Ludwig-Maximilians-Universität
München

The Application of the Genetic Algorithm to Game Theory

Yakov Benilov

Master's Thesis
March 2006

I would like to thank Professor Martin Schottenloher for the advice and assistance I received during this thesis; my reviewers Markus and Jelena for their useful suggestions and corrections; Professor Marks for providing the life-saving experiment resources; and my parents, for their patience and encouragement while I wrote this work.

Contents

List of Tables

List of Figures

List of Program Code

Chapter 0

Introduction

The genetic algorithm (GA) is a powerful computational technique for optimisation. The aim of this thesis is to establish a formal language for applying this technique in the context of strategic game theory, and to illustrate it with worked examples drawn from real-world game theory problems.

Due to their heuristic nature, the methods outlined here can be used during the exploratory stages of problem analysis (to get an idea of the solution landscape, or to find approximate solutions), before the analytical theory has been developed.

While important work has already been done in the field of GA applications to game theory, it has been predominantly approached from the economic and computer theoretic angle; as a consequence, some of the work has lacked mathematical rigour. This thesis attempts to formalise some of the previously developed concepts, using game theoretic language that has been developed in the previous 15 years, by introducing a genetic algorithm specifically for games.

The main points of interest in this thesis are a revisit of an important 1987 experiment staged by Robert Axelrod (described in [Axelrod, 1987]), with the aim of clearing up some of the finer implementation points using more formal notation, and an illustration of how the genetic game algorithm (GGA) may be used to heuristically search for Nash equilibria in extensive games, using a model called the Contract Game (taken from [Huck et al., 2003]). Both include experimental analysis, using the simulation that was created based on the theory in this thesis.

The application of Markov chain theory to the analysis of genetic algorithms has yielded some interesting results concerning convergence. The research done in this thesis can be extended by further investigation into the possibility of transferring Markov theory to the genetic algorithms in the domain of games.

The chapter breakdown is as follows: Chapters 1 and 2 provide introductions to the genetic algorithm and game theory, respectively. Chapter 3 ties the exposition from Chapters 1 and 2 together by providing several versions of the GGA. Chapter 4 applies the material from Chapter 3 to Axelrod's experiment. Chapter 5 focuses on the Contract Game model, first tackling it analytically and then using the GGA; a comparison of the two approaches is then made. Chapter 6 is the conclusion. The appendices contain details about the simulation code for this thesis (Appendix A), and various miscellaneous information (Appendices B and C).

Chapter 1

The Genetic Algorithm

The purpose of this chapter is to define a vocabulary of terms and concepts that are necessary for our discussion of genetic algorithms (Section 1.1), to give a basic introduction to genetic algorithms (Section 1.2), and to illustrate the presented ideas using one specific algorithm: the Canonical Genetic Algorithm (CGA) (Section 1.3).

1.1 Introduction

A genetic algorithm (GA) is an algorithmic search technique used to find approximate solutions to optimisation and search problems. The GA's primary application is in situations where a multidimensional, non-linear function needs to be maximised/minimised, and the solution need not be exact, but rather "good enough". Genetic algorithms belong to the class of methods known as "weak methods" in the Artificial Intelligence community because they makes relatively few assumptions about the problem that is being solved - this makes GAs ideal for "feeling out" a problem domain and finding solution candidates, prior to launching into in-depth theoretical analysis.

The GA has three main components, (the first two of) which mimic similar concepts in biological evolution:

1. a sequence of chromosome populations
2. a genetic mechanism which allow a population to be generated from its predecessor; this mechanism mirrors the main evolutionary processes - fitness evaluation, selection, recombination and mutation.
3. a terminating condition

We shall now give definitions and explanations of the above terms:

- In this context, a **chromosome** (or **binary string**) $b = (b_1, b_2, ..., b_m)$ of length m is a sequence of m **genes**. Each gene is a binary number: $b_i \in \{0, 1\}\ \forall i$.
- A **population of size** n is a collection of n chromosomes of equal length; because a unique chromosome can appear more than once in a population, we represent it with an n-tuple, rather than a set.
- The genetic mechanism (mentioned earlier) is best thought of as a stochastic function Ω that transforms one population into another (of equal size); thus, it is possible for us to define a population sequence $(P(i))_{i \geq 0}$, such that

$$P(j+1) = \Omega(P(j))\ \forall j \geq 0 \tag{1.1}$$

  The i^{th} population in the population sequence is often referred to as **generation** i, or **the** i^{th} **generation**. The 0^{th} population (generation 0), commonly called the **initial population**, is the starting point for the algorithm and is passed to it as a parameter.
- **Encoding/decoding** connects points in the problem domain to chromosomes - the initial population may be comprised of encoded points

from the problem domain (for example, if the approximate area of the fitness maximum is known before the GA is run, the initial population may be "seeded" by points from this area), and a chromosome from the final generation can be decoded back into a problem domain point when the GA run terminates.

- **Fitness evaluation** is a process that assigns a quantitative value to chromosome, based on some metric. That metric may be dynamic (that is, the fitness of a chromosome relies on what the other chromosomes in the population are) or static (the fitness of a chromosome is independent of the other chromosomes in the population).
- **Selection** is a process that can be thought of as a gate-keeper - it regulates which chromosomes from one generation play a part in the next generation, and which do not. Selection improves the overall population fitness by preventing the propagation of chromosomes with low fitness values.
- **Crossover**, or **recombination** is a process that can create two new chromosomes (children) from two existing chromosomes (parents); each child shares genes with both of its parents. Crossover facilitates the creation of chromosomes that combine the "best" parts of its parents.
- **Mutation** is a process that stochastically makes small changes to chromosomes. Mutation helps prevent premature homogeny of a population and facilitates discovery of previously unvisited optima in the search space.
- The **terminating condition**, when satisfied, signals the end of the GA run - this condition may be chosen to assert whether the best or average fitness has reached a certain (minimum) level, or perhaps the

condition may be a time constraint (that is, it may assert whether a generation has been reached or not).

- When we talk about **population sequence convergence**, we refer to the situation when a high level of homogeny exists within the population sequence over several generations; this usually implies that the chromosomes represent a (local or global) maximum of the fitness function.

The simple description of the GA inner workings is that starting with an initial population of chromosomes, subsequent generations are created by putting the previous generation through the genetic mechanisms. The GA is designed so that both the maximum and average fitness of strategies in each generation are predominantly increasing[1] with time - new populations continue to be generated until the terminating condition is satisfied.

In a strict interpretation, *the genetic algorithm* refers to a model introduced and investigated by John Holland [Holland, 1975] and by students of Holland (e.g. [DeJong, 1975]). It is still the case that most of the existing theory for genetic algorithms applies either solely or primarily to the model introduced by Holland, as well as variations on *the canonical genetic algorithm* (see Section 1.3.1)[2].

1.1.1 Remarks

Unlike some other global search methods, genetic algorithms does not use gradient information; this also makes their use appropriate in problems involving non-differentiable functions, or functions with multiple local optima.

[1]The fitness increase is not monotonic - fluctuations on the local time-frame are normal, but overall growth is generally observed.

[2][Whitley, 1994]

The fact that GAs make relatively few assumptions about the problem that is being solved, is an advantage when creating a software implementation of a GA: time can be saved by using "off-the-shelf" components, instead of creating them from scratch - numerous software GA frameworks (which implement the commonly used mutation, selection and crossover functions) exist already.

The GA's generality brings a certain degree of robustness, but the downside is that domain-specific methods, where they exist, often out-perform the GA in terms of computational cost. A common technique is to try to take the best from both worlds, and to create hybrid algorithms from the combination of GAs and existing methods.

When adapting the GA to their specific needs, problem solvers need to make sure that they are performing the correct optimisation - unless an appropriate choice of fitness function is made, the output of the genetic algorithm may not be useful to the original problem.

The theory of Markov chains has been demonstrated to be a very powerful tool for the theoretical analysis of GAs. There are mainly two approaches to modeling GAs as Markov chains. The first approach, called population Markov Chain model, views the sequence of population in GAs as finite Markov chains on population space(Eiben, Aarts, and Hee (1991), Fogel (1994), Rudolph (1994)), Leung, Gao and Xu (1997)), while the second approach models the GAs by identifying the states of the population with probability vectors over the individual space[3] (Reynolds and Gomatam (1996), Vose (1996)).

[3]See Section 1.3.1 for explanation of what an individual space is.

1.2 Genetic Algorithm Specifics

An implementation of a genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to "reproduce" than those chromosomes which are poorer solutions. The "goodness" of a solution is typically defined with respect to the current population.

It is helpful to view the main execution loop of the genetic algorithm as a two stage process. It starts with the fitness evaluation of the current population. Selection is applied to the current population to create an intermediate population. Then crossover (recombination) and mutation are applied to the intermediate population to create the next population. Crossover can be viewed as creating the next population from the intermediate population. Crossover is applied to randomly paired chromosomes with a probability denoted p_c (the population should already be sufficiently shuffled by the random selection process). Pick a pair of chromosomes. With probability p_c "recombine" these chromosomes to form two new chromosomes that are inserted into the next population. The process of evaluation, selection, crossover and mutation forms one generation in the execution of a genetic algorithm.

We can summarise these steps in a flowchart (Figure 1.1).

The pseudo-code representation of the GA can be seen in Listing 1.1. Usually there are only two main components of most genetic algorithms that are problem dependent: the problem encoding and the evaluation function. The remaining components can be reused, and only their parameters (such as the mutation parameter, crossover parameter, population size) are tuned to fit the simulation.

As we mentioned earlier, we can divide the fitness metrics into two groups - dynamic and static. With a static fitness function, the fitness of a chromo-

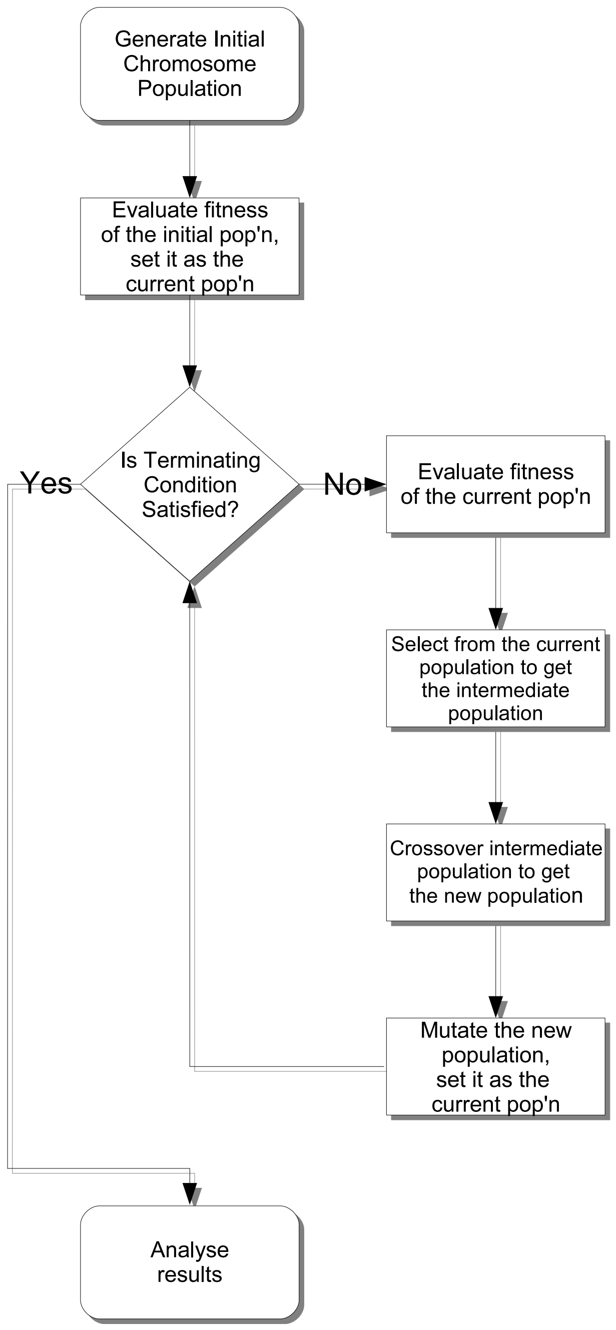

Generate Initial Chromosome Population
Evaluate fitness of the initial pop'n, set it as the current pop'n
Is Terminating Condition Satisfied?
Yes
No
Evaluate fitness of the current pop'n
Select from the current population to get the intermediate population
Crossover intermediate population to get the new population
Mutate the new population, set it as the current pop'n
Analyse results

Figure 1.1: The GA flowchart

```
i=0                         // set generation number to zero
initpopulation P(0)         // initialise a usually random population of individuals
evaluate P(0)               // evaluate fitness of all initial individuals of population
while (not done) do         // test for termination condition (time, fitness, etc.)
begin
    evaluate P(i)           // evaluate the fitness
    i = i + 1               // increase the generation number
    select P(i) from P(i-1) // select a sub-population for offspring reproduction
    recombine P(i)          // recombine the genes of selected parents
    mutate P(i)             // perturb the mated population stochastically
end
```

Listing 1.1: Genetic Algorithm Pseudo-code

some in a population is independent of the fitness of the other chromosomes in the same population - the fitness value of a chromosome is absolute. With a dynamic fitness function, the fitness values are interdependent within a population - the fitness value of a chromosome is relative. This means that with a static fitness function, chromosomes from different populations can be compared and ranked by their fitness values; this is not possible with a dynamic fitness function, because a chromosome's fitness only makes sense in the context of the population that it is in. The type of fitness function depends on the nature of the landscape being searched by the GA - if the GA is optimising a variable Examples of both types appear in this thesis: the original Axelrod experiment in Chapter 4 uses a static fitness function, while the contract game experiment in Chapter 5 uses a dynamic fitness function.

1.3 The Canonical Genetic Algorithm

We now provide an example of a genetic algorithm: the canonical genetic algorithm. The CGA defines specific selection, crossover and mutatation functions, but the fitness, the encoding/decoding functions and the terminating condition all remain problem/simulation specific. Here, we adopt the more mathematically formal notation that shall be used extensively in later chapters.

1.3.1 Formal Definition

Taken from Gao [1998]:

We consider the GAs with binary string representations of the encoding length l and the fixed population size n. The set of chromosomes, or individuals (encoded feasible solutions) is denoted by $B = \{0,1\}^l$ and is called the *individual space*. The set of populations with size n is denoted by B^n. Particularly, we call $B^2 = B \times B$ the *parents space*. The fitness function $f : B \to R_0^+$ can be derived from the objective function of the optimization problem by a certain decoding rule.

With respect to selection in the CGA, the probability that chromosomes in the current population are copied (i.e., duplicated) and placed in the intermediate generation is proportion to their fitness. We view the population as mapping onto a roulette wheel, where each individual is represented by a space that proportionally corresponds to its fitness. By repeatedly spinning the roulette wheel, individuals are chosen using "stochastic sampling with replacement" to fill the intermediate population.

Definition 1 (Roulette Wheel Selection)**.** *The* ***proportional selection operator****, ${}^{GA}T_s^f : B^n \to B^2$, selects a pair of parents from the given population for reproduction, based on the relative fitness (which is defined by function f) of the individual in the population. Given the population $\vec{X}$, the probability of selecting $(X_i, X_j) \in B^2$ as the parents is*

$$P\{{}^{GA}T_s^f(\vec{X}) = (X_i, X_j)\} = \frac{f(X_i)}{\sum_{X \in \vec{X}} f(X)} \cdot \frac{f(X_j)}{\sum_{X \in \vec{X}} f(X)} \qquad (1.2)$$

with $1 \leq i \leq n$, $1 \leq j \leq n$.

For crossover, the CGA uses a technique known as 1-point crossover. Given two parents, this function generates the first child by first choosing a random position, and then substituting with the crossover probability p_c, the gene segment after the chosen position in the first parent by the gene segment after the chosen position in the second parent. The second child is formed from the left-over segments from the parents. With probability $1-p_c$, the children are simply the parents.

Definition 2 (1-point Crossover)**.** *For $x = (x_1, ..., x_l) \in B$, $y = (y_1, ..., y_l) \in B$, ${}^{GA}T_c : B^2 \rightarrow B^2$ is defined as:*

$$P({}^{GA}T_c(x, y) = (x, y)) \quad = \quad 1 - p_c \tag{1.3}$$

$$P({}^{GA}T_c(x, y) = (z_k, w_k)) \quad = \quad \frac{p_c}{l} \ \forall k = 1, ..., l \tag{1.4}$$

where $z_k = (x_1, ..., x_k, y_{k+1}, ..., y_l)$ and $w_k = (y_1, ..., y_k, x_{k+1}, ..., x_l)$.

Example 1 (1-point Crossover)**.** *Consider binary chromosomes 1101001100 and yxyyxyxxyy (in the latter, the values 0 and 1 are denoted by x and y). Using a single randomly chosen recombination point, 1-point crossover occurs as follows:*

```
11010 \/ 01100
yxyyx /\ yxxyy
```

Swapping the fragments between the two parents produces the following offspring:

11010yxxyy and yxyyx01100

The mutation used in the CGA flips the selected bit (as opposed to generating a random replacement for it).

Definition 3 (Mutation)**.** *The **mutation operator**, ${}^{GA}T_m : B \to B$, operates on the individual by independently perturbing each gene in a probabilistic manner and can be specified as follows:*

$$P\{{}^{GA}T_m(X) = Y\} = p_m^{|X-Y|}(1 - p_m)^{1-|X-Y|} \tag{1.5}$$

where p_m is the mutation probability.

Finally, we can give the recursive definition of the population sequence in the CGA.

Definition 4. *Based on the genetic operators defined above and a given initial population $\vec{X}(0)$ of size n, the **canonical genetic algorithm (CGA)** can be represented as the following iteration of populations:*

$$\vec{X}(k+1) = \{T_m^i(T_c^i(T_s^i(\vec{X}(k)))),\ i = 1, ..., n\},\ k \geq 0 \tag{1.6}$$

where (T_m^i, T_s^i), $i = 1, ..., n$ are independent versions of $({}^{GA}T_m,\ {}^{GA}T_s)$ and

$$(T_c^{2j-1}, T_c^{2j}) =^{GA} T_c^j,\ j = 1, ..., \frac{n}{2} \tag{1.7}$$

where ${}^{GA}T_c^j$ are independent versions of ${}^{GA}T_c$.

Chapter 2

Game Theory Notions

In this chapter, we define, discuss and illustrate game theoretic concepts that shall be used in consequent chapters. The strategic games section (Section 2.2) covers the basic concept of a strategic game, payoff functions and symmetric games. The extensive games section (Section 2.3) briefly covers extensive games with both perfect and imperfect information, repeated games, strategies in such games, and player recall.

2.1 Notes

This chapter is heavy on exposition; the main reason behind this is that many of the definitions build upon each other, as can be seen in Figure 2.1. Nonetheless, several ideas have been omitted for simplicity; for example, all the games involved are pure strategy games (no mixed strategies), and do not involve chance. There is further explanation as to why mixed strategies do not feature, in Section 3.3.

Most of the material in this chapter features in [Osborne and Rubinstein, 1994], albeit edited and presented with the narrow focus on what is required for later on.

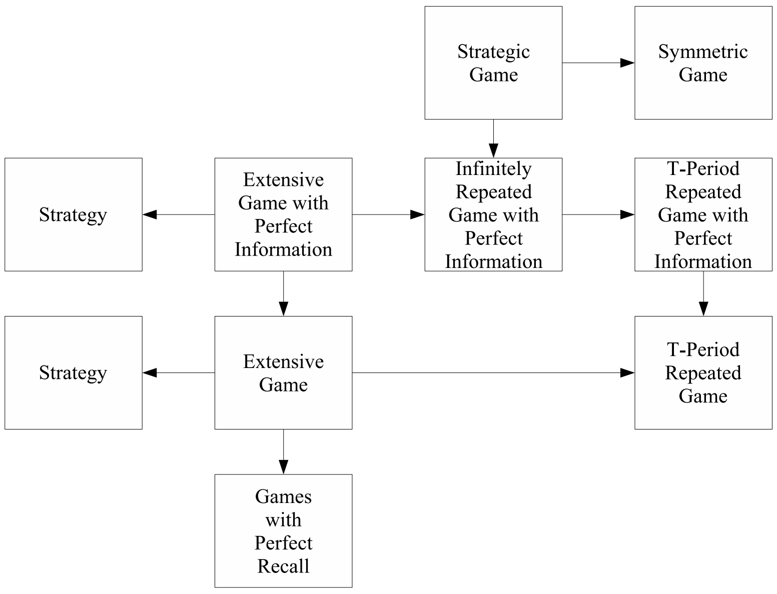

Strategic Game
Symmetric Game
Strategy
Extensive Game with Perfect Information
Infinitely Repeated Game with Perfect Information
T-Period Repeated Game with Perfect Information
Strategy
Extensive Game
T-Period Repeated Game
Games with Perfect Recall

Figure 2.1: Game Definition Dependencies

2.2 Strategic Games

Before we launch into the theoretical definitions, we shall introduce probably the most widely known game, the Prisoner's Dilemma Game.

Definition 5 (Prisoner's Dilemma)**.** *The* ***Prisoner's Dilemma (PD)*** *is a two-player game in which each player has only two pure strategies: cooperation (C) and defection (D). In any given round, the two players receive R points if both cooperate and only P points if both defect; a defector who plays a cooperator gets T points, while the cooperator receives S (with $T > R > P > S$ and $2R > T + S$).*

$$A = \begin{pmatrix} R & S \\ T & P \end{pmatrix}, \; B = \begin{pmatrix} R & T \\ S & P \end{pmatrix}, \tag{2.1}$$

Example 2.

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix}, \; B = \begin{pmatrix} 3 & 5 \\ 0 & 1 \end{pmatrix}, \tag{2.2}$$

The 2-player game with payoff matrices (A,B) is an example of the Prisoner's Dilemma Game, as $5 > 3 > 1 > 0$ *and* $2 \times 3 > 5 + 0$.

The models we study assume that each decision-maker is "rational" in the sense that he is aware of his alternatives, forms expectations about any unknowns, has clear preferences, and chooses his action deliberately after some process of optimisation. In the absence of uncertainty, the following elements constitute a model of rational choice:

- A set A of actions from which the decision-maker makes a choice
- A set C of possible consequences of these actions
- A consequence function $g : A \to C$ that associates a consequence with each action
- A preference relation (a complete transitive reflexive binary relation) $\succsim$ on the set C.

Sometimes the decision-maker's preferences are specified by giving a utility function $U : C \to \mathbb{R}$, which defines a preference relation $\succsim$ by the condition $x \succsim y$ if and only if $U(x) \geq U(y)$.

Given any set $B \subseteq A$ of actions that are feasible in some particular case, a rational decision-maker chooses a feasable action $a^* \in B$, which is optimal in the sense that $g(a^*) \succsim g(a)$ for all $a \in B$; alternatively he solves the problem $max_{a \in B} U(g(a))$.

A strategic game is a model of interactive decision-making in which each decision-maker chooses his plan of action at once and for all, and these choices are made simultaneously. The model consists of a finite set N of players and, for each player i, a set A_i of actions and a preference relation on the set of action profiles (a profile is a collection of values of some variable, one for each player). We refer to an action profile $a = (a_j)_{j \in N}$ as an outcome, and denote the set $\times_{j \in N} A_j$ of outcomes by A.

Notation. *For any profile $x = (x_j)_{j \in N}$ and any $i \in N$ we let x_{-i} be the list $(x_j)_{j \in N \setminus \{i\}}$ of elements of the profile x for all players except i.*

The formal definition of a strategic game is the following.

Definition 6 (Strategic Game)**.** *A **strategic game**, or **normal form game** consists of*

1. *a finite set N (the set of **players**)*
2. *a nonempty set A_i (the set of **actions** available to player i) for each player $i \in N$*
3. *a preference relation $\succsim_i$ on $A = \times_{j \in N} A_j$ (the **preference relation** of player i) for each player $i \in N$*

If the set A_i of actions of every player i is finite then the game is finite.

Definition 7 (Payoff Function)**.** *Under a wide range of circumstances the preference relation $\succsim_i$ of player i in a strategic game can be represented by a **payoff function** $u_i : A \to \mathbb{R}$ (also called a utility function), in the sense that $u_i(a) \geq u_i(b)$ whenever $a \succsim_i b$. We refer to values of such a function as payoffs (or utilities). Frequently we specify a player's preference relation by giving a payoff function that represents it. In such a case we denote the game by $\langle N, (A_i), (u_i) \rangle$ rather than $\langle N, (A_i), (\succsim_i) \rangle$.*

Before we illustrate the concept of strategic games with an example, we would like to introduce a special class of strategic games - symmetric games. Such games have the property that the participating decision-makers are not affected by which player "role" (out of the player set N) they have been assigned to (for instance, employer-employee games, or incumbent-challenger games have roles); rather, each player has the same actions available to them, with the same consequences for a symmetric choice of actions (rock-paper-scissors is an example of a symmetric game).

For the moment, we only define symmetry for the simplest type of strategic games - 2-player games.

Definition 8 (Two-player Symmetric Game)**.** *Let $G = \langle \{1,2\}, (A_i), (\succsim_i) \rangle$ be a two-player strategic game; G is called* ***symmetric*** *if it satisfies the following:*

- $A_1 = A_2$, *and*
- $(a_1, a_2) \succsim_1 (b_1, b_2)$ *if and only if* $(a_2, a_1) \succsim_2 (b_2, b_1)$ *for all* $a \in A$ *and* $b \in A$.

For $H = \langle \{1,2\}, (A_i), (u_i) \rangle$, *the second criterion becomes* $u_1(a_1, a_2) = u_2(a_2, a_1)$ *for all* $a_1, a_2 \in A$.

We can now formalise the Prisoner's Dilemma (from Example 5) using the definitions that we have introduced.

Example 3 (Prisoner's Dilemma)**.** *Prisoner's Dilemma is a strategic game of the form* $\langle \{1,2\}, (A_i), (\succsim_i) \rangle$, *with:*

- $A_1 = A_2 = \{C, D\}$, *and*
- $(D, C) \succsim_1 (C, C) \succsim_1 (D, D) \succsim_1 (C, D)$
- $(C, D) \succsim_2 (C, C) \succsim_2 (D, D) \succsim_2 (D, C)$

It is trivial to see that it satisfies the symmetry property from Definition 8.

2.3 Extensive Games

2.3.1 Extensive Games with Perfect Information

An extensive game is a detailed description of the sequential structure of the decision problems encountered by the players in a strategic situation. There is perfect information in such a game if each player, when making any decision, is perfectly informed of all the events that have previously occurred. We initially restrict attention to games in which no two players make decisions at the same time and all relevant moves are made by players (no randomness ever intervenes) - the first restriction is removed later on.

Definition 9 (Extensive Game with Perfect Information). *An **extensive game with perfect information** has the following components.*

1. *A set N (the set of **players**)*
2. *A set H of sequences (finite or infinite) that satisfies the following three properties.*
   - *The empty sequence $\emptyset$ is a member of H.*
   - *If $(a^k)_{k=1,\ldots,K} \in H$ (where K may be infinite) and $L < K$ then $(a^k)_{k=1,\ldots,L} \in H$.*
   - *If an infinite sequence $(a^k)_{k=1}^{\infty}$ satisfies $(a^k)_{k=1,\ldots,L} \in H$ for every positive integer L then $(a^k)_{k=1}^{\infty} \in H$.*

   *(Each member of H is a **history**; each component of a history is an **action** taken by a player.) A history $(a^k)_{k=1,\ldots,K} \in H$ is **terminal** if it is infinite or if there is no a^{K+1} such that $(a^k)_{k=1,\ldots,K+1} \in H$. The set of terminal histories is denoted Z.*
3. *A function P that assigns to each nonterminal history (each member of $H \setminus Z$) a member of N. (P is the **player function**, $P(h)$ being the player who takes an action after the history h.)*
4. *A preference relation $\succsim_i$ on Z (the **preference relation** of player i) for each player $i \in N$.*

We interpret such a game as follows. After any nonterminal history h player $P(h)$ chooses an action from the set

$$A(h) = \{a : (h, a) \in H\}$$

(here, if h is a history of length k, (h, a) denotes the history of length $k + 1$ consisting of h followed by a). The empty history is the starting point of the game; it is referred to as the *initial history.* At this point player $P(\emptyset)$ chooses a member of $A(\emptyset)$. For each possible choice a^0 from this set player $P(a^0)$ subsequently chooses a member of the set $A(a^0)$; this choice determines the next player to move, and so on. A history after which no more choices have to be made is terminal. Note that a history may be an infinite sequence of actions.

Here is an example, illustrating the above definition.

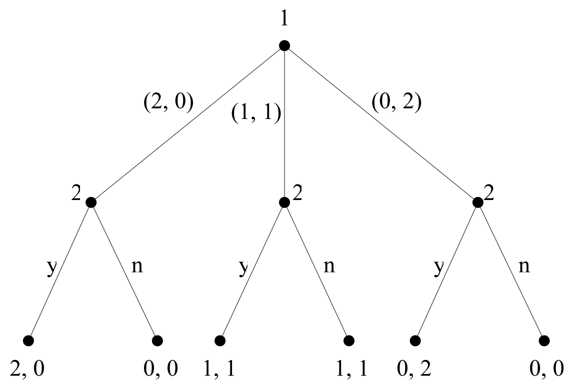


Figure 2.2: Extensive Game with Perfect Information Example

Example 4 (Extensive Game with Perfect Information)**.** *Two people use the following procedure to share two desirable identical indivisible objects. One of them proposes an allocation, which the other then either accepts or rejects. In the event of rejection, neither person receives either of the objects. Each person cares only about the number of objects he obtains.*

$\langle N, H, P, (\succsim_i)\rangle$ is an extensive game that models the individuals' predicament; here

1. *$N = \{1, 2\}$;*
2. *H consists of the ten histories: $\emptyset$, $(2,0)$, $(1,1)$, $(0,2)$, $((2,0),y)$, $((2,0),$ $n)$, $((1,1),y)$, $((1,1),n)$, $((0,2),y)$, $((0,2),n)$;*
3. *$P(\emptyset) = 1$ and $P(h) = 2$ for ever nonterminal history $h \neq \emptyset$*
4. *$((2,0),y) \succ_1 ((1,1),y) \succ_1 ((0,2),y) \sim_1 ((2,0),n) \sim_1 ((1,1),n) \sim_1 ((0,2),n)$ and $((0,2),y) \succ_2 ((1,1),y) \succ_2 ((2,0),y) \sim_2 ((0,2),n) \sim_2 ((1,1),n) \sim_2 ((2,0),n)$*

A convenient representation of this game is shown in Figure 2.2. The small circle at the top of the diagram represents the initial history $\emptyset$ (the starting point of the game). The 1 above this circle indicates that $P(\emptyset) = 1$ (player 1 makes the first move). The three line segments that emanate from the circle correspond to the three members of $A(\emptyset)$ (the possible actions of player 1 at the initial history); the labels beside these line segments are the names of the actions, $(k, 2-k)$ being the proposal to give k of the objects to player 1 and the remaining $2-k$ to player 2. Each line segment leads to a small disk beside which is the label 2, indicating that player 2 takes an action after any history of length one. The labels beside the line segments that emanate from these disks are the names of player 2's actions, y meaning "accept" and n meaning "reject". The numbers below the terminal histories are payoffs that represent the players' preferences (the first number in each pair is the payoff of player 1 and the second is the payoff of player 2). Figure 2.2 suggests an alternative definition of an extensive game in which the basic component is a tree (a connected graph with no cycles). In this formulation each node corresponds to a history and any pair of nodes that are connected corresponds to an action; the names of the actions are not part of the definition.

Player Strategy

A strategy of a player in an extensive game is a plan that specifies the action chosen by the player for every history after which it is his turn to move, even for histories that, if the strategy is followed, are never reached.

Definition 10 (Player Strategy in an Extensive Game with Perfect Information)**.** *A **strategy of player** $i \in N$ in an extensive game with perfect information $\langle N, H, P, (\succsim_i)\rangle$ is a function that assigns an action in $A(h)$ to each nonterminal history $h \in H \setminus Z$ for which $P(h) = i$.*

Example 5 (Player Strategy in an Extensive Game with Perfect Information)**.** *Consider the game from Example 4 (that is displayed in Figure 2.2). Player 1 takes an action only after the initial history $\emptyset$, so that we can identify each of her strategies with one of three possible actions that she can take after this history: $(2, 0)$, $(1, 1)$ and $(0, 2)$. Player 2 takes an action after each of the three histories $(2, 0)$, $(1, 1)$ and $(0, 2)$, and in each case has two possible actions. Thus we can identify each of his strategies as a triple $a_2b_2c_2$ where a_2, b_2 and c_2 are the actions that he chooses after the histories $(2, 0)$, $(1, 1)$ and $(0, 2)$. The interpretation of player 2's strategy $a_2b_2c_2$ is that it is a contingency plan: if player 1 chooses $(2, 0)$ then player 2 will choose a_2; if player 1 chooses $(1, 1)$ then player 2 will choose b_2; and if player 1 chooses $(0, 2)$ then player 2 will choose c_2.*

Simultaneous Moves

To model situations in which players move simultaneously after certain histories, each of them being fully informed of all past events when making his choice, we can modify the definition an extensive game with perfect information (Definition 9) as follows.

Definition 11 (Extensive Game with Perfect Information and Simultaneous Moves)**.** *An* ***extensive game with perfect information and simultaneous moves*** *is a tuple $\langle N, H, P, (\succsim_i)\rangle$ where N, H, $\succsim_i$ for each $i \in N$ are the same as in Definition 9, P is a function that assigns to each nonterminal history a set of players, and H and P jointly satisfy the condition that for every nonterminal history h there is a collection $\{A_i(h)\}_{i\in P(h)}$ of sets for which $A(h) = \{a : (h, a) \in H\} = \times_{i\in P(h)} A_i(h)$.*

A history in such a game is a sequence of vectors; the components of each vector a^k are the actions taken by the players whose turn it is to move after the history $(a^l)_{l=1}^{k-1}$. The set of actions among which each player $i \in P(h)$ can choose after the history h is $A_i(h)$; the interpretation is that the choices of the players in $P(h)$ are made simultaneously. A strategy of player $i \in N$ in such a game is a function that assigns an action in $A_i(h)$ to every nonterminal history h for which $i \in P(h)$.

2.3.2 Repeated Games with Perfect Information

The model of a repeated game captures the situation in which players repeatedly engage in a strategic game G, which we refer to as the constituent game. We restrict attention to games in which the action set of each player is compact and the preference relation of each player is continuous (a preference relation $\succsim$ on A is continuous if $a \succsim b$ whenever there are sequences $(a^k)_k$ and $(b^k)_k$ in A that converge to a and b respectively for which $a^k \succsim b^k$ for all k). On each occasion that G is played, the players choose their actions simultaneously. When taking an action, a player knows the actions previously chosen by all players. We model this situation as an extensive game with perfect information and simultaneous moves, as follows.

Definition 12 (Infinitely Repeated Game with Perfect Information)**.** *Let $G = \langle N, (A_i), (\succsim_i)\rangle$ be a strategic game; let $A = \times_{i\in N} A_i$. An* ***infinitely***

repeated game with perfect information *of G is an extensive game with perfect information and simultaneous moves $\langle N, H, P, (\succsim_i^*) \rangle$ in which*

1. *$H = \{\emptyset\} \cup (\cup_{t=1}^{\infty} A^t) \cup A^{\infty}$ (where $\emptyset$ is the initial history and A^{∞} is the set of infinite sequences $(a^t)_{t=1}^{\infty}$ of action profiles in G)*

2. *$P(h) = N$ for each nonterminal history $h \in H$*

3. *$\succsim_i^*$ is a preference relation on A^{∞} that extends the preference relation $\succsim_i$ in the sense that it satisfies the following condition of weak separability: if $(a^t) \in A^{\infty}$, $a \in A$, $a' \in A$ and $a \succsim_i a'$ then*

$$(a^1, ..., a^{t-1}, a, a^{t+1}, ...) \succsim_i^* (a^1, ..., a^{t-1}, a', a^{t+1}, ...)$$

for all values of t.

We now introduce the concept of a finitely repeated game. The formal description of a finitely repeated game is very similar to that of an infinitely repeated game.

Definition 13 (T-period Repeated Game with Perfect Information)**.** *For any positive integer T a **T-period finitely repeated game with perfect information** of the strategic game $\langle N, (A_i), (\succsim_i) \rangle$ is an extensive game with perfect information that satisfies the conditions in Definition 12 when the symbol ∞ is replaced by T. We restrict attention to the case in which the preference relation $\succsim_i^*$ of each player i in the finitely repeated game is represented by the function $\sum_{t=0}^{T} u_i(a^t)/T$, where u_i is a payoff function that represents i's preference in the constituent game.*

Definition 14 (Canonical Iterated Prisoner's Dilemma)**.** *The canonical version of the Iterated Prisoner's Dilemma (IPD) is a T-period repeated game with perfect information, with the Prisoner's Dilemma (Definition 3) as its constituent game.*

2.3.3 Extensive Games with Imperfect Information

In each of the models we have introduced previously, the players are not perfectly informed, in some way, when making their choices. In a strategic game a player, when taking an action, does not know the actions that the other players take. In an extensive game with perfect information, a player does not know the future moves planned by the other players. The model that we define here - an extensive game with imperfect information - differs in that the players may in addition be imperfectly informed about some (or all) of the choices that have already been made.

The following definition generalises that of an extensive game with perfect information (Definition 9) to allow players to be imperfectly informed about past events when taking actions. It does not incorporate the generalisation in which more than one player may move after any history (Definition 11), nor does it allow for exogenous uncertainty: moves may not be made by "chance". The latter is not incorporated in our definition not as a result of any incompatibilities in the concepts, but purely for simplicity's sake; we will not require it for the examples and analyses that come later.

Definition 15 (Extensive Game)**.** *An* ***extensive game*** *is a tuple* $\langle N, H, P, (\mathbb{I}_i), (\succsim_i)\rangle$ *where* N, H, $\succsim_i$ *for each* $i \in N$ *are the same as in Definition 9, and a partition* $\mathbb{I}_i$ *of* $\{h \in H : P(h) = i\}$ *for each player* $i \in N$ *with the property that* $A(h) = A(h')$ *whenever* h *and* h' *are in the same member of the partition. For* $I_i \in \mathbb{I}_i$ *we denote by* $A(I_i)$ *the set* $A(h)$ *and by* $P(I_i)$ *the player* $P(h)$ *for any* $h \in I_i$*. (*$\mathbb{I}_i$ *is the* ***information partition*** *of player i; a set* $I_i \in \mathbb{I}_i$ *is an* ***information set*** *of player i.)*

We interpret the histories in any given member of $\mathbb{I}_i$ to be indistinguishable to player i. Thus the game models a situation in which after any history $h \in I_i \in \mathbb{I}_i$ player i is informed that some history in I_i has occured but is not infomed that the history h has occured. The condition $A(h) = A(h')$

whenever h and h' are in the same member of $\mathbb{I}_i$ captures the idea that if $A(h) \neq A(h')$ then player i could deduce, when he faced $A(h)$, that the history was not h', contrary to our interpretation of $\mathbb{I}_i$.

If $\langle N, H, P, (\mathbb{I}_i)_{i\in N}, (\succsim_i)_{i\in N}\rangle$ is an extensive game (as in Definition 15) and every member of the information partition of every player is a singleton, then $\langle N, H, P, (\succsim_i)_{i\in N}\rangle$ is an extensive game with perfect information (as in Definition 9).

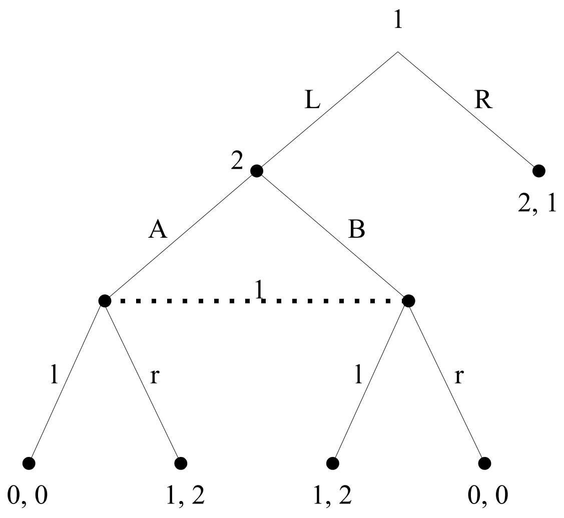


Figure 2.3: Extensive Game Example

Example 6 (Extensive Game)**.** *An example of an extensive game with imperfect information is shown in Figure 2.3. In this game player 1 makes the first move, choosing between L and R. If she chooses R, the game ends. If she chooses L, it is player 2's turn to move; he is informed that player 2 chose L and chooses A or B. In either case it is player 1's turn to move, and when doing so she is not informed whether player 2 chose A or B, a fact indicated in the figure by the dotted line connecting the ends of the histories after which player 1 has to move for the second time, choosing an action from* *the set $\{l, r\}$. Formally, we have $P(\emptyset) = P(L, A) = P(L, B) = 1$, $P(L) = 2$, $\mathbb{I}_1 = \{\emptyset, \{(L, A), (L, B)\}\}$, and $\mathbb{I}_2 = \{\{L\}\}$ (player 1 has two information sets and player 2 has one). The numbers under the terminal histories are players' payoffs. (The first number in each pair is the payoff of player 1 and the second is the payoff of player 2.)*

In Definition 15, we do not allow more than one player to move after any history. However, there is a sense in which an extensive game as we have defined it can model such a situation. To see this, consider Example 6 above. After player 1 chooses L, the situation in which players 1 and 2 are involved is essentially the same as that captured by a game with perfect information in which they choose actions simultaneously. (This is the reason that in much of the literature the definition of an extensive game with perfect information does not include the possibility of simultaneous moves.) With this in mind, we can see that we need only trivial (if technically messy) adjustments, in order to represent the games from Definitions 12 and 13 as extensive games with imperfect information.

Before we introduce a specific type of repeated game with imperfect information (Definition 17), we extend the concept of symmetry that was first introduced in the 2-player symmetric games definition (Definition 8).

Definition 16 (m-player Symmetric Strategic Game)**.** *An m-player strategic game $\langle N,\ (A_i), (u_i)\rangle$ (where $|N| = m$) is symmetric if the following conditions hold:*

1. *Every player has the same action space: $A = A_1 = A_2 = ... = A_m$.*
2. *Every player has a symmetric payoff function in the following sense: pick two action profiles $a, a' \in A$ and a pair of players $i, j \in N$ arbitrarily. If $a_i = a'_j$ and a_{-i} can be obtained from a'_{-j} by a permutation of actions, then $u_i(a) = u_j(a')$.*

We illustrate the definition with an example.

Example 7 (m-player Symmetric Strategic Game)**.** *A 3-player Rock-Paper-Scissors game* $\langle\{1,2,3\},\{R,P,S\},(u_i)\rangle$ *has the following rules:*

- *a starting pot of winnings is split between 3 players at the end of each game*
- *if players pick one of each strategy, or everyone picks the same strategy, then the pot is shared equally*
- *if one player's strategy beats the others' strategies, she wins the pot*
- *if two players' strategies are the same and beat the third's, then they share the pot*

This game is symmetric if the payoff profiles are as in Figure 2.4 (tuples correspond to payoffs for players (I,II,III)).

I = R		III		
		R	P	S
II	R	(2,2,2)	(0,0,6)	(3,3,0)
	P	(0,6,0)	(0,3,3)	(2,2,2)
	S	(3,0,3)	(2,2,2)	(6,0,0)

I = P		III		
		R	P	S
II	R	(6,0,0)	(3,0,3)	(2,2,2)
	P	(3,3,0)	(2,2,2)	(0,0,6)
	S	(2,2,2)	(0,6,0)	(0,3,3)

I = S		III		
		R	P	S
II	R	(0,3,3)	(2,2,2)	(0,6,0)
	P	(2,2,2)	(6,0,0)	(3,0,3)
	S	(0,0,6)	(3,3,0)	(2,2,2)

Figure 2.4: Rock-Paper-Scissors Payoffs

Definition 17 (m-Player Symmetric T-period Repeated Game)**.** *Let the constituent game* $G = \langle N,(A_i),(\succsim_i)\rangle$ *be an m-player symmetric strategic game;*

let $A = \times_{i \in N} A_i$. A ***T-period repeated game*** *of G is an extensive game $\langle N, H, P, (\mathbb{I}_i), (\succsim_i)\rangle$ in which*

1. *$H = \{\emptyset\} \cup (\cup_{t=1}^{T} A^t)$ (where $\emptyset$ is the initial history and A^T is the set of T-length sequences $(a^t)_{t=1}^{T}$ of action profiles in G)*

2. *$P(h) = N$ for each nonterminal history $h \in H$*

3. *the preference relation $\succsim_i^*$ of each player i is represented by the function $\sum_{t=0}^{T} u_i(a^t)/T$, where u_i is a payoff function that represents i's preference in the constituent game.*

4. *$\mathbb{I}_i = \mathbb{I}_j$ for all $i, j \in N$*

A player's strategy in an extensive game with perfect information is a function that specifies an action for every history after which the player chooses an action (Definition 10). The following definition is an extension to a general extensive game.

Definition 18 (Player Strategy in an Extensive Game)**.** *A* ***strategy of player*** *$i \in N$ in an extensive game $\langle N, H, P, (\mathbb{I}_i), (\succsim_i)\rangle$ is a function that assigns an action in $A(I_i)$ to each information set $I_i \in \mathbb{I}_i$.*

Chapter 3

The Genetic Game Algorithm

In Chapter 1, we discussed the concept of the genetic algorithm in broad terms. In this chapter, the goal is to find a meaningful way to apply the genetic algorithm to problems in game theory. To that end, we shall take various definitions that we introduced in Chapter 2 and combine them with the ideas from Chapter 1; our results will be several formal definitions (covering the different flavours of games) that we will call the Genetic Game Algorithm[1] (GGA).

After an discussion of the motivations behind the GGA and how it differs from the GA (Section 3.1), we shall define two versions of the GGA - one for symmetric strategic games (Section 3.2.1), and one for 2-player symmetric repeated games (Section 3.2.2). Constraints and limitations of the given algorithms are discussed in Section 3.3, and more general versions of the GGA can be found in Appendix C.

To the author's best knowledge, almost all of the material in this chapter (at least in its present form), is original.

[1]This name should be interpreted as "Genetic Algorithm for Games", rather than an "Algorithm for Genetic Games".

3.1 Overview

3.1.1 Motivations

The main motivation for the GGA stems from the desire to have an algorithmic method for finding the best strategy in a given action/strategy space, just as the GA is an algorithmic method for finding the best individual in a given population space. Another goal of the GGA is to formalise some of the prior economic and game theoretic experimental work on strategy evolution. For me personally, there were numerous times when the understanding of an interesting experiment was hindered by the imprecise language used in its exposition. By providing some precise but flexible definitions (built on the very strict game theoretic definitions and results that has been developed over the last 15 years), an attempt is made to alleviate these issues. Another benefit of formalisation is that for any experiments that utilise the GGA, the simulation implementation time is reduced (this is because a mathematically stipulated model is the most precise specification that a software implementer can hope for, meaning the written software can be written quicker and with fewer bugs).

Beside strategy evolution, the GGA can be applied to other situations involving discrete population, discrete-time dynamics, such as experiments investigating population convergence or equilibrium points.

3.1.2 The GGA and the GA

In Chapter 1, we formalised only those parts of the GA that were domain and problem independent, and even then, not all of them (for instance, we introduced the concept of a terminating condition, which is but we did not formally define it). As we are focusing on a specific domain of games - which comes with its own formal language and structure - we can now precise in our definitions.

The GGA is different to the GA in that the evaluation process (from the GA) is broken down into fitness, encoding and decoding in the GGA, which are collectively called the evaluation functions; the GGA flowchart (Figure 3.1) reflects this decomposition. The most important one of all, the fitness function, is (the only place in the GGA) where games are played. While we can now specify the domains of the evaluation functions, they are in fact problem-specific - we shall see several examples of these when we analyse problems in the next two chapters.

3.2 Two GGA Definitions

Two versions of the GGA are presented in this chapter: Definition 19 (for m-player symmetric strategic games) and Definition 20 (for a 2-player symmetric T-period repeated games). More general versions of the GGA can be found in Appendix C - since they are not necessary for use in later chapters, and are not different enough conceptually to warrant extra attention, we do not present them here.

3.2.1 The GGA for Symmetric Strategic Games

Before we introduce the simplest version of the GGA, we need to define some notation:

Notation. *Wherever a function of the form $T_z : X \to Y$ has been defined, $T_{z,k}$ will always be understood to be*

$$T_{z,k} : X^k \to Y^k,\ (x_1, x_2, ..., x_k) \mapsto (T_z(x_1), T_z(x_2), ..., T_z(x_k)) \tag{3.1}$$

for $x_1, ..., x_k \in X$.

Notation. $\mathbb{R}_0^+$ *refers to the set* $\{x \in \mathbb{R} | x \geq 0\}$

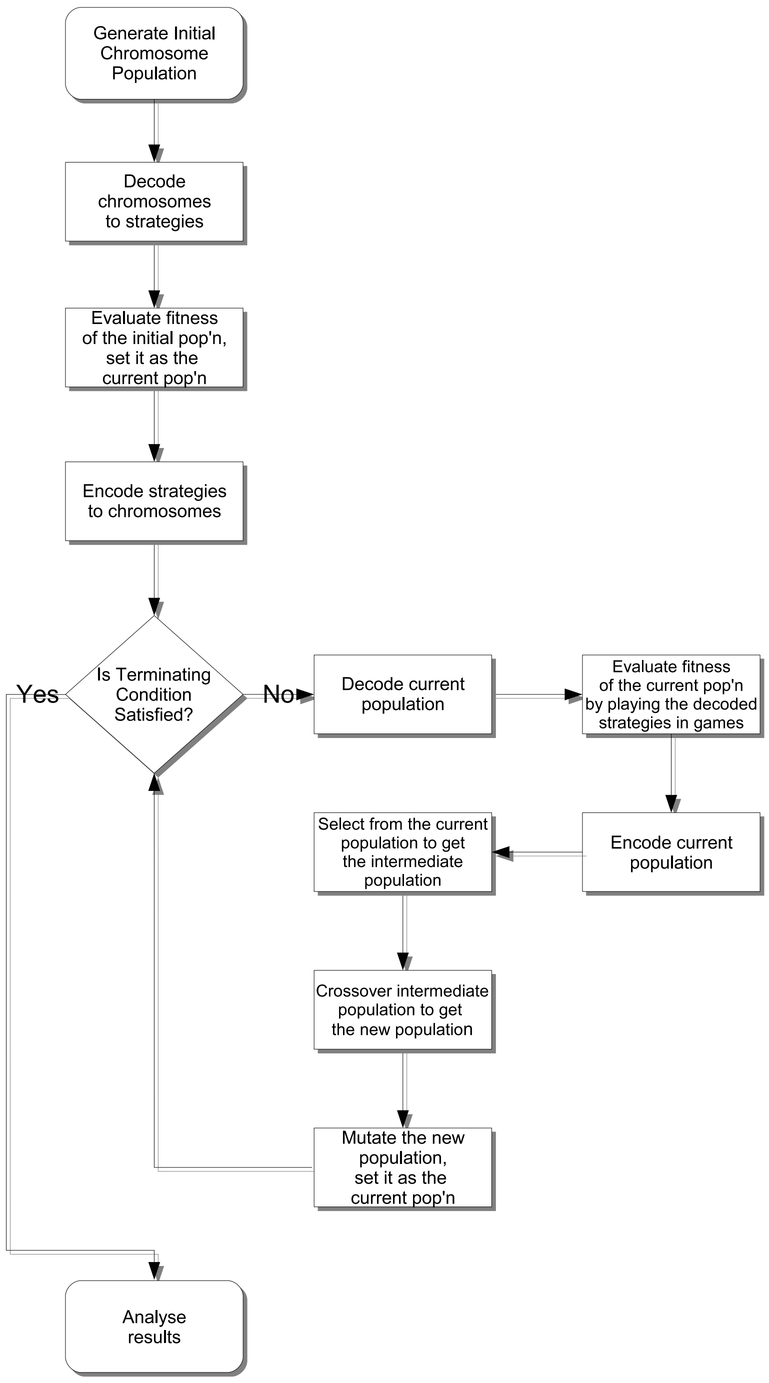

Generate Initial Chromosome Population
Decode chromosomes to strategies
Evaluate fitness of the initial pop'n, set it as the current pop'n
Encode strategies to chromosomes
Is Terminating Condition Satisfied?
Yes
No
Decode current population
Evaluate fitness of the current pop'n by playing the decoded strategies in games
Encode current population
Select from the current population to get the intermediate population
Crossover intermediate population to get the new population
Mutate the new population, set it as the current pop'n
Analyse results

Figure 3.1: The GGA Flowchart

Definition 19 (The Genetic Game Algorithm for an m-player Symmetric Strategic Game)**.** *The GGA for an m-player symmetric game $G = \langle N, (A_i), (u_i) \rangle$ consists of:*

1. *a set $D \subseteq A (= A_1 = A_2)$, the* ***action subset****, containing 2^k elements (for some $k \in \mathbb{N}$),*

2. *the evaluation functions:*

    - *$T_e : D \to B$ (an invertible encode function),*
    - *$T_d : B \to D$ (the inverse of encode, the decode function),*
    - *$T_f : D^n \to (\mathbb{R}_0^+)^n$ (fitness)*

    where $B = \{0, 1\}^k$ (k is as in point 1),

3. *the genetic functions:*

    - *$(T_s^i)_{i=1}^{n/2} : B^n \times (\mathbb{R}_0^+)^n \to B^2$ (selection) - its inputs are the current population and the population's fitness,*
    - *$(T_c^i)_{i=1}^{n/2} : B^2 \to B^2$ (crossover) - its inputs are two parent chromosomes,*
    - *$T_m : B \to B$ (mutation) - its input is the chromosome undergoing mutation*

4. *the terminating condition function $T_t : B^n \times \mathbb{N} \to \{true, false\}$ - its input is a population and its generation number,*

5. *an n-tuple of actions, $\vec{Y} \in D^n$ (with n a multiple of 2), called the initial population,*

Then the ***population sequence*** $(\vec{X}(p))_{p\in\{0,1,2,...,c\}}$, $\vec{X}(p) \in B^n$ *is obtained using the following:*

$$\vec{X}(p) = \begin{cases} T_{e,n}(\vec{Y}) & \text{for } p = 0 \\ (p_1, p_2, ..., p_n) & \text{for } 1 \le p \le c \end{cases} \tag{3.2}$$

where $\forall i = 1, ..., \frac{n}{2}$,

$$(p_{2i-1}, p_{2i}) = T_{m,2}(T_c^i(T_s^i(\vec{X}(p-1), T_f(T_{d,n}(\vec{X}(p-1)))))) \tag{3.3}$$

In the above expressions, the ***terminating generation*** $c \in \mathbb{N}$ *is a number that satisfies the following conditions:*

$$0 \le j < c \Rightarrow T_t(\vec{X}(j), j) = false \text{ , and} \tag{3.4}$$

$$T_t(\vec{X}(c), c) = true. \tag{3.5}$$

Remark 1. *For a game* $G = \langle N, A, (u_i)\rangle$ *and action subset* $D \subseteq A$ *(G and D as in Definition 19), the action subset induces a game* $G' = \langle N,\ D,\ (u_i)\rangle$*. This means that* G'*, not* G*, is the game in which the GGA is searching for the best action. Thus, it is imperative that* D *approximates* A *as closely as possible, otherwise the optimum solution from* D *may not be a close enough approximation of the optimum solution from* A*.*

The GGA from Definition 19 is used in the analysis of the Contract Game problem from Chapter 5.

3.2.2 The GGA for Symmetric Repeated Games

The major difference between the GGA for extensive games and the GGA for strategic games, in that the ”individuals” encoded in the chromosomes are strategies, rather than actions.

As before, only one (narrow) extensive game GGA version is given here; this is done so that we can maintain our focus and move forward to our examples. Again, more general versions are provided in the Appendix.

Definition 20 (The Genetic Game Algorithm for an m-player Symmetric T-period Repeated Game)**.** *The GGA for an T-period repeated game* $G = \left\langle N, H, \; P, (\mathbb{I}_i), (\succsim_i) \right\rangle$ *(see Definition 17) with an m-player symmetric constituent game* $\langle N, (A_i), (u_i) \rangle$*, is defined in exactly the same way as in Definition 19, except that instead of instead of* D*, we have* W*, subset of player strategies for the game* G*, (strategies are as defined in Definition 18), with* W *containing* 2^k *elements (for some* $k \in \mathbb{N}$*).*

Remark 2. *The argument here is similar to the one made in Remark 1: the GGA is trying to find the "best" strategy from the strategy subset* W*, not from the set of all strategies for the game; thus, for the GGA search to be useful, we need to pick* W *so that the strategies within have enough complexity to be useful in the problem being solved.*

The GGA from Definition 20 is used in the analysis of the Axelrod experiment from Chapter 4.

3.3 Constraints and Limitations

There are certain limitations regarding which games can be fitted to the GGA:

1. *The cardinality of the action/strategy subset must be power of 2.*
   If the cardinality is not a power of 2, then there will exist chromosomes which do not correspond to any action/strategy - this would mean that the mutation and crossover operators are not closed.

2. *The number of individuals in each of the populations (in the population sequence) must be a multiple of 2.*
   This restriction is in place only for the sake of notation simplicity in the crossover operator, which tends to be symmetric. In practice, large populations are usually used and this restriction becomes a non-issue.

3. *The GGA cannot model dynamics that feature non-integer populations, such as replicator dynamics.*
   This incompatibility is not of crucial importantance, as continuous population dynamics have already been the focus of much in-depth research, yielding results that eclipse anything presented here regarding discrete population models ([Weibull, 1995] is a great resource on this topic).

Chapter 4

Axelrod's Evolutionary Experiment

In 1979, Robert Axelrod (University of Michigan) hosted a tournament to see what kinds of strategies would perform best over the long haul in the Iterated Prisoner's Dilemma (IPD) game. The fourteen entries (plus the "random" strategy entered by Axelrod himself) - all computerized IPD strategies - were submitted not just by game theorists, but also by economists, biologists, computer scientists and psychologists. The tournament pitted the entries against each other in a round-robin format (that is, each contestant is matched in turn against every other contestant), with 200 rounds of PD played during each "match", and was run five times to smooth out random effects. Tit-For-Tat (TFT), the winning strategy (that is, the one that averaged the highest score overall), entered by Anatol Rapoport (a mathematical psychologist), was the simplest of all submitted strategies, with just two rules:

1. in the first round, cooperate
2. in each subsequent round, play the opponent's action from the previous round

Axelrod staged a secound tournament, and had sixty-two entry submissions from 6 countries (plus the "random" strategy, as before). The rules were only slightly modified from the first tournament: games were now of a random length with median 200, rather than exactly 200 rounds; this avoided the complications from programs having special cheating rules for the last game. Surprisingly, given that every submitter had full information about the structure and results of the first tournament, Tit-For-Tat once again emerged as the winner.

After his tournaments, Axelrod went on to stage several "evolutionary" tournaments (or rather experiments, since these did not involve submitted strategies). These experiments modelled the players in the IPD game as stimulus-response automata - the stimulus was the state of the game, defined as both players' actions over the previous several moves, and the response was the next period's action (or actions) - and investigated the question of what the best-performing IPD automaton strategy is. The focus of this chapter will be on the specific experiment presented in [Axelrod, 1987] (and revisited in [Marks, 1989]) - we shall describe the experimental setup as a GAA.

This is done with two aims in mind: to illustrate the GGA and at the same time, to try to improve on one of the weaker aspects of Axelrod's unquestionably important and influential work - its mathematically loose style of exposition. This perceived weakness should not be interpreted as a challenge to the rigour or the correctness of the experiment itself. For me personally, as I studied the aforementioned papers on this experiment, the informal approach at times hindered my understanding of the material; consequently, this chapter is designed to serve as a companion to Axelrod's and Marks' research, by clarifying some of the murkier points and "colouring in" the sketches that they lay out.

We shall start with an overview in Section 4.1, which is roughly broken down into the the following parts, mirroring Definition 20: the game, the strategy subset available to players, the evaluation functions, the genetic functions, and the initial population. Then we proceed to formally describe the experiment in detail in Sections 4.2, 4.3, 4.4, 4.5 and 4.6, using the definitions from the previous chapters. In Section 4.7, we discuss the simulations that we built to test our definitions, and the experiments that we ran on them.

4.1 Overview

We are going to describe Axelrod's experimental setup as a GGA, specifically the version from Definition 20.

4.1.1 The Game

The game that the experiment revolves around is the Iterated Prisoner's Dilemma (which is a T-period repeated game with the Prisoner's Dilemma as the constituent game). What needs to be decided is whether we should model the situation with the canonical, perfect information version (Definition 14), or its imperfect information equivalent.

Unlike the tournament that we analyse in this chapter (which involves stimulus-response automata), Axelrod's first tournament pitted programmed strategy subroutines against each other; although the extensive game being played was not explicitly defined in Axelrod's paper, one fact about his subroutines - that they were allowed to have persistent local variables - helped determine what that definition should be.

In simple terms, persistent local variables allow the subroutines to "remember" between the rounds of the repeated game; hence, a strategy subroutine can choose to remember every move that it and its opponent made. The implication stemming from this fact, is that the most appropriate game for our model is the repeated game with perfect information (that is, there is an information set for each history in the game).

4.1.2 The Strategy Subset

The strategy subset, being explored in this experiment, contains all strategies which have:

- an action for every possible combination of moves over the previous 3 rounds of the game; each player makes one of two moves - cooperate or defect - at each of the 3 rounds, which brings it to $2^6 = 64$ possible combinations, and hence, 64 actions. The strategy's current action is determined solely by what happened in the previous 3 rounds.

- a "fake" history (or "false memory"), which is used by the strategy only in the first 3 rounds, when there isn't enough real history to determine an action.

4.1.3 The Evaluation Functions

The binary representation of the strategy described above is quite straightforward; since the Prisoner's Dilemma is a symmetric game with only two possible actions, we can simply represent "cooperate" as 0, and "defect" as 1. Overall, each strategy is represented in the chromosome space by a 70 bit chromosome. The first 6 bits store the fake history, and the remaining bits store instructions regarding which action to take for each of the 64 possible histories over the previous 3 rounds.

$$\underbrace{\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{6}}_{\textit{phantom history}}\underbrace{\boxed{7}\boxed{8}\boxed{9}\ldots\boxed{68}\boxed{69}\boxed{70}}_{\textit{behaviour rules}}$$

Strategy	In Axelrod	Name	Author(s)
v_1	K60R	TFT with Check for Random	J.Graaskamp & K.Katzen
v_2	K91R	Revised State Transition	J.Pinckley
v_3	K40R	Discoverer	R.Adams
v_4	K67R	Tranquilizer	C.Feathers
v_5	K76R	Tester	D.Gladstein
v_6	K77R	Adjuster	S.Feld
v_7	K85R	Slow-Never	R.Falk & J.Langsted
v_8	K47R	Fink	R.Hufford

Table 4.1: The Predetermined Strategies Used to Measure Fitness

Fitness

In his 1984 report, Axelrod specified a set (let us call it T8) of eight strategies from his second tournament (that are listed in Table 4.1); the T8 strategies could be used as representatives of the complete set of 63 strategies entered in the tournament. Using the following equation:

$$\begin{aligned} f(w) = & \quad c_0 + & \sum_{i=1}^{8} c_k w_k \\ = & \quad 110.55 + & (0.1574)\, w_2 + (0.1506)\, w_1 + (0.1185)\, w_3 \\ + & \quad (0.0876)\, w_4 + & (0.0579)\, w_6 + (0.0492)\, w_7 \\ + & \quad (0.0487)\, w_5 + & (0.0463)\, w_8 \end{aligned}$$

where w_i is the score strategy w gets playing 151 rounds of the IPD against v_i, Axelrod reported that the estimates correlated with the actual tournament scores at a variance of 0.98 (so 98% of the variance in his tournament scores is explained by knowing a strategys performance against the T8). Consequently, Axelrod defines the evolutionary experiment's population fitness using the above equation (an example fitness calculation can be seen in Table 4.2).

Opponent	Outcome after 151 Rounds	w_k	$c_k w_k$
v_1	STRRPTS...R	420	63.252
v_2	RRRRRRR...R	453	71.3022
$\vdots$	$\vdots$	$\vdots$	$\vdots$
v_8	$\dots$	$\dots$	$\dots$
		$\sum_{i=1}^{8} c_k w_k$	271.05
		$f(w)$: $c_0 + \sum_{i=1}^{8} c_k w_k$	381.60

Table 4.2: Sample Fitness Calculation for strategy w

(R, S, T, and P refer to the four possible outcomes of the Prisoner's Dilemma, as defined in Definition 5.)

This static[1] fitness measure was engineered with two assumptions in mind:

1. The 8 IPD strategies that it features provide an accurate approximation of the 63 strategies entered into Axelrod's second tournament

2. Those 63 strategies are representative of the entire population of IPD strategies, and performance against these 63 strategies provides an accurate approximation of performance against all IPD strategies.

If these assumptions did not hold, it could mean that the optimal strategy, found using the static fitness, would be suboptimal with respect to the entire set of all IPD strategies. In fact, [Nachbar, 1988] challenges the second assumption, arguing that the results from Axelrod's second tournament are tainted by the entrants' prior knowledge of the results of the first tournament, which may have been suboptimal.

[1] See discussion in Section 1.2.

Axelrod was aware of such doubts, because he introduces an alternative fitness function, one that it no longer relies on these assumptions. During fitness calculation for a strategy, the function pits that strategy against every other in the population (including itself) in an IPD game, and averages the outcome - it is a dynamic fitness function.

4.1.4 The Genetic Functions

Selection

The technique used here for selecting chromosomes (for crossover and mutation) is called "sigma scaling". First, the mean and the standard deviation (SD) of the fitness values is calculated before the selection; strategies with fitness less than one SD lower than the mean are discarded (that is, they play no part in forming the next generation), strategies with fitness over on SD higher than the mean are selected twice, and the remaining strategies are each selected only once.

Crossover

The standard one-point crossover technique (as in the CGA - see Definition 2) is used: a gene position on the parent chromosomes is (uniformly) randomly selected; then with the crossover probability p_c, the children are created from both parent chromosomes being sliced at that point and their tail segments switched, or with probability $1 - p_c$ the children chromosomes are simply clones of the parent chromosomes.

Mutation

The CGA mutation technique (see Definition 3) is used: each gene is perturbed in a probabilistic manner.

4.1.5 The Initial Population

The initial population is drawn randomly, so each chromosome is generated through 70 Bernoulli($\frac{1}{2}$) trials.

4.1.6 The Terminating Condition

The terminating condition is a trigger condition that interrupts the GGA at the 50th generation.

4.2 Detailed Analysis: The Game

As discussed in Section 4.1.1, the game being played is the Canonical IPD from Definition 14 (which is a T-period repeated game with perfect information, with the Prisoner's Dilemma as the constituent game), with $T = 151$. The payoffs in the PD constituent game are as in Example 2.

We give some important strategies for an IPD game $G = \left\langle N, H, P, (\succsim_i^*)\right\rangle$ with constituent PD game $\left\langle N,\ A, (\succsim_i)\right\rangle$:

Example 8 (AllC Strategy for the IPD)**.** *The AllC strategy always plays "cooperate":*

$$AllC(h) = C\ \forall h \in H \setminus Z \tag{4.1}$$

Example 9 (AllD Strategy for the IPD)**.** *The AllD strategy always plays "defect":*

$$AllD(h) = D\ \forall h \in H \setminus Z \tag{4.2}$$

Example 10 (Grim Strategy for the IPD)**.** *The Grim strategy, (which chooses C, both initially and for as long as both players have chosen C in every period* *in the past; otherwise, it chooses D) is defined as:*

$$Grim(h) = \begin{cases} D & \textit{if } h = (a_1, ..., a_n) \neq \emptyset \textit{ and } \exists k \in \{1, ..., n\} \textit{ s.t. } a_k \neq (C, C) \\ C & \textit{otherwise} \end{cases} \tag{4.3}$$

for $h \in H \setminus Z$ and $a_1, ..., a_n \in A$.

Example 11 (TFT Strategy for the IPD)**.** *The Tit-For-Tat strategy, described at the start of this chapter, is defined as:*

$$TFT(h) = \begin{cases} C & \textit{if } h = \emptyset \\ a_{-i} & \textit{if } h = (h', a) \in H \setminus (Z \cup \{\emptyset\}) \textit{ for some } h' \in H, a \in A \end{cases} \tag{4.4}$$

where $i \in N$ is the player playing the TFT strategy.

4.3 The Strategy Subset & Evaluation Functions: A First Look

Our first method of characterising the strategies involves representing them as look-up tables - a response action is provided for each of the 64 possible outcomes of the previous 3 rounds.

4.3.1 The Strategy Subset

The strategy subset W is the set of strategies of the form $\langle(\alpha, \beta, \gamma), m\rangle$. (α, β, γ) is the false memory of the strategy, that gets used by the strategy when the game has not been running long enough for 3 rounds' worth of history to have accumulated yet. α, β, γ are all action profiles and each represents a round of the game that the strategy thinks has happened, with α being the oldest memory (3 rounds ago) and γ being the newest memory (last round). The function $m : \mathbb{I} \to A$ maps each of 64 history equivalence classes in the history partition $\mathbb{I}$ to an action - m is the look-up table part of the strategy. The history partition $\mathbb{I}$ is defined through the following relation: $h_1 \sim h_2$ if $x(h_1) = x(h_2)$, where

$$x(h) := \begin{cases} (a,b,c) & \text{if } h = (h', a, b, c) \text{ for some } h' \in H, \text{ a,b,c} \in A \\ (\gamma, b, c) & \text{if } h = (b, c) \\ (\beta, \gamma, c) & \text{if } h = (c) \\ (\alpha, \beta, \gamma) & \text{if } h = \emptyset \end{cases} \tag{4.5}$$

Let $n : H \to \mathbb{I}$ be defined by $h \mapsto I$ if $h \in I$. Then a strategy $w \in W$ is defined by:

$$h \mapsto m(n(h)) \tag{4.6}$$

4.3.2 Encode and Decode

The encode and decode functions map between an Axelrod strategy $\langle(\alpha, \beta, \gamma), m\rangle$ and its binary representation $(b_1, ..., b_{70})$, with $b_i \in \{0, 1\}$. We shall try to formulate the decode function: $T_d : B \to W$, with $(b_1, ..., b_{70}) \mapsto w$.

We can break m down further into two functions: $m = t \circ s$. $s : \mathbb{I} \to \{1, ..., 64\}$ enumerates the history partitions, and is defined by Table B.1 (in the table, if α for strategy w is designated DC, that implies that $\alpha_w = D$ and $\alpha_{-w} = C$). $t : \{1, ..., 64\} \to A$ is defined by:

$$k \mapsto \begin{cases} C & \text{if } b_{k+6} = 0 \\ D & \text{if } b_{k+6} = 1 \end{cases} \tag{4.7}$$

To complete the definition of the decode function, we need to specify α,

β and γ. For strategy w:

$$\alpha_w := \begin{cases} C & \text{if } b_1 = 0 \\ D & \text{if } b_1 = 1 \end{cases} \tag{4.8}$$

$$\alpha_{-w} := \begin{cases} C & \text{if } b_2 = 0 \\ D & \text{if } b_2 = 1 \end{cases} \tag{4.9}$$

$$\beta_w := \begin{cases} C & \text{if } b_3 = 0 \\ D & \text{if } b_3 = 1 \end{cases} \tag{4.10}$$

$$\vdots \tag{4.11}$$

$$\gamma_{-w} := \begin{cases} C & \text{if } b_6 = 0 \\ D & \text{if } b_6 = 1 \end{cases} \tag{4.12}$$

This kind of explicit definition - enumerating all the histories using a table and linking that number to gene position - is fine in the 3 round memory case; however, were we to increase the number of rounds that strategies look into the past, the table would grow and such an approach would become cumbersome. We need a more general way in which to define our decode and encode functions, and we develop one below.

4.4 Machines And Agents

We first discuss special types of strategies: machines and agents. We then adapt these concepts to specify strategies that are equivalent (that is, behave in the same way under the same input) to those in Section 4.3.1, provided that both map (encode and decode) to the same chromosome.

Definition 21 (Machine)**.** *For an infinitely (or T-period) repeated game of $G = \langle N, (A_i), (\succsim_i)\rangle$, we define a **machine** of player i to be a four-tuple $\langle Q_i, q_i^0, g_i, \tau_i\rangle$ in which*

1. *Q_i is a set of* ***states***

2. *$q_i^0 \in Q_i$ is the* ***initial state***

3. *$g_i : Q_i \rightarrow A_i$ is the* ***output function****, that assigns an action to each state*

4. *$\tau_i : Q_i \times A \rightarrow Q_i$ is the* ***transition function****, that assigns a state to every pair consisting of a state and an action profile*

The set Q_i is unrestricted. In the first period, the state of the machine is q_i^0 and the machine chooses the action $g(q_i^0)$. Whenever the machine is in some state q_i, it chooses the action $g_i(q_i)$ corresponding to that state. The transition function τ_i specifies how the machine moves from one state to another: if the machine is in state q_i and a is the action profile chosen then its state changes to $\tau_i(q_i, a)$.

We shall now give some example machines for the canonical IPD game.

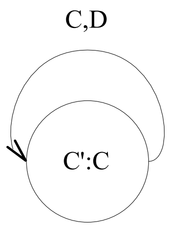


Figure 4.1: The AllC Strategy Machine in the IPD

Example 12 (Machine)**.** *The simplest machine $\langle Q_i, q_i^0, g_i, \tau_i \rangle$ that carries out the AllC strategy (Example 8), is defined as follows:*

1. *$Q_i = \{C'\}$*

2. *$q_i^0 = C'$*

3. *$g_i = C$*

4. *$\tau_i = C'$*

This machine is illustrated in Figure 4.1.

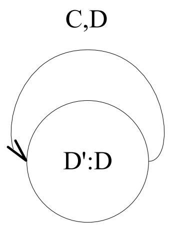


Figure 4.2: The AllD Strategy Machine in the IPD

Example 13 (Machine)**.** *The simplest machine $\langle Q_i, q_i^0, g_i, \tau_i \rangle$ that carries out the AllD strategy (Example 9), is defined as follows:*

1. *$Q_i = \{D'\}$*
2. *$q_i^0 = D'$*
3. *$g_i = D$*
4. *$\tau_i = D'$*

This machine is illustrated in Figure 4.2.

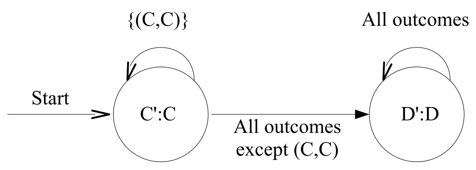


Figure 4.3: The Grim Strategy Machine in the IPD

Example 14 (Machine)**.** *The simplest machine $\langle Q_i, q_i^0, g_i, \tau_i \rangle$ that carries out the grim strategy (Example 10), is defined as follows:*

1. $Q_i = \{C', D'\}$
2. $q_i^0 = C'$
3. $g_i(C') = C$ *and* $f_i(D') = D$
4. $\tau_i(C', (C, C)) = C'$ *and* $\tau_i(X, (Y, Z)) = D'$ *if* $(X, (Y, Z)) \neq (C', (C, C))$

This machine is illustrated in Figure 4.3.

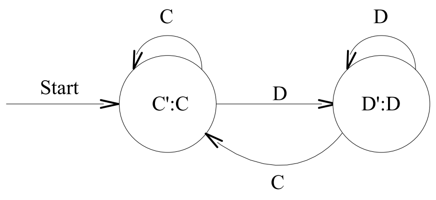


Figure 4.4: The Tit-For-Tat Strategy Machine in the IPD

Example 15 (Machine). *The simplest machine* $\langle Q_i, q_i^0, g_i, \tau_i \rangle$ *that carries out the Tit-For-Tat strategy (Example 11), is defined as follows:*

1. $Q_i = \{C', D'\}$
2. $q_i^0 = C'$
3. $g_i(C') = C$ *and* $f_i(D') = D$
4. $\tau_i(X, (Y, Z)) = Z$ *if* Y *is this machine's last move and* Z *is the opponent's last move*

This machine is illustrated in Figure 4.4.

In Definition 21, the input of the transition function consists of the current state and the profile of *all* the players' current actions. It is more natural to take as the input the current state and the list of actions chosen by the *other* players. This fits the natural description of a "strategy" as a plan of how to behave in all possible circumstances that are consistent with one's plans. However, since the game theoretic definition (Definition 18) requires that a strategy specify an action for all possible histories, *including* those that are inconsistent with the player's own strategy, we have to include as an input into the transition function the action of the player himself.

We now introduce a type of machine that better captures the consistency described above.

Definition 22 (Agent). *For a two-player infinitely (or T-period) repeated game of $G = \langle \{1, 2\}, (A_i), (u_i) \rangle$, we define an **agent** of player i to be a four-tuple $\langle Q_i, q_i^0, g_i, \tau_i \rangle$ in which Q_i, $q_i^0 \in Q_i$ and $g_i : Q_i \to A_i$ are as in Definition 21, only here τ_i (the **transition function**) is defined as $\tau_i : Q_i \times A_j \to Q_i$ (where $i \neq j$).*

But how does an agent link in with the concept of strategy in a repeated game?

Definition 23 (Agent as a Strategy). *For an infinite (or T-period) repeated game $\langle N, H, P, (\mathbb{I}_i), (\succsim_i) \rangle$ and provided that at any round of the game, player $i \in N$ can deduce the outcome of the previous round using its information set, an agent $\langle Q_i, q_i^0, g_i, \tau_i \rangle$ (representing player i) specifies the player's strategy $w : \mathbb{I}_i \to A_i$ as:*

$$I_i \in \mathbb{I}_i \mapsto g_i(q) \tag{4.13}$$

$q \in Q_i$ here is the current state of the machine; $q := q_i^0$ at before the first round, and q is updated to $\tau_i(q, a_j)$ ($i \neq j$) after the strategy has made its move, with a_j being deduced from the information set I_i.

4.5 The Strategy Subset & Evaluation Functions Using Agents

Since we are playing a game with perfect information, the outcome of the previous round can be deduced from the history; thus, an agent as a strategy (Definition 23) makes sense in this context, and we can apply the theory from the previous subsection.

4.5.1 The Strategy Subset

Now, our strategy subset is the set of all agents $\langle Q, q^0, g, \tau \rangle$ that share the specific Q and τ outlined below; q^0 and g - the parts of the agent that are encoded in the chromosomes - are discussed in Section 4.5.2. These agents determine player strategies in the way that was discussed at the end of the previous section.

States

Each Axelrod agent has 64 states

$$Q = \{q_1, q_2, ..., q_{63}, q_{64}\} \tag{4.14}$$

Now, we introduce two functions that will help us relate the agent's state with actions in the game:

- $own(k)$: action taken by this agent, k rounds ago.
- $other(k)$: action taken by this agent's opponent, k rounds ago.

We designate $own'(k)$ and $other'(k)$ to be the binary equivalents of $own(k)$ and $other(k)$, that is:

$$own'(k) := \begin{cases} 0 & \text{if } own(k) = C \\ 1 & \text{if } own(k) = D \end{cases} \tag{4.15}$$

(and similarly for $other'(k)$).

So, if the agent is currently in state q_l, (with l being the state identifier, or index) then the following equation holds:

$$l = \left(\sum_{k=1}^{3} 4^{3-k} [own'(k) + 2 \times other'(k)] \right) + 1 \tag{4.16}$$

Once expanded, this equation becomes

$$\begin{aligned} l \quad &= 2^4 \times own'(1) \quad +2^5 \times other'(1) \\ &+2^2 \times own'(2) \quad +2^3 \times other'(2) \\ &+2^0 \times own'(3) \quad +2^1 \times other'(3) + 1 \end{aligned}$$

The Transition Function

After each "round" of the repeated game, each machine changes its state. The new state (the post-transition state $q_{l'}$ with index l') relies only on the machine's pre-transition state q_l and the opponent's move a: $q_{l'} := \tau(q_l, a)$.

Given a machine's pre-transition state q_l, we can surmise which actions the machine (and its opponent) took over the last 3 moves (i.e. the values of the functions $own(k)$ and $other(k)$), using Equation 4.16:

$$own'(3) := (l-1) mod\, 2 \tag{4.17}$$

$$other'(3) := \frac{(l-1) - 2^0 \cdot own'(3)}{2} mod\, 2 \tag{4.18}$$

$$own'(2) := \frac{(l-1) - 2^0 \cdot own'(3) - 2^1 \cdot other'(3)}{4} mod\, 2 \tag{4.19}$$

$$\vdots \tag{4.20}$$

$$other'(1) := \frac{(l-1) - 2^0 \cdot own'(3) - 2^1 \cdot other'(3) - 2^2 \cdot own'(2) - 2^3 \cdot other'(2) - \ldots}{32} mod\, 2 \tag{4.21}$$

We define new functions ${}^{new}own(k)$ and ${}^{new}other(k)$, which help us define the post-transition state $q_{l'}$ (g is, as before, the agent's output function):

$$^{new}own'(k) := \quad own'(k-1) \qquad \text{for } k = 2,3 \tag{4.22}$$

$$^{new}other'(k) := \quad other'(k-1) \qquad \text{for } k = 2,3 \tag{4.23}$$

$$^{new}own'(1) := \begin{cases} 1 & \text{if } g(q_l) = D \\ 0 & \text{if } g(q_l) = C \end{cases} \tag{4.24}$$

$$^{new}other'(1) := \begin{cases} 1 & \text{if } a = D \\ 0 & \text{if } a = C \end{cases} \tag{4.25}$$

Finally, we can use the values of $^{new}own'(k)$ and $^{new}other'(k)$ with Equation 4.16 to obtain an expression for the post-transition state index l':

$$\begin{aligned} l' &= 2^4 \times {}^{new}own'(1) & +2^5 \times {}^{new}other'(1) \\ &+2^2 \times {}^{new}own'(2) & +2^3 \times {}^{new}other'(2) \\ &+2^0 \times {}^{new}own'(3) & +2^1 \times {}^{new}other'(3) + 1 \end{aligned}$$

Combining the above with Equations 4.22, 4.23, 4.24 and 4.25, we obtain:

$$\begin{aligned} l' &= 2^4 \times 1_{\{g(q_l)=D\}} & +2^5 \times 1_{\{a=D\}} \\ &+2^2 \times own'(1) & +2^3 \times other'(1) \\ &+2^0 \times own'(2) & +2^1 \times other'(2) + 1 \end{aligned}$$

In summary, the transition function τ is defined as $(q_l, a) \mapsto q_{l'}$, where l' is defined as above.

Example 16 (Transition Function)**.** *Say, a strategy is in state q_{53}, $g(q_{53}) = C$ and $a = C$. Using the above equations, we can evaluate own, other, ^{new}own and ^{new}other:*

$$\begin{aligned} (own(1), other(1), own(2), \ldots, other(3)) &= (D, D, C, D, C, C) \\ (^{new}own(1), ^{new}other(1), ^{new}own(2), \ldots, ^{new}other(3)) &= (C, C, D, D, D, C) \\ (^{new}own'(1), ^{new}other'(1), \ldots, ^{new}other'(3)) &= (0, 0, 1, 1, 1, 0) \end{aligned}$$

Consequently

$$l' = 2^4 \times 0 + 2^5 \times 0 + 2^2 \times 1 + 2^3 \times 1 + 2^0 \times 1 + 2^1 \times 0 + 1 = 4 + 8 + 1 + 1 = 14 \quad (4.26)$$

and hence the new state index evaluates to 14 *(that is, the post-transition state is* q_{14}*).*

4.5.2 Encode And Decode

The decode function gives us the strategy from its binary representation. In our case here, all that is left for us to do, in order to complete the specification of the agent (and hence a strategy), is to provide the agent's initial state and its output function, given its chromosome $(b_1, ..., b_{70})$.

The Initial State

As we have discussed earlier, the first 6 genes of an agent's chromosome specify its false memory; this false memory allows us to define the "own" and "other" functions before the first round of the game. This, in turn, allows us to apply Equation 4.16 to work out what the initial state is (here m refers to the index of q^0; that is, $q^0 = q_m$):

$$\begin{aligned} m \; := & \; 2^4 \times b_5 \quad +2^5 \times b_6 \\ & +2^2 \times b_3 \quad +2^3 \times b_4 \\ & +2^0 \times b_1 \quad +2^1 \times b_2 + 1 \end{aligned}$$

The Output Function

We have already defined the machine states and the initial state for each machine. The output function $g : Q \rightarrow A$ can be defined as:

$$g(q_j) := \begin{cases} C & \text{if } b_{j+6} = 0 \\ D & \text{if } b_{j+6} = 1 \end{cases} \tag{4.27}$$

for $j \in \{1, ..., 64\}$.

While the strategy subset and the evaluation functions definitions yielded with this approach are still specific to the 3-round memory scenario, it would be trivial to adapt them to a scenario with a different memory depth.

4.6 Fitness, Genetic Functions & The Population

4.6.1 Fitness

Definition 24 (Axelrod Static Fitness)**.** *We define the fitness function $T_f : W^n \rightarrow (\mathbb{R}_0^+)^n$ as:*

$$(w_1, w_2, ..., w_n) \mapsto (f(w_1), f(w_2), ..., f(w_n)) \tag{4.28}$$

where $f : W \rightarrow \mathbb{R}_0^+$ is defined as

$$w \mapsto c_0 + \sum_{i=1}^{8} c_i \cdot U_w(v_i) \tag{4.29}$$

with $c_0 = 110.55$, $c_1 = 0.1506$, $c_2 = 0.1574$, $c_3 = 0.1185$, $c_4 = 0.0876$, $c_5 = 0.0487$, $c_6 = 0.0579$, $c_7 = 0.0492$ and $c_8 = 0.0463$.

$U_w(s)$ is the outcome (for w) of a 151-round Iterated Prisoner's Dilemma game against strategy s:

$$U_w(s) = \sum_{i=1}^{151} u_1(a^i, b^i) \tag{4.30}$$

where $a^i = w(h_{i-1})$, $b^i = s(h_{i-1})$, u the payoff function for the PD, $h_0 = \emptyset$ and $h_i = (\{a^1, b^1\}, \{a^2, b^2\}, ..., \{a^i, b^i\})$.

Definition 25 (Axelrod Dynamic Fitness)**.** *We define the dynamic fitness function ${}^{dyn}T_f : W^n \to (\mathbb{R}_0^+)^n$ as:*

$$(w_1, w_2, ..., w_n) \mapsto (f(w_1), f(w_2), ..., f(w_n)) \tag{4.31}$$

where $f : W \to \mathbb{R}_0^+$ is defined as

$$w \mapsto \frac{1}{n} \sum_{i=1}^{n} U_w(w_i) \tag{4.32}$$

*Here, $U_w(s)$ is the **average round** outcome (for w) of a 151-round Iterated Prisoner's Dilemma game against strategy s:*

$$U_w(s) = \frac{1}{151} \sum_{i=1}^{151} u_1(a^i, b^i) \tag{4.33}$$

where $a^i = w(h_{i-1})$, $b^i = s(h_{i-1})$, u the payoff function for the PD, $h_0 = \emptyset$ and $h_i = (\{a^1, b^1\}, \{a^2, b^2\}, ..., \{a^i, b^i\})$.

4.6.2 The Genetic Functions

Selection

T_s is the sigma scaling selection function: it eliminates all the strategies that are 1 standard deviation below the average population fitness, and replaces them with clones of the fittest strategies.

Definition 26 (Sigma Scaling Selection)**.** *We define the selection function $T_s^i : B^n \times (\mathbb{R}_0^+)^n \to B^2$ as:*

$$((c_j)_{j=1}^n, (f_j)_{j=1}^n) \mapsto (d_{2i-1}, d_{2i}), \text{ where} \tag{4.34}$$

$$d_i = c_i \cdot 1_{\{g(i)>0\}} + c_{(n-p(i))} \cdot 1_{\{g(i)\leq 0\}} \tag{4.35}$$

$$p(k) = \sum_{l=1}^{k} 1_{\{g(l)>0\}} + 1,\ 1 \leq k \leq N, \tag{4.36}$$

$$g(l) = max(0, \frac{f_l - \mu}{\sigma} + 1), \tag{4.37}$$

$$\mu = \frac{1}{n}\sum_{k=1}^{n} f_k,\ \sigma = \left(\frac{1}{n} \cdot \sum_{k=1}^{n} (f_k - \mu)^2\right)^{\frac{1}{2}} \tag{4.38}$$

In 4.35, we use a variable of the form $c_{(k)}$; $(c_{(k)})_{k=1}^{n}$ is a permutation of the population tuple $(c_k)_{k=1}^{n}$ in such a way, that $c_{(1)} < c_{(2)} < ... < c_{(n)}$ (this is equivalent to the concept of order statistic in probability theory).

g (from 4.37) is the *sigma scaling* of the fitness function f.

An alternative (and much simpler) selection definition is a variant of the roulette wheel selection (as in Definition 1); the only difference is that the probability distribution of the "wheel" is based on the the sigma scaled fitness (and not the "pure" fitness):

$$P\{T_s((c_k)_{k=1}^{n}, (f_k)_{k=1}^{n}) = (c_i, c_j)\} = \frac{g(i)}{\sum_{k=1}^{n} g(k)} \cdot \frac{g(j)}{\sum_{k=1}^{n} g(k)} \tag{4.39}$$

Crossover

We define the 1-point crossover function $T_c : B^2 \to B^2$ ($= T_c^i$ for all $i = 1, ..., \frac{n}{2}$) as:

$$((b_1, ..., b_n), (c_1, ..., c_n)) \mapsto ((b_1, ..., b_Y, c_{Y+1}, ..., c_N), (c_1, ..., c_Y, b_{Y+1}, ..., b_n))$$

for $(b_1, ..., b_n), (c_1, ..., c_n) \in B$ and Y is a discrete r.v., drawing members from the set $\{1, 2, ..., n\}$ with uniform probability $\frac{1}{n}$.

$$P(Y = k) = \frac{1}{n}\, for\ k \in \{1, 2, ..., n\} \tag{4.40}$$

Mutation

The mutation function is simply the mutation function from Definition 3: $T_m =^{GA} T_m$.

4.6.3 The Initial Population

The initial population in this experiment was randomly generated, so we can just give the initial chromosome population:

$$\vec{X}(0) = ((Z_{1j}, Z_{2j}, ..., Z_{70j}))_{j=1}^{n} \tag{4.41}$$

where $Z_k \sim Z$: Bernoulli($\frac{1}{2}$) $\forall k$.

4.6.4 The Terminating Condition

Definition 27 (b-trigger Terminating Condition)**.** *A b-trigger terminating condition ends a GGA run at the b^{th} generation:*

$$^{b}T_t(\vec{X}, k) = \begin{cases} false & \text{if } k < b \\ true & \text{if } k \geq b \end{cases} \tag{4.42}$$

The Axelrod terminating condition is a 50-trigger condition.

4.7 Verifying the Definitions

We would like to verify that the GGA functions defined in this chapter faithfully reflect Axelrod's setup; we do this by implementing them in a simulation, and comparing the simulation's output to the results from [Axelrod, 1987] and [Marks, 1989].

4.7.1 Simulation Implementation

In order to verify the theory, the GGA and the problem-specific functions described in earlier sections, were implemented in a Java simulation (the details, including code snippets, can be found in Appendix A.1).

Several problems were encountered during the implementation.

Firstly, the reliance of the fitness function upon 8 specific pre-programmed strategies meant that it could not be replicated without the original source code[2]. Further difficulties arose during integration of the strategy code (which was written in FORTRAN 77) into the simulation code (that was written in Java). Time constraints meant that the preferred solution - creating a bridge adapter using the JNI[3] - was not feasible. Auto-translation of the FORTRAN code to Java was also ruled out, as none of the available open-source tools proved adequate for the task. In the end, the strategies were ported to pure Java by hand.

4.7.2 Results Comparison

Preliminary Test

As a preliminary test, the fitness function was applied to 3 strategies: AllC (Example 8), AllD (Example 9) and TFT (Example 11), just like Marks did in his paper. The comparison can be seen in Table 4.3.

Strategy	Marks Result	GGA Result
AllD	319.831	319.4498
AllC	398.513	347.6118
TFT	427.198	378.3799

Table 4.3: Fitness Calibration

The matching of the AllD fitness values and the matching of the order in which the strategies ranked was encouraging; disappointingly, the AllC and TFT fitness values did not match. It is likely that one or more mistakes were introduced during the porting process.

[2]Luckily, Prof. Marks came to the rescue and was kind enough to send me the code when I contacted him.

[3]Java Native Interface

Simulation with Static Fitness

Axelrod's simulations were run using a population size of twenty individuals per generation. A run consisted of 50 generations. Forty runs were conducted under identical conditions to allow an assessment of the variability of the results.

Here is a quote from [Axelrod, 1987], describing the outcome:

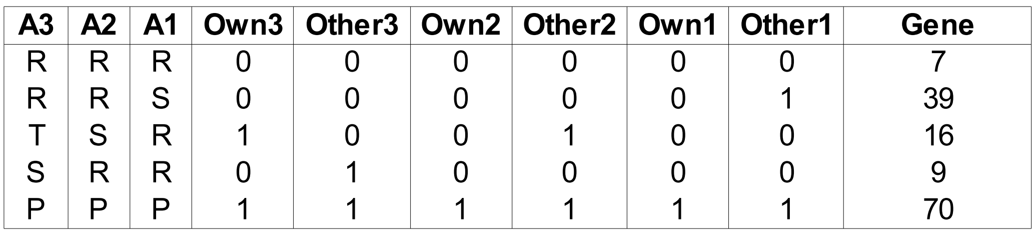
> Most of the strategies that evolved in the simulation actually resemble TIT FOR TAT, having many of the properties that make TIT FOR TAT so successful. For example, five behavioral alleles in the chromosomes evolved in the vast majority of the individuals to give them behavioral patterns that were adaptive in this environment and mirrored what TIT FOR TAT would do in similar circumstances. These patterns are:
>
> 1. Do not rock the boat: continue to cooperate after three mutual cooperations (which can be abbreviated as C after RRR).
> 2. Be provocable: defect when the other player defects out of the blue (D after receiving RRS).
> 3. Accept an apology: continue to cooperate after cooperation has been restored (C after TSR).
> 4. Forget: cooperate when mutual cooperation has been restored after an exploitation (C after SRR).
> 5. Accept a rut: defect after three mutual defections (D after PPP).
>
> While most of the runs evolve populations whose rules are very similar to TIT FOR TAT, in eleven of the forty runs, the median

> rule actually does substantially better than TIT FOR TAT[4]. In these eleven runs, the populations evolved strategies that manage to exploit one of the eight representatives at the cost of achieving somewhat less cooperation with two others. But the net effect is a gain in effectiveness.

We would like to replicate Axelrod's runs, and see whether we get the same results as him. Before we can do that, we need to translate his technical vocabulary into our language. For a rule given in the form $A_3A_2A_1$, we can calculate the genes to look for using the *other/own* lookup Table 4.4 - the calculations can be seen in Figure 4.5 and are summarised in Table 4.5.

A_i	own(i)	other(i)
R	0	0
T	1	0
S	0	1
P	1	1

Table 4.4: Converting Notations



A3	A2	A1	Own3	Other3	Own2	Other2	Own1	Other1	Gene
R	R	R	0	0	0	0	0	0	7
R	R	S	0	0	0	0	0	1	39
T	S	R	1	0	0	1	0	0	16
S	R	R	0	1	0	0	0	0	9
P	P	P	1	1	1	1	1	1	70

Figure 4.5: Calculating the Genes to Watch

In order for us to assess how similar our results are to Axelrod's, we need to answer the following questions:

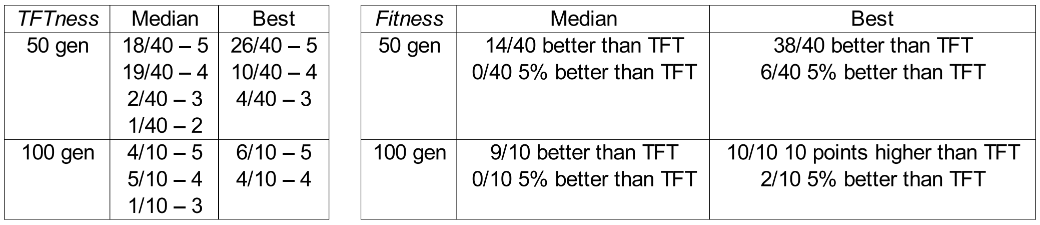
[4]"Substantially" is later explained to be a 5% increase on Tit-for-Tat's fitness.

Axelrod Rule	**Check for**
C after RRR	0 at gene 7
D after RRS	1 at gene 39
C after TSR	0 at gene 16
C after SRR	0 at gene 9
D after PPP	1 at gene 70

Table 4.5: Confirming Results

1. How does the fitness of the strategies extracted after each run compare with TFT?

2. To what degree do the strategies extracted after each run exhibit "TFT-ness" (that is, how many of Axelrod's 5 criteria from Table 4.5 do the strategies satisfy)?



TFTness	Median	Best
50 gen	18/40 – 5 19/40 – 4 2/40 – 3 1/40 – 2	26/40 – 5 10/40 – 4 4/40 – 3
100 gen	4/10 – 5 5/10 – 4 1/10 – 3	6/10 – 5 4/10 – 4

Fitness	Median	Best
50 gen	14/40 better than TFT 0/40 5% better than TFT	38/40 better than TFT 6/40 5% better than TFT
100 gen	9/10 better than TFT 0/10 5% better than TFT	10/10 10 points higher than TFT 2/10 5% better than TFT

Figure 4.6: Simulation Results Using Static Fitness

40 runs of the simulation were made, each running for 50 generations. A further 10 runs were made, each running for 100 generations. For each run, both the best strategy of the final generation, and the median strategy of the final generation were examined. The results are summarised in Figure 4.6 (in the TFTness table, the meaning of "A/B - C" for a cell labelled "50 gen - Median" means that out of the B runs which ran for 50 generations, A of the median values passed C TFTness tests).

It can be seen than none of our median results were "substantially" (5%) better than TFT fitness; however our best strategy results seem to fit Axelrod's findings more closely. And our TFTness results from the best strategies confirm Axelrod's assertions that the best strategies are those that exhibit TFT-like properties.

Simulation with Dynamic Fitness

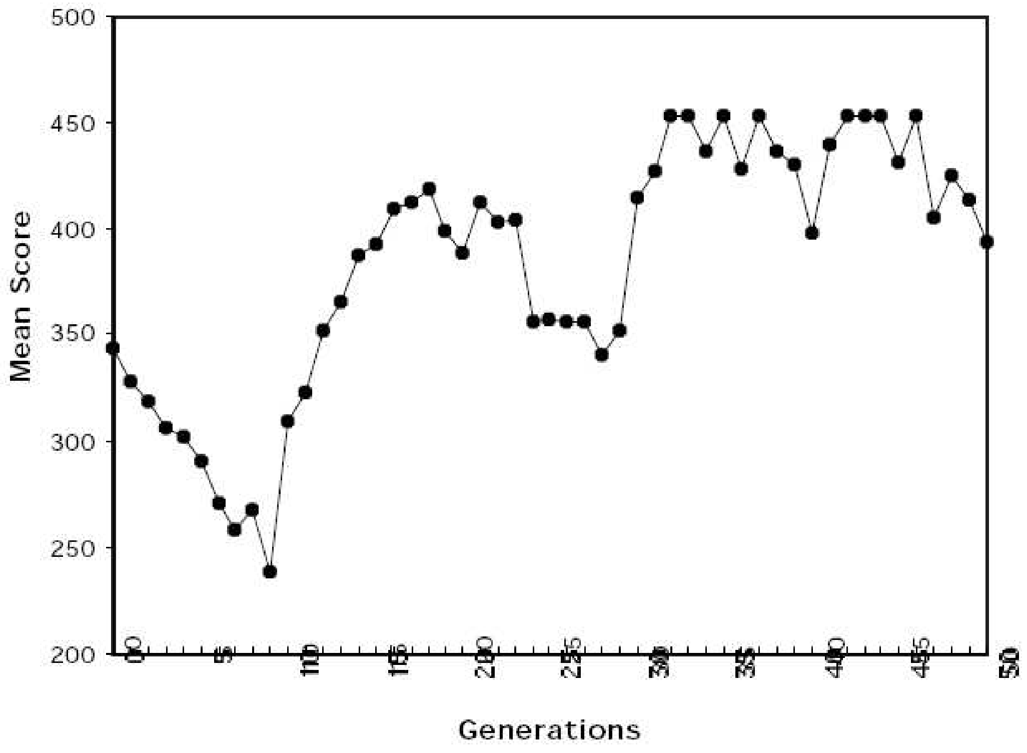


Figure 4.7: A Typical Run from Axelrod's Dynamic Fitness Experiment

Axelrod also ran 10 simulation runs using the dynamic fitness; he describes his results in the following quote:

> A typical run is shown in Figure 4.7. From a random start, the population evolves away from whatever cooperation was initially

> displayed. The less cooperative rules do better than the more cooperative rules because at first there are few other players who are responsive - and when the other player is unresponsive the most effective thing for an individual to do is simply defect. This decreased cooperation in turn causes everyone to get lower scores as mutual defection becomes more and more common. However, after about ten or twenty generations the trend starts to reverse. Some players evolve a pattern of reciprocating what cooperation they find, and these reciprocating players tend to do well because they can do very well with others who reciprocate without being exploited for very long by those who just defect. The average scores of the population then start to increase as cooperation based upon reciprocity becomes better and better established. So the evolving social environment led to a pattern of decreased cooperation and decreased effectiveness, followed by a complete reversal based upon an evolved ability to discriminate between those who will reciprocate cooperation and those who won't. As the reciprocators do well, they spread in the population resulting in more and more cooperation and greater and greater effectiveness.

After simulation runs with the dynamic fitness, all of the runs exhibit the initial dip in cooperation that Axelrod described. However, after 50 generations (Figure 4.8), only 2 of the 10 runs show the recovery in cooperation that Axelrod talks about in his typical case[5]: most of the remaining runs exhibit low levels of cooperation. For runs over 100 generations (Figure 4.9), 4 out of 10 exhibit high levels of cooperation: it seems that once cooperation is established, it is more stable than non-cooperation.

[5]Axelrod's paper does not make it explicitly clear, as to what "typical" means statistically.

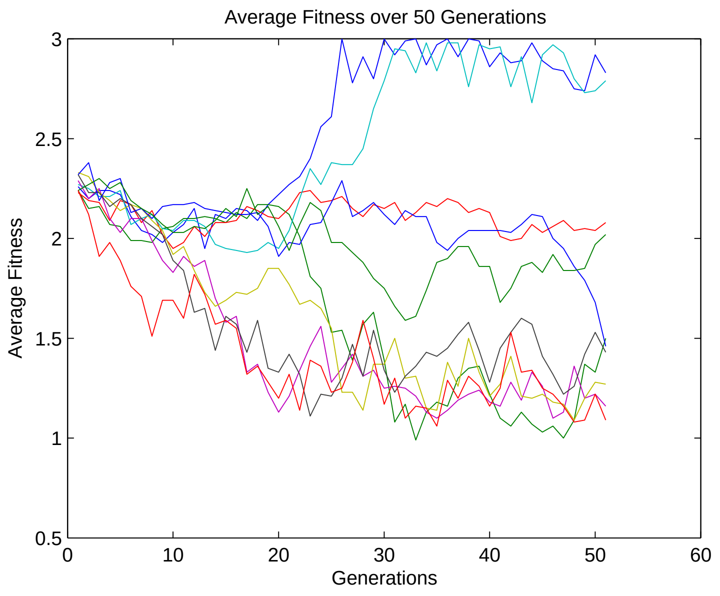


Figure 4.8: Dynamic Fitness Results over 50 Generations

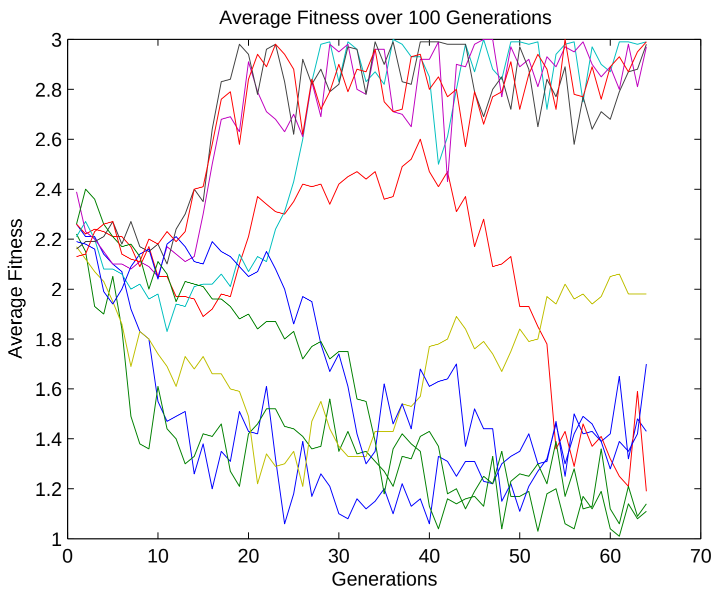


Figure 4.9: Dynamic Fitness Results over 100 Generations

The above experimental results exhibited enough similarities with Axelrod's results to inspire confidence in the methods used; it is likely that the discreptancies between our and Axelrod's results are a consequence of the fitness function not being identical and of slight variations in the genetic parameters.

Chapter 5

The Contract Game

This chapter will revolve around a mathematical economics paper entitled "Social norms and economic incentives in firms" [Huck et al., 2003], that studies the equilibria in a model (which we shall call the **contract game**) that is similar to the public goods game from game theoric literature[1]. Our goal will be to investigate the usefulness of the GGA as a problem analysis tool, by comparing the analytic deductions derived within the paper to the results obtained from applying the GGA to the contract game model.

Section 5.1 introduces the language and the components of the contract game, and explains our motivations for studying it - the game's formal definition is given in Section 5.2. Section 5.3 presents the analytic approach for finding the game's equlibria, Section 5.4 outlines how this can be done using the GGA, and Section 5.5 analyses and compares the results from the two approaches. Section 5.6 contains some remarks on the preceding sections, and Section 5.7 is a brief conclusion.

The background section, and the underlying mathematics in the analytic

[1]The public goods game, a relative of the iterated prisoner's dilemma, was designed to illustrate such problems as voluntary payment of taxes and contribution to team and community goals.

approach section have been taken from [Huck et al., 2003] (although they appear in heavily edited form).

5.1 Background

[Huck et al., 2003] studies the interplay between economic incentives and social norms in firms. Its main focus is on efficiency norms, that is, norms that arise from a firm's workers' desire for, or peer pressure towards, social efficiency for the workers as a group.

In this context, a social norm (among the workers) is taken to be an informal (unwritten) rule that is enforced either by the workers themselves or is internalised by these, but is not enforced by a third party, such as an employer. Once established in a group, a social norm is self-enforcing: expecting the others to adhere to the norm, each worker wants to adhere as well.

For a firm owner, social norms concerning work efforts are important because they affect profits. Suppose, for example, that a workers compensation not only depends on his or her own effort but also on the effort of other workers. In the presence of such externalities, peer pressure, whether explicit or internalised, might penalise those who deviate from what is good for the group. And depending on the type of externality, which in its turn depends on the economic incentives in the firm, equilibrium output may be higher or lower than without the norm. On the one hand, if an increase in one worker's effort increases other workers' (expected) income, as in team production, a social norm may induce high efforts. On the other hand, if an increase in one workers effort reduces others' (expected) income, such as under relative-performance pay schemes or under piece-rate schemes that are adjusted according to past performance, social norms may instead keep workers back from working hard.

Group norms are analysed in a simple model of team incentives. Each individual worker's effort level is unobserved by the firm owner, but the total output can be observed and verified without error. We study the following basic scenario: first, the principal, who is the owner of the firm, chooses a base wage paid to all workers and a bonus proportional to the team's output. Then, the agents, that is, the workers in the team, simultaneously choose their individual work efforts. We study the effects of a social norm concerning work effort among these team members.

A key observation is that social norms may cause multiplicity of equilibria under a given contract. Intuitively, multiplicity arises when a social norm introduces a coordination problem into the agents' effort choices: others' high or low efforts may serve as a norm for the individual worker.

5.2 The Model

Consider a firm with $n+1$ staff members: $n > 1$ identical employees (workers) and one boss (principal) - referred to as $1, ..., n$ and p respectively. Each worker i receives the same wage w; the risk neutral, profit-maximising principal pays the wages and is the residual claimant, receiving π. Each worker chooses to exert a certain effort $x_i \geq 0$; the principal chooses a contract, consisting of base wage a and a bonus rate b (both of which are assumed to be nonnegative). The firm's production technology is linear, with the output y being the sum of all workers' efforts; the wage received by each worker is an affine function of the firm's output, and is also dependent on the base wage and the bonus rate.

The interaction between the staff is in two-stages: at first, the principal chooses a contract (a, b), and then all the workers observe the contract and simultaneously choose their individual efforts x_i. After everyone has made a decision, the total output y is revealed and the utilities are calculated

- however, the levels of effort that each individual worker exerted remains known only to that worker. The principal's utility is a function of the number of employees, the contract and the output; the workers' utilities are additively separable, linear-quadratic in income and efforts, and they may contain a term v representing social preferences.

Let us summarise the situation described above:

Definition 28 (The Contract Game)**.** *The* ***contract game*** $\langle N, H, P, (u_i)\rangle$ *is an extensive game with perfect information and simultaneous moves, as in Definition 11, in which:*

- $N = \{1, 2, ..., n, p\}$
- $H = \emptyset \cup \{(a,b) |\, a, b > 0\,\} \cup \{((a,b),(x_1, x_2, ..., x_n)) |\, x_i \geq 0 \ \text{ for all } i\}$
- $P(h) := \begin{cases} p & \text{if } h = \emptyset \\ N \setminus \{p\} & \text{otherwise} \end{cases}$
  for all non-terminal histories h
- *for player p,*
$$u_p((a,b),(x_i)_{i=1}^n) = \pi = y - nw = y - n(a + by/n) \tag{5.1}$$
  where $y = \sum_{i=1}^n x_i$*; for players* $i = 1, ..., n$,
$$u_i((a,b),(x_i)_{i=1}^n) = w - \frac{1}{2}x_i^2 - v(x_i, \hat{x}_{-i}, b) \tag{5.2}$$
  where $v : \mathbb{R}^3 \to \mathbb{R}$ *is a continuous function representing the social preference (to be specified later), and for player i,* $\hat{x}_{-i} = \sum_{j\neq i} \frac{x_j}{n-1}$ *is the average of the other workers' efforts.*

Now that we have defined the game, we would like to solve it, by finding the subgame perfect equilibria. Exactly what a subgame perfect equilibrium is, or how we go about finding it, is addressed in the next section.

5.3 Calculating The Subgame Perfect Equilibrium

To this point, we have been only concerned with games and their strategies, without looking at the dynamics of their interactions. Any discussion of game dynamics must start with the concept of Nash equilibrium, which we cite in Definition 30; however, besides the definition, we present little explanation or exposition, as ample coverage of this topic exists in game theoretic literature.

5.3.1 Subgame Perfect Equilibrium

Definition 29 (Outcome in an Extensive Game)**.** *For each strategy profile $s = (s_i)_{i \in N}$ in an extensive game $\langle N, H, P, (\succsim_i) \rangle$ we define the* ***outcome*** *$O(s)$ of s to be the terminal history that results when each player $i \in N$ follows the precepts of s_i. That is, $O(s)$ is the (possibly infinite) history $(a^1, ..., a^K) \in Z$ such that for $0 \leq k \leq K$ we have $s_{P(a^1,...,a^k)}(a^1, ..., a^k) = a^{k+1}$.*

The first solution concept we define for an extensive game ignores the sequential structure of the game; it treats the strategies as choices that are made once and for all before play begins.

Definition 30 (Nash Equilibrium of an Extensive Game with Perfect Information)**.** *A Nash equilibrium of an extensive game with perfect information $\langle N, H, P, (\succsim_i) \rangle$ is a strategy profile s^* such that for every player $i \in N$ we have*

$$O(s^*_{-i}, s^*_i) \succsim_i O(s^*_{-i}, s_i) \tag{5.3}$$

for every strategy s_i of player i.

Example 17 (Nash Equilibrium of an Extensive Game with Perfect Information)**.** *We revisit Example 4 (illustrated in Figure 2.2); the Nash equilibria of that game are $((2,0), yyy)$, $((2,0), yyn)$, $((2,0), yny)$, $((2,0), ynn)$,*

$((1,1), nyy)$, $((1,1), nyn)$, $((0,2), nny)$, $((2,0), nny)$, $((2,0), nnn)$. The first four results are in the division $(2,0)$, the next two results are in the division $(1,1)$, the next result is in the division $(0,2)$ and the last two results are in the division $(0,0)$. All of the equilibria except $((2,0), yyy)$ and $((1,1), nyy)$ involve an action of player 2 that is implausible after some history (since he rejects a proposal that gives him at least one of the objects).

Definition 31 (A Subgame of an Extensive Game with Perfect Information)**.** *The subgame of the extensive game with perfect information $\Gamma = \langle N, H, P, (\succsim_i)\rangle$ that follows the history h is the extensive game $\Gamma(h) = \langle N, H|_h, P|_h, (\succsim_i|_h)\rangle$, where $H|_h$ is the set of sequences h' of actions for which $(h, h') \in H$, $P|_h$ is defined by $P|_h(h') = P(h, h')$ for each $h' \in H|_h$, and $\succsim_i|_h$ is defined by $h' \succsim_i|_h h"$ if and only if $(h, h') \succsim_i (h, h")$.*

The notion of equilibrium we now define requires the action prescribed by each player's strategy to be optimal, given the other players' strategies, after every history. Given a strategy s_i of player i and a history h in the extensive game Γ, denote by $s_i|_h$ the strategy that s_i induces in the subgame $\Gamma(h)$ (i.e. $s_i|_h(h') = s_i(h, h')$ for each $h' \in H|_h$); denote by O_h the outcome function $\Gamma(h)$.

Definition 32 (Subgame Perfect Equilibrium for an Extensive Game with Perfect Information)**.** *The subgame perfect equilibrium of an extensive game with perfect information $\Gamma = \langle N, H, P, (\succsim_i)\rangle$ is a strategy profile s^* such for every player $i \in N$ and every nonterminal history $h \in H \setminus Z$ for which $P(h) = i$ we have*

$$O_h(s^*_{-i}|_h, s^*_i|_h) \succsim_i |_h O(s^*_{-i}|_h, s_i) \tag{5.4}$$

for every strategy s_i of player i in the subgame $\Gamma(h)$.

Equivalently, we can define a subgame perfect equilibrium to be a strategy profile s^* in Γ for which for any history h the strategic profile $s^*|_h$ is a Nash equilibrium of the subgame $\Gamma(h)$.

The notion of subgame perfect equilibrium eliminates Nash equilibria in which the players' threats are not credible. For example, in the game from Example 4 the only subgame perfect equilibria are $((2,0), yyy)$ and $((1,1), nyy)$.

5.3.2 Back to the Problem

As we have already mentioned, we are looking for the Contract Game's subgame perfect equilibrium strategy profile. More precisely, we will focus on symmetric equilibria, that is, equilibria in which all workers use the same strategy x and, thus, exert the same effort under any given contract (a, b).

Lemma 1. *In every symmetric equilibrium of the Contract Game, the principal chooses the base wage to be zero ($a = 0$).*

Proof. A uniform effort profile $(x, x, ..., x)$ constitutes a symmetric Nash equilibrium in the corresponding subgame if and only if the common effort x is a worker's best reply when the others exert effort x:

$$\begin{aligned} x \in \xi(x) &= arg\, max_{t \geq 0}\, \psi(t, x) && (5.5) \\ &= arg\, max_{t \geq 0} \left\{ a + \frac{b}{n} t - \frac{1}{2} t^2 - v(t, x, b) \right\} && (5.6) \end{aligned}$$

From the above, it is clear that the workers' utility maximisation does not depend on the base wage a. Hence, since such a salary is costly to the principal, he will minimise it to $a = 0$ in every subgame perfect equilibrium. □

No Social Preferences

We first consider the simplest version of the Contract Game: the case when the workers' preferences are not social, that is, when the social preference is everywhere zero.

Lemma 2. *When the social preference $v \equiv 0$, the unique subgame perfect equilibrium strategy profile is $(s_p, s_w, s_w, ..., s_w)$, where $s_p := (0, \frac{1}{2})$ and $s_w(\{(a,b)\}) := \frac{1}{2n}$.*

Proof. We determine the subgame perfect equilibrium strategy using backwards induction, so we consider the workers' utility first; with $v \equiv 0$, Equation 5.2 becomes:

$$u_i = w - \frac{1}{2}x_i^2 \tag{5.7}$$

Thus, each worker solves

$$max_{x_i \geq 0}\left\{\frac{b}{n}x_i - \frac{1}{2}x_i^2\right\} \tag{5.8}$$

Consequently, the unique Nash equilibrium effort level, given any contract (a, b), is $x_i = \frac{b}{n}$ for all workers i. We solve for subgame perfect equilibrium by inserting the equilibrium effort into the expression for the principal's utility, to obtain

$$u_p = (1-b)b - na \tag{5.9}$$

By Lemma 1 (and since u_p is a decreasing function of the base wage), the principal will set $a = 0$, and maximise his utility by setting $b = \frac{1}{2}$. Substituting this back into the workers' effort x_i, we get $x_i = \frac{1}{2n}$. □

In sum: in the absence of a social norm, there exists a unique subgame perfect equilibrium. In this equilibrium, the principal offers zero base salary, and offers 50/50 split of the firms' revenue with the team of workers.

The Team Optimum

We have assumed, up to now, that the workers do not cooperate, and only optimise their own utility. However, were the workers interested in the overall good of the team, (that is, trying to maximise the sum of the worker utilities, not the individual utilities), they would be exert a different effort level to the individual optimum.

Definition 33 (Team Optimum Effort)**.** *Let x be the symmetric effort exerted by every worker, in order to maximise the sum of the worker utilities:*

$$max_{x \geq 0} \left\{ \sum_i u_i \right\} \tag{5.10}$$

We will call this effort level the ***team optimum effort*** *for the workers as a collective under contract (a, b), and denote it by $o(a, b)$.*

Lemma 3. *In the case when the workers' preferences are not social, for any given contract (a, b), the team optimum effort x is equal to b.*

Proof. Equation 5.7 gives us the worker utility in the case where $v \equiv 0$, and so in order to calculate the team optimum, we need to solve

$$max_{x \geq 0} \left\{ \sum_i u_i \right\} = max_{x \geq 0} \{ f(x) \} = max_{x \geq 0} \left\{ n \left(a + bx - \frac{1}{2} x^2 \right) \right\} \tag{5.11}$$

Clearly, we require $0 = f'(x)$, which is satisfied at $x = b$. □

The Efficiency Preference

We assume here that each worker i has social preferences with the following qualitative feature: if others cause higher externalities, or worker i expects them to do so, then also i wants to work more. This is the idea of peer pressure. Such peer pressure can be internalised or external: arising in the mind of the defector (say, by way of reduced self-esteem), and/or arising from other team members' irritation or anger from losing due to i's shirking. Here we study an internalised social norm, as the defectors are not punished in our model.

Through our choice of efficiency preference, we would like to capture two phenomena: workers derive disutility when deviating from the team optimum, and this disutility is increased or reduced based on whether the others in the team work more or less, respectively - that is, if a worker is slacking off along with everyone else, the disutility is much less than when the slacker is surrounded by hard workers. We formalise such preferences in the following remark.

Remark 3. *We assume that the social disutility term $v(x_i, \hat{x}_{-i}, b)$ is convex in the worker's own effort, with the minimum at $x_i = b$ (the team optimum effort level), and non-decreasing in $\hat{x}_{-i}$, the other workers' average effort.*

Proposition 1. *Suppose that $v : \mathbb{R}^+ \to \mathbb{R}$ is continuous, and that $v(t, x, b)$, for any $b, x > 0$, is convex in t with the minimum at $t = b$. Then there exists at least one symmetric Nash equilibrium effort x for each $b > 0$, and that all such x lie in the interval $[\frac{b}{n}, b)$*

Proof. In the model where workers had no social preferences, we proved the existence of a subgame perfect equilibrium by construction. For this current model, we can guarantee the existence by the following lemma (taken from [Huck et al., 2003]) - the lemma also proves that all the solutions lie in the closed interval $[\frac{b}{n}, b]$. Corollary 1 finishes the proof by showing that $x = b$ cannot be a solution. □

Lemma 4. *Suppose that v is as in Proposition 1. Then there exists at least one symmetric Nash equilibrium effort x for each $b > 0$, and that all such x lie in the interval $[\frac{b}{n}, b]$.*

Proof. Suppose $b > 0$. A common effort level x is a Nash equilibrium associated with contract (a, b) if and only if it satisfies 5.6. First note that, for each $x \geq 0$, the set $\xi(x)$ is a nonempty, convex and compact subset of the interval $[\frac{b}{n}, b]$. No t outside this interval belongs to the set $\xi(x)$, since the maximand is strictly increasing in t to the left of this interval, and it is strictly decreasing to the right of the interval. Hence, $\xi(x)$ is a nonempty and compact subset of $[\frac{b}{n}, b]$, by Weierstrass' Maximum Theorem applied to the continuous maximand. Moreover, if v is convex, then the maximand is concave, and thus $\xi(x)$ is convex. Second, the so-defined solution correspondence ξ, which maps others' mean effort x to own optimal efforts $t \in \xi(x)$, is upper hemi-continuous by Berge's Maximum Theorem. Being a compact- and convex-valued upper hemi-continuous correspondence from $\mathbb{R}_+$ to $[\frac{b}{n}, b] \subset \mathbb{R}_+$, ξ has at least one fixed point x in $[\frac{b}{n}, b]$, by Kakutani's Fixed-Point Theorem. □

So, given a v with the properties outlined in Remark 3, we know that all the equilibrium efforts x lie in $[\frac{b}{n}, b]$.

Corollary 1. *Given a v with the properties outlined in the above Lemma, and the additional property that it is continuously differentiable in its first argument, $x \neq b$.*

Proof. If v is continuously differentiable in its first argument, then we can combine Equation 5.6, with the fact that all the equilibrium efforts x are internal, (that is, $x > 0$), to surmise that

$$0 = \psi_1(t, x)|_{t=x} \tag{5.12}$$

Then:

$$0 = \frac{b}{n} - v_1'(x, x, b) - x \tag{5.13}$$

$$x = \frac{b}{n} - v_1'(x, x, b) \tag{5.14}$$

We have already required v to have a minimum at $t = b$ for any $b, x > 0$; thus,

$$v_1'(t, x, b)|_{t=b} = 0 \text{ for all } x > 0 \tag{5.15}$$

It is clear that after substituting 5.15 into 5.14, $x = b$ cannot satisfy the resulting equation. □

Peer Pressure

If v is proportional to the square of the worker's deviation from the socially optimal effort, then it satisfies all of the requirements (set out in Remark 3):

$$v(x_i, \hat{x}_{-i}, b) = \frac{1}{2}(o(a,b) - x_i)^2 p(o(a,b) - \hat{x}_{-i}) \tag{5.16}$$

where, in the simple case considered here, $o(a,b) = b$, and where $p : \mathbb{R}_+ \to \mathbb{R}_+$ is continuously differentiable with $p' \leq 0$. Hence,

$$v(x_i, \hat{x}_{-i}, b) = \frac{1}{2}(b - x_i)^2 p(b - \hat{x}_{-i}) \tag{5.17}$$

We will refer to the proportionality factor $p(b - \hat{x}_{-i})$ as peer pressure. The more others deviate from the social optimum, the less peer pressure does a deviating worker feel.

In this special case, the necessary and sufficient first order condition 5.14 (for x being an equilibrium effort) becomes $x = F(x)$, where

$$F(x) = \frac{1 + np(b-x)}{1 + p(b-x)} \cdot \frac{b}{n} \tag{5.18}$$

The function $F : \mathbb{R}_+ \to \mathbb{R}_+$ is clearly continuous and non-decreasing, with $F(0) \geq \frac{b}{n}$ and $F(b) < b$. Hence, we can immediately verify that there indeed exists at least one fixed-point under F, and that all fixed points belong to the interval $[\frac{b}{n}, b)$, where every fixed point is the effort level in a symmetric Nash equilibrium and vice versa.

We illustrate the above exposition by way of a numerical example.

Example 18 (Peer Pressure)**.** *Suppose that the peer-pressure function p is given by*

$$p(b-x) = \begin{cases} \alpha \, exp(-\beta(b-x)^2) & \text{if } x \leq b \\ \alpha & \text{if } x > b \end{cases} \tag{5.19}$$

for $\alpha, \beta > 0$. Here, α represents the utility weight placed on norm adherence and β the sensitivity of this weight to others' norm adherence.

After substituting the above p function into Equation 5.18, we can find the equilibrium solutions numerically[2]*. Figure 5.1 is a graph of the equilibrium (b,x) pairs for the number of workers* $k = 8$, $\alpha = 4$, *and* $\beta = 40$. *Note that for some bonus rates* ($b = 0.4$, *say), there are in fact up to 3 worker effort equilibrium levels.*

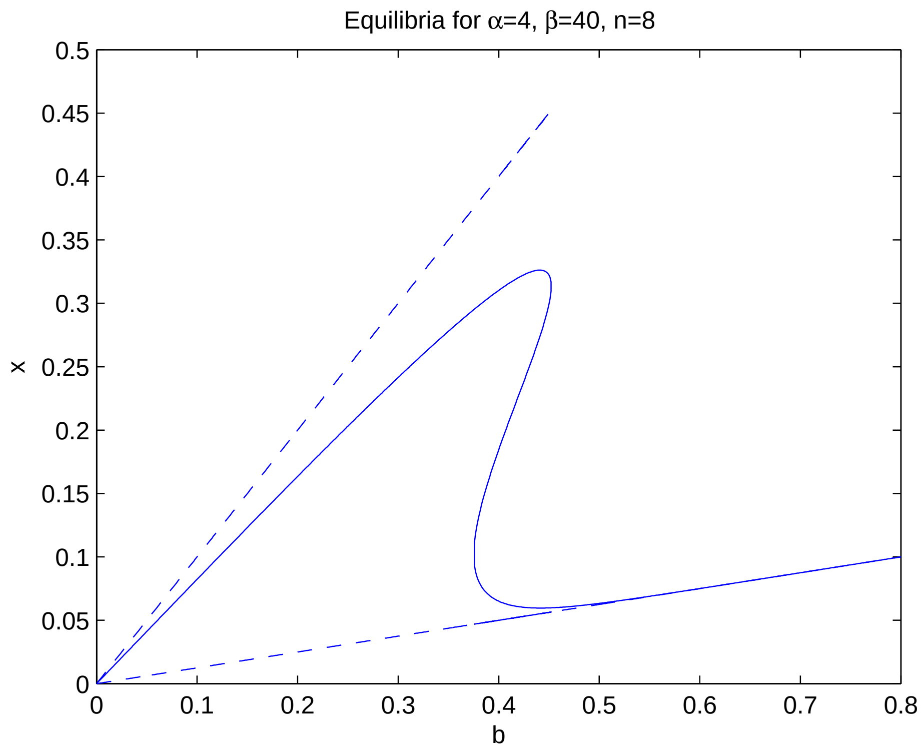


Figure 5.1: Nash equilibrium efforts x for different bonus rates b

[2]The Matlab code needed for the calculations and the plotting, was written myself and can be found in Appendix A.2

5.4 Applying the GGA

As we have already mentioned at the start of this chapter, our goal is to investigate the effectiveness of the GGA as a problem analysis tool. To that end, we shall apply the GGA to the contract game model and compare the results with the analytic results obtained in the previous section.

We will be attempting to find the symmetric subgame perfect equilibrium strategy profile $(s_p, s_w, s_w, ..., s_w)$; however, we shall take Lemma 1 as given, meaning that our problem can be reduced to finding the equilibrium action profile $((0, b), x, x, ..., x)$ - that is, finding the equilibrium action x in response to a given contract $(0, b)$ ($a = 0$ by the lemma). Consequently, we shall run the algorithm several times over, varying the b value each time.

It is worth noting that since it is impossible to search the entire infinite set of possible worker effort strategies (for reasons discussed in Section 3.3), we have to work with the discrete version of the contract game from here onwards - that is, the worker strategies can take values only from a finite subset of the entire solution space (we shall specify this subset in the next subsection).

5.4.1 The GGA Specification

The game used for fitness calculations is not actually the contract game (Definition 28), but a reduced subgame of it:

Definition 34 (The Reduced Contract Subgame)**.** *The* ***reduced contract subgame*** $\langle N \setminus \{p\}, A, (v_i) \rangle$ *is a strategic n-player symmetric game in which:*

- $A = \mathbb{R}_0^+$,
- $v_i(x_1, x_2, ..., x_n) = u_i((0, b), (x_i)_{i=1}^n$ *for* $i = 1, ..., n$,
- N, $(u_i)_{i=1}^n$ *are as in Definition 28, and*

- *b is a pre-defined constant.*

Because the subgame is a symmetric strategic game, we shall be using the symmetric strategic game GGA (Definition 19).

The chromosomes in the population represent the worker actions in response to a contract $(0, b)$. We recycle the genetic functions (selection, mutation, crossover) from the Axelrod experiment, except that we use slightly different parameters (a mutation probability of 0.02, and a crossover probability of 0.6). The terminating condition is a trigger condition (see Definition 27) that interrupts the GGA at the 200^{th} generation. The details of the remaining parts of the GGA are discussed below.

5.4.2 The Strategy Subset and Encoding/Decoding

The worker effort information is encoded in the chromosomes using Gray encoding:

Definition 35 (Gray Decoding)**.** *Assume that a real value v is represented by a chromosome, with associated step size t and minimum value m; in order to recover v, the following steps need to be followed:*

1. *The chromosome b is decoded to an integer i by treating the chromosome as the integer's base-2 representation*
2. *i is multiplied by the step size t to get value j ($j = s \times i$)*
3. *v is equal to the offset of j by the minimum value m ($v = j + m$)*

Example 19 (Gray Decoding)**.** *If $b = 0101$, $t = 0.01$ and $m = -0.2$, then:*

1. $i = 1 \times 2^0 + 1 \times 2^2 = 5$
2. $j = 5 \times 0.01 = 0.05$

3. $v = j + m = 0.03$

so the decoded value is 0.03*.*

We now give a definition of the encode and decode functions:

- Encode $T_e : D \to B$ with $v \mapsto (b_1, ..., b_l)$ is determined by a recursive definition:
$$b_1 = \frac{v-m}{t}\, mod\, 2^{l-1} \tag{5.20}$$
$$b_2 = \left(\frac{v-m}{t} - b_1 \cdot 2^{l-1}\right) mod\, 2^{l-1} \tag{5.21}$$
$$\vdots$$
$$b_k = \left(\frac{v-m}{t} - \sum_{j=1}^{k-1} b_j \cdot 2^{l-j}\right) mod\, 2^{l-k} \tag{5.22}$$
- Decode $T_d : B \to D$ with $(b_1, ..., b_l) \mapsto v$ is defined by $v = t \cdot \sum_{k=0}^{l} 2^k\, b_{k+1} + m$.

In the above, l is the length of the chromosome, t is the step size, and m is the minimum value.

In our case, the chromosome length $l = 12$, the step size $t = 0.000625$ and the minimum value $m = 0$, meaning that the subset of worker efforts being explored by the GGA is:

$$D = \{x \in X \mid x \text{ is a multiple of } 0.000625\} \tag{5.23}$$

where $X = [0, 0.000625 \times 2^{12}) = [0, 2.56)$.

5.4.3 The Population

The initial population is generated using the same random Bernoulli technique as in the Axelrod experiment. With regards to the size of the initial (and all subsequent) populations, we apply a restriction: the size of the population is equal to the number of workers in the game being played - this restriction simplifies the definition of the fitness function (this will become apparent when the fitness functions are introduced).

5.4.4 Fitness

Fitness Version 1

The fitness function should encourage optimality in the worker strategies - workers that get the best utility out of their current situation should be rated highly. Keeping this in mind, the first version of the fitness function is very simple: the fitness of a worker is his relative utility from one round of the contract game, played against the principal contract $(0, b)$ and the other strategies in the current population[3] (relative utility means that the minimum utility across all of the workers in the population is subtracted from each player's utility - consequently, there is always a strategy that has a fitness of 0, and all fitness values are non-negative).

Here is the definition of the first fitness function: ${}^1T_f : D^n \to (\mathbb{R}_0^+)^n$ with $s = (s_1, ..., s_n) \mapsto (f_1(s), ..., f_n(s))$, where

$$f_l(s) = u_l(s) - u_{min}(s) \tag{5.24}$$

and

$$u_{min}(s) = min_{k \in \{1,..,n\}}\, u_k(s) \tag{5.25}$$

(u_k is as defined in Equation 5.2).

[3]This is where the restriction mentioned in Section 5.4.3 comes in.

5.5 Results Analysis

- b (which determines the principal's action $(0, b)$) is the simulation's sole input parameter
- the output of the simulation is the population sequence (which contains all of the populations, from the initial population through to the final generation[4]) and their fitness values
- we focus only on the fittest worker strategy x from the final generation; we consider the point (b, x) to be the result from this run of the simulation
- for each b, 3 runs are made; we plot a graph using the average of the 3 x values
- the b input values were from the interval $[0, 0.8]$, starting at 0.01, step size 0.01

5.5.1 Results Analysis Under Fitness Version 1

We first ran the GGA against the simpler version of the contract game: the version with no social preferences ($v \equiv 0$). We expected the GGA to tell us that $x = \frac{b}{n}$ is the subgame perfect strategy for each b.

Under the fitness function ${}^1T_f$, a higher utility of a strategy should lead to a higher relative utility, so in theory one would expect all but the best-response efforts to eventually become extinct. In practice, however, this was not the case: for all values of b, the fittest worker effort from the final generation was equal or close to 0, instead of the expected value $x = \frac{b}{n}$ (and these fittest strategies were representative of the populations they were from - in every case, the populations had converged to 0). How can we explain

[4]The final generation is the last generation before the terminating condition is triggered.

such marked discrepancies between the theoretical expectations and practical results?

Let us first consider the case where every worker exerts the same effort, $x = \frac{b}{n}$; then the total output $y = b$ and by Equation 5.2, the utility for each worker is:

$$\begin{aligned} u_w &= w - \frac{x^2}{2} && (5.26)\\ &= \frac{by}{n} - \frac{x^2}{2} && (5.27)\\ &= \frac{b^2}{n} - \frac{b^2}{2n^2} && (5.28)\\ &= \frac{2b^2 - b^2}{2n^2} && (5.29)\end{aligned}$$

Now, say worker 1 chooses to exert $x_1 = 0$ instead, while all the other workers continue to exert $x = \frac{b}{n}$; then the total output becomes $y = \frac{(n-1)b}{n}$, and the utility for worker 1 is:

$$\begin{aligned} u_{w_1} &= w && (5.30)\\ &= \frac{by}{n} && (5.31)\\ &= \frac{b^2(n-1)}{n} && (5.32)\\ &= \frac{2b^2 - 2b^2}{2n^2} < \frac{2b^2 - b^2}{2n^2} = u_w && (5.33)\end{aligned}$$

So worker 1 earns a lower utility by playing $x = 0$ instead of $x = \frac{b}{n}$.

However, if we take into consideration the utilities for the other workers:

$$\begin{aligned} u_{w'} &= w - \frac{x^2}{2} && (5.34)\\ &= \frac{b^2(n-1)}{n} - \frac{b^2}{2n^2} < \frac{b^2(n-1)}{n} = u_{w_1} && (5.35)\end{aligned}$$

we suddenly see that after playing $x = 0$, worker 1 has the best fitness of all, as opposed to having the same fitness as everyone else after playing $x = \frac{b}{n}$.

Thus, under ${}^1T_f$, player 1 is able to exert a suboptimal (for himself) effort, but due to free-riding, come out on top evolutionarily. We need to rethink our choice of fitness function.

5.5.2 Fitness Version 2

With ${}^2T_f$, the second version of the fitness function, we attempt to remove the relativity aspect that doomed the first version to failure. Since we are trying to encourage convergence to the best response, the new fitness definition will reflect how good the strategy's chosen action is, compared to the best move it could have made in the same situation (that is, under the assumption that the opponents take the same actions). Under ${}^2T_f$, a strategy's fitness is the ratio of the payoff from the action it actually made, to the payoff from the best move that it could have made (and like for ${}^1T_f$, the fitness values are shifted so that they are non-negative).

The definition of the second fitness function is: ${}^2T_f : D^n \to (\mathbb{R}_0^+)^n$ with $s = (s_1, ..., s_n) \mapsto (h_1(s), ..., h_n(s))$, where

$$h_l(s) = g_l(s) - min_{c \in \{1,..,n\}}\, g_c(s) \tag{5.36}$$

$$g_l(s_l, s_{-l}) = \frac{f_l(s_l, s_{-l})}{arg\ max_{t \in D}\ f_l(t, s_{-l})} \tag{5.37}$$

(D is as in Equation 5.23), and, as before

$${}^1T_f(a) = (f_1(s), ..., f_n(s)) \tag{5.38}$$

5.5.3 Results Analysis Under Fitness Version 2

From the plot of the averaged results (Figure 5.2), distinct intervals of continuity and discontinuity can be seen. Since we expect multiple equilibria at certain intervals, averaging the results may not give us a clear picture of what is going on in the simulation. We try a new approach: we process

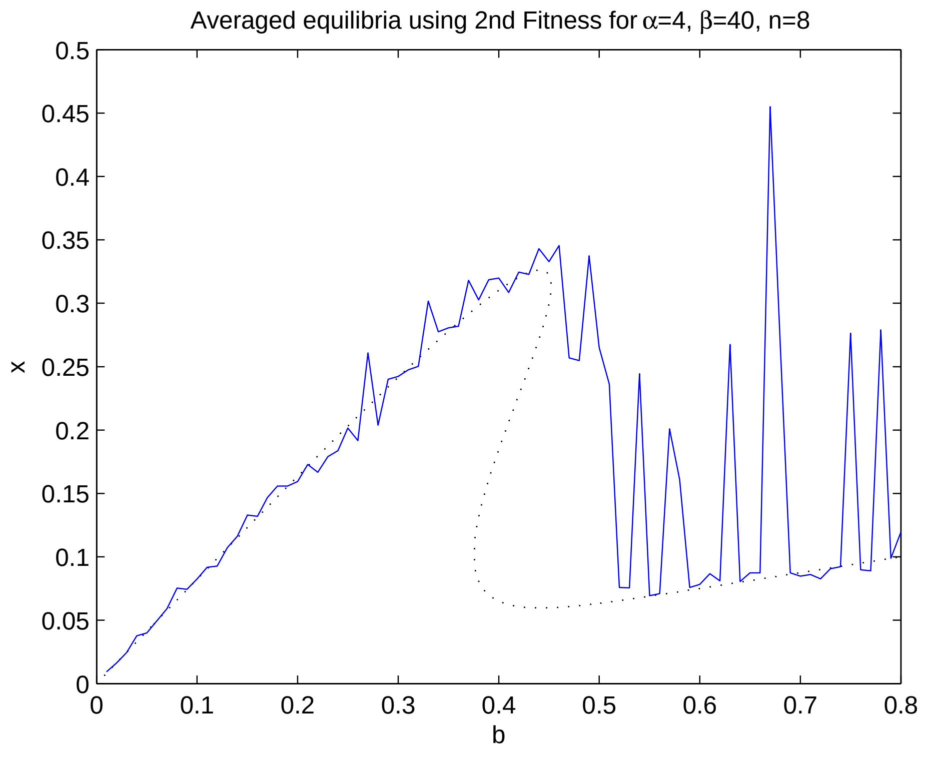


Figure 5.2: Averaged equilibria using ${}^2T_f$

the results, grouping seemingly continuous subsequences together, averaging them and separating anomalous results out. A plot of the processed results (Figure 5.3) reveals more about the behaviour of the simulation.

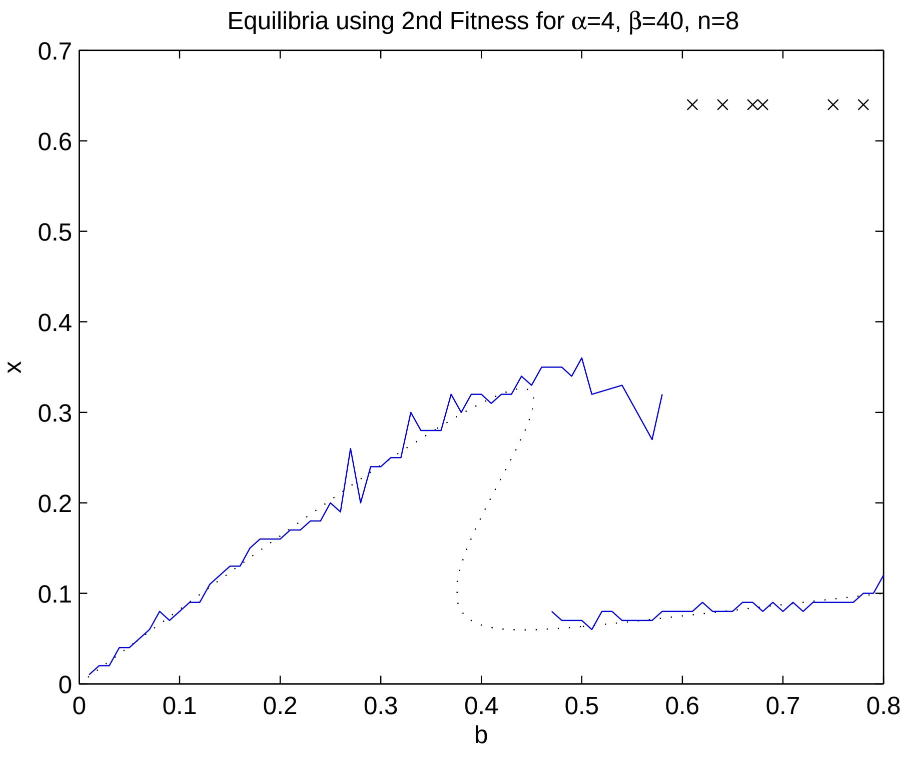


Figure 5.3: Equilibria using ${}^2T_f$

We can now see just what was causing the spikes on the interval $b = [0.6, 0.8]$ in Figure 5.2: there were 6 outliers (anomalous points), all at around $x = 0.64$. Their consistent reappearance upon reruns of the simulation meant that they cannot be discounted in the analysis, and need some investigation.

So, the root cause of the outliers is hidden somewhere within the functions and parameters used in the GGA. Upon close examination, one factor is an obvious candidate: the small population size. As a result of the restriction from Section 5.4.3, the population size is tied to the number of workers in the underlying contract game. So, if we wanted to analyse the game with (the number of workers) $k = 8$, we would be forced to use a population size of 8. It is a known genetic algorithms problem that a population of such size can lead to premature convergence[5]. And indeed, some exploratory simulation modifications seemed to suggest that this was the case: increasing the number of workers in the game (and with it, the population size) decreased the number of outliers. A new version of the fitness function, one that would somehow divorce population size from the worker number, is required.

5.5.4 Fitness Version 3

${}^3T_f$ extends ${}^2T_f$ by loosening the restriction from Section 5.4.3 - when using ${}^3T_f$, the number of strategies in the population is now allowed to exceed (as well as equal) the number of players in the contract game. In its fitness evaluation of a strategy, ${}^2T_f$ uses all of the strategies in the population (one for each worker in the contract game), because the population size is equal to the worker number. With ${}^3T_f$, this is not possible, as there could potentially be more strategies than worker places; hence ${}^3T_f$ uses only a subset of the population - this subset always contains the strategy being evaluated, the other strategies in the subset are picked randomly from the main population, and its size is exactly the number of workers in the game.

The definition of the third fitness function is: ${}^3T_f : D^n \to (\mathbb{R}_0^+)^n$ with $s = (s_1, ..., s_n) \mapsto (h_{1,1}(s), ..., h_{j,1}, ..., h_{n,1}(s))$, where

$$(h_{l,1}, ..., h_{l,n}) =^2 T_f(\sigma_k(\pi_l(s))) \tag{5.39}$$

[5]Premature convergence refers to the following situation: if an individual that is fitter than most of its competitors emerges early on in the course of the GA run, it may reproduce so abundantly that it drives down the population's diversity too soon, leading the algorithm to converge on the local optimum that that individual represents rather than searching the fitness landscape thoroughly enough to find the global optimum.

for all $l = 1, 2, ..., n$,

$$\pi_l(s) = (s_l, \pi'(s_{-l})) \tag{5.40}$$

($\pi'(s_{-l})$ is a uniformly distributed random permutation of s_{-l}, and $\pi_l(s)$ is a tuple with s_l being its first element and and $\pi'(s_{-l})$ making up the other $n-1$ elements), and

$$\sigma_k(t) = (t_1, t_2, ..., t_k) \tag{5.41}$$

if $t = (t_1, t_2, ..., t_n)$ and $n \geq k$.

5.5.5 Results Analysis Under Fitness Version 3

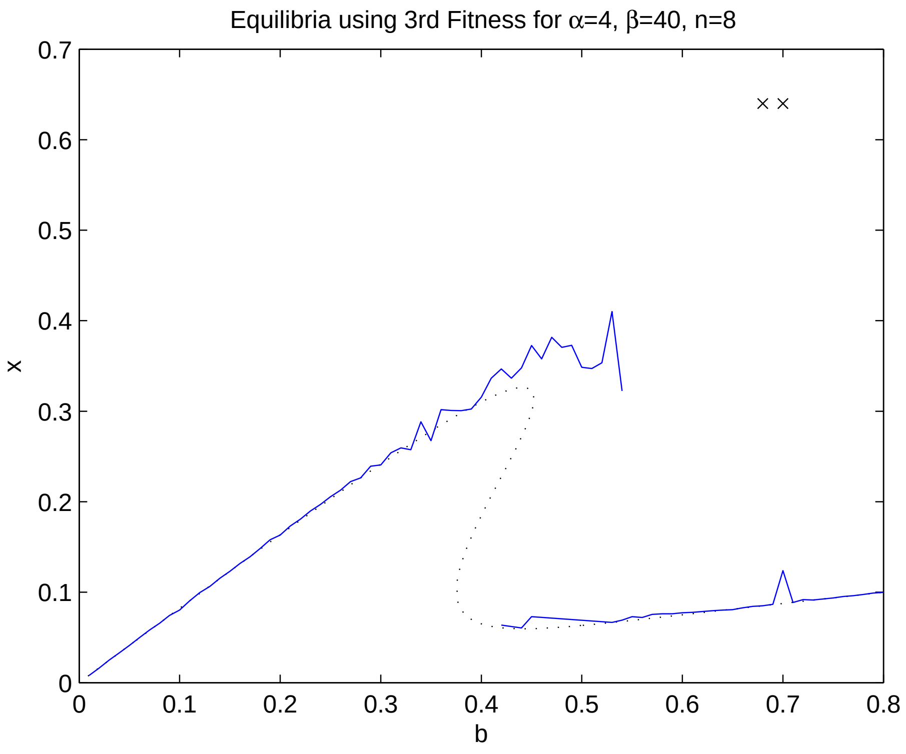


Figure 5.4: Equilibria using ${}^3T_f$

The results under ${}^3T_f$ (Figure 5.4) appear to confirm the suspicions that the outliers are related to population size, rather than the number of workers in the game; in the results from the simulation run with a larger population size (20 instead of 8) but using the old number of workers ($m = 8$), the outliers are almost completely eliminated (this was confirmed by reruns of the simulation).

For $b \in [0, 0.42]$ and for $b \in [0.54, 0.8]$, the simulation and theoretical results match closely. On the interval $[0.42, 0.54]$, the simulation produces two levels: a high effort and a low effort. Here we can see that the genetic algorithm has trouble discerning between optimal and slightly suboptimal equilibria - even when the higher effort is suboptimal, it is still an "attractor" and the simulation converges to it regularly. It is likely that a simulation running for longer (a larger number of generations) is more likely to converge to the optimum, rather than the suboptimum.

The theoretical middle equilibrium (that exists only between 0.376 and 0.452) is never attained; this is not unexpected as only the high- and low-effort equilibria are stable under adaptive dynamics; a small deviation (up, down) from the medium-effort equilibrium induces a movement (up, down) towards the high- or low-effort equilibrium level ([Huck et al., 2003]).

5.6 Remarks

- The reader may have noticed that none of the fitness functions discussed above were specifically engineered to achieve a symmetric equilibrium, but the populations invariably converged, with high probability, to a common value after a certain number of rounds. This property, a by-product of the underlying genetic algorithm, was a result of careful choices of crossover and mutation probabilities: mutation was chosen to be suitably low ($p = 0.02$) so as to prevent the population from fluctuating chaotically, and crossover probability was less than 1 ($p = 0.6$) so that the parent strategies had a good chance of being included in the next generation's population, (this meant that subsequent populations became increasingly homogenous).

- ${}^2T_f$ and ${}^3T_f$ make the GGA far more computationally expensive than ${}^1T_f$; ${}^1T_f$ requires the game utility calculation (which is a constant-time operation) to be carried out only once for every generation, and hence its execution time is $O(1)$. ${}^2T_f$ evaluates the best possible effort for a strategy in its current situation, and thus requires the game being run d times for every strategy, where d is the number of possible values the worker effort can take ($d = |D|$, where D is as in Equation 5.23). This is done for every one of k players in the contract game being played, thus making the cost $O(kd)$. ${}^3T_f$ does a similar search for the best effort in the situation as ${}^2T_f$, but it divorces the number of players in the contract game from the number of strategies in each population (which we shall denote by n) and so its cost is $O(nd)$ (of course, $n \geq k$ is a necessary condition).

5.7 Conclusion

Armed with only the minimal theoretical analysis of the problem, we have been able to predict, to a reasonably high degree, the theoretical solution for the Contract Game. We have shown that the GGA has its place as a "pre-emptive" problem analysis tool, (one that can give us some idea of what results we can expect to get from the analytic methods) and illustrated some of the pitfalls to be avoided when applying this technique.

Chapter 6

Conclusion

It is hoped that this thesis managed to successfully formalise the main concepts in the application of the genetic algorithm to games, and that adequate examples and illustrations have been provided to demonstrate how this technique can be a useful tool in game analysis.

The next step along this research path could involve looking for interrelationships between Markov chain theory (that has been so successfully applied to the analysis of genetic algorithms) and equilibrium convergence in games, as well as exploring other problems in which the genetic game algorithm could prove useful.

Appendix A

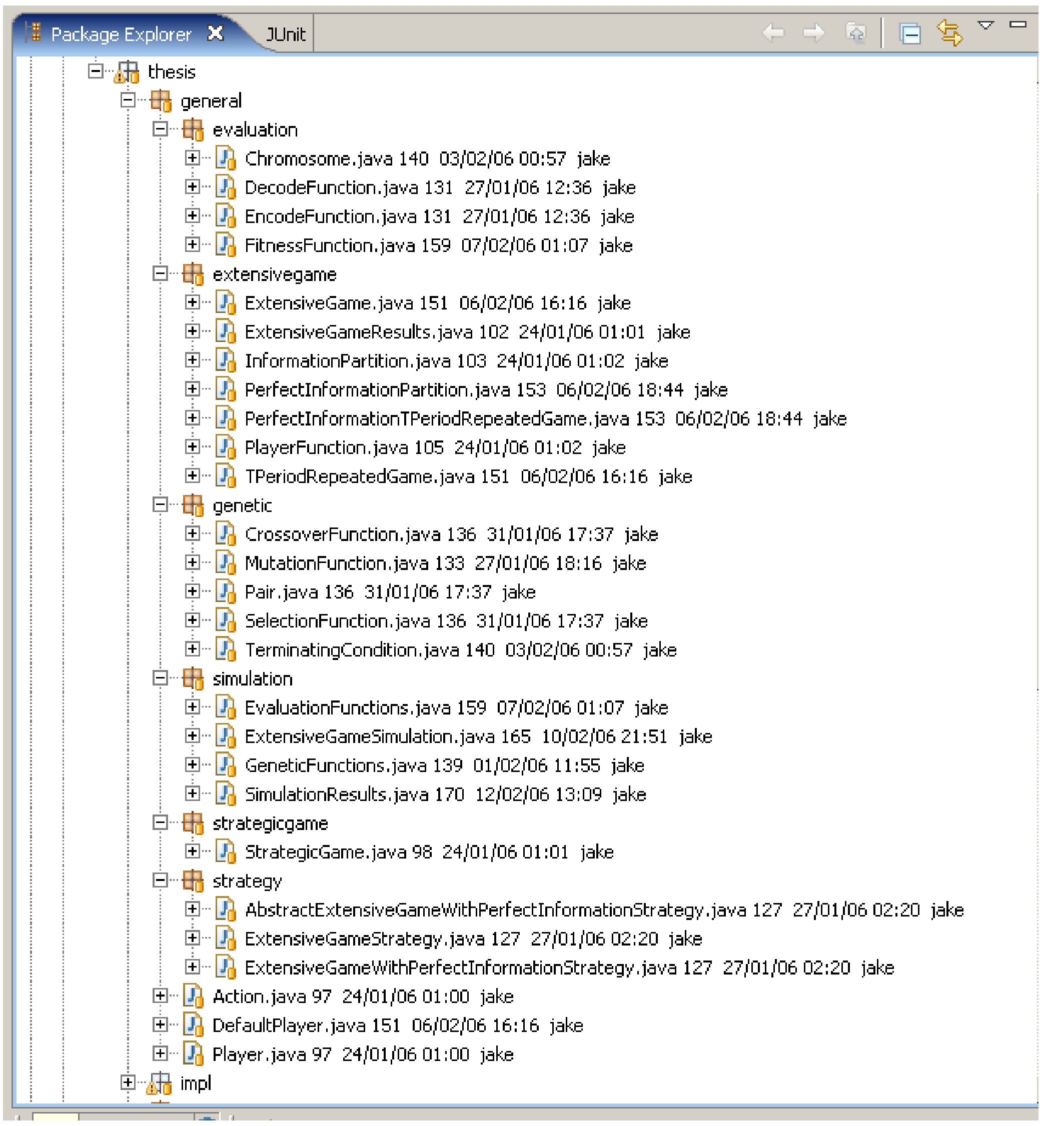
Code Listings

This appendix discusses the details of the GGA simulation (Section A.1), and the Matlab code used in the Contract Game chapter (Section A.2).

A.1 The GGA Simulation

The simulation for the Axelrod and the Contract Game experiments is written in Java and contains more than 4800 lines of code. It does not require extensive explanation here, as the aim throughout was to try to imitate the theoretic function definitions as closely as possible, and this was largely possible.

A.1.1 Common Code



```
thesis
  general
    evaluation
      Chromosome.java 140  03/02/06 00:57  jake
      DecodeFunction.java 131  27/01/06 12:36  jake
      EncodeFunction.java 131  27/01/06 12:36  jake
      FitnessFunction.java 159  07/02/06 01:07  jake
    extensivegame
      ExtensiveGame.java 151  06/02/06 16:16  jake
      ExtensiveGameResults.java 102  24/01/06 01:01  jake
      InformationPartition.java 103  24/01/06 01:02  jake
      PerfectInformationPartition.java 153  06/02/06 18:44  jake
      PerfectInformationTPeriodRepeatedGame.java 153  06/02/06 18:44  jake
      PlayerFunction.java 105  24/01/06 01:02  jake
      TPeriodRepeatedGame.java 151  06/02/06 16:16  jake
    genetic
      CrossoverFunction.java 136  31/01/06 17:37  jake
      MutationFunction.java 133  27/01/06 18:16  jake
      Pair.java 136  31/01/06 17:37  jake
      SelectionFunction.java 136  31/01/06 17:37  jake
      TerminatingCondition.java 140  03/02/06 00:57  jake
    simulation
      EvaluationFunctions.java 159  07/02/06 01:07  jake
      ExtensiveGameSimulation.java 165  10/02/06 21:51  jake
      GeneticFunctions.java 139  01/02/06 11:55  jake
      SimulationResults.java 170  12/02/06 13:09  jake
    strategicgame
      StrategicGame.java 98  24/01/06 01:01  jake
    strategy
      AbstractExtensiveGameWithPerfectInformationStrategy.java 127  27/01/06 02:20  jake
      ExtensiveGameStrategy.java 127  27/01/06 02:20  jake
      ExtensiveGameWithPerfectInformationStrategy.java 127  27/01/06 02:20  jake
    Action.java 97  24/01/06 01:00  jake
    DefaultPlayer.java 151  06/02/06 16:16  jake
    Player.java 97  24/01/06 01:00  jake
  impl
```

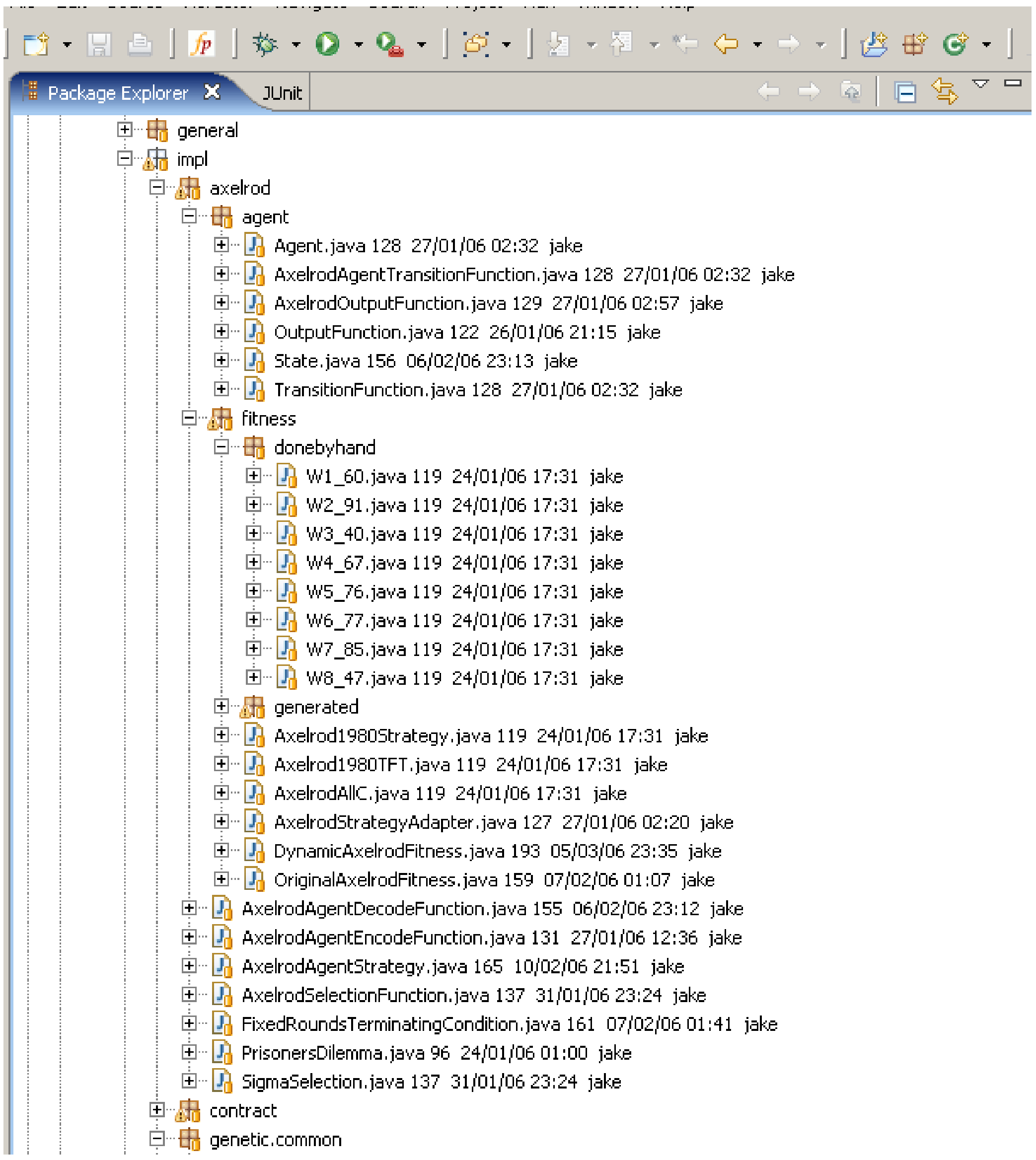
Figure A.1: Common Classes and Interfaces

```
        public void run() {
                while(!termCond.isSatisfied()) {
                        roundCount++;
                        //System.out.println("Round "+roundCount);
                        List<Chromosome> newPopulation = new LinkedList<
                            Chromosome>();
                        List<Chromosome> lastPopulation = populationSequence
                            .get(roundCount-1);
                        List<Double> fitness = populationFitnessSequence.get
                            (roundCount - 1);
                        for (int i = 0; i < populationSize/2; i++) {
                                Pair<Chromosome> pair;
                                pair = geneticFns.selectionFns[i].select(
                                    lastPopulation, fitness);
                                pair = crossoverPair(pair);
                                pair = mutatePair(pair);

                                newPopulation.add(pair.child1);
                                newPopulation.add(pair.child2);
                        }

                        addAndEvaluatePopulation(newPopulation);
                }

                results = new SimulationResults<T>(populationSequence,
                    populationFitnessSequence, evalFns);
        }
```

Listing A.1: The Main Loop of the Simulation

A.1.2 Axelrod Simulation

```
/*
 * Created on 18-Dec-2005
 */
package thesis.impl.axelrod;

import java.util.HashSet;
import java.util.Hashtable;
import java.util.Map;
import java.util.Set;
```



```
general
impl
  axelrod
    agent
      Agent.java 128  27/01/06 02:32  jake
      AxelrodAgentTransitionFunction.java 128  27/01/06 02:32  jake
      AxelrodOutputFunction.java 129  27/01/06 02:57  jake
      OutputFunction.java 122  26/01/06 21:15  jake
      State.java 156  06/02/06 23:13  jake
      TransitionFunction.java 128  27/01/06 02:32  jake
    fitness
      donebyhand
        W1_60.java 119  24/01/06 17:31  jake
        W2_91.java 119  24/01/06 17:31  jake
        W3_40.java 119  24/01/06 17:31  jake
        W4_67.java 119  24/01/06 17:31  jake
        W5_76.java 119  24/01/06 17:31  jake
        W6_77.java 119  24/01/06 17:31  jake
        W7_85.java 119  24/01/06 17:31  jake
        W8_47.java 119  24/01/06 17:31  jake
      generated
      Axelrod1980Strategy.java 119  24/01/06 17:31  jake
      Axelrod1980TFT.java 119  24/01/06 17:31  jake
      AxelrodAllC.java 119  24/01/06 17:31  jake
      AxelrodStrategyAdapter.java 127  27/01/06 02:20  jake
      DynamicAxelrodFitness.java 193  05/03/06 23:35  jake
      OriginalAxelrodFitness.java 159  07/02/06 01:07  jake
    AxelrodAgentDecodeFunction.java 155  06/02/06 23:12  jake
    AxelrodAgentEncodeFunction.java 131  27/01/06 12:36  jake
    AxelrodAgentStrategy.java 165  10/02/06 21:51  jake
    AxelrodSelectionFunction.java 137  31/01/06 23:24  jake
    FixedRoundsTerminatingCondition.java 161  07/02/06 01:41  jake
    PrisonersDilemma.java 96  24/01/06 01:00  jake
    SigmaSelection.java 137  31/01/06 23:24  jake
  contract
  genetic.common
```

Figure A.2: The Axelrod Classes

```
import thesis.general.Action;
import thesis.general.Player;
import thesis.general.strategicgame.StrategicGame;

public class PrisonersDilemma implements StrategicGame {
        Set<Player> players;
        Set<Action> actions;

        public static final Player p1 = new Player() {
                public String getName() {
                        return "P1";
                }
                public String toString() { return getName(); }
        };

        public static final Player p2 = new Player() {
                public String getName() {
                        return "P2";
                }
                public String toString() { return getName(); }
        };

        public static final Action cooperate = new Action() {
                public String getName() {
                        return "Cooperate";
                }
                public String toString() { return getName(); }
        };

        public static final Action defect = new Action() {
                public String getName() {
                        return "Defect";
                }
                public String toString() { return getName(); }
        };

        public PrisonersDilemma() {
                players = new HashSet<Player>();
                players.add(p1);
                players.add(p2);

                actions = new HashSet<Action>();
                actions.add(cooperate);
                actions.add(defect);
        }

        public Set<Player> getPlayers() {
                return players;
        }

        public Set<Action> getActions(Player p) {
                return actions;
        }

        public Map<Player, Double> getUtility(Map<Player, Action>
            actionProfile) {
                Action p1Action = actionProfile.get(p1);
                Action p2Action = actionProfile.get(p2);
                Hashtable<Player, Double> payoffs = new Hashtable<Player,
                    Double>();

                if (p1Action == cooperate && p2Action == cooperate) {
                        payoffs.put(p1, 3.0);
                        payoffs.put(p2, 3.0);
                }
                else if (p1Action == cooperate && p2Action == defect) {
                        payoffs.put(p1, 0.0);
                        payoffs.put(p2, 5.0);
                }
                else if (p1Action == defect && p2Action == cooperate) {
                        payoffs.put(p1, 5.0);
                        payoffs.put(p2, 0.0);
                }
                else {
                        // both defect
                        payoffs.put(p1, 1.0);
                        payoffs.put(p2, 1.0);
                }
                return payoffs;
        }

        public String toString() {
                return "Prisoner's Dilemma";
        }
}
```

Listing A.2: The Prisoner's Dilemma Game

```
/*
 * Created on 20-Dec-2005
 */
package thesis.impl.axelrod.fitness;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import thesis.general.Player;
import thesis.general.evaluation.FitnessFunction;
import thesis.general.extensivegame.ExtensiveGameResults;
import thesis.general.extensivegame.PerfectInformationTPeriodRepeatedGame;
import thesis.general.extensivegame.TPeriodRepeatedGame;
import thesis.general.strategicgame.StrategicGame;
import thesis.general.strategy.ExtensiveGameStrategy;
import thesis.impl.axelrod.AxelrodAgentStrategy;
import thesis.impl.axelrod.PrisonersDilemma;
import thesis.impl.axelrod.fitness.donebyhand.W1_60;
import thesis.impl.axelrod.fitness.donebyhand.W2_91;
import thesis.impl.axelrod.fitness.donebyhand.W3_40;
import thesis.impl.axelrod.fitness.donebyhand.W4_67;
import thesis.impl.axelrod.fitness.donebyhand.W5_76;
import thesis.impl.axelrod.fitness.donebyhand.W6_77;
import thesis.impl.axelrod.fitness.donebyhand.W7_85;
import thesis.impl.axelrod.fitness.donebyhand.W8_47;

public class OriginalAxelrodFitness implements FitnessFunction<
    AxelrodAgentStrategy> {
        private ExtensiveGameStrategy[] axStrategies;
        private TPeriodRepeatedGame game;
        private int stages;
        double[] w ;

        public OriginalAxelrodFitness() {
                stages = 151;
                StrategicGame constituentGame = new PrisonersDilemma();
                game = new PerfectInformationTPeriodRepeatedGame(
                    constituentGame, stages);

                axStrategies = new ExtensiveGameStrategy[9];
                axStrategies[1] = new AxelrodStrategyAdapter(game, new W1_60
                    ());
```

```
            axStrategies[2] = new AxelrodStrategyAdapter(game, new W2_91
                ());
            axStrategies[3] = new AxelrodStrategyAdapter(game, new W3_40
                ());
            axStrategies[4] = new AxelrodStrategyAdapter(game, new W4_67
                ());
            axStrategies[5] = new AxelrodStrategyAdapter(game, new W5_76
                ());
            axStrategies[6] = new AxelrodStrategyAdapter(game, new W6_77
                ());
            axStrategies[7] = new AxelrodStrategyAdapter(game, new W7_85
                ());
            axStrategies[8] = new AxelrodStrategyAdapter(game, new W8_47
                ());
    }

    public double evaluate(ExtensiveGameStrategy strategy) {
            w = new double[9];
            Map<Player, ExtensiveGameStrategy> strategyProfile = new
                HashMap<Player, ExtensiveGameStrategy>();
            // set the given strategy to player 1
            strategyProfile.put(PrisonersDilemma.p1, strategy);
            for (int i = 1; i <= 8; i++) {
                    strategyProfile.put(PrisonersDilemma.p2,
                        axStrategies[i]);
                    ExtensiveGameResults results = game.runGame(
                        strategyProfile);
                    // extract how well the given strategy fared against
                         our set axelrod strategies
                    w[i] = results.getUtility().get(PrisonersDilemma.p1)
                        ;
                    // don't forget to remove the normalisation
                    w[i] = w[i] * stages;
            }
            return 110.55 + (0.1574) * w[2] + (0.1506) * w[1] + (0.1185)
                 * w[3]
                    + (0.0876) * w[4] + (0.0579) * w[6] + (0.0492) * w
                        [7]
            + (0.0487) * w[5] + (0.0463) * w[8];

    }

    public double[] getResults() {
            return w;
```

```
        }

        public double evaluate(AxelrodAgentStrategy strategy, List<
            AxelrodAgentStrategy> population) {
                return evaluate(strategy);
        }

}
```

Listing A.3: Axelrod Static Fitness (Definition 24)

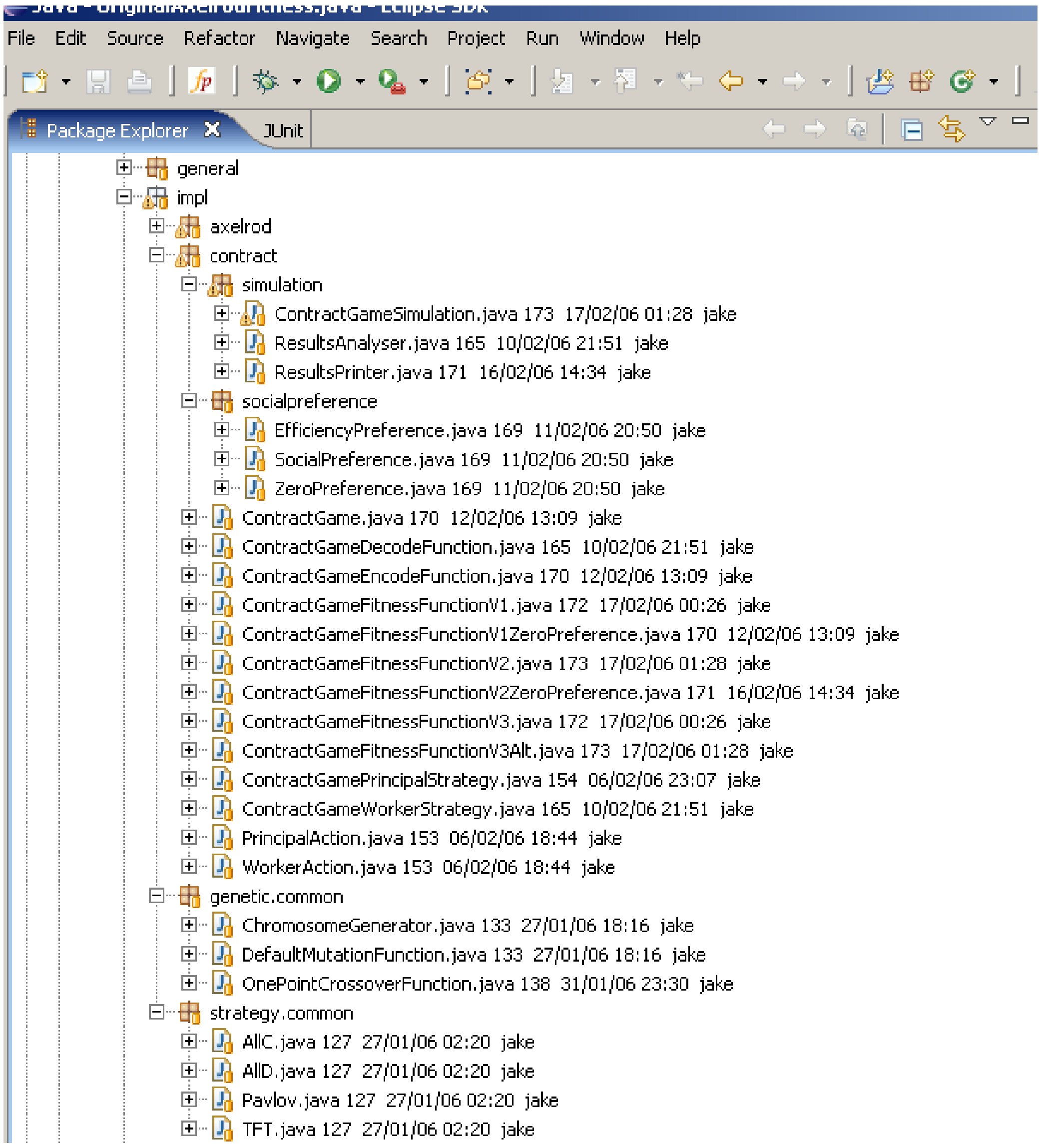
```
/*
 * Created on 20-Dec-2005
 */
package thesis.impl.axelrod.fitness;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import thesis.general.Player;
import thesis.general.evaluation.FitnessFunction;
import thesis.general.extensivegame.ExtensiveGameResults;
import thesis.general.extensivegame.PerfectInformationTPeriodRepeatedGame;
import thesis.general.extensivegame.TPeriodRepeatedGame;
import thesis.general.strategicgame.StrategicGame;
import thesis.general.strategy.ExtensiveGameStrategy;
import thesis.impl.axelrod.AxelrodAgentStrategy;
import thesis.impl.axelrod.PrisonersDilemma;

public class DynamicAxelrodFitness implements FitnessFunction<
    AxelrodAgentStrategy> {
        private TPeriodRepeatedGame game;
        private int stages;

        public DynamicAxelrodFitness() {
                stages = 151;
                StrategicGame constituentGame = new PrisonersDilemma();
                game = new PerfectInformationTPeriodRepeatedGame(
                    constituentGame, stages);
        }

        public double evaluate(ExtensiveGameStrategy
            strategyWeWantResultsFor, ExtensiveGameStrategy otherStrategy) {
```

```
            Map<Player, ExtensiveGameStrategy> strategyProfile = new
                HashMap<Player, ExtensiveGameStrategy>();
            // set the given strategy to player 1
            strategyProfile.put(PrisonersDilemma.p1,
                strategyWeWantResultsFor);
            strategyProfile.put(PrisonersDilemma.p2, otherStrategy);
            ExtensiveGameResults results = game.runGame(strategyProfile)
                ;
            // extract how well the given strategy fared against our set
                 axelrod strategies
            return results.getUtility().get(PrisonersDilemma.p1);
    }


    public double evaluate(AxelrodAgentStrategy strategy, List<
        AxelrodAgentStrategy> population) {
            double sum = 0;
            for (AxelrodAgentStrategy popnMember : population) {
                    sum = sum + evaluate(strategy, popnMember);
            }
            return sum/(double) population.size();
    }

}
```

Listing A.4: Axelrod Dynamic Fitness (Definition 25)

A.1.3 Contract Game Simulation

```
/*
 * Created on 06-Feb-2006
 */
package thesis.impl.contract;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
```


File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit

```
general
impl
  axelrod
  contract
    simulation
      ContractGameSimulation.java 173  17/02/06 01:28  jake
      ResultsAnalyser.java 165  10/02/06 21:51  jake
      ResultsPrinter.java 171  16/02/06 14:34  jake
    socialpreference
      EfficiencyPreference.java 169  11/02/06 20:50  jake
      SocialPreference.java 169  11/02/06 20:50  jake
      ZeroPreference.java 169  11/02/06 20:50  jake
    ContractGame.java 170  12/02/06 13:09  jake
    ContractGameDecodeFunction.java 165  10/02/06 21:51  jake
    ContractGameEncodeFunction.java 170  12/02/06 13:09  jake
    ContractGameFitnessFunctionV1.java 172  17/02/06 00:26  jake
    ContractGameFitnessFunctionV1ZeroPreference.java 170  12/02/06 13:09  jake
    ContractGameFitnessFunctionV2.java 173  17/02/06 01:28  jake
    ContractGameFitnessFunctionV2ZeroPreference.java 171  16/02/06 14:34  jake
    ContractGameFitnessFunctionV3.java 172  17/02/06 00:26  jake
    ContractGameFitnessFunctionV3Alt.java 173  17/02/06 01:28  jake
    ContractGamePrincipalStrategy.java 154  06/02/06 23:07  jake
    ContractGameWorkerStrategy.java 165  10/02/06 21:51  jake
    PrincipalAction.java 153  06/02/06 18:44  jake
    WorkerAction.java 153  06/02/06 18:44  jake
  genetic.common
    ChromosomeGenerator.java 133  27/01/06 18:16  jake
    DefaultMutationFunction.java 133  27/01/06 18:16  jake
    OnePointCrossoverFunction.java 138  31/01/06 23:30  jake
  strategy.common
    AllC.java 127  27/01/06 02:20  jake
    AllD.java 127  27/01/06 02:20  jake
    Pavlov.java 127  27/01/06 02:20  jake
    TFT.java 127  27/01/06 02:20  jake
```

Figure A.3: The Contract Game Classes

```
import thesis.general.Action;
import thesis.general.DefaultPlayer;
import thesis.general.Player;
import thesis.general.extensivegame.ExtensiveGame;
import thesis.general.extensivegame.ExtensiveGameResults;
import thesis.general.extensivegame.InformationPartition;
import thesis.general.extensivegame.PerfectInformationPartition;
import thesis.general.extensivegame.PlayerFunction;
import thesis.general.strategy.ExtensiveGameWithPerfectInformationStrategy;
import thesis.impl.contract.socialpreference.SocialPreference;

public class ContractGame implements ExtensiveGame<
    ExtensiveGameWithPerfectInformationStrategy> {
        public static Player principal = new Player() {
                public String getName() { return "principal"; }
        };

        public Player workers[];

        private static PlayerFunction playerFunction = new PlayerFunction()
            {
                public Set<Player> getPlayer(List<Map<Player, Action>>
                    history) {
                        if (history.isEmpty()) {
                                return Collections.singleton(principal);
                        }
                        else {
                                return workerSet;
                        }
                }
        };

        private static Set<Player> workerSet;
        private static Set<Player> playerSet;
        private InformationPartition infoPartition = new
            PerfectInformationPartition();

        private int numberOfWorkers;

        private SocialPreference socialPreference;

        @SuppressWarnings("unchecked")
    public ContractGame(int numberOfWorkers, SocialPreference
        socialPreference) throws IllegalArgumentException {
            if (numberOfWorkers <= 1) {
                    throw new IllegalArgumentException("Number of
                        workers must be > 1");
            }
            else {
                    this.numberOfWorkers = numberOfWorkers;
            }

            workers = new Player[numberOfWorkers];
            for (int i = 0; i < numberOfWorkers; i++) {
                    workers[i] = new DefaultPlayer("worker "+ (i+1));
            }

            this.socialPreference = socialPreference;

            playerSet = new HashSet(Arrays.asList(workers));
            playerSet.add(principal);
    }

    public Set<Player> getPlayers() {
            return playerSet;
    }

    public InformationPartition getInformationPartition(Player p) {
            return infoPartition;
    }

    public PlayerFunction getPlayerFunction() {
            return playerFunction;
    }

    public ExtensiveGameResults runGame(
                    Map<Player,
                        ExtensiveGameWithPerfectInformationStrategy>
                        strategyProfile) {
            // set up an initial (empty) history
            List<Map<Player, Action>> history = new LinkedList<Map<
                Player, Action>>();

            // init the action profile
            Map<Player, Action> actionProfile = new HashMap<Player,
                Action>();
        // first the principal makes a move
        actionProfile.put(principal, strategyProfile.get(principal).
            getMove(history, principal));

        history.add(actionProfile);
        actionProfile = new HashMap<Player, Action>();

        // calculate the action profile for each player and for the
            current history
        for (Player p : workers) {
                // work out the player's information partition
                InformationPartition infoPartition =
                    getInformationPartition(p);
                // work out the information set from the history
                Set<List<Map<Player, Action>>> infoSet =
                    infoPartition.getInformationSet(history);
                ExtensiveGameWithPerfectInformationStrategy strategy
                     = strategyProfile.get(p);
                Action playerAction = strategy.getMove(infoSet, p);
                actionProfile.put(p, playerAction);
        }
        // we need to update the history
        history.add(actionProfile);

        return new ExtensiveGameResults(history, getUtility(history)
            );
}

public Map<Player, Double> getUtility(List<Map<Player, Action>>
    history) {
        PrincipalAction principalsAction = (PrincipalAction) history
            .get(0).get(principal);
        Map<Player, Action> workersActions = history.get(1);

        Map<Player, Double> utility = new HashMap<Player, Double>();

        double a = principalsAction.getBaseWage();
        double b = principalsAction.getBonus();

        Map<Player, Double> workersEfforts = new HashMap<Player,
            Double>();
        for (Player worker : workersActions.keySet()) {
                double x = ((WorkerAction) workersActions.get(worker
                    )).getValue();
                workersEfforts.put(worker, x);

                double average = averageOfOthers(worker,
                    workersActions);
                utility.put(worker, calculateWorkerUtility(x,
                    average, a, b));
        }

        // calculate the utility for the principal
        utility.put(principal, calculatePrincipalUtility(a,b,
            workersEfforts));

        return utility;
}

public static double averageOfOthers(Player worker, Map<Player,
    Action> workersActions) {
        double sum = 0;
        for (Player otherWorker : workersActions.keySet()) {
                if (!(worker == otherWorker)) {
                        sum = sum + ((WorkerAction) workersActions.
                            get(otherWorker)).getValue();
                }
        }
        double average = sum / (double) (workersActions.size() - 1);
        return average;
}

public double calculatePrincipalUtility(double a, double b, Map<
    Player, Double> workersEfforts) {
        double y = 0;
        int n = workersEfforts.size();

        for (Player worker : workersEfforts.keySet()) {
                y = y + workersEfforts.get(worker);
        }
        double principalsUtility = y - n * (a + b * y / (double) n);
        return principalsUtility;
}

public double calculateWorkerUtility(double x, double xhat, double a
    , double b) {
        double y = xhat * (numberOfWorkers - 1) + x;
        double wage = a + (b * y / (double) numberOfWorkers);
```

```
                return wage - 0.5 * x * x - socialPreference.evaluate(x,
                    xhat, b);
        }
}
```

Listing A.5: The Contract Game (Definition 28)

```
/*
 * Created on 07-Feb-2006
 */
package thesis.impl.contract;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import thesis.general.Player;
import thesis.general.evaluation.FitnessFunction;
import thesis.general.extensivegame.ExtensiveGameResults;
import thesis.general.strategy.ExtensiveGameWithPerfectInformationStrategy;
import thesis.impl.contract.socialpreference.EfficiencyPreference;
import thesis.impl.contract.socialpreference.SocialPreference;

public class ContractGameFitnessFunctionV1 implements FitnessFunction<
    ContractGameWorkerStrategy> {
        protected ContractGame game;
        private ContractGamePrincipalStrategy principal;

        /**
         * This fitness function uses the efficiency preference with
             parameters alpha, beta
         * @param numberOfWorkers
         * @param alpha
         * @param beta
         * @param a
         * @param b
         */
        public ContractGameFitnessFunctionV1(int numberOfWorkers, double
            alpha, double beta, double a, double b) {
                this(numberOfWorkers, a, b, new EfficiencyPreference(alpha,
                    beta));
        }
    protected ContractGameFitnessFunctionV1(int numberOfWorkers, double
        a, double b, SocialPreference p) {
            game = new ContractGame(numberOfWorkers, p);
            principal = new ContractGamePrincipalStrategy(a,b);
    }

    public double evaluate(ContractGameWorkerStrategy strategy, List<
        ContractGameWorkerStrategy> population) {
            ExtensiveGameResults results = playGame(population);
            return getStrategyUtility(strategy, population, results) -
                getMinimum(results);
    }

    protected double getStrategyUtility(ContractGameWorkerStrategy
        strategy, List<ContractGameWorkerStrategy> population,
        ExtensiveGameResults results) {
            int indexOfStrategyBeingEvaluated = population.indexOf(
                strategy);
            Player playerOfStrategyBeingEvaluated = game.workers[
                indexOfStrategyBeingEvaluated];

            return results.getUtility().get(
                playerOfStrategyBeingEvaluated);
    }

    protected ExtensiveGameResults playGame(List<
        ContractGameWorkerStrategy> population) {
            Map<Player, ExtensiveGameWithPerfectInformationStrategy>
                strategyProfile = new HashMap<Player,
                ExtensiveGameWithPerfectInformationStrategy>();
            strategyProfile.put(ContractGame.principal, principal);

            for (int i = 0; i < population.size(); i++) {
                    strategyProfile.put(game.workers[i], population.get(
                        i));
            }
            return game.runGame(strategyProfile);
    }

    private double getMinimum(ExtensiveGameResults results) {
            double minimum = Double.MAX_VALUE;
            for (Player p : results.getUtility().keySet()) {
                    minimum = Math.min(minimum, results.getUtility().get
                        (p));
                }
                return minimum;
        }
}
```

Listing A.6: ${}^1T_f$ (Equation 5.24)

```
/*
 * Created on 07-Feb-2006
 */
package thesis.impl.contract;

import java.util.List;

import thesis.general.extensivegame.ExtensiveGameResults;
import thesis.impl.contract.socialpreference.EfficiencyPreference;
import thesis.impl.contract.socialpreference.SocialPreference;

public class ContractGameFitnessFunctionV2 extends
    ContractGameFitnessFunctionV1 {
        protected double minValue, maxValue, step;

        /**
         * This fitness function uses the efficiency preference with
             parameters alpha, beta
         * @param numberOfWorkers
         * @param alpha
         * @param beta
         * @param a
         * @param b
         */
        public ContractGameFitnessFunctionV2(int numberOfWorkers, double
            alpha, double beta, double a, double b, double minValue, double
            maxValue, double step) {
                this(numberOfWorkers, a, b, new EfficiencyPreference(alpha,
                    beta), minValue, maxValue, step);
                this.minValue = minValue;
                this.maxValue = maxValue;
                this.step = step;
        }

        public ContractGameFitnessFunctionV2(int numberOfWorkers, double a,
            double b, SocialPreference p, double minValue, double maxValue,
            double step) {
                super(numberOfWorkers, a, b, p);
                this.minValue = minValue;
                this.maxValue = maxValue;
                this.step = step;
        }

        public double evaluate(ContractGameWorkerStrategy strategy, List<
            ContractGameWorkerStrategy> population) {
                ExtensiveGameResults results;

                results = playGame(population);
                double actualUtility = getStrategyUtility(strategy,
                    population, results);

                double maximumUtility = Double.MIN_VALUE;
                int index = population.indexOf(strategy);
                for (int i = 0; i * step + minValue < maxValue; i++) {
                        ContractGameWorkerStrategy alternativeStrategy = new
                             ContractGameWorkerStrategy(minValue + i*step,
                            null);

                        population.set(index, alternativeStrategy);
                        results = playGame(population);
                        double newUtility = getStrategyUtility(
                            alternativeStrategy, population, results);
                        population.set(index, strategy);

                        maximumUtility = Math.max(maximumUtility, newUtility
                            );
                }

                //return Math.max(actualUtility / maximumUtility, 0);
                return actualUtility / maximumUtility;
        }
}
```

Listing A.7: ${}^{2}T_f$ (Equation 5.36)

```
/*
 * Created on 07-Feb-2006
 */
package thesis.impl.contract;

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

import thesis.impl.contract.socialpreference.SocialPreference;

public class ContractGameFitnessFunctionV3Alt extends
    ContractGameFitnessFunctionV2 {
        public ContractGameFitnessFunctionV3Alt(int numberOfWorkers, double
            a, double b, SocialPreference p, double minValue, double
            maxValue, double step) {
                super(numberOfWorkers, a, b, p, minValue, maxValue, step);
        }

        public ContractGameFitnessFunctionV3Alt(int numberOfWorkers, double
            alpha, double beta, double a, double b, double minValue, double
            maxValue, double step) {
                super(numberOfWorkers, alpha, beta, a, b, minValue, maxValue
                    , step);
        }

        public double evaluate(ContractGameWorkerStrategy strategy, List<
            ContractGameWorkerStrategy> population) {
                List<ContractGameWorkerStrategy> subPopulation =
                    pickSubpopulationContainingStrategy(strategy, population
                    );
                return super.evaluate(strategy, subPopulation);
        }

        protected List<ContractGameWorkerStrategy>
            pickSubpopulationContainingStrategy(ContractGameWorkerStrategy
            strategy,
                        List<ContractGameWorkerStrategy> population) {
                List<ContractGameWorkerStrategy> clonedPopulation = new
                    LinkedList<ContractGameWorkerStrategy>(population);
                List<ContractGameWorkerStrategy>
                    strategiesThatPlayedInTheLastGame = new LinkedList<
                    ContractGameWorkerStrategy>();

                Collections.shuffle(clonedPopulation);

                strategiesThatPlayedInTheLastGame.add(strategy);
                int counter = 0;
                while (strategiesThatPlayedInTheLastGame.size() != game.
                    workers.length) {
                        ContractGameWorkerStrategy s = clonedPopulation.get(
                            counter);
                        if (s != strategy) {
                                strategiesThatPlayedInTheLastGame.add(s);
                        }
                        counter++;
                }

                return strategiesThatPlayedInTheLastGame;
        }
}
```

Listing A.8: ${}^{3}T_f$ (Equation 5.39)

A.2 Matlab Programs

```
function [p_0] = p(b,x,alpha,beta)

if(x >= b)
    p_0 = alpha;
elseif(x < b)
    p_0 = alpha * exp(-beta * (b-x)^2);
end
```

Listing A.9: The peer-pressure f'n p(b,x)

```
function [p_0, p_1, p_2, F, F1st, F2nd] = get_fns(b,n,alpha,beta)
p_0  = @(x)p(b,x,alpha,beta);
p_1  = @(x)-2 * beta * (b - x) * p_0(x);
p_2  = @(x)(-2 * beta + 4 * (beta.^2) * (b - x).^2) * p_0(x);

F    = @(x) (b/n)*(1 + n * p_0(x)) / ( 1 + p_0(x));
F1st = @(x)(b * (1/n - 1) * p_1(x)) / ((1 + p_0(x)).^2);
F2nd = @(x) b * (1/n - 1) * (2 * p_1(x).^2 - p_2(x) * (1 + p_0(x))) / ((1 +
    p_0(x)).^3);
```

Listing A.10: The f'n that declares F, p and their 1st and 2nd derivatives

```
function [x1, x2, x3] = zeros_fn(b,n,alpha,beta)
```

```
x1 = 0; x2 = 0; x3 = 0;
[p_0, p_1, p_2, F, F1st, F2nd] = get_fns(b,n,alpha,beta);

g = @(x)F(x) - x;
g1st = @(x)F1st(x) - 1;
g2nd = F2nd;

% first we determine whether g" has 0 or 1 roots
z1 = fzero(g2nd, b/2);
if( z1 ~= NaN && z1 > (b/n) && z1 < b)
    % g" has 1 root at z1
    num_g2nd_roots = 1;
else
    % g" has 0 roots
    num_g2nd_roots = 0;
end

% now we determine whether g' has 0, 1 or 2 roots
if (num_g2nd_roots == 1)
    % g' has 0 or 2 roots
    if (sign(g1st(z1)) == sign(g1st(b)))
        num_g1st_roots = 0;
    else
        num_g1st_roots = 2;
        y1 = fzero(g1st, [(b/n) z1]);
        y2 = fzero(g1st, [z1    b ]);
    end
else
    % g' has 0 or 1 roots
    if (sign(g1st(z1)) == sign(g1st(b)))
        num_g1st_roots = 0;
    else
        num_g1st_roots = 1;
        y1 = fzero(g1st, [(b/n)  b]);
    end
end

% now we determine whether g has 1, 2 or 3 roots
if (num_g1st_roots == 0)
    % g has only 1 root
    x1 = fzero(g, [(b/n)  b]);
elseif (num_g1st_roots == 1)
    % g has 2 roots
    x1 = fzero(g, [(b/n) y1]);
    x2 = fzero(g, [y1      b]);
else
    % g' has 2 roots, so g has 1 or 3 roots
    if (sign(g(b/n)) == sign(g(y1)))
        % only 1 root
        x1 = fzero(g, [y2 b]);
    elseif (sign(g(y2)) == sign(g(b)))
        % only 1 root
        x1 = fzero(g, [b/n y1]);
    else
        % 3 roots
        x1 = fzero(g, [(b/n) y1]);
        x2 = fzero(g, [y1    y2]);
        x3 = fzero(g, [y2     b]);
    end
end
```

Listing A.11: The f'n that finds the equilibrium x value(s) for a given b

```
n=8;
alpha=4;
beta=40;

clear y; clear k;
limit=0.8;
step=0.001;
y=0:step:limit;
m=size(y);
for j=1:m(2)
    [x1,x2,x3] = zeros_fn(y(j),n,alpha,beta);
    k(j,1) = y(j);
    k(j,2) = x1;
    k(j,3) = x2;
    k(j,4) = x3;

    if (x3 ~= 0 && k(j-1,4)== 0)
        a1 = j
    end

    if (j>1 && x3 == 0 && k(j-1,4) ~= 0)
        a2 = j
    end
end
```

```
xseg1 = k(1:a2-1,1);
xseg2 = k(a1:a2-1,1);
xseg3 = k(a1:m(2),1);

yseg1 = [k(1:a1-1,2);k(a1:a2-1,4)];
yseg2 = k(a1:a2-1,3);
yseg3 = k(a1:m(2),2);

xaxis = [xseg1;flipud(xseg2);xseg3];
yaxis = [yseg1;flipud(yseg2);yseg3];

plot (xaxis, yaxis);
hold on;
plot (xaxis(1:0.5/step), xaxis(1:0.5/step), '--b');
plot (xaxis, xaxis/n, '--b');
xlabel('b')
ylabel('x')
title(strcat('Equilibria for \alpha=',num2str(alpha),', \beta=',num2str(beta
    ),', worker number k=',num2str(n)))
hold off;
figure(gcf);
```

Listing A.12: The script that calculates and plots the equilibrium (b,x) pairs

Appendix B

Miscellaneous

B.1 History - State Mapping

q	α	β	γ
1	CC	CC	CC
2	DC	CC	CC
3	CD	CC	CC
4	DD	CC	CC
5	CC	DC	CC
6	DC	DC	CC
7	CD	DC	CC
8	DD	DC	CC
9	CC	CD	CC
10	DC	CD	CC
11	CD	CD	CC
12	DD	CD	CC
13	CC	DD	CC
14	DC	DD	CC
15	CD	DD	CC
16	DD	DD	CC
17	CC	CC	DC
18	DC	CC	DC
19	CD	CC	DC
20	DD	CC	DC
21	CC	DC	DC
22	DC	DC	DC
23	CD	DC	DC
24	DD	DC	DC
25	CC	CD	DC
26	DC	CD	DC
27	CD	CD	DC
28	DD	CD	DC
29	CC	DD	DC
30	DC	DD	DC
31	CD	DD	DC

32	DD	DD	DC
33	CC	CC	CD
34	DC	CC	CD
35	CD	CC	CD
36	DD	CC	CD
37	CC	DC	CD
38	DC	DC	CD
39	CD	DC	CD
40	DD	DC	CD
41	CC	CD	CD
42	DC	CD	CD
43	CD	CD	CD
44	DD	CD	CD
45	CC	DD	CD
46	DC	DD	CD
47	CD	DD	CD
48	DD	DD	CD
49	CC	CC	DD
50	DC	CC	DD
51	CD	CC	DD
52	DD	CC	DD
53	CC	DC	DD
54	DC	DC	DD
55	CD	DC	DD
56	DD	DC	DD
57	CC	CD	DD
58	DC	CD	DD
59	CD	CD	DD
60	DD	CD	DD
61	CC	DD	DD
62	DC	DD	DD
63	CD	DD	DD
64	DD	DD	DD

Table B.1: The Enumeration of History Partitions

Appendix C

More on the GGA

In Chapter 3, we introduced versions of the GGA that deal with certain symmetric games. In this appendix, we shall try to introduce new versions that accomodate asymmetric games as well (that is, games with more than one ”type” of player).

The biggest issue to contend with when switching from symmetric to asymmetric games is encoding/decoding. Let us say, for instance, that we are looking for equilibrium (action or strategy) profiles in a given game. If the game is symmetric, then the natural choice is for each chromosome to represent one action/strategy, because all of the players are the same. If the game is asymmetric, then there are at least two natural representation schemes. One involves making one chromosome represent one profile, with different parts of the chromosome encoding actions/strategies for a different player:

$$\underbrace{\boxed{1}\boxed{2}\boxed{3}\ldots\boxed{6}}_{\textit{player } 1}\underbrace{\boxed{7}\boxed{8}\boxed{9}\ldots\boxed{12}}_{\textit{player } 2}\ldots\underbrace{\boxed{68}\boxed{69}\boxed{70}}_{\textit{player } n}$$

With this approach, problems arise when trying to engineer a fitness function for the algorithm: how do we define the fitness of a action/strategy profile?

The alternative solution would be to for each chromosome to represent only one action/strategy (same as for a symmetric game), but this would require a population to exist for every different type of player in the game, each with its own encoding/decoding scheme. This is the approach that we shall explore deeper in this appendix.

C.1 Symmetry Partitions

The concept of a symmetry partition attempts to capture which players are the "same" as other players in the game; here, "same" implies the inter-changeability inter-relation which all of the players in a symmetric game have.

Definition 36 (Symmetry Partition)**.** *The partition $\mathbb{J}$ of the player set N in an n-player strategic game $G = \langle N, (A_i), (u_i) \rangle$ is a* ***symmetry partition*** *if the following conditions hold: for every partition $J \in \mathbb{J}$,*

1. *Every player in J has the same action space*
2. *Every player in J has a symmetric payoff function in the following sense: pick two action profiles $a, a' \in A$ and a pair of players $i, j \in J$ arbitrarily. If $a_i = a'_j$ and a_{-i} can be obtained from a'_{-j} by a permutation of actions, then $u_i(a) = u_j(a')$.*

We can then write G as a a game $\langle \mathbb{J}, (A_j), (u_j) \rangle$, where $\mathbb{J} = \{1, 2, ..., k\}$ is now the enumeration of the player symmetry partitions, and A_j and u_j are the action set and utility function for the j^{th} symmetry partition.

Straight away we can see that an m-player symmetric game has only one set in its symmetry partition, which contains every player. On the other hand, for an m-player game in which every player has her own strategies and payoffs, the symmetry partition is a set of singletons, one for each player.

Example 20 (Symmetry Partition). *We take Example 7 and modify it so it now has the following rules:*

- *players II and III effectively play a 2-player rock-paper-scissors game for the pot*
- *if they tie, the pot is shared between all 3 players*
- *it does not matter which strategy player I plays*

The game's payoff profiles are as in Figure C.1 (tuples correspond to payoffs for players (I,II,III)). From them, we can say that the symmetry partition of

I = R,P,S		III		
		R	P	S
II	R	(2,2,2)	(0,0,6)	(0,6,0)
	P	(0,6,0)	(2,2,2)	(0,0,6)
	S	(0,0,6)	(0,6,0)	(2,2,2)

Figure C.1: Modified Rock-Paper-Scissors Payoffs

this game is $\{\{II, III\}, \{I\}\}$.

Definition 37 (The Genetic Game Algorithm for a Strategic m-player Game). *The GGA for a strategic m-player game* $G = \langle \mathbb{J}, (A_j), (u_j) \rangle$, *consists of:*

1. *a collection* $(D_j)_{j \in \mathbb{J}}$, *with each* $D_j \subseteq A_j$, *(*A_j *being the symmetry partition j's actions), and* D_j *having* 2^{k_j} *elements for some* $k_j \in \mathbb{N}$,
2. *the evaluation functions:*
   - ${}^jT_e : D_j \rightarrow B_j$ *(an invertible encode function),*
   - ${}^jT_d : B_j \rightarrow D_j$ *(the inverse of encode, the decode function)*

- $^{j}T_f : \times_{l \in \mathbb{J}}(D_l)^n \rightarrow ((\mathbb{R}_0^+)^n)^{|\mathbb{J}|}$ *(fitness)*

where $B_j = \{0,1\}^{k_j}$ *(k_j is as in point 1),and* $j \in \mathbb{J}$,

3. *the genetic functions:*

   - $({}^{j}T_s^i)_{i=1}^{n/2} : B_j^n \times (\mathbb{R}_0^+)^n \rightarrow B_j^2$ *(selection)*
   - $({}^{j}T_c^i)_{i=1}^{n/2} : B_j^2 \rightarrow B_j^2$ *(crossover)*
   - ${}^{j}T_m : B_j \rightarrow B_j$ *(mutation)*

4. *the terminating condition function* $T_t : B^n \times \mathbb{N} \rightarrow \{true, false\}$ *- its input is a population and its generation number,*

5. *an m-vector* $(\vec{Y}_j(0))_{j \in \mathbb{J}}$ *of strategy n-tuples,* $\vec{Y}_j(0) \in D_j^n$ *(with n a multiple of 2), called the initial population,*

Then the ***population sequence*** $(\vec{X}_j(p))_{p \in \mathbb{N}}$, $\vec{X}_j(p) \in B_j^n$ *is obtained using the following:*

$$\vec{X}_j(p) = \begin{cases} {}^{j}T_{e,n}(\vec{Y}_j) & for\ p = 0 \\ (p_1, p_2, ..., p_n) & for\ 1 \leq p \leq c \end{cases} \tag{C.1}$$

where $\forall i = 1, ..., \frac{n}{2}$, $\forall j \in \mathbb{J}$,

$$(p_{2i-1}, p_{2i}) =^{j} T_{m,2}({}^{j}T_c^i({}^{j}T_s^i(\vec{X}_j(p-1),^{j} T_f({}^{j}T_{d,n}(\vec{X}_j(p-1)))))) \tag{C.2}$$

In the above expressions, the ***terminating generation*** $c \in \mathbb{N}$ *is a number that satisfies the following conditions:*

$$0 \leq j < c \Rightarrow T_t(\vec{X}(j), j) = false\ ,\ and \tag{C.3}$$

$$T_t(\vec{X}(c), c) = true. \tag{C.4}$$

The GGA for extensive games is adapted in a similar way:

Definition 38 (The Genetic Game Algorithm for an Extensive m-player Game)**.** *The GGA for an extensive m-player game* $G = \langle N, H, P, (\mathbb{I}_i), (\succsim_i)\rangle$ *is defined in exactly the same way as in Definition 37, except that:*

- *instead of* $(D_j)_{j\in N}$*, we have* $(W_j)_{j\in N}$*, a collection of subsets of player* j *strategies for the game* G*, with* W_j *having* 2^{k_j} *elements for some* $k_j \in \mathbb{N}$*, and*
- *we replace every instance of* $\mathbb{J}$ *with* N

Bibliography

R. Axelrod. The Evolution of Strategies in the Iterated Prisoner's Dilemma. *Genetic Algorithms and Simulated Annealing*, 1987.

K. DeJong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, 1975.

Y. Gao. An Upper Bound on the Convergence Rates of Canonical Genetic Algorithms. *Complexity International*, 5, 1998.

J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

S. Huck, D. Kubler, and J. Weibull. Social norms and economic incentives in firms. March 2003.

R. E. Marks. Niche strategies - the prisoners dilemma computer tournaments revisited. 1989.

J. Nachbar. The evolution of cooperation revisited, mimeo. June 1988.

M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, 1994. ISBN 0-262-15041-7.

J. W. Weibull. *Evolutionary Game Theory*. The MIT Press, 1995. ISBN 0-262-73121-5.

D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 1994.

Declaration

I hereby declare that the work submitted in this thesis was independently collected and written, and that no sources or references, other than those cited, were used.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstst ändig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den ....

(Yakov Benilov)