Modelling Exchange Using the Prisoner's Dilemma and Genetic Programming

Laurie Hirsch Masoud Saeedi

Sheffield Hallam University
Sheffield S1 1WB
UK
Email: l.hirsch@shu.ac.uk m.h.saeedi@shu.ac.uk

Abstract

In this paper we show how exchange, co-operation and other complex strategies found in nature can be modelled using the prisoner's dilemma game and genetic programming. We are able to produce and evolve different strategies represented by computer programs that can play the prisoner's dilemma against a set of predefined strategies or against other programs in the population (co-evolution). Although the game is simple the number of possible strategies for playing it is huge. Genetic programming provides an efficient search mechanism capable of identifying and propagating strategies that do well in a particular environment. Our implementation provides a distinct advantage over previous investigations into the prisoner's dilemma using genetic algorithms. In particular strategies can be based upon the entire history of a game at any point, rather than on recent moves only. We incorporate the use of list data structures as terminals and provide list-searching capability in the function set so that potentially large volumes of data can be utilised by the evolved programs.

Keywords : Prisoner's Dilemma, Genetic Programming, co-evolution, large terminal sets.

1 Introduction

Analysis of the prisoner's dilemma game has proved useful in many fields of study including biology, sociology, psychology, political science and economics.[Poundstone 1992]. It has also been applied to the modelling of a form of animal behaviour know as 'reciprocal altruism'. This is where unrelated organisms co-operate with each other even when it would appear that such action is not advantageous in terms of inclusive genetic fitness.

Useful research in modelling this behaviour in a software environment using genetic algorithms to play the prisoner's dilemma has been developed [Alexrod 1987]. Here we build on this research using the more recently developed genetic programming system [Koza 1992]

2 Genetic Programming

Genetic programming (GP) is an automatic programming technique developed by John Koza in 1992. GP is a variant of the genetic algorithm (GA) and is similarly inspired by evolutionary processes occurring in nature, in particular by the exchange of strands of DNA occurring in sexually reproducing organisms. Whereas GA's use fixed length character strings as the genetic material GP uses executable LISP-like program trees. A fixed size and structure of evolved programs in a GP system does not need to be specified in advance. These trees serve as both the genetic material of an individual and as the solution in program code; there is no intermediate representation.

A population of computer programs is initially created by randomly combining functions and terminals from a predefined set relevant to the problem domain. A fitness function that can take an arbitrary GP individual from the population and return an assessed fitness for that individual must also be provided. The fitness value returned is used to favour the selection of programs with higher fitness that will be used to form the next generation.

GP's genetic operators are customised to deal with GPs tree structured individuals. The three most common operators are subtree crossover, point mutation and reproduction. The mutation operator takes a single individual and replaces an arbitrary subtree in this individual with a new, randomly generated subtree. The crossover operation swaps sub-trees between two selected fit individuals to produced two new programs for the next generation. The reproduction operation takes a fit individual and copies it into the next generation.

The process of measuring and creating new generations of programs is repeated. Differences in measured fitness are exploited such that in general we will see an improvement in the fitness of the programs. Unspecified programs of variable size, structure and complexity are likely to evolve. Termination occurs after a predefined fitness has been achieved by one or more programs or after a pre-set number of generations has been completed. In this case the best individual occurring in the entire run represents the candidate solution to the particular problem.

3 The Prisoner's Dilemma

The prisoner's dilemma is a game for 2 players where both players are trying to achieve a maximum score but where their interests are not necessarily opposed.

Each player has a choice of two moves usually referred to as co-operate(C) or defect (D). This is because the original version of the game was based upon a hypothetical situation in which two individuals are detained after a crime. Confessing to the crime is seen as a defection and co-operation is equivalent to a refusal to admit to the crime. They are kept isolated from each other but both offered a series of more or less appealing sentences and rewards, which are dependent upon the actions of both individuals.

The various outcomes can be given scores as in the table below (score for A first).

	B Co-operates(C)	B Defects(D)
A Co-operates(C)	2,2 (CC)	0,3 (CD)
A Defects(D)	3,0 (DC)	1,1 (DD)

It is the ranking of the outcomes that defines a game as a prisoner's dilemma. Wherever we find we have a choice to cooperate (C) or defect (D) in a game with another player and the outcomes are ranked such that

DC > CC > DD > CD

then we are playing the prisoner's dilemma. The moves are simultaneous and each player is unable to determine the others move until after the move.

Where there is one move in a game of prisoner's dilemma there are only 2 possible strategies: (C, D). There is no incentive to cooperate in this game i.e. whatever move the other player makes it is best to defect. When the game is iterated so that 2 players repeat the game over many moves we find that there are a large number of strategies and that co-operation can be advantageous provided that it can in some way secure co-operation from the other player.

A distinction can be made between 'blind strategies' such as always co-operate (ALLC), always defect (ALLD) and play a random move (RANDOM) and those that took some account of what had occurred in the previous moves. For example ALLC would get a good score when matched with itself but a 0 score when matched with ALLD.

One of the most successful strategies is known as 'Tit For Tat (TFT). This specifies *co-operation on the first move and then always copy the other players last move*. TFT will do well against itself or against strategies like ALLC but is 'provocable' in that it will defect in response to defection, and thus get the best possible score of 1 against ALLD. Tit-for-Tat represents a simple way to gain co-operation in a competitive situation without risking heavy loses from other individuals who will not co-operate.

TFT will start to co-operate at any time its partner does. TFT as a strategy based on reciprocity but does not take advantage of the blind strategies such as ALLC where it is always best to defect.

4 Using Genetic Algorithms with the Iterated Prisoner's Dilemma(IPD)

Alexrod has used a genetic algorithm to evolve strategies for prisoner's dilemma. This was done by specifying each allowable strategy as a string of genes on a chromosome. Each game has 4 possible outcomes giving a total of 64 possible histories for the last 3 moves. To determine its choice of co-operate or defect a strategy would only need to determine what to do in each of the possible situations that could arise.

Alexrod used binary strings (chromosomes) to encode strategies based upon the last three moves in the IPD. [Alexrod 1987]

Strategies represented by chromosomes were then evolved using crossover and mutation over a number of generations. Alexrod found that most strategies that evolved in the simulation resembled TFT having many of the properties that make TFT successful such as ‘continue to co-operate after mutual co-operation is established’, ‘be provocable e.g. defect after the other player defects’ ‘accept an ‘apology’ i.e. continue to co-operate after co-operation has been restored’

5 Implementing the IPD using Genetic Programming

The objective of the application was to repeat and develop the work done by Alexrod on the iterated prisoner’s dilemma (IPD) problem using genetic programming. It was also hoped that this more flexible approach would enable further investigation of co-evolution and co-operation.

The following table represents the scoring system used in our implementation.

	B Co-operates	B Defects
A Co-operates	3,3	0,5
A Defects	5,0	1,1

While the ranking is the same the values are changed slightly. This is to meet a second requirement in IPD, which is that, the average of co-operation and defection is less than that of continuous co-operation. Thus if two players are alternately co-operating while the other is defecting their average score will be 2.5 which would be less than the average of 3 for two co-operating players.

5.1 List Implementation

Although the search space in the genetic algorithm implementation described above is huge we have developed an implementation using genetic programming that is not constrained to examining only the last 3 moves of the game. Because we are evolving computer programs we can include powerful list-searching functions that allow for the development of strategies based on the entire history of the game at any point. This implementation also allows us to examine the problem of large terminal sets in relation to genetic programming.

5.2 Terminal Set

The terminal set *t* consists of only 2 variables.

t = {opponents-move-list, players-move-list}

The variables are lists that record the moves of each player. After a move in a game of prisoner’s dilemma the move of each player is added to the front of their lists. Thus the first element of the terminal *players-move-list* will be the players own last move. After move 5 in a game of IPD the opponents-move list variable might look like (DDDDC) which indicates co-operation on the first move followed by defections on the following 4 moves.

5.3 Function Set

All the functions in the implementation will return lists and will accept lists as arguments. This will ensure that all programs produced, either randomly in generation 0, or via genetic operators will be valid programs that will run and return a result, which will be the programs next move (closure property).

NOT-LIST(a) : this function is similar to the Boolean function NOT. It takes one argument of type list and will return the complement of the first element of that list and return it as a one element list containing just that value e.g. if the argument a = (CCCDCD) then NOT-LST(a) will return (D)

ANDC(*a b*) : this function is similar to the Boolean function AND. It takes two arguments of type list and will return the one element list (C) if both FIRST(*a*) and FIRST(*b*) are equal to ‘C’ else will return (D)

ORC(a b) : this function is similar to the Boolean function OR. It takes two arguments of type list and will return the list (C) if either FIRST(a) OR FIRST(b) are equal to ‘C’ else will return (D)

REST-LST(*a*) : this function is a modified version of the LISP function REST and is used to return a list minus its first element. If REST(*a*) is a list then this is returned else *a* is returned. This is to ensure that an atom, which would not be an acceptable argument for other functions, is never returned e.g. REST(C) = (C), REST(CDDCD) = (DDCD)

MEMBERD(*a*) will return the list (D) if the list a contains a ‘D’ in any position else (C)

REVERSE(*a*) this is the common LISP function and will simply return the list in reverse order.

5.4 Wrapper

At the end of evaluating a program (i.e. executing) a list is returned. In order to find the move produced by the program being evaluated we extract the first element of the list produced.

5.5 Initial conditions

Because the functions must have non-empty lists as arguments the 2 terminals were initialised to the list '(C). This does introduce a bias into the environment but this is somewhat reduced by not scoring the first 10 moves of any game.

5.6 Example

For the examples we assume that:
opponents-move-list = (CCDCC)

and
players-move-list = (DCCDC)

An example program that might be produced randomly in generation 0 or via genetic operators

(ORC opponents-move-list (REST-LST opponents-move-list))

(REST-LST opponents-move-list) evaluates to (CDCC)

the entire program evaluates to

(ORC (CCDCC) (CDCC))

and will return (C) to the wrapper which in turn will return 'C' i.e. that will be the programs move at that point.

The program

(ORC opponents-move-list opponents-move-list)

is the strategy TIT-FOR-TAT since the initial state of each list is (C).

An example of the strategy ALLC would be the program

(ORC opponents-move-list (NOT-LST opponents-move-list))

An example of ALLD would be the program

(ANDC opponents-move-list (NOT-LST opponents-move-list))

By repeating the function *REST-LST* the program can look back at previous moves e.g. the program

(REST-LST (REST-LST (REST-LST players-move-list)

will return the players own move, 4 moves previously.

6 Experiments

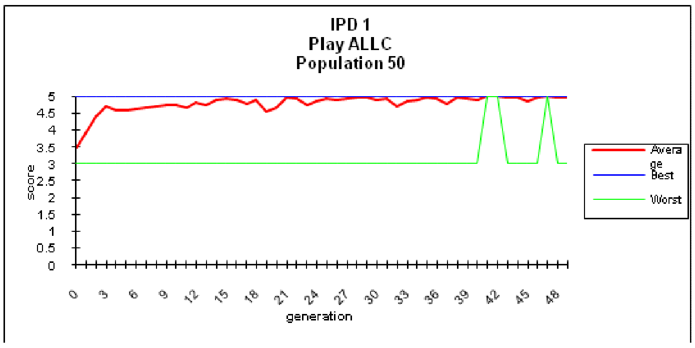
The genetic programming system for the iterated prisoner's dilemma (IPD) was used to implement a number of experiments. These are described together with graphs summarising the results below. Some of the best and worst programs are listed to illustrate the type of evolution occurring at this level. All IPD games were 150 moves long and the fitness of a program was the average score obtained.

6.1 Playing Blind Strategies

Blind strategies are those which take no account of the history of the game at any point. The best strategy against any blind strategy is simply to always defect (i.e. ALLD) as there is no hope of changing the opponents move by co-operating.

6.1.1 Experiment 1: playing ALLC

ALLC is the first of the 'blind' strategies to be used as an opponent. ALLC will always return 'C' i.e. will always 'co-operate'. When playing ALLC the worst possible average score is 3 and this is obtained when ALLC plays itself. The best possible score is 5, which is obtained by ALLD.



With a population this size it is highly likely that at least one individual will be playing the best strategy (ALLD) in generation 0 as is shown on this run.

Example program scoring 5 (best score):

(ANDC (REST-LST(REST-LST (NOT-LST OPPONENTS-MOVE-LIST)))
(REST-LST (NOT-LST (REVERSE PLAYERS-MOVE-LIST))))

This program appeared by chance alone in generation 0. The function ANDC takes 2 arguments. Where both arguments contain 'C' at the front of the list then (C) will be returned by the function else (D). In this case the opponents move is always 'C'. (REST-LST (NOT-LST OPPONENTS-MOVE-LIST) will therefore return (D) so the entire program will also always return 'D' i.e. ALLD which is the best strategy against ALLC.

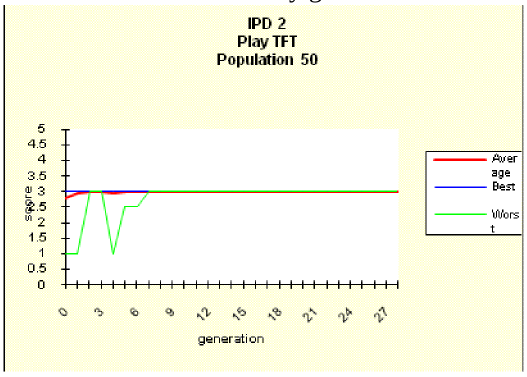
6.2 *Playing More Complex Strategies based on previous moves.*

The game of IPD become more interesting where strategies are used which take into account the history of the game. This opens the possibility of communication via the moves and of developing co-operation.

6.2.1 Experiment 2: Playing Tit-For-Tat (TFT)

TFT is the simple strategy, which specifies co-operation on move 1 and then repeating the opponents' previous move. Co-operation is the best strategy when playing this opponent. Defection against TFT will achieve the maximum score of 5 for one move but will 'provoke' TFT into subsequent defection resulting in the low score of 1 for both players. To return to mutual co-operation after defection a player must accept a 0 score for one move before TFT will begin to co-operate again. Note that the blind strategy ALLC will do as well against TFT as TFT playing itself or some other strategy based on similar principles.

In this run evolution to the optimum strategy occurred very quickly so that all individuals were playing a version of TFT or ALLC by generation 8.



Example Program:

(REST-LST OPPONENTS-MOVE-LIST)

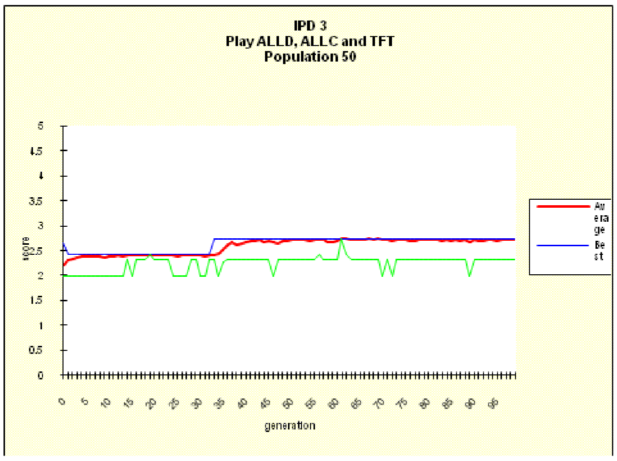
This program is a version of the TFT strategy and will score 3 against TFT

6.3 *Playing a combination of Strategies*

Evolving strategies that produce good solutions against a combination of other strategies is far more demanding as a program may need to identify the strategy of its opponent in order to play the best game.

6.3.1 Experiment 3: Playing ALLC, ALLD and TFT

A good program must be able to identify the kind of strategy being played (by provocation if necessary), and to respond in the best way to the information received about the opponent. TFT does well against this combination but does not exploit the weakness of ALLC i.e. TFT will co-operate with ALLC when defecting (ALLD) is the highest scoring option.



Individuals achieved better than TFT by being able to identify exploitable individuals such as ALLC. This was not achieved by any human programmer or strategist in the tournaments held by Alexrod in the 1980's. We should note that to achieve this an individual must be able to do three things:

- to be able to discriminate between one individual and another based upon only the behaviour the other player shows or is provoked into showing
- to adjust its own behaviour to exploit an individual that is identified as exploitable
- to be able to achieve this discrimination and exploitation without producing poor results with other individuals

If the opponent was identified as unexploitable (e.g. TFT*)* an 'apology' is made and mutual co-operation was established whilst exploitable individuals (e.g. ALL*C*) were effectively exploited.

Notice that there are two distinct stages of rapid growth in average fitness the first occurring in the first few generations and the second at about generation 35. Analysis of the programs reveals that the first period of evolution stabilised at a strategy of TFT or similar which scores an average 2.3 against the three opponents.

A new strategy was discovered which was able to exploit ALLC while co-operating with TFT and defecting against ALLD after 30 generations. This then spread rapidly throughout the population. This may be an example of what evolutionary biologists term punctuated equilibrium.
Example Program:

```
(ANDC (ANDC
(NOT-LST (REST-LST (REST-LST (MEMBERD
OPPONENTS-MOVE-LIST))))
(REST-LST (REST-LST (REVERSE (REST-LST
OPPONENTS-MOVE-LIST)))))
(ORC (REVERSE (REST-LST OPPONENTS-
MOVE-LIST))
(REVERSE (ANDC (REST-LST OPPONENTS-
MOVE-LIST)
(REST-LST OPPONENTS-MOVE-LIST)))))
```

This program appeared in generation 34 and scored 3.0 (significantly better than TFT: score 2.3). The key element of the program is the branch

```
(NOT-LST (REST-LST (REST-LST (MEMBERD
OPPONENTS-MOVE-LIST)))
```

The function REST-LST has no effect here. MEMBERD will return (D) if there is a 'D' anywhere in opponents previous moves else '(C)' and NOT-LST will complement the move. In pseudo English we might specify this strategy as *'co-operate if your opponent has ever defected else defect'* and so effectively identifies strategies similar to the blind strategy ALLC that are open to exploitation.

On its own this strategy would not be successful against strategies like ALLD since it would co-operate with them. However the strategy conjuncts (via the ANDC function) with other branches which effectively implement TFT. In the above program the branch below must also evaluate to (C) before the move 'C' is played.

```
(REST-LST (REST-LST (OPPONENTS-MOVE-
LIST)))
```

To summarise the key features of this successful strategy we might say *co-operate if your opponent has ever defected AND has recently co-operated ELSE defect*

A shorter version of the program would be:

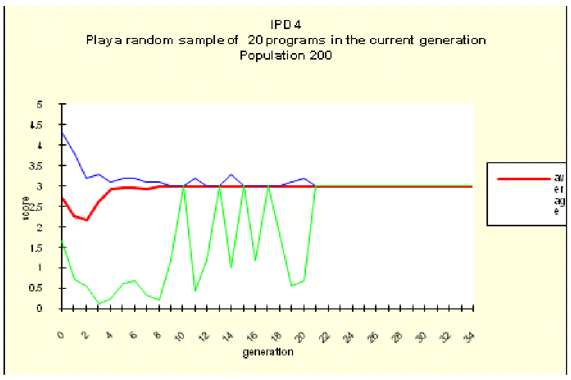
```
(ANDC(NOT-LST (MEMBERD OPPONENTS-
MOVE-LIST)) OPPONENTS-MOVE-LIST))
```

6.4 Co-evolution

These experiments involve evaluating each programs fitness by taking the average score when played against a number of other opponents from the population of programs

6.4.1 Experiment 4: Playing a sample of the population

In this experiment each program plays a sample of 20 programs selected randomly from the population.



The average fitness reduces dramatically in the early generations before rising to stabilise at a co-operative strategy.

Example program:

```
(ANDC (REST-LST (REST-LST (ORC PLAYERS-
MOVE-LIST PLAYERS-MOVE-LIST)))
(REST-LST (NOT-LST (REVERSE PLAYERS-
MOVE-LIST))))
```

This was the best of run program occurring in generation 0 and scoring 4.2. The program is in fact a version of the blind strategy ALLD and did so well because a large proportion of its random sample of opponents were playing a strategy similar to ALLC.

7 Results

In all cases the average fitness eventually improved over its initial value i.e. evolution occurred in the system.

When playing fixed strategies or a combination of fixed strategies, the average, best and worst fitness improves from generation 0 toward some maximum value.

In the co-evolutionary example the average, best and worst fitness values all decrease initially before improving and stabilising at a point where all individuals are co-operating (score 3.0). This initial decline occurred as co-operating individuals were exploited by defectors who could achieve high fitness and therefore proliferate. The point at which the fitness of the population began to increase coincided with the ‘discovery’ of the Tit-For-Tat strategy and its subsequent spread through the population.

Where the environment is co-evolving co-operative behaviour becomes the dominant strategy even though in the early generations strategies which refuse to co-operate achieve higher scores

Experiment 4 revealed two distinct areas of evolution.

8 CONCLUSIONS

The prisoner’s dilemma is a simple non-zero-sum game that has proved useful in modelling many different kinds of exchange. In many biological settings, where two organism are regularly interacting and can remember some aspects of the prior exchange then the strategic situation may become an iterated prisoner’s dilemma. Artificial models of animal behaviour are created via the IPD application and these may prove useful to understanding the often-complex strategic situation occurring between interacting animals.

Genetic programming provides a powerful and flexible tool for the evolution of strategies for playing the game in different environments. Using computer programs rather than character strings to encode strategies provides a more readable and natural media for representing strategies. Co-operative behaviour was easily evolved and novel strategies produced in the experiments. Strategies were based on the entire game at any point rather than on recent moves only as in previous work using the genetic algorithm. Using data structures (lists in this case) as terminals in a GP system and providing functions that are able to search these structures provides a way for GP systems to incorporate memory and large sets of data.

The experiments described above show how co-operative behaviour might evolve through a system of communication based on previous interaction. Short-term benefits can be gained by the exploitation of co-operative behaviour but can lead to the loss of more significant reward achieved by co-operating in the longer term. This is of course only true were individuals are able to identify ‘cheating’ behaviour and react by withdrawing future co-operation.

Software inspired from features of biological systems has proved powerful in problems involving search and optimisation. It is possible to view the evolution of organisms also as a problem of searching for or optimising organisms reproductive success in relation to their environment (which includes other evolving organisms). Modelling the natural situation via software simulation may greatly enhance our understanding of the process of natural selection itself and allows us to view the biological world from the greatly advantageous standpoint of having numerous examples both natural and artificial to investigate.

Bibliography

[Alexrod 1980] Alexrod R, 1980 Effective Choice in the Prisoner's Dilemma, Journal of Conflict (24)

[Alexrod 1984] Alexrod R 1984 The Evolution of Co-operation New York: Basic Books.

[Alexrod 1987] Alexrod R. 1987 The Evolution of Strategies in the Iterated Prisoner's Dilemma in L. Davies(ed.), Genetic Algorithms and Simulated Annealing, London : Pitman

[Altenberg 1994] Altenberg L 1994 The Evolution of Evolability in Kinnear K, Advances in Genetic Programming [KI94]

[Badcock 89] Badcock C, 1989 The Problem of Altruism Harper Collins

[Haynes 1995] Haynes T, Wainwright S February 1995, A Simulation of Adaptive Agents in a Hostile Environment, Proceedings of the 1995 ACM/SIGAPP Symposium on Applied Computing, ACM Press

[Haynes 1995b] Haynes T, Wainwright S, Shoenefeld D + S, July 1995 Strongly Typed Genetic Programming in Evolving Co-operation Strategies, Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kaufnann

[Holland 1992] Holland J. 1992, Adaptation in Natural and Artificial Systems, second edition, MIT Press

[Kinnear 1994] Kinnear K E 1994 editor Advances in Genetic Programming, MIT

[Koza 1992] Koza J Genetic Evolution and Co-Evolution of Computer Programs in Langton G., Taylor C., Farmer J., Rasmussen S 1992 Artificial Life II, Addison-Wesley[LA92]

[Koza 1992] Koza John R.1992 Genetic Programming: On the Programming of Computers by means of Natural Selection, MIT

[Koza 1994] Koza J 1994 Introduction to Genetic Programming in Kinnear K, Advances in Genetic Programming [KI94]

[Lindgren 1992] Lindgren Evolutionary Phenomena in Simple Dynamics in Langton G., Taylor C., Farmer J., Rasmussen S 1992 Artificial Life II, Addison-Wesley

[Luke 1996] Sean Luke and Lee Spector, Evolving teamwork and coordination with genetic programming. In http://www-cs-faculty.stanford.edu/~koza/J Koza, D Goldberg, D Fogel, R Riolo , *Genetic Programming 1996: Proceedings of the First Annual Conference* , pages 150-156, Stanford University, CA, USA, 28-31 July 1996. MIT Press

[Luke 1997] Luke, S. et al. 1997. Co-evolving Soccer Softbot Team Coordination with Genetic Programming. In Proceedings of the RoboCup-97 Workshop at the 15th International Joint Conference on Artificial Intelligence (IJCAI97). H. Kitano, ed. 115--118. IJCAI.

[Poundstone 1992] Poundstone W 1992 Prisoner's Dilemma, Oxford University Press