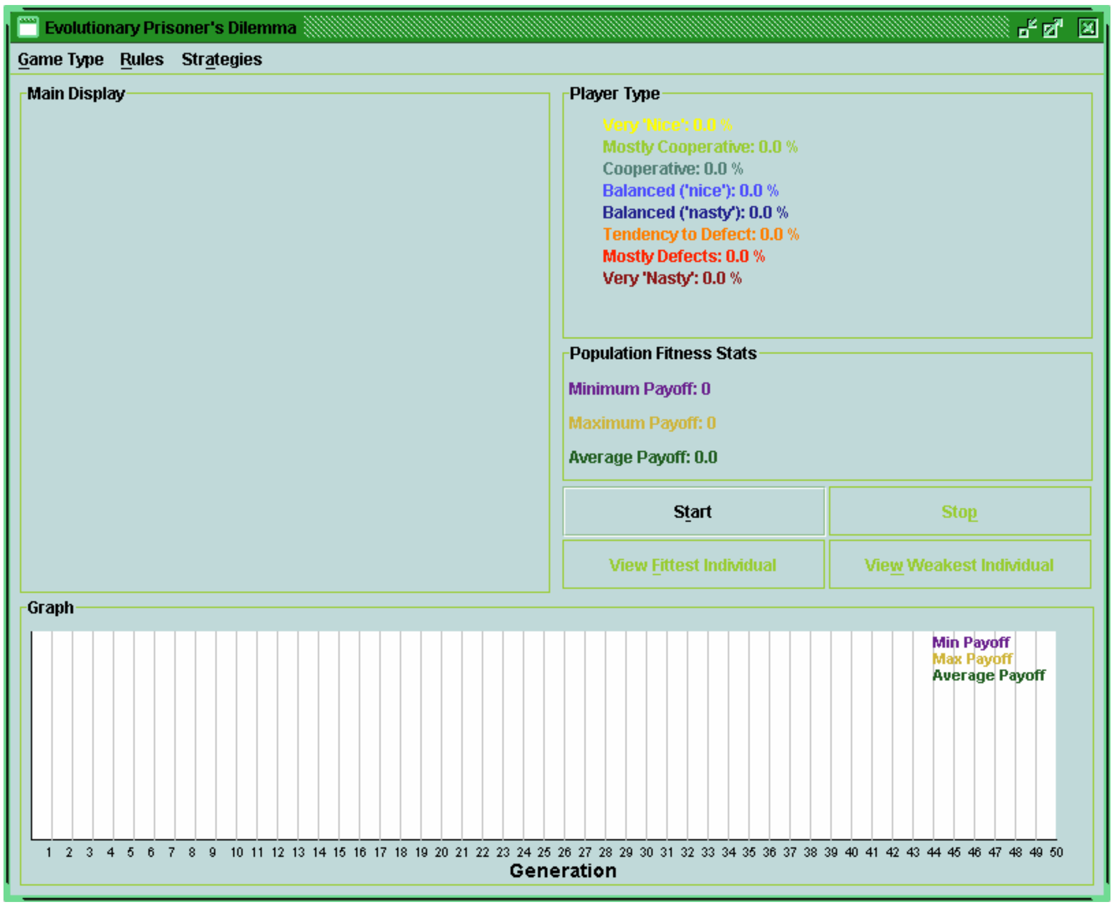
8. Game Types

The following section describes the three separate game windows which may be viewed. These windows may be viewed one at a time using the menu bar Game Type option.
[Warning: Switching between a Tournament Simulation and Spatial Simulation will cause any active simulations to end]

a. Tournament

The tournament window allows you to run evolution simulations on populations of Prisoners playing an Iterated Prisoner's Dilemma tournament.

When you first open a tournament window the following will be displayed:



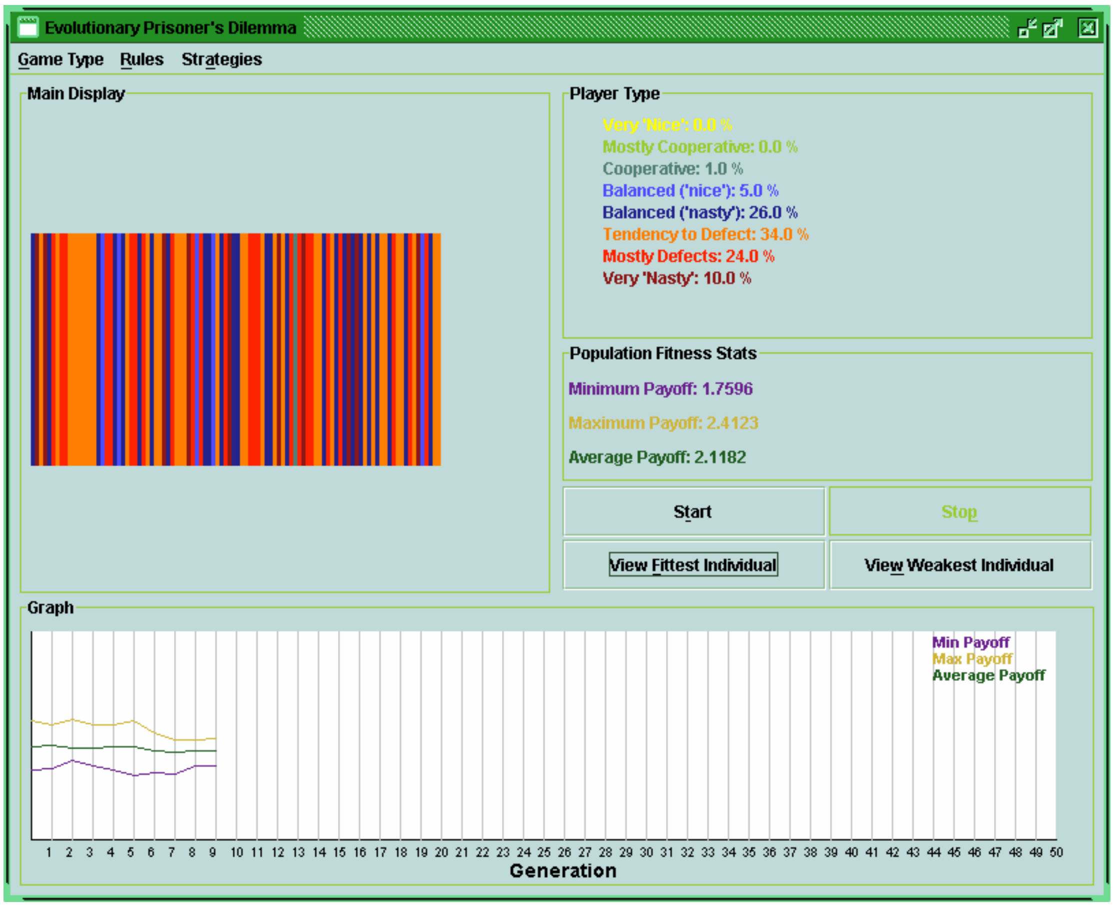
Very 'Nice': 0.0 %
Mostly Cooperative: 0.0 %
Cooperative: 0.0 %
Balanced ('nice'): 0.0 %
Balanced ('nasty'): 0.0 %
Tendency to Defect: 0.0 %
Mostly Defects: 0.0 %
Very 'Nasty': 0.0 %

Population Fitness Stats
Minimum Payoff: 0
Maximum Payoff: 0
Average Payoff: 0.0

Tournament Initial Window

This window consists of the following components:

- Main Display: This displays a representation of the current population during evolution; each Prisoner in the population is represented by a narrow rectangle. The colour of this rectangle indicates the player's strategy type.
- Player Type: Provides a legend to the colours displayed on the main display and also provides strategy population percentage information
- Population Fitness Stats: This pane displays the maximum, minimum and average payoffs of individuals in the population.
- Graph: Provides a line graph charting the maximum, minimum and average payoffs for each generation of evolution [Legend appears in upper right corner].



Very 'Nice': 0.0 %
Mostly Cooperative: 0.0 %
Cooperative: 1.0 %
Balanced ('nice'): 5.0 %
Balanced ('nasty'): 26.0 %
Tendency to Defect: 34.0 %
Mostly Defects: 24.0 %
Very 'Nasty': 10.0 %

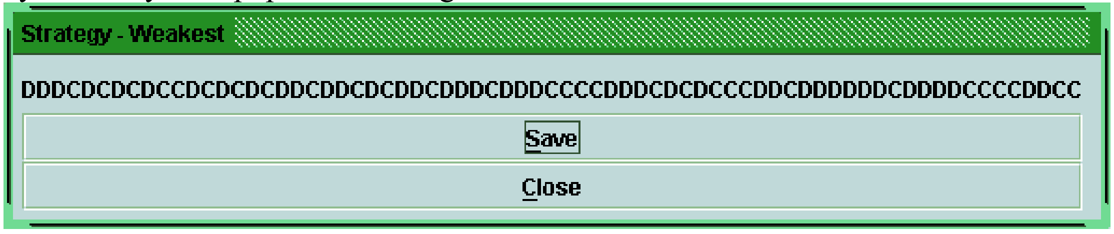
Minimum Payoff: 1.7596
Maximum Payoff: 2.4123
Average Payoff: 2.1182

Tournament Window

This window provides the following controls (in addition to the Rules Settings):

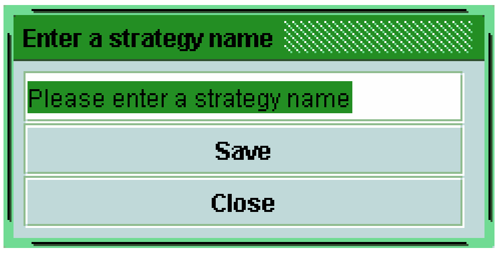
- Start – Clears the current population and statistics and starts the evolution of a new random population of prisoners playing an Iterated Prisoner's Dilemma tournament.
- Stop – Stops the current evolution but leaves all displays and statistics intact.
- View Fittest Individual – Opens a dialog box showing the strategy which currently has the highest score.
- View Weakest Individual - Opens a dialog box showing the strategy which currently has the lowest score.

The strategy dialog boxes which open to display the fit/weak strategies update dynamically with population changes.



DDDCDCDCDCCDCDCDCDDCDDCDCDDCDDDCDDDCCCCDDDCDCDCCCDDCDDDDDDCDDDDCCCCDDCC

These dialogs allow you to save a Prisoner's strategy in the strategy manager. Clicking save in this dialog will open a dialog box [see below] which allows you to enter a name under which to save the strategy. The strategy that was visible when you clicked save will be the one added to the strategy manager.

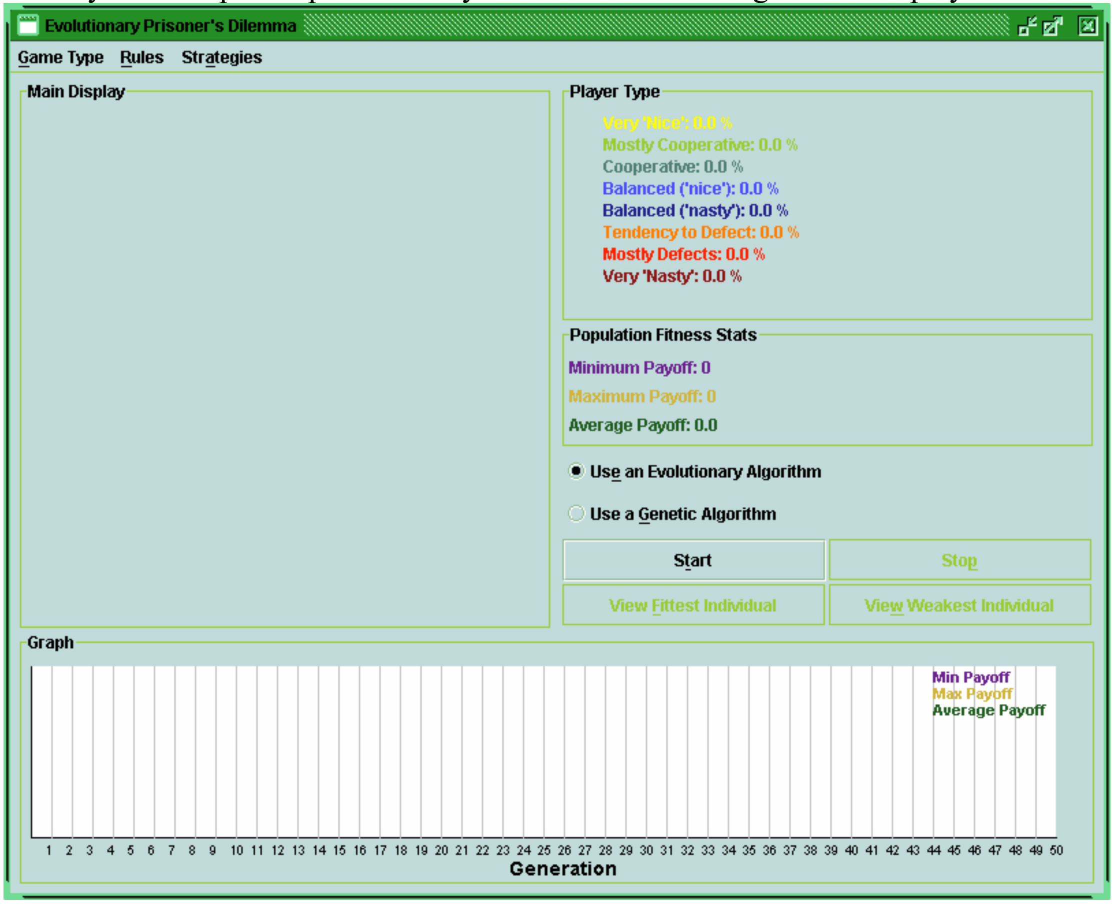


Having entered a strategy name clicking save will add the strategy to the strategy manager for later use.
[Warning: The strategy will be saved in the next free position in the strategy manager; if no empty spaces exist the strategy will not be saved]
[Warning: When the application is closed any strategies saved in the strategy manager will be lost.]

b. Spatial Society

The spatial society window allows you to run evolution simulations on populations of Prisoners playing an Iterated Prisoner's Dilemma tournament on a spatial grid.

When you first open a spatial society window the following will be displayed:



Very 'Nice': 0.0 %
Mostly Cooperative: 0.0 %
Cooperative: 0.0 %
Balanced ('nice'): 0.0 %
Balanced ('nasty'): 0.0 %
Tendency to Defect: 0.0 %
Mostly Defects: 0.0 %
Very 'Nasty': 0.0 %

Minimum Payoff: 0
Maximum Payoff: 0
Average Payoff: 0.0

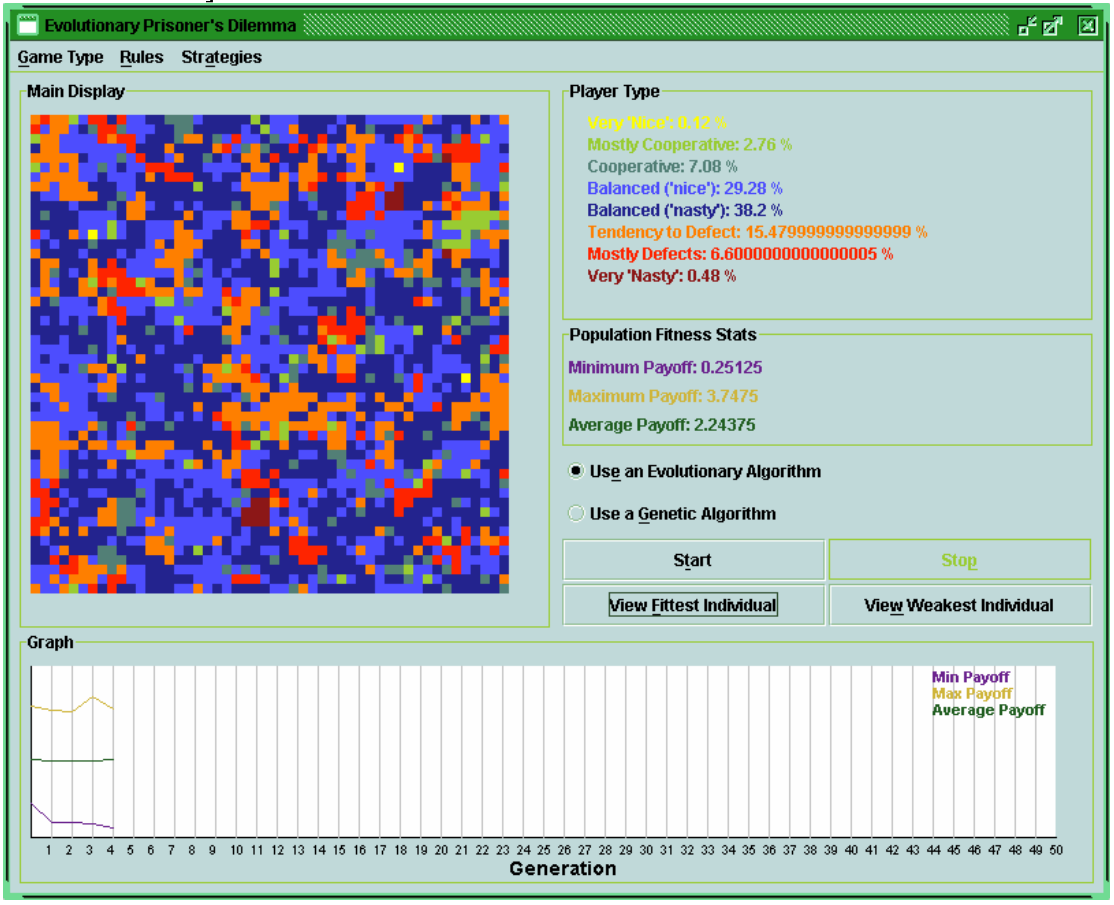
(● Use an Evolutionary Algorithm)

Spatial Society Initial Window

This window consists of the following components:

- Main Display: This displays a representation of the current population during evolution; each Prisoner in the population is represented by a small rectangle. The colour of this rectangle indicates the player's strategy type.

- Player Type: Provides a legend to the colours displayed on the main display and also provides strategy population percentage information
- Population Fitness Stats: This pane displays the maximum, minimum and average payoffs of individuals in the population.
- Graph: Provides a line graph charting the maximum, minimum and average payoffs for each generation of evolution [Legend appears in upper right corner].



Very 'Nice': 0.12 %
Mostly Cooperative: 2.76 %
Cooperative: 7.08 %
Balanced ('nice'): 29.28 %
Balanced ('nasty'): 38.2 %
Tendency to Defect: 15.479999999999999 %
Mostly Defects: 6.6000000000000005 %
Very 'Nasty': 0.48 %

Minimum Payoff: 0.25125
Maximum Payoff: 3.7475
Average Payoff: 2.24375

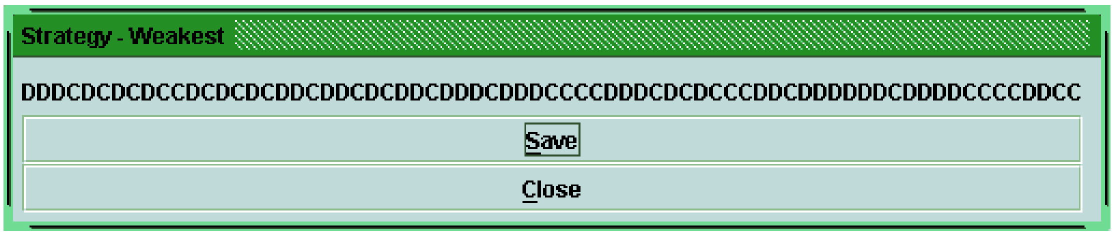
[✓ Use an Evolutionary Algorithm] Use a Genetic Algorithm

Spatial Society Window

This window provides the following controls (in addition to the Rules Settings):

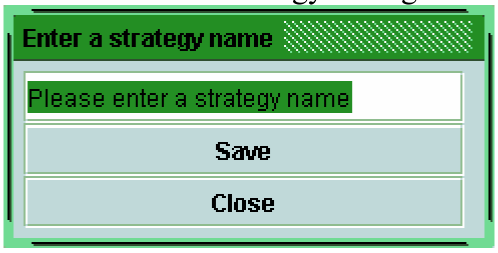
- Start – Clears the current population and statistics and starts the evolution of a new random population of prisoners playing an Iterated Prisoner's Dilemma on a spatial grid.
- Stop – Stops the current evolution but leaves all displays and statistics intact.
- Evolutionary Algorithm/Genetic Algorithm: Allows you to choose which type of algorithm to use for evolution.
- View Fittest Individual – Opens a dialog box showing the strategy which currently has the highest score.
- View Weakest Individual - Opens a dialog box showing the strategy which currently has the lowest score.

The strategy dialog boxes which open to display the fit/weak strategies update dynamically with population changes.



DDDCDCDCDCCDCDCDCDDCDDCDCDDCDDDCDDDCCCCDDDCDCDCCCDDCDDDDDDCDDDDCCCCDDCC

These dialogs allow you to save a Prisoner's strategy in the strategy manager. Clicking save in this dialog will open a dialog box [see below] which allows you to enter a name under which to save the strategy. The strategy that was visible when you clicked save will be the one added to the strategy manager.



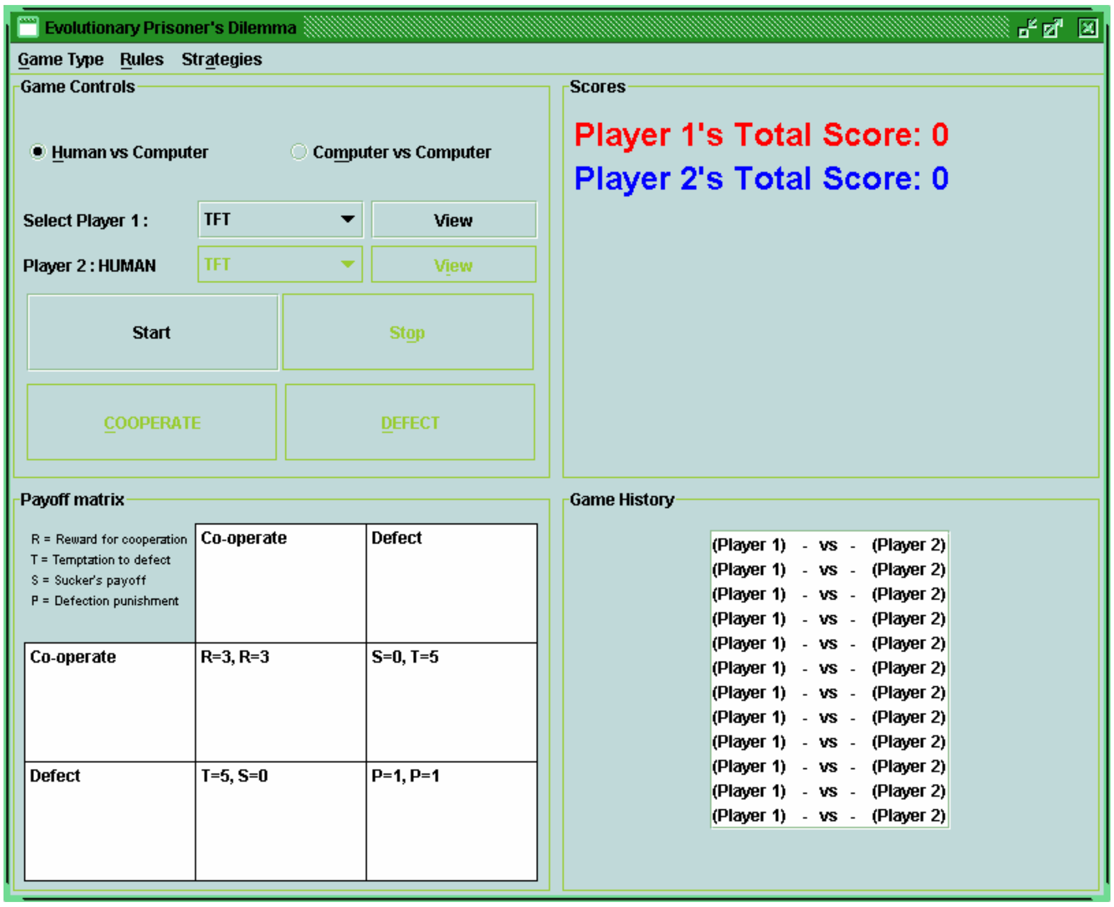
Please enter a strategy name

Having entered a strategy name clicking save will add the strategy to the strategy manager for later use.
[Warning: The strategy will be saved in the next free position in the strategy manager; if no empty spaces exist the strategy will not be saved]
[Warning: When the application is closed any strategies saved in the strategy manager will be lost.]

c. Interactive

This window allows you to play games of the Prisoner's Dilemma against saved strategies and pit saved strategies against one another

When first opened this window will display the following:



[● Human vs Computer] Computer vs Computer

Select Player 1 : TFT

Player 2 : HUMAN TFT

Player 1's Total Score: 0

Player 2's Total Score: 0

R = Reward for cooperation
T = Temptation to defect
S = Sucker's payoff
P = Defection punishment

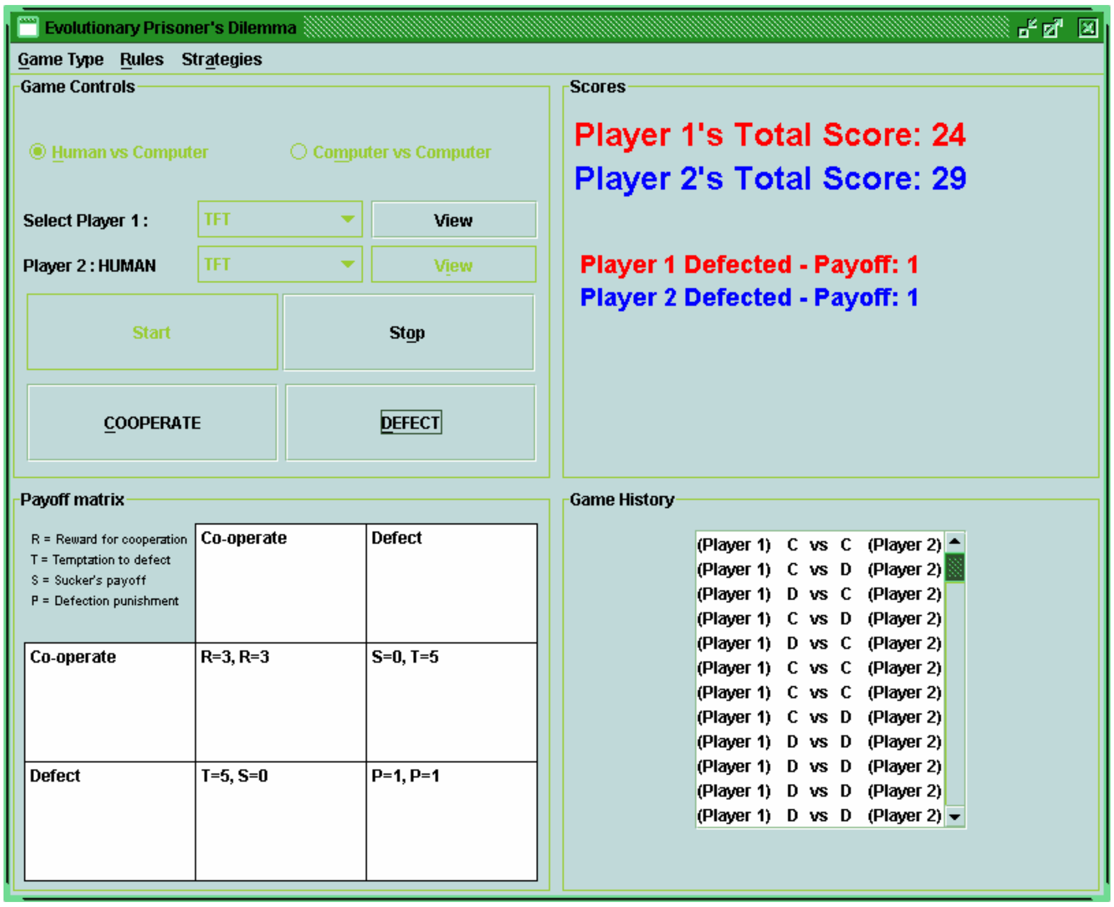
	Co-operate	Defect
Co-operate	R=3, R=3	S=0, T=5
Defect	T=5, S=0	P=1, P=1

```
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
(Player 1)  -  vs  -  (Player 2)
```

Interactive Window

The sub-windows which make up this window are:

- Scores: Displays the total payoffs received by each player. Also displays the results of the last game move. Any status messages, such as errors or warnings will also be displayed in this window.
- Game History: Displays a record of the moves played by both players in a scrollable list.
- Payoff matrix: Displays the payoff matrix for the current rule settings. This is aimed to aid the understanding of the Prisoner's Dilemma.
- Game Controls: Allow you to do the following
  - Choose a "Human vs. Computer" or "Computer vs. Computer" game.
  - Select the strategies to compete. These strategies are loaded from the strategy manager and may be viewed by clicking the View Button. You should select one strategy from the drop down box for "Human vs. Computer", this strategy will be your opponent. For "Computer vs. Computer" you should select two strategies to compete.
  - Start – Clicking start in "Computer vs. Computer" mode will play the Iterated Prisoner's Dilemma between the selected players as specified by the Rules Settings. The status of this game will be displayed in the panels described above.
  - Start – In "Human vs. Computer" mode this will start a game of IPD between you and the computer strategy. You may select your game move by clicking the cooperate or defect button. The opponents corresponding move will be displayed in the scores panel.
  - Stop – Clicking stop will end the active game but leave all displays and statistics intact.



Evolutionary Prisoner's Dilemma

Game Type   Rules   Strategies

Game Controls

[✓ Human vs Computer]   Computer vs Computer

Select Player 1 : TFT   View

Player 2 : HUMAN   TFT   View

Start   Stop

COOPERATE   DEFECT

Scores

Player 1's Total Score: 24

Player 2's Total Score: 29

Player 1 Defected - Payoff: 1

Player 2 Defected - Payoff: 1

Payoff matrix

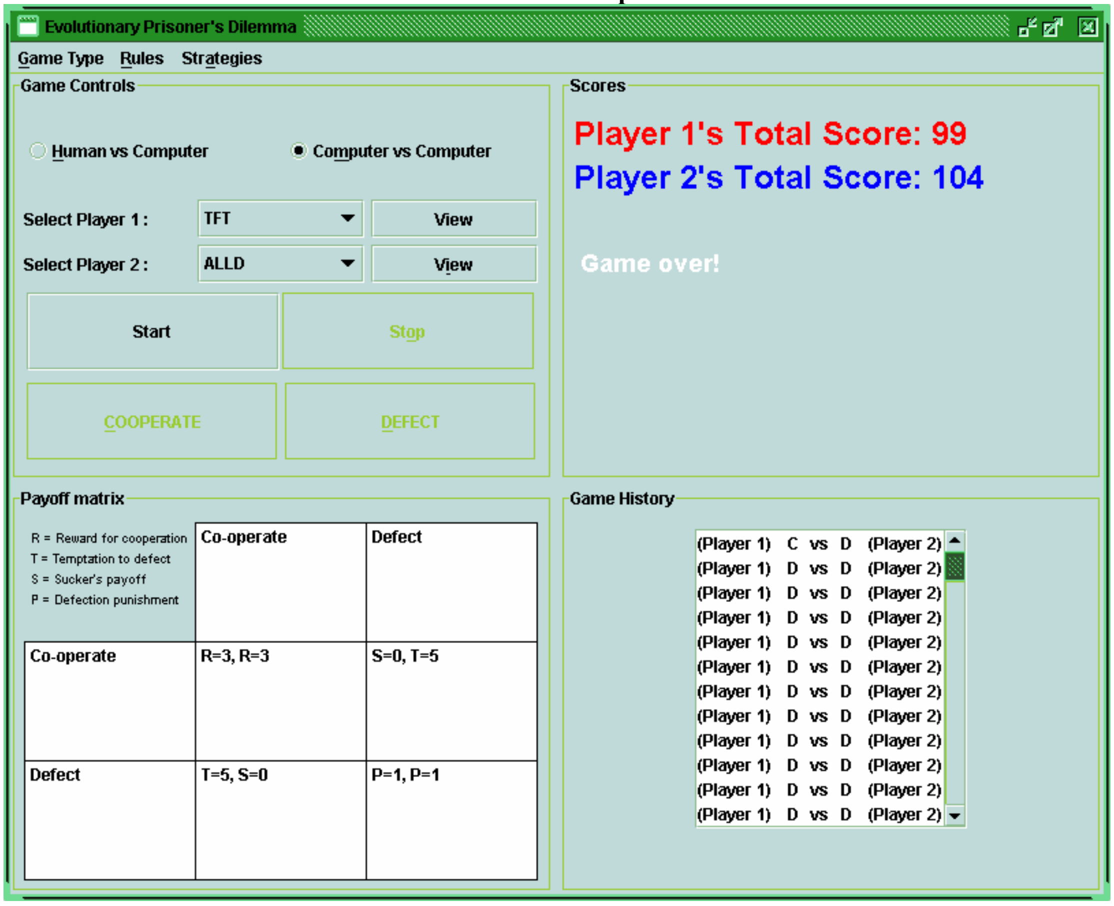
R = Reward for cooperation
T = Temptation to defect
S = Sucker's payoff
P = Defection punishment
	Co-operate	Defect

Co-operate	R=3, R=3	S=0, T=5
Defect	T=5, S=0	P=1, P=1

Game History

```
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  D  (Player 2)
(Player 1)  D  vs  C  (Player 2)
(Player 1)  C  vs  D  (Player 2)
(Player 1)  D  vs  C  (Player 2)
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  C  (Player 2)
(Player 1)  C  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
```

Human vs. Computer



Evolutionary Prisoner's Dilemma

Game Type   Rules   Strategies

Game Controls

Human vs Computer   [✓ Computer vs Computer]

Select Player 1 : TFT   View

Select Player 2 : ALLD   View

Start   Stop

COOPERATE   DEFECT

Scores

Player 1's Total Score: 99

Player 2's Total Score: 104

Game over!

Payoff matrix

R = Reward for cooperation
T = Temptation to defect
S = Sucker's payoff
P = Defection punishment
	Co-operate	Defect

Co-operate	R=3, R=3	S=0, T=5
Defect	T=5, S=0	P=1, P=1

Game History

```
(Player 1)  C  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
(Player 1)  D  vs  D  (Player 2)
```

Computer vs. Computer

Appendix B – Source Code

The following pages contain the source code implementation of some of the projects main components. This is not a full source listing, extracts from some of the important classes are provided for those interested. Full source code listings and API documentation are available at http://www.computing.dcu.ie/~aerrit-cacs4/prisoner/

Code listing

Package ie.errity.pd

Extract from Prisoner.java

```
/**
 *This class represents a Prisoner with a strategy to play the
 *Prisoner's Dilemma {@link  ie.errity.pd.Game Game}.
 *@author      Andrew Errity 99086921
 */
public class Prisoner implements Cloneable
{
        private String name;
        final private BitSet Strategy;

        private int Score;        //total payoffs recieved
        private Moves m;          //table used to decode strategy

        /**
         *Create a new Prisoner to play the prisoners dilemma
         *@param na   the Prisoner's name
         *@param strat         a <code>BitSet</code> representing the player's
strategy
         */
        public Prisoner(String na, BitSet strat)
        {
                name = na;
                Strategy = strat;
                Score= 0;
                m = new Moves();

        }

.
.
.
        /**
         *Gets the Prisoner's next game move
         *@param iteration      current iteration number
         *@param History game history represented as a <code>BitSet</code>
         *<BR>History should represent previous moves as C=1 and D=0
         *<BR>This players move should be followed by the opponents move, one
         *such pair for each previous iteration of PD
         *@return <code>true</code> or <code>false</code> {C or D}
         */
        public boolean play(int iteration, BitSet History)
        {
                // if first move return start move
                if(iteration == 0)
                        return Strategy.get(0);
                // if second move
                else if(iteration == 1)
                {
                        if(History.get(1)) //opponent Cooperated
                                return Strategy.get(1);
                        else //opponent Defected
                                return Strategy.get(2);
                }
                // if third move
                else if(iteration == 2)
                {
                        if(History.get(1) && History.get(3)) //opponent CC
                                return Strategy.get(3);
                        else if(History.get(1) && !History.get(3)) //opponent CD
                                return Strategy.get(4);
                        else if(!History.get(1) && History.get(3)) //opponent DC
                                return Strategy.get(5);
                        else if(!History.get(1) && !History.get(3)) ////opponent DD
                                return Strategy.get(6);
                }
                // if normal move use normal strategy
                else
                {
                        //Get last 3 sets of moves
                        BitSet hist = History.get( (iteration*2) - 6, (iteration*2));
                        int x = m.get(hist);
                        return Strategy.get(x+7); //adjust index to skip setup info
                }
                return false;
        }

  /**
   *Convert the Prisoner's strategy to a string of C's and D's
   *@return a string representation of the Prisoner's strategy
   */
   public String toString()
        {
                String p = new String();
                for(int i = 0; i < 71; i++)
                {
                        if(Strategy.get(i))
                                p = p + 'C';
                        else
                                p = p + 'D';
                }
                return p;
        }
```

Moves.java

```
package ie.errity.pd;

import java.util.Hashtable;
import java.util.BitSet;

/**
 *Table of the 64 possible histories of a 3 game sequence,
 *indexed by numbers from 0-63.
 *@author Andrew Errity 99086921
 */
public class Moves
{
        private Hashtable moves;

        /**
         *Creates a table of the 64 possible histories of a 3 game sequence,
         *indexed by numbers from 0-63.
         */
        public Moves()
        {
                moves = new Hashtable(64);
                String s;

                //Build table of all possible moves (3-game history)
        for(int n = 0; n <64; n++)
        {
                s = Integer.toString(n,2);
                while(s.length() < 6)
                {
                        s = '0' + s; //pad to length 6
                }

                BitSet temp = new BitSet(6);
                for(int i = 0; i < 6; i++)
                        if(s.charAt(i) == '0')
                                temp.set(i);
                moves.put(temp,new Integer(n));
        }
        }

        /**
         *Decodes a 3 game history to an index number
         *@param h    a 3 game <code>bitset</code> history
         *@return an index number between 0-63
         */
        public int get(BitSet h)
        {
                Integer x = (Integer)moves.get(h);
        return x.intValue();
        }

}
```

Extract from Rules.java

```
/**
 *This class represents the rules for a {@link  ie.errity.pd.Game Game}
 *of the Prisoner's Dilemma.
 *<BR>Payoff Matrix used is
 *<PRE>
 *          -------------------------
 *          |Cooperate    |Defect  |
 *-----------------------------------
 *|Cooperate|     R,R     |  S,T   |
 *-----------------------------------
 *|Defect   |     T,S     |  P,P   |
 *-----------------------------------
 *</PRE>
 *@author       Andrew Errity 99086921
 */
public class Rules
{
        private int       iterations, generations, players;

        private int T,S,R,P;
        private double mutateP, crossP;         //mutate & crossover probabilities
.
.
.
        /**
         *Create new game rules with specified parameters<BR>
         *@param it the number of PD games to play in an iterated PD
         *@param t the T value for the above payoff matrix
         *@param s the S value for the above payoff matrix
         *@param r the R value for the above payoff matrix
         *@param p the P value for the above payoff matrix
         *@param m the probability of mutation
         *@param c the probability of crossover
         *@param gen the maximum number of generations
         *@param pl the number of players
         */
        public Rules(int it, int t,int s, int r, int p, double m, double c, int gen, int pl)
        {
                iterations = it; //iterations of IPD
                T = t; //Temptation to defect
                S = s; //Sucker's Payoff
                R = r; //Reward for mutual cooperation
                P = p; //Punishment for mutual defection
```

```
		mutateP = m; //probability of mutation
		crossP = c; //probability of crossover
		generations = gen;
		players = pl;
	}
```

Extract from Game.java

```
	/**Play a game of IPD according to the rules*/
	public void Play()
	{
		//Init
		int length = rules.getIterations();
		int iteration = 0;
		P1Score = 0;
		P2Score = 0;
		BitSet P1History = new BitSet();
		BitSet P2History = new BitSet();
		boolean P1move, P2move;

		//Play the specified number of PD games
		for(iteration = 0; iteration < length; iteration++)
		{
			//Get each players move
			P1move = P1.play(iteration,P1History);
			P2move = P2.play(iteration,P2History);

			//Update scores according to payoffs
			if(P1move && P2move) //CC
			{
				P1Score += rules.getR();
				P2Score += rules.getR();
			}
			else if(P1move && !P2move) //CD
			{
				P1Score += rules.getS();
				P2Score += rules.getT();

			}
			else if(!P1move && P2move) //DC
			{
				P1Score += rules.getT();
				P2Score += rules.getS();

			}
			else if(!P1move && !P2move) //DD
			{
				P1Score += rules.getP();
				P2Score += rules.getP();
				}

				//Update player histories
				if(P1move)
				{
					P1History.set(iteration*2);
					P2History.set((iteration*2)+1);
				}
				else
				{
					P1History.clear(iteration*2);
					P2History.clear((iteration*2)+1);
				}
				if(P2move)
				{
					P1History.set((iteration*2)+1);
					P2History.set((iteration*2));
				}
				else
				{
					P1History.clear((iteration*2)+1);
					P2History.clear((iteration*2));
				}
			}
			//Update each players score
			P1.updateScore(P1Score);
			P2.updateScore(P2Score);
		}
```

Extract from Tournament.java

```
		/**
		 *Play the tournament
		 *@return the Tournament results, an array of payoffs corresponding to the
		 *player's scores
		 */
		public int[] Play()
		{
			int i;
			Game g;
			Prisoner Clone[] = new Prisoner[numPlayers];

			//Every player plays themselves and every other player
			for(i = 0; i < numPlayers; i++)
			{
				Clone[i] = (Prisoner)Players[i].clone();
				Clone[i].setScore(0);
				for(int j = 0; j < numPlayers; j++)
				{
					g = new Game(Clone[i],Players[j],rules);
```

```
                g.Play();
            }
            Results[i] = Clone[i].getScore();
        }
        //The total pay-offs each player recieved are returned
        Players = Clone;
        done = true;
        return Results;
    }
```

Package ie.errity.pd.genetic

Extract from Genetic.java

```java
/**
 *Provides Genetic operations
 *@author Andrew Errity 99086921
 */
public class Genetic
{

	/**
	 *Mate two parents using random, one point crossover
	 *@param parenta first parent (<code>BitSet</code> representation)
	 *@param parentb second parent (<code>BitSet</code> representation)
	 *@return an array containing the two children (<code>BitSet</code>
	 *representation)
	 */
	public static BitSet[] crossover(BitSet parenta,BitSet parentb)
	{
		Random rand = new Random();
		BitSet child1 = new BitSet(71);
		BitSet child2 = new BitSet(71);

		//One point splicing
		int slicePoint = rand.nextInt(71); //rnd num between 0-70
		BitSet a = (BitSet)parenta.clone();
		a.clear(slicePoint,71);
		BitSet b = (BitSet)parenta.clone();
		b.clear(0,slicePoint);
		BitSet c = (BitSet)parentb.clone();
		c.clear(slicePoint,71);
		BitSet d = (BitSet)parentb.clone();
		d.clear(0,slicePoint);

		//Combine start of p1 with end of p2
		child1.or(a);
		child1.or(d);
		//Combine start of p2 with end of p1
		child2.or(c);
		child2.or(b);

		//Return the children
		BitSet[] offspring = {child1, child2};
		return offspring;
	}

	/**
	 *Mutate (Flip a bit in the bitset) with probability mProb
	 *@param original the entity to be mutated
	 *@param mProb the probability of a bit being mutated
 *@return the (possibly) mutated entity
 */
public static BitSet mutate(BitSet original, double mProb)
{
        Random rand = new Random();
        for(int m = 0; m < 71; m++)
        {
                //Small possibility a bit copied from parent to child is mutated
                if(rand.nextDouble() <= mProb)
                        original.flip(m);
        }
        //Return the (possibly) strategy
        return original;
}


/**
 *Linear fitness scaling of an array of
 *{@link  ie.errity.pd.Prisoner Prisoners}
 *<BR>Based on the {@link  ie.errity.pd.Prisoner Prisoner's} Scores
 *@param curPopulation the array of {@link  ie.errity.pd.Prisoner Prisoners}
 * to be scaled
 *@return the scaled fitnesses
 */
public static int[] scale(Prisoner[] curPopulation)
{
        //init
        int min, max, total = 0;
        double avg = 0;
        int fs = 0;
        double a,b,fsmax, delta = 0;
        final int cmult = 2;
        int popSize = curPopulation.length;
        int [] scaled = new int[popSize];

        //Calculate min, max and average payoffs
        min = curPopulation[0].getScore();
        max = curPopulation[0].getScore();
        total = curPopulation[0].getScore();
        for(int i = 1; i < popSize; i++)
        {
                if(curPopulation[i].getScore() < min)
                        min = curPopulation[i].getScore();
                if(curPopulation[i].getScore() > max)
                        max = curPopulation[i].getScore();
                total += curPopulation[i].getScore();
        }
        avg = total/popSize;
```

```
		//Calculate scaling factors
		if( min > ((cmult*avg - max) / (cmult - 1)) ) //non-negative test
		{
			delta = max - avg;
			fsmax = cmult * avg;
			a = (cmult-1) * avg / delta;
			b = avg * ((max-fsmax)/delta);
		}
		else //negative: scale as much as possible
		{
			delta = avg - min;
			a = avg / delta;
			b = -1 * min * avg / delta;
		}

		//scale each player's fitness value
		for(int s = 0; s < popSize; s++)
		{
			fs = (int) ((a*curPopulation[s].getScore()) + b);
			scaled[s] = fs;
		}

		return scaled;
	}
```

Extract from Spatial.java

```
	/**
	 *Evolve a new generation using a Genetic algorithm
	 */
	public void Mate()
	{
		int [] parent1 = new int[2];
		int [] parent2 = new int[2];
		int [] weak = new int[2];
		BitSet[] Offspring = new BitSet[2];

		//Fitness scale the payoffs
		scaledScores = Genetic.scale(scores);

		//SELECTION
		parent1 = fitSelect();
		parent2 = fitSelectMate(parent1);


		//CROSSOVER
		if(rand.nextDouble() <= rules.getCrossP())
			Offspring = Genetic.crossover(world[ parent1[0] ][ parent1[1]
].getStrat(),world[parent2[0]][parent2[1]].getStrat());
```

```
        else //CLONE
        {
                Offspring[0] = world[parent1[0]][parent1[1]].getStrat();
                Offspring[1] = world[parent2[0]][parent2[1]].getStrat();
        }

        //MUTATION
        Offspring[0] = Genetic.mutate(Offspring[0], rules.getMutateP());
        Offspring[1] = Genetic.mutate(Offspring[1], rules.getMutateP());

        //REPLACEMENT
        weak = weakReplace(parent1); //find who to replace
        world[ weak[0] ][ weak[1] ] = new Prisoner(Offspring[0]);
        update(weak[0],weak[1]); //update scores

        weak = weakReplace(parent2); //find who to replace
        world[ weak[0] ][ weak[1] ] = new Prisoner(Offspring[1]);
        update(weak[0],weak[1]); //update scores

}


/**
 *Evolve a new generation using an Evolutionary algorithm
 */
public void Evolve()
{
        //fitness scaling
        scaledScores = Genetic.scale(scores);

        //pick random player
        int x = 0;
        int y = 0;
        try{
        x = rand.nextInt(X);
        y = rand.nextInt(Y);
        }
        catch(Exception e){}
        //find fittest surrounding
        int [] fittest = fitSelectMate(new int[]{x,y});

        //replace current with fittest
        if(scaledScores[ fittest[0] ][ fittest[1] ] > scaledScores[x][y])
        {
                //Very small mutatations
                BitSet fitB = world[ fittest[0] ][ fittest[1] ].getStrat();
                if(rand.nextDouble() <= rules.getMutateP())
                        fitB.flip(rand.nextInt(71));

                Prisoner fit = new Prisoner(fitB);
                world[x][y] = fit;
                update(x,y);
        }

}



/**
 *Fitness proportional selection (roulette wheel)
 *@return the [x,y] coordinates selected
 */
private int[] fitSelect()
{
        double t1, fitSum = 0;
        int target;

        //Set Target Fitness
        for(int i = 0; i < X; i++)
                for(int j = 0; j < Y; j++)
                                fitSum += scaledScores[i][j];
        t1 = fitSum * rand.nextDouble();
        target = (int)t1;

        //Build up a sum of fitness
        //the individual who's fitness causes the sum to
        //exceed the target is selected
        int fitness = 0;
        for(int i = lastP[0] + 1; ;i++)
                for(int j = lastP[1] + 1; ;j++)
                {
                        if(i >= X)
                                i = 0;
                        if(j >= Y)
                                j = 0;
                        if(i != lastP[0] || j != lastP[1])
                                fitness += scaledScores[i][j];
                        if(fitness >= target)
                        {
                                lastP[0] = i;
                                lastP[1] = j;
                                return lastP;
                        }
                }

}
```

Extract from Breeder.java

```
    /**
     *Breeds the next generation (panmictic mating) of an array of
     *{@link  ie.errity.pd.Prisoner Prisoners}
     *@param c     initial population (raw fitness of population must be
     *calculated previously)
     *@return the next generation
     */
    public Prisoner[] Breed(Prisoner[] c)
    {
        BitSet Offspring[]; //stores children
        curPopulation = c;      //population to breed
        popSize = curPopulation.length;
        Prisoner newPopulation[] = new Prisoner[popSize];;
        int P1,P2,d;     //parents and index

        //fitness scaling
        scaledScores = Genetic.scale(curPopulation);

        //Breed new population
        d = 0;
        while(d < popSize)
        {
            Offspring = new BitSet[2];

            //Selection
            P1 = selectRoulette();
            P2 = selectRoulette();

            //Cross Over
            if(rand.nextDouble() <= crossP)
                Offspring =
Genetic.crossover(curPopulation[P1].getStrat(),curPopulation[P2].getStrat());
            else //clone
            {
                Offspring[0] = curPopulation[P1].getStrat();
                Offspring[1] = curPopulation[P2].getStrat();
            }

            //Mutation
            Offspring[0] = Genetic.mutate(Offspring[0],mutateP);
            Offspring[1] = Genetic.mutate(Offspring[1],mutateP);

            //Replacement
            newPopulation[d] = new Prisoner(Offspring[0]);
            if( (d+1) < popSize)    //in case of odd population
                newPopulation[d+1] = new Prisoner(Offspring[1]);

            d+=2;
```

```
		}

		curPopulation = newPopulation;
		repaint();		//update display (if any)
		return curPopulation; //return the bred population
	}

	/**
	 *Roulette wheel selection
	 */
	private int selectRoulette()
	{
		double t1, fitSum = 0;
		int target;

		//Set Target Fitness
		for(int j = 0; j < popSize; j++)
				fitSum += scaledScores[j];
		t1 = fitSum * rand.nextDouble();
		target = (int)t1;

		//Build up a sum of fitness
		//the individual who's fitness causes the sum to
		//exceed the target is selected
		int fitness = 0;
		int nextparent = lastparent;
		while(fitness < target)
		{
			nextparent++;
			if(nextparent >= popSize)
				nextparent = 0;

			if(nextparent != lastparent)
				fitness += scaledScores[nextparent];
		}
		lastparent = nextparent;
		return nextparent; //return index of selected player
	}
```

ie.errity.pd.graphics

Extract from TournamentPanel.java

```
/**
 *Start Evolution
 *<BR>Runs in a seperated thread so GUI remains responsive
 */
public void start()
{
        if(stopped == true)
        {
                //Separate thread for computationally intensive evolution
                final SwingWorker worker = new SwingWorker()
                {
        public Object construct()
   {
        //init
        int maxIndex, minIndex;
        //capture current settings
                int gen = rules.getGenerations();
                int numP = rules.getNumPlayers();
                Rules r1 = rules;
                graphPanel.setMax(numP*r1.getIterations()*r1.getT());
                //Clear old data
                graphPanel.clear();
                prisTypes = new double[]{0,0,0,0,0,0,0,0};

                /*Non HTML labels
                minLbl.setText("Minimum Payoff: " + (new Integer(0)).toString());
                maxLbl.setText("Maximum Payoff: " + (new Integer(0)).toString());
                avgLbl.setText("Average Payoff: " + (new Double(0)).toString());
                */

                //HTML labels
                minLbl.setText("<html><font color=#6B238E>Minimum Payoff: " +
(new Integer(0)).toString() + "</font>");
                maxLbl.setText("<html><font color=#CFB53B>Maximum Payoff: " +
(new Integer(0)).toString() + "</font>" );
                avgLbl.setText("<html><font color=#215E21>Average Payoff: " +
(new Double(0)).toString() + "</font>");


                type0.setText("<html><font color=" + color[0] +">" + typeNames[0] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type1.setText("<html><font color=" + color[1] +">" + typeNames[1] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type2.setText("<html><font color=" + color[2] +">" + typeNames[2] +
": " + (new Double(0)).toString() + " %" + "</font>");
```

```
                type3.setText("<html><font color=" + color[3] +">" + typeNames[3] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type4.setText("<html><font color=" + color[4] +">" + typeNames[4] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type5.setText("<html><font color=" + color[5] +">" + typeNames[5] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type6.setText("<html><font color=" + color[6] +">" + typeNames[6] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type7.setText("<html><font color=" + color[7] +">" + typeNames[7] +
": " + (new Double(0)).toString() + " %" + "</font>");



                b1.setRules(r1);
                Prisoner[] pris = Prisoner.getRand(numP);

                Tournament t1;
        for(int i = 0; i < gen; i++)
                {
                        if(stopped)     //if stop clicked end

                                break;

                        t1 = new Tournament(pris,r1);
                        t1.Play();

                        //***** reporting ******************
                        prisTypes = new double[]{0,0,0,0,0,0,0,0};
                        prisTypes[pris[0].getType()]  = prisTypes[pris[0].getType()] +
1;
                        maxIndex = 0;
                        minIndex = 0;
                        min = pris[0].getScore();
                        max = pris[0].getScore();
                        total = pris[0].getScore();
                        for(int j = 1; j < numP; j++)
                        {
                                if(pris[j].getScore() < min)
                                {
                                        min = pris[j].getScore();
                                        minIndex = j;
                                }
                                if(pris[j].getScore() > max)
                                {
                                        max = pris[j].getScore();
                                        maxIndex = j;
                                }
                                total += pris[j].getScore();
                                prisTypes[pris[j].getType()]   =
prisTypes[pris[j].getType()] + 1;
                        }
                        avg = total/numP;
                        graphPanel.addData(min,max,avg);


                        type0.setText("<html><font color=" + color[0] +">" +
typeNames[0] + ": " + (new Double((prisTypes[0]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type1.setText("<html><font color=" + color[1] +">" +
typeNames[1] + ": " + (new Double((prisTypes[1]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type2.setText("<html><font color=" + color[2] +">" +
typeNames[2] + ": " + (new Double((prisTypes[2]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type3.setText("<html><font color=" + color[3] +">" +
typeNames[3] + ": " + (new Double((prisTypes[3]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type4.setText("<html><font color=" + color[4] +">" +
typeNames[4] + ": " + (new Double((prisTypes[4]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type5.setText("<html><font color=" + color[5] +">" +
typeNames[5] + ": " + (new Double((prisTypes[5]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type6.setText("<html><font color=" + color[6] +">" +
typeNames[6] + ": " + (new Double((prisTypes[6]/(double)numP)*100)).toString() +
" %" + "</font>");
                        type7.setText("<html><font color=" + color[7] +">" +
typeNames[7] + ": " + (new Double((prisTypes[7]/(double)numP)*100)).toString() +
" %" + "</font>");

                        //Have now calculated player fitnesses, allow user to view
prisoners
                        weakDlg.setStrat(pris[minIndex]);
                        fitDlg.setStrat(pris[maxIndex]);
                        weakBtn.setEnabled(true);
                        fitBtn.setEnabled(true);

                        /*Non HTML labels
                        minLbl.setText("Minimum Payoff: " + (new
Double((double)min/(numP*(double)r1.getIterations()))).toString());
                        maxLbl.setText("Maximum Payoff: " + (new
Double((double)max/(numP*(double)r1.getIterations()))).toString());
                        avgLbl.setText("Average Payoff: " + (new
Double((double)avg/(numP*(double)r1.getIterations()))).toString());
                        */

                        //HTML Labels
                        minLbl.setText("<html><font color=#6B238E>Minimum
Payoff: " +  (new Double((double)min/(numP*(double)r1.getIterations()))).toString()
+ "</font>");
                        maxLbl.setText("<html><font color=#CFB53B>Maximum
Payoff: " + (new Double((double)max/(numP*(double)r1.getIterations()))).toString()
+ "</font>" );
                        avgLbl.setText("<html><font color=#215E21>Average Payoff:
" + (new Double((double)avg/(numP*(double)r1.getIterations()))).toString() +
"</font>");


                        //**** reporting end ************
                        if(stopped)             //if stop clicked end
                                break;
                        //Evolve a generation
                        pris = b1.Breed(pris);
                }
                //Finished - Enable/Disable Buttons as required
                stopped = true;
                stopBtn.setEnabled(false);
                startBtn.setEnabled(true);
     return pris;
        }
           };
                //Starting - Enable/Disable Buttons as required
                stopBtn.setEnabled(true);
                startBtn.setEnabled(false);
                stopped = false;
                worker.start(); //Start evolution in background thread
        }
 }

/**
 *Stops Evolution
 *<BR>Ensures background thread is closed
 */
public void stop()
{
        stopped = true;
        //Finished - Enable/Disable Buttons as required
        stopBtn.setEnabled(false);
        startBtn.setEnabled(true);
}
```

Extract from SpatialPanel.java

```
/**
 *Start Evolution
 *<BR>Runs in a seperated thread so GUI remains responsive
 */
public void start()
```

```
{
        if(stopped == true)
        {
                //Separate thread for computationally intensive evolution
        final SwingWorker worker = new SwingWorker()
                {
        public Object construct()
   {
        int maxIndexA, minIndexA, maxIndexB, minIndexB;

        boolean mate = GA;   //capture current settings
        Rules r1 = rules;         //capture current settings
                int gen = rules.getGenerations();
                int numP = rules.getNumPlayers();

                //Clear old data
                grid.clear();
                graphPanel.clear();
                graphPanel.setMax(8*r1.getIterations()*r1.getT());
                prisTypes = new double[]{0,0,0,0,0,0,0,0};

                /*Non HTML labels
                minLbl.setText("Minimum Payoff: " + (new Integer(0)).toString());
                maxLbl.setText("Maximum Payoff: " + (new Integer(0)).toString());
                avgLbl.setText("Average Payoff: " + (new Double(0)).toString());
                */

                //HTML labels
                minLbl.setText("<html><font color=#6B238E>Minimum Payoff: " +
(new Integer(0)).toString() + "</font>");
                maxLbl.setText("<html><font color=#CFB53B>Maximum Payoff: " +
(new Integer(0)).toString() + "</font>" );
                avgLbl.setText("<html><font color=#215E21>Average Payoff: " +
(new Double(0)).toString() + "</font>");

                type0.setText("<html><font color=" + color[0] +">" + typeNames[0] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type1.setText("<html><font color=" + color[1] +">" + typeNames[1] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type2.setText("<html><font color=" + color[2] +">" + typeNames[2] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type3.setText("<html><font color=" + color[3] +">" + typeNames[3] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type4.setText("<html><font color=" + color[4] +">" + typeNames[4] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type5.setText("<html><font color=" + color[5] +">" + typeNames[5] +
": " + (new Double(0)).toString() + " %" + "</font>");
                type6.setText("<html><font color=" + color[6] +">" + typeNames[6] +
": " + (new Double(0)).toString() + " %" + "</font>");
```

```
		type7.setText("<html><font color=" + color[7] +">" + typeNames[7] +
": " + (new Double(0)).toString() + " %" + "</font>");


		Prisoner[][] pris = Prisoner.getRand(numP,numP);

		grid.setRules(r1);
		grid.setPlayers(pris);
		grid.Play(); //initialisation **may be slow with >75^2 players
		int [][] scores;
		//Evolve the set number of generations
	for(int g = 0; g < gen; g++)
		{
			if(stopped)	//if stop clicked end
				break;

			//***** reporting ******************
			prisTypes = new double[]{0,0,0,0,0,0,0,0};
			scores = grid.getScores();
			min = scores[0][0];
			max = scores[0][0];
			minIndexA = 0;
			minIndexB = 0;
			maxIndexA = 0;
			maxIndexB = 0;
			total = 0;
			for(int i = 0; i < scores.length; i++)
			{
				for(int j = 0; j < scores[0].length; j++)
				{
					if(scores[i][j] < min)
					{
						min = scores[i][j];
						minIndexA = i;
						minIndexB = j;
					}
					if(scores[i][j] > max)
					{
						max = scores[i][j];
						maxIndexA = i;
						maxIndexB = j;
					}
					total += scores[i][j];

					prisTypes[pris[i][j].getType()]		=
prisTypes[pris[i][j].getType()] + 1;
				}
			}
			avg = total/(scores.length*scores[0].length);
			graphPanel.addData(min,max,avg);
```

```
                        type0.setText("<html><font color=" + color[0] +">" +
typeNames[0] + ": " + (new
Double((prisTypes[0]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type1.setText("<html><font color=" + color[1] +">" +
typeNames[1] + ": " + (new
Double((prisTypes[1]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type2.setText("<html><font color=" + color[2] +">" +
typeNames[2] + ": " + (new
Double((prisTypes[2]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type3.setText("<html><font color=" + color[3] +">" +
typeNames[3] + ": " + (new
Double((prisTypes[3]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type4.setText("<html><font color=" + color[4] +">" +
typeNames[4] + ": " + (new
Double((prisTypes[4]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type5.setText("<html><font color=" + color[5] +">" +
typeNames[5] + ": " + (new
Double((prisTypes[5]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type6.setText("<html><font color=" + color[6] +">" +
typeNames[6] + ": " + (new
Double((prisTypes[6]/(double)(numP*numP))*100)).toString() + " %" + "</font>");
                        type7.setText("<html><font color=" + color[7] +">" +
typeNames[7] + ": " + (new
Double((prisTypes[7]/(double)(numP*numP))*100)).toString() + " %" + "</font>");


                        //Have now calculated player fitnesses, allow user to view
prisoners
                        weakDlg.setStrat(pris[minIndexA][minIndexB]);
                        fitDlg.setStrat(pris[maxIndexA][maxIndexB]);
                        weakBtn.setEnabled(true);
                        fitBtn.setEnabled(true);

                        /*Non HTML Labels
                        minLbl.setText("Minimum Payoff: " + (new
Double((double)min/(8*(double)r1.getIterations()))).toString());
                        maxLbl.setText("Maximum Payoff: " + (new
Double((double)max/(8*(double)r1.getIterations()))).toString());
                        avgLbl.setText("Average Payoff: " + (new
Double((double)avg/(8*(double)r1.getIterations()))).toString());
                        */
                        //HTML Labels
                        minLbl.setText("<html><font color=#6B238E>Minimum
Payoff: " +  (new Double((double)min/(8*(double)r1.getIterations()))).toString() +
"</font>");
                        maxLbl.setText("<html><font color=#CFB53B>Maximum
Payoff: " + (new Double((double)max/(8*(double)r1.getIterations()))).toString() +
"</font>" );
```

```
                        avgLbl.setText("<html><font color=#215E21>Average Payoff:
" + (new Double((double)avg/(8*(double)r1.getIterations()))).toString() + "</font>");
                        //***** reporting end *******

                        //Evolve a generation
                        if(!mate)
                        {
                        for(int m = 0; m < (numP*numP)/4; m++)
                        {
                                if(stopped)     //if stop clicked end
                                        break;
                                grid.Evolve(); //use the evolutionary Algorithm
                        }
                        }
                        else
                        { for(int m = 0; m < (numP*numP)/10; m++)
                          {
                                if(stopped)     //if stop clicked end
                                        break;
                                grid.Mate();    //use the Genetic Algorithm
                          }
                        }
                        grid.repaint();
                }
                //Finished - Enable/Disable Buttons as required
                stopped = true;
                stopBtn.setEnabled(false);
                startBtn.setEnabled(true);
                ea.setEnabled(true);
                ga.setEnabled(true);
     return pris;
        }
                };
                //Starting - Enable/Disable Buttons as required
                stopBtn.setEnabled(true);
                startBtn.setEnabled(false);
                ea.setEnabled(false);
                ga.setEnabled(false);
                stopped = false;
                worker.start(); //Start evolution in background thread
        }
}

/**
 *Stops Evolution
 *<BR>Ensures background thread is closed
 */
public void stop()
{
        stopped = true;
        //Finished - Enable/Disable Buttons as required
        stopBtn.setEnabled(false);
        startBtn.setEnabled(true);
        ea.setEnabled(true);
        ga.setEnabled(true);
}
```

Extract from InteractivePanel.java

```
//Computer vs Computer Play
//Plays all iterations of the PD between the two selected players
//Play is performed in a background thread so GUI remains responsive
//Method is responsible for updating the GUI as required
private void move()
{
if(stopped)     //not already playing a game
{
//Create a seperate thread to play the IPD in
final SwingWorker worker = new SwingWorker()
{
public Object construct()
{
        boolean P1move, P2move;

        //Play IPD and update GUI as neccesary
        for(iteration = 0; iteration < itMax; iteration++)
        {
                if(stopped)
                        break;
                P1move = opponent.play(iteration,history);
                P2move = opponent2.play(iteration,P2History);

                if(P1move && P2move) //CC
                {
                        P1Score = rules.getR();
                        P2Score = rules.getR();
                        opponentMove.setText("<html><font color=#FF0000
size=+1>Player 1 Cooperated - Payoff: " + (new Integer(P1Score)).toString()+
"</font>");
                        yourMove.setText("<html><font color=#0000FF
size=+1>Player 2 Cooperated - Payoff: " + (new Integer(P2Score)).toString()+
"</font>");

                        hist1[iteration] = "(Player 1)   C   vs   C   (Player 2)";
                        list1.setListData(hist1);
                }
                else if(P1move && !P2move) //CD
                {
                        P1Score = rules.getS();
                        P2Score = rules.getT();
                        opponentMove.setText("<html><font color=#FF0000
size=+1>Player 1 Cooperated - Payoff: " + (new Integer(P1Score)).toString()+
"</font>");
                        yourMove.setText("<html><font color=#0000FF
size=+1>Player 2 Defected - Payoff: " + (new Integer(P2Score)).toString()+
"</font>");

                        hist1[iteration] = "(Player 1)    C   vs   D    (Player 2)";
                        list1.setListData(hist1);
                }
                else if(!P1move && P2move) //DC
                {
                        P1Score = rules.getT();
                        P2Score = rules.getS();
                        opponentMove.setText("<html><font color=#FF0000
size=+1>Player 1 Defected - Payoff: " + (new Integer(P1Score)).toString()+
"</font>");
                        yourMove.setText("<html><font color=#0000FF
size=+1>Player 2 Cooperated - Payoff: " + (new Integer(P2Score)).toString()+
"</font>");

                        hist1[iteration] = "(Player 1)    D   vs   C    (Player 2)";
                        list1.setListData(hist1);
                }
                else if(!P1move && !P2move) //DD
                {
                        P1Score = rules.getP();
                        P2Score = rules.getP();
                        opponentMove.setText("<html><font color=#FF0000
size=+1>Player 1 Defected - Payoff: " + (new Integer(P1Score)).toString()+
"</font>");
                        yourMove.setText("<html><font color=#0000FF
size=+1>Player 2 Defected - Payoff: " + (new Integer(P2Score)).toString()+
"</font>");

                        hist1[iteration] = "(Player 1)    D   vs   D    (Player 2)";
                        list1.setListData(hist1);
                }

                //Update player histories
                if(P1move)
                {
                        history.set(iteration*2);
                        P2History.set((iteration*2)+1);
                }
                else
                {
                        history.clear(iteration*2);
                        P2History.clear((iteration*2)+1);
                }
```

```
                    if(P2move)
                    {
                              history.set((iteration*2)+1);
                              P2History.set((iteration*2));
                    }
                    else
                    {
                              history.clear((iteration*2)+1);
                              P2History.clear((iteration*2));
                    }
                    //Update Scores
                    P1Total += P1Score;
                    P2Total += P2Score;
                    opponentScore.setText("<html><font color=#FF0000 size=+2>Player
1's Total Score: " +  (new Integer(P1Total)).toString() + "</font>");
                    yourScore.setText("<html><font color=#0000FF size=+2>Player 2's
Total Score: " +  (new Integer(P2Total)).toString() + "</font>");

          }
          //When all iterations completed
          opponentMove.setText("<html><font color=#FFFFFF size=+1>Game
over!</font>");
          yourMove.setText("<html><font color=#0000FF size=+1> </font>");
          //Enable/Disable buttons as required
          cBtn.setEnabled(false);
          dBtn.setEnabled(false);
          stopBtn.setEnabled(false);
          playBtn.setEnabled(true);
          prisList.setEnabled(true);
          if(!humanp)
          {
                    prisList2.setEnabled(true);

          }
          human.setEnabled(true);
          comp.setEnabled(true);
          stopped = true; //Game has ended
          return opponent;
}
};
worker.start(); //Start the game thread
}
}
```