Using Symbolic Regression to Infer Strategies from Experimental Data

John Duffy[1] and Jim Engle–Warnick[1]

University of Pittsburgh, Pittsburgh, PA 15260 USA

Abstract. We propose the use of a new techinque–symbolic regression–as a method for inferring the strategies that are being played by subjects in economic decision making experiments. We begin by describing symbolic regression and our implementation of this technique using genetic programming. We provide a brief overview of how our algorithm works and how it can be used to uncover simple data generating functions that have the flavor of strategic rules. We then apply symbolic regression using genetic programming to experimental data from the repeated "ultimatum game." We discuss and analyze the strategies that we uncover using symbolic regression and conclude by arguing that symbolic regression techniques should at least complement standard regression analyses of experimental data.

1 Introduction

A frequently encountered problem in the analysis of data from economic decision–making experiments is how to infer subjects' strategies from their actions. The standard solution to this inference problem is to make some assumptions about how actions might be conditioned on or related to certain strategically important variables and then conduct a regression analysis using either ordinary least squares or discrete dependent variable methods. A well-known difficulty with this approach is that the strategic specification that maps explanatory variables into actions may be severely limited by the researcher's view of how subjects ought to behave in the experimental environment. While it is possible to experiment with several different strategic specifications, this is not the common practice, and in any event, the set of specifications chosen remains limited by the imagination of the researcher.

In this paper, we propose the use of a new technique – symbolic regression using genetic programming – as a means of inferring the strategies that are being played by subjects in economic decision-making experiments. In contrast to standard regression analysis, symbolic regression involves the breeding of simple computer programs or functions that are a good fit to a given set of data. These computer programs are built up from a set of model primitives, specified by the researcher, which include logical if–then–else operations, mathematical and Boolean operators (and, or, not), numerical constants, and current and past realizations of variables relevant to problem that is being solved. These programs can be generated for each subject in a population and may be depicted in a decision tree format that facilitates their interpretation as individual strategies.

The genetic programming algorithm that we develop for breeding and selecting programs is an automated, domain–independent process that involves large populations of computer programs that compete with one another on the basis of how well they predict the actions played by experimental subjects. These computer programs are selected for breeding purposes based on Darwin's principle of survival of the fittest and they also undergo naturally occurring genetic operations such as crossover (recombination) that are appropriate for genetically mating computer programs. Following several generations of breeding computer populations, the algorithm evolves programs that are highly fit in terms of their ability to predict subject actions. The directed, genetic search process that genetic programming embodies, together with the implicit parallelism of a population–based search process has proven to be a very powerful tool for function optimization in many other applications.[4]

The advantage of symbolic regression over standard regression methods is that in symbolic regression, the search process works simultaneously on both the model specification problem and the problem of fitting coefficients. Symbolic regression would thus appear to be a particularly valuable tool for the analysis of experimental data where the specification of the strategic function used is often difficult, and may even vary over time. We begin by describing genetic programming and how it can be used to perform symbolic regression analysis. We then show that our algorithm is capable of uncovering simple data generating functions that have the flavor of strategic rules. We apply our symbolic regression algorithm to experimental data from the repeated ultimatum game. We discuss and analyze the strategies that we uncover using symbolic regression and we conclude by arguing that symbolic regression should at least complement standard regression analyses of experimental data.

2 Symbolic Regression Using Genetic Programming

The use of genetic programming for symbolic regression was first proposed by John Koza[4] as one of several different applications of genetic programming. In addition to symbolic regression, genetic programming has been successfully applied to solving a large number of difficult problems such as pattern recognition, robotic control, the construction of neural network architectures, theorem proving, air traffic control and the design of electrical circuits and metallurgical processes. The genetic programming paradigm, as developed by Koza and other artificial intelligence researchers, is an approach that seeks to automate the process of program induction for problems that can be solved on a computer i.e. for problems that are computable. The basic idea is to use Holland's[3] genetic algorithm to search for a computer program that constitutes the best (approximate) solution to a computable problem given appropriate input data and a programming language. A genetic algorithm

is a stochastic, directed search algorithm based on principles of population genetics that artificially evolves solutions to a given problem. Genetic algorithms operate on populations of finite length, (typically) binary strings (patterned after chromosome strings) that encode candidate solutions to a well–defined problem. These strings are decoded and evaluated for their fitness, i.e. for how well each solution comes to solving the problem objective. Following Darwin's principle of survival of the fittest, strings with relatively higher fitness values have a relatively higher probability of being selected for mating purposes to produce the succeeding 'generation' of candidate solutions. Strings selected for mating are randomly paired with one another and, with certain fixed probabilities, each pair of 'parent' strings undergo versions of such genetic operations as crossover (recombination) and mutation. The strings that result from this process, the 'children', become members of the next generation of candidate solutions. This process is repeated for many generations so as to (artificially) evolve a population of strings that yield very good, if not perfect solutions to a given problem. Theoretical work on genetic algorithms, e.g. [2] reveals that these algorithms are capable of quickly and efficiently locating the regions of large and potentially complex search spaces that yield highly fit solutions to a given problem. This quick and efficient search is due to the use of a population–based search, and to the fact that the genetic operators ensure that highly fit substrings, called schema, (or subtrees in genetic programming) increase approximately exponentially in the population. These schema constitute the "building blocks" used to construct increasingly fit candidate solutions. Indeed, Holland[3] has proven that genetic algorithms optimize on the trade–off between searching for new solutions (exploration) and exploiting solutions that have worked well in the past.

Genetic programming is both a generalization and an extension of the genetic algorithm that has only recently been developed by Koza (1992) and others. In genetic programming the genetic operators of the genetic algorithm e.g. selection, crossover and mutation operate on a population of variable rather than fixed length character strings that are not binary, but are instead interpretable as executable computer programs in a particular programming language, typically LISP. In LISP (or in similar LISP–like environments), program structures, known in LISP as Symbolic expressions, or 'S–expressions' can be represented as dynamic, hierarchical decision trees in which the non–terminal nodes are functions or logical operators, and the terminal nodes are variables or constants that are the arguments of the functions or operators. The set of non–terminal functions and operators and the set of terminal variables and constants are specified by the user and are chosen so as to be appropriate for the problem under study.

In our application, each decision tree (self executing computer program) is viewed as a potential strategy for one subject, playing a particular role in a particular economic decision-making game. The fitness of each candidate

decision tree is simply equal to the number of times the program makes the same decision as the experimental subject over the course of the experimental session, given the same information that was available to the subject at the time of the decision. Koza has termed the problem of finding a function, in symbolic form, that fits a finite sample of data as symbolic regression. While there may be other ways of performing a symbolic regression, we know from the work of Holland that a genetic-algorithm-based search will be among the most efficient, hence, the use of genetic programming for symbolic regression analysis. As mentioned in the introduction, the major advantage of symbolic regression using genetic programming over standard regression methods is that one does not have to prespecify the functional form of the solution, i.e., symbolic regression is data–to–function regression. Instead, one simply specifies two sets of model primitives: (1) the set of non–terminal functions and operators, N, and (2) the set of terminal variables or constants, T. The dynamical structure of the player's strategy (in our application) is then evolved using genetic operations and the grammar rules of the programming language (LISP).

3 An Illustration

Our application of symbolic regression using genetic programming is perhaps best illustrated by an example. We will consider the well-known two player, repeated ultimatum game, using data from an experiment conducted by Duffy and Feltovich[1] where subjects played this game for 40 periods. The symbolic regression technique that we illustrate here can easily be applied to other experimental data sets as will (hopefully) become apparent from the description that follows.

In the ultimatum game, there are two players, A and B, sometimes referred to as proposer and responder. The proposer (player A) proposes a split of a \$10 pie and player B can either accept or reject the proposed split, with acceptance meaning implementation of the offer and rejection resulting in nothing for either player. In the Duffy–Feltovich experiment, the proposers (player As) could propose only integer dollar amounts, e.g. a split of \$6 for A and \$4 for B; this set–up greatly simplifies our implementation of symbolic regression using genetic programming. In this experiment, players of both types were randomly paired in each of 40 periods. Here we will focus on the strategy of player Bs, the responders, to the proposals of player As.

The large number of observations (40) for each subject allows us to run separate regressions for each subject, in an effort to uncover individual strategies. This will allow us to look for heterogeneity in the strategic behavior of subjects who were assigned to play the same role in the experiment. In data sets with a smaller number of observations per subject, one could use the symbolic regression technique to search for the strategy that best characterizes a population of players of a given type.

4 The Regression Model

The first step in conducting a symbolic regression is to specify a grammar for the programming language that will be used to evolve the structures (computer programs) that characterize the play of the game. A generative grammar for a programming language simply specifies the rules by which the set of non–terminal symbols and terminal symbols may be combined. Non–terminal symbols are those requiring further input, and terminal symbols are those that do not require any further input. For example, the non–terminal symbolic logic operator "if" requires three additional inputs, denoted in brackets { }: if {condition} then {do something} else {do other thing}. The inputs to the non–terminal "if" symbol may themselves be either non–terminals or terminals. An example of a terminal is a variable, constant or action requiring no further input. For example, in modeling the behavior of responders in the ultimatum game, the input for {do something} in the if expression above might be the terminal action accept; the input for do other thing might be the terminal action reject.

As in spoken languages, the grammar of a programming language is intended to be extremely general, admitting a wide variety of different symbolic operators and expressions. Rather than constructing the grammar of a programming language from scratch, the practice in genetic programming is to make use of the grammar of an existing, high-level programming language like LISP or APL. In this paper we make use of grammar of LISP. The advantage of LISP is that the input structures are all symbolic text arrays which are readily converted into programs (and vice versa). Furthermore, parse tree manipulations are easily implemented and program structures are free to vary in size up to some maximum length.

Our implementation of LISP is simulated using C++, but other programming languages can also be used, including, of course, LISP itself.

4.1 Grammar

We us the Backus–Nauer form grammar as described in Geyer–Schulz.[2] This grammar consists of non–terminal and terminal nodes of a tree, and a structure with which to build the tree. An example grammar that allows for nested "if" statements, and which we use for the ultimatum game is given in Figure 1. The grammar we use for the ultimatum game specifies that the set of non–terminals includes nested if–then statements, logical and, or, and not statements, and the mathematical operators $<$, $>$, and $=$. The set of terminals includes the past 3 proposals made by the player As that a player B has met (a1–a3), along with the player B's own past 3 responses (b1–b3). Also included is player A's current proposal, denoted a0. In our application, the internal representation of a player A proposal is an integer from 0-9 which denotes the amount of the \$10 prize that a player A proposes to keep for him or herself (thus \$10 - the player A's proposal is the

Node	Possible Derivations
<fe>	= (<f6><f0><fe><fe>) or (<f6><f0><f2><fe>) or (<f6><f0><fe><f2>) or (<f6><f0><f2><f2>)
<f0>	= (<f7><f0><f0>) or (<f8><f0>) or (<f1>) or (<f10>)
<f1>	= (<f9><f3><f3>) or (<f9><f4><f5>)
<f2>	= (<f10 >)
<f3>	= (<f11>)
<f4>	= (<f12>)
<f5>	= (<f13>)
<f6>	= “if”
<f7>	= “or” or “and”
<f8>	= “not”
<f9>	= “<” or “>” or “=”
<f10>	= “0” or “1” or “b1” or “b2” or “b3”
<f11>	= “4” or “5” or “6” or “7” or “8” or “a0” or “a1” or “a2” or “a3”
<f12>	= “T”
<f13>	= “5” or “10” or “15” or “20” or “25” or “30” or “35

Fig. 1. “Nested If Statement” Grammar for Ultimatum Game

amount to be received by player B). The internal representation of a player B's response is either a 0 or a 1 with a 0 representing reject, and a 1 representing accept. The set of terminals also includes the set of integers from 4–8, which player Bs may use to condition their decisions; we chose this set of integers since most player A proposals are for dollar amounts in this range. Finally, we include time, T, as an additional terminal symbol, along with integer values for 5-period intervals of play. If T is chosen, then the number referenced comes from $\langle f13\rangle$ as indicated by the grammar $(\langle f9\rangle\langle f4\rangle\langle f5\rangle)$. If one of the mathematical operators, $<$, $>$, or $=$ are chosen, the two numbers compared come from $\langle f11\rangle$ as indicated by the grammar $(\langle f9\rangle\langle f3\rangle\langle f3\rangle)$. The nodes $\langle f2\rangle - \langle f5\rangle$ simply add parentheses to nodes $\langle f10\rangle - \langle f13\rangle$, so that our algorithm understands these symbols to be terminal nodes. We can summarize the textual aspects of the grammar by noting that the set of non–terminals, $\mathcal{N} = \{if,\ or,\ and,\ not,\ <,\ >,\ =\}$ and the set of terminals, $\mathcal{T} = \{a0,\ a1,\ a2,\ a3,\ b1,\ b2,\ b3,\ 4,\ 5,\ 6,\ 7,\ 8,\ T,\ 5,\ 10,\ 15,\ 20,\ 25,\ 30,\ 35\}$.

In addition to specifying the set of non–terminals and terminals, the grammar also specifies how operations may be performed on the set $\mathcal{N}(\mathcal{T})$. The starting node, as specified in Figure 1, allows for four different initial derivations of a decision tree, (individual strategy) all of which begin with the non–terminal logical operator “if” = node $\langle f6\rangle$. As noted above, this operator requires three inputs, and the grammar in Figure 1 specifies restrictions on these inputs. For instance, the second input, which is the condition statement that the if operator evaluates, must always come from node $\langle f0\rangle$, which in turn requires either a Boolean operator from nodes $\langle f7\rangle - \langle f8\rangle$, $(and,\ or\ ,\ not)$, or a mathematical operator from node $\langle f9\rangle$ or a terminal from node

$\langle f10 \rangle$. The other two inputs in $\langle fe \rangle$ are designed to be as general as possible, with either node $\langle fe \rangle$ or node $\langle f0 \rangle$ possible for each input position. Similarly, the rules for non–terminal nodes $\langle f0 \rangle$ take account of the input needs of the operators in the first position. For instance, an "and" or "or" operator requires two inputs, whereas a "not" operator requires only one, and terminals from node $\langle f10 \rangle$ require no inputs. In addition to defining the structure of if–then statements and logical statements the $\langle fe \rangle$ and $\langle f0 \rangle$ nodes also call on their own nodes, thus allowing for nested versions of both types of statements. The internal representation of these various nodes is discussed in Appendix A1.

When constructing a tree we first choose uniformly from one of the four given derivations for $\langle fe \rangle$. Note that each of these derivations begins with the conditional if statement. Given a particular choice for $\langle fe \rangle$, we then proceed from left to right and choose uniformly from the possible derivations for each non–terminal node until only terminal nodes remain. To illustrate how this is done, we will derive a rule using the grammar of Figure 1 for the ultimatum game.

4.2 Deriving a rule

Consider the following rule for player B, the responder, in the ultimatum game: reject if the current period offer is greater than 5, otherwise accept. In the syntax of LISP, the symbolic expression is written as:

$$(if((> (a0)(5)))(0)(1).$$

Parentheses are used to control the evaluation of the expression, with the expression in the innermost set of parentheses being evaluated first. Figure 2 shows how the construction of this rule proceeds starting with a random choice for $\langle fe \rangle$. At each step we take the first non–terminal node, working from left to right through the rule, and replace it with one of its derivations. The process continues until the only remaining nodes are terminal. The result is a valid, interpretable decision rule. To see the internal representation for this process, see appendix A2.

4.3 Tree representation

What we have really derived (and what provides for a better interpretation) is a dynamic, hierarchical decision tree which consists of non–terminal and terminal nodes. The genetic program begins with a randomly generated population of these decision trees. Over many generations, the program creates trees of variable length up to a certain maximum depth using the non–terminal nodes to perform genetic operations in a search to find the best fit tree. The tree for the above rule is given in Figure 3. The depth of the tree is defined as the number of non–terminal nodes, in this case 13. When generating rules,

Symbol	Derivation	Resulting Rule
start	---	<fe>
<fe>	4th	(<f6><f0><f2><f2>)
<f6>	1st	(if<f0><f2><f2>)
<f0>	3rd	(if(<f1>)<f2><f2>)
<f1>	1st	(if((<f9><f3><f3>))<f2><f2>)
<f9>	2nd	(if((><f3><f3>))<f2><f2>)
<f3>	1st	(if((>(<f11>)<f3>))<f2><f2>)
<f11>	6th	(if((>(a0)<f3>))<f2><f2>)
<f3>	1st	(if((>(a0)(<f11>)))<f2><f2>)
<f11>	2nd	(if((>(a0)(5)))<f2><f2>)
<f2>	1st	(if((>(a0)(5)))(<f10>)<f2>)
<f10>	1st	(if((>(a0)(5)))(0)<f2>)
<f2>	1st	(if((>(a0)(5)))(0)(<f10>))
<f10>	2nd	(if((>(a0)(5)))(0)(1))

Fig. 2. Derivation of a Decision Rule

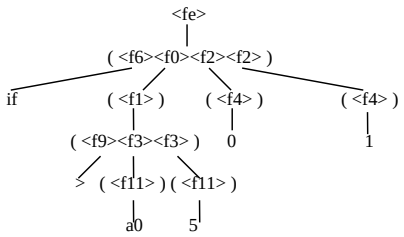

Fig. 3. Tree for Rule: (if((>(a0)(5)))(0)(1))

a maximum allowable depth is chosen, and whenever a random tree is generated that is larger, it is thrown out and replaced. The depth can be thought of as a measure of the complexity of the tree (or strategy).

4.4 Genetic Operation – Crossover

The crossover operation first selects two rules to be parents. It then randomly chooses one of the non–terminal nodes in the first parent, finds all identical nodes in the second parent and uniformly chooses one of these. It then cuts the two subtrees at these nodes, swaps them and recombines the subtrees with the parent trees. By cutting and swapping at the same nodes, the crossover operation ensures that the resulting recombined trees are always syntactically

(and semantically) valid programs. If the crossover operation results in a tree that exceeds the maximum depth, the tree is discarded and crossover is repeated until two valid trees result or until a maximum number of attempts is exceeded.

As an example, consider again the rule derived above, and also consider another rule for the second parent, say, if T is less than 30, then accept if the current offer is less than seven and reject otherwise, else reject. The trees are shown in Figures 4 and 5.

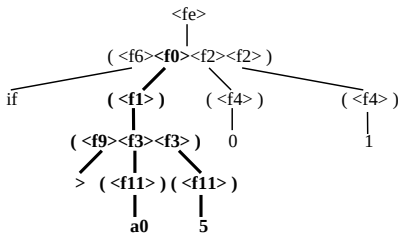

Fig. 4. Parent 1 for Crossover: (if((>(a0)(5)))(0)(1))

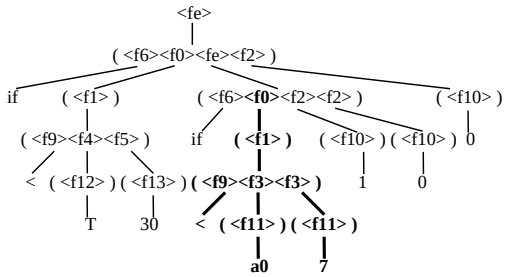

Fig. 5. Parent 2 for Crossover: (if((<(T)(30)))(if((<(a0)(7)))(1)(0))(0))

To illustrate crossover, suppose we randomly choose the first node $\langle f0 \rangle$ in the second level of the first parent tree. We then have to choose an $\langle f0 \rangle$ in

the second parent, so suppose we take the $\langle f0 \rangle$ in the third level of the second parent tree. Both nodes are highlighted in the Figures 4 and 5. Next, follow these non–terminal $\langle f0 \rangle$ nodes in each parent until their paths terminate at terminal nodes. These are the subtrees that will be swapped between the parents to create new offspring, or children. One of these children is illustrated in Figure 6: Note that the new strategy is quite different from the

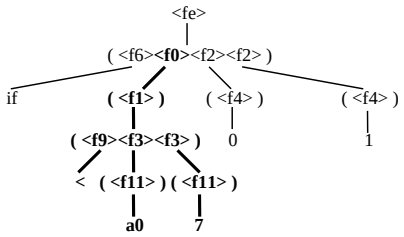

Fig. 6. Child 1 from Crossover: (if((<(a0)(7)))(0)(1))

parent strategy (parent 1) from which it came, as this new strategy instructs the player to reject if the current offer is less than 7.

The internal representation of the crossover operation is provided in appendix A3.

A few genetic programs also include a mutation operation that is in addition to the crossover operation. However, as Koza[4] has pointed out, the position–independence of subtrees in genetic programming would seem to obviate the need for a separate mutation operation; in effect, the crossover operation by itself serves as a kind of macromutation. Indeed, Koza[4] shows that the addition of a separate mutation operator does not lead to any substantial improvement in the performance of genetic programming. Following Koza and most other genetic programming researchers, we chose not to use a mutation operation in our application of genetic programming.

4.5 Evaluation of the rule

The decision in LISP form must be interpreted in order to determine what decisions it makes when playing the game. This is accomplished by recursively calling an evaluation function which begins returning values as it reaches terminal nodes. The details of this part of the program are available on request.

5 The Algorithm

The following algorithm is used in our application of symbolic regression to the ultimatum game data.

1. Randomly generate a population of n rules.
2. Play the rules against the same opponents that each experimental subject faced and evaluate their fitness. If stopping criterion is met, then stop.
3. Choose k rules to survive to the next generation, with the probability of survival proportional to relative fitness.
4. Choose $n-k$ parent rules on which to perform crossover with probability of being chosen proportional to relative fitness. The resulting recombined decision rules are the parent's offspring or "children."
5. The next "generation" of rules is changed to include only the k survivors and the $n-k$ children. Go to step 2 and repeat steps 2–5.
6. End the algorithm after a maximum number of generations, or if a perfect fitness score is achieved.

Note that a genetic program can be seen as the piecing together of building blocks, or sub–trees, which contribute a greater than average fitness to the complete rule, and which are small enough to survive crossover. The fact that reproduction and crossover are performed based on relative fitness increases the probability that a better than average fit subtree is used in future generations, and thus help to direct the search, as in Holland's genetic algorithm.

6 Parameters and Fitness Specification

In our regression analysis of the ultimatum game data we used the grammar as described in Figure 1. We considered population sizes of $n = 200$ trees, with a maximum depth of 150. The number of trees that were selected for copying intact into the next generation was $k = 20$ or 10%. The remaining 180 trees were created through crossover alone. The selection of trees to be copied into the next generation as well as the selection of trees for crossover purposes was based on an adjusted and normalized fitness criterion. First, each decision rule was decoded to determine its raw fitness, which is simply the number of actions out of 37 that it correctly predicted for a particular player B. (Recall we are allowing for 3 lagged values and we have 40 observations). Let us denote the raw fitness score of decision rule i at generation t by $f(i,t)$. The adjusted fitness measure of rule i at generation t is given by:

$$af(i,t) = \frac{1}{1+f(i,t)}.$$

This transformation converts the raw fitness value into the $[0,1]$ interval, and also ensures that small fitness differences are sufficiently exaggerated, which

becomes an important issue as the population of decision rules becomes increasingly fit over time. Note too, that this adjustment implies that the most fit rules are now those with the lowest adjusted fitness values. The normalized fitness value takes the adjusted fitness value, and makes it a relative, population–wide measure. The normalized fitness of rule i at generation t is defined by

$$nf(i,t) = \frac{af(i,t)}{\sum_{j=1}^{n} af(i,t)}.$$

The sum of all normalized fitness values is 1. When selection and crossover decisions are made they are made on the basis of this normalized fitness value. In particular, the 10% of strings selected for reproduction as well as the strings selected for crossover purposes are selected randomly (with replacement) with probability that is inversely proportional to normalized fitness values. These fitness measures and methods for determining the next generation from the current generation are standard in the genetic programming literature (see, e.g. [4]).

7 Regression Results for the Ultimatum Game

We selected 8 individual player Bs from the ultimatum game experimental data. The criterion for selecting a particular player B was that the player rejected a Player A offer at least twice in a 40–round game; we focused on these cases, as other cases had too little variation in actions played to detect any meaningful strategy other than "always accept". We present the data for each of these 8 player B subjects in the 8 examples of Table 7 below. In each example, the first line of data reveals the integer amount that a player A proposed to keep for him/herself. The second line represents the player B's actual response: 0 = reject, 1 = accept.

For each player B history, we report the results of our symbolic regression in several ways. First, we provide the binary string strategy that was generated by the best–of–generation rule, along with this rule's raw fitness value and the mean raw fitness value of the population of 200 rules at various generations (iterations of the algorithm). The symbolic regression was stopped after a maximum of 30 generations, or earlier if a perfect raw fitness score was achieved. If there was more than one best of generation rule at each generation that we report, we chose to report the most parsimonious rule, i.e. the rule with the minimum depth. We also provide the best of generation rules themselves, in both symbolic (computer program) form, and in a decoded and reduced ("plain English") form.

Recall that a perfect raw fitness score in this application is 37. This perfect score was achieved (within 30 generations) in only one of the 8 examples, example number 4. In this example the player B's strategy was uncovered to be of the form: reject any proposed split that gives player A more than \$7, otherwise accept. This strategy was uncovered by our genetic program

algorithm after only 7 generations. Of course, this strategy is fairly easy to uncover by carefully examining the history of play for this particular player B. However, the point of our exercise is that we can use our technique to automate this inference process. Furthermore, inference may not be so easy if the player's strategy is not deterministic as in this example, or is time–varying due to learning.

Indeed, time dependence is shown to matter in Example 5, where the minimum depth, best–of–generation rule at the end of 30 generations yielded a raw fitness score of 30. This rule had player B playing the same action played in the previous period for the first 10 periods of the game. Following period 10, the strategy was to always accept any player A proposal. This rule, of course, misses the player B's rejections of player A offers of 8 following period 10, but it is able to detect that a change has occurred in the proposals being made by player As starting around period 10 (in this case, it is a change in the magnitude of player A proposals, which become smaller). Of course, if we allowed for a more continuous dimension for time T this rule might be further improved. Note that the $T < 10$ substring survives in the best of generation string for some time (the last 10 generations). The survival of this substring illustrates the notion that highly fit building blocks are more likely to be chosen over time. The actual strategy of the player B in example 5, like the rule in example 4, is a deterministic rule: a close study of the history of play of player B in example 5 reveals that this player always rejected proposals that were greater than 7. While our algorithm was not able to detect this rule in 30 generations with a population of 200 strings, when we increased the population size to 500 strings, we were able to perfectly uncover this player B's deterministic rule in 10 generations.

The other 6 examples in Table 7 yield best–of–generation rules that are more difficult to interpret, with ending raw fitness scores of 30 or less. We see in these rules that time conditioning is of some importance. For instance, in Example 6, the best of generation rule at the end of the simulation run (30 generations) is able to detect a break in player B behavior around period 20, and the rule conditions its actions on this break–point. We also see the survival over many generations of highly fit substrings in the best of generation rules. For example, the substring "if accepted at $t-3$" appears in the minimum depth, best fit rule of generations 10, 20 and 30 in Example 3 as does the substring "if $t > 35$" in Example 8. The survival of such highly fit substrings would be more apparent if we reported all of the rules, not just the minimum depth, best–of–generation rule. Finally, we see a lot of strategies that condition on the actions played by the player B in the past, as well as on the proposals made by player As in the past, rather than on the current $t = 0$ proposal. While the responders in the experiment may have been conditioning on this type of information, – they did have the past history of 10 rounds of play on their computer screens – we suspect that what the genetic program is actually doing is using conditioning on past actions as a means

of implementing probabilistic choices. In our current implementation, there is no randomization mechanism; therefore, the genetic program may have to invent such a mechanism in order to advance the search for more fit strings. We plan to implement the possibility of randomized choices in future work, by including in the set of terminals a floating point constant chosen randomly from the set of real numbers on the unit interval.

8 Summary and Conclusions

In this paper we have discussed how genetic programming can be used to conduct a symbolic regression analysis. We have then applied this procedure to experimental data from the repeated ultimatum game. Our aim was to uncover player's strategies from the actions these players played, a perennial inference problem in the analysis of experimental data. We find that our algorithm is frequently capable of uncovering simple, deterministic strategies, but has more difficulty with strategies that appear to be random or time-varying. While simple rules may be detected by careful inspection of the data, the fact that our algorithm can automate this process is an appealing feature. We believe that with further work, as well as with longer simulation run–times, we will be able to make further progress on the recovery of random or time–varying rules.

The main advantage of our approach over standard regression analyses is that we do not have to prespecify the structure of the regression equation. Instead, we only need provide a set of terminals and nonterminals as part of the grammar of some language that is then used to derive decision trees (e.g. LISP). Finally, our technique can be used to assess the degree of homogeneity of strategic play among subjects assigned to play the same role in economic decision making experiments. If strategic behavior is not sufficiently homogeneous, as we have found in our simple examples, then it may not be reasonable to pool data across subjects as is frequently done in analyses of experimental data.

References

1. Duffy, J. and Feltovich, N., Does Observation of Others Affect Learning in Strategic Environments? An Experimental Study, *International Journal of Game Theory*, Vol. 28 pp. 131–152, 1999.
2. Geyer–Schulz, A., Fuzzy Rule–Based Expert Systems and Genetic Machine Learning, Second Ed., Physica–Verlag, 1997.
3. Holland J., Adaptation in Natural and Artificial Systems, University of Michigan Press, 1975.
4. Koza, J.R., Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.

Table 7: Symbolic Regressions on Player B actions in the Ultimatum Game: 8 examples.

Example 1

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	665876665676677667766788667656666656666		
B	10100111100110010001100010011101101110000	best rule fitness	mean fitness
rule			
1	10110000110011011100111011000100100001	25	17.62
10	10110000110011011100111011000100111100	26	19.20
20	00010000010001001100011001000000000000	23	19.67
30	10111000111011011101110111011001101100 1	25	19.71

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

1 (IF(B2)(0)(1))
10 (IF((<(T)(35)))(IF(B2)(0)(1))(B1))
20 (IF(NOT(AND(NOT(NOT(B2)))(NOT(NOT(NOT((=(T)(15))))))))(IF(B3)(0)(1))(IF((>(5)(8)))(0)(0)))
30 (IF(NOT(NOT(NOT(B2))))(1)(IF(NOT(B3))(B1)(IF(B3)(0)(IF(B2)(0)(1)))))

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

```
 1   if accepted at t-2, reject
     else accept
10   if t < 35
          if accepted at t-2, reject
          else accept
     else repeat action at t-1
20   if rejected at t-2 and t is not 15
          if accepted at t-3, reject
          else accept
     else reject
30   if rejected at t-2, accept
     else if rejected at t-3, repeat action at t-1
     else if accepted at t-3, reject
     else accept
```

Results of expanded search with population size 500

(if(not((>(7)(a0))))(if(((>(8)(4))(0)(0))(1)), fitness = 29, generation 21

Example 2

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	6546577776665666656576666556666676666656		
B	11101100011110011111011001100111010010011	best rule fitness	mean fitness
rule			
1	10110111111011111101101111011110111011	24	17.55
10	11100001111111111111111110111111111111	25	20.44
20	11100111111111111111101101111110101011	25	21.56
30	00100111111011111001101100101001011011	27	22.49

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

```
 1 (IF(OR(NOT((<(T)(10)))(NOT(NOT(NOT((=(7)(5)))))))(IF(0)(B1)(IF((<(A0)(A3)))(1)
   (IF(B2)(B1)(1))))(IF(B3)(IF((>(T)(20)))(IF(1)(B2)(1))(B2))(0)))
10 (IF((>(A0)(A2)))(B3)(IF(AND(1)((<(T)(15))))(IF((=(T)(30)))
   (IF(NOT(AND((=(A1)(7)))(OR((<(A0)(A3)))(AND((=(T)(10)))(1)))))(1)(B2))(B1))(1)))
20 (IF((>(A0)(A2)))(B3)(IF(NOT(NOT(B2)))(IF((=(T)(30)))(IF(NOT((>(A2)(A0))))(1)(B2))(B1))(1)))
30 (IF((=(T)(30)))(IF(AND((=(T)(10)))((<(T)(20))))(B1)(B3))(IF(B2)(IF(B2)
   (IF(NOT((>(A0)(A2))))(B1)(0))(B1))(IF((<(T)(5)))
   (IF(AND(NOT(NOT(NOT(B3))))(AND((<(A3)(A2)))(OR(NOT(0))(AND(B3)(B3)))))(0)(B1))(1))))
```

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

```
 1   if offer at t-0 > offer at t-3, accept
     else if accepted at t-2, repeat action at t-1
     else accept
10   if offer at t-0 > offer at t-2, repeat action at t-3
     else if t < 15, repeat action at t-1
     else accept
20   if offer at t-0 > offer at t-2, repeat action at t-3
     else if accepted at t-2
          if t = 30
               if offer at t >= offer at t-2, accept
               else repeat action at t-2
          else repeat action at t-1
30   if t = 30, repeat action at t-3
     else if accepted at t-2
          if not offer at time t > offer at time t-2, repeat action at t-1
          else reject
     else if t < 5
          if rejected at t-3 and offer at t-3 < offer at t-3, reject
          else repeat action at t-1
     else accept
```

Results of expanded search with population size 500

no improvement over original search

Example 3

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	566797772977763757778888778177886878777		
B	11000111001111111111000111011110010101 11	best rule fitness	mean fitness
rule			
1	00011100111111111100011111111110010101	27	20.32
10	00111100111111111110011110111100111111	28	23.52
20	00111101111111111110011110111100111111	29	24.50
30	01111101111111111110111110111101011111	29	25.63

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

```
 1 (IF((>(5)(A0)))(1)(IF(NOT((>(T)(30))))(IF((=(T)(25)))(B1)(IF(1)(B1)(0)))(B2)))
10 (IF(B3)(B1)(1))
20 (IF(OR(0)(B3))(IF((<(A2)(6)))(IF(1)(IF(AND(B1)((<(A0)(A1))))(1)(1))(1))(B1))
   (IF(0)(0)(IF((>(7)(7)))(IF((=(T)(20)))(B2)(0))(1))))
30 (IF(B2)(IF(B3)(IF(AND(0)(B3))(B3)(B1))(1))(IF((=(A1)(6)))(B3)(1)))
```

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

```
 1   if offer at t-0< 5, accept
     else if t <=30, repeat action at t-1
     else repeat action at t-2
10   if accepted at t-3, repeat action at t-2
     else accept
20   if accepted at t-3
          if offer at t-2 < 6, accept
          else repeat action at t-1
     else accept
30   if accepted at t-2
          if accepted at t-3, repeat action at t-1
          else accept
     else if offer at t-1 = 6, repeat action at t-3
     else accept
```

Example 4

Data/rule comparisons in generations 1, 6, & 7

subject			
A	95767669778667977784777888777787674677 86		
B	01111110110111011011110001110111111101	best rule fitness	mean fitness
rule			
1	11111111111111111111111111111111111111	28	22.18
6	10110001111001101110000111101111111001	30	25.95
7	11110110111011101110001110111111101	37	26.96

Rule expressions (minimum depth among best fit in generations 1, 6, & 7)

```
1 (IF(NOT(0))(1)(1))
6 (IF((>(A0)(A3)))(0)(1))
7 (IF((>(A0)(7)))(0)(1))
```

Minimum depth best fit rule interpretations in generations 1, 6 & 7

```
1   always accept
6   if offer at time 0 > offer at time t-3, reject
    else accept
7   if offer at time 0 > 7, reject
    else accept
```

Example 5

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	667659899667777876677874778786778777777		
B	11111000011111101111101111011011101111111	best rule fitness	mean fitness
rule			
1	11110001111111111111111111111111111111	29	23.11
10	11100001111111111111111111111111111111	30	27.08
20	11100001111111111111111111111111111111	30	27.62
30	11100001111111111111111111111111111111	30	27.65

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

```
 1  (IF(NOT(B1))(IF(B1)(B1)(B2))(IF(B1)(1)(1)))
10  (IF(NOT(0))(IF(NOT(B1))(IF((>(7)(A3)))(B1)(B3))(1))(IF(NOT(B1))(B1)(1)))
20  (IF(0)(1)(IF((<(T)(10)))(B1)(1)))
30  (IF((<(T)(10)))(IF(OR(OR(1)((=(T)(25))))(NOT(B1)))(B1)(1))(1))
```

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

```
 1      if rejected at t-1, repeat action at t-2
        else accept
10      if rejected at t-1
                if offer at t-3 > 7, repeat action at t-1
                else repeat action at t-3
        else accept
20      if t < 10, repeat action at t-1
        else accept
30      if t < 10, repeat action at t-1
        else accept
```

Results of expanded search with population size 500

(if(not((>(8)(a0))))(0)(1)), fitness = 37, generation 10

Example 6

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	576977777669839688777766777876775787758		
B	10100000111001010011111111111111110000010	best rule fitness	mean fitness
rule			
1	00000001100000000011111111111111000000	28	22.79
10	00000011101101001111111111111111000011	29	24.91
20	00000001101001011111111111111111010000	29	25.42
30	00000011000100001111111111111111000000	30	26.02

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

1 (IF((>(A2)(A3)))(B2)(IF(B1)(B2)(B1)))
10 (IF((=(A0)(7)))(IF((<(4)(A3)))(B1)(B3))(IF(B2)(IF(1)(IF(B2)(B1)(0))(B1))(IF((<(T)(5)))
(IF((>(T)(10)))(IF((=(T)(30)))(IF(NOT(1))(1)(IF(NOT(1))
(IF((>(T)(25)))(B1)(B3))(B3)))(0))(0))(1))))
20 (IF((=(8)(A3)))(IF((<(4)(A3)))(B1)(B3))(IF(B2)(IF(B2)(IF(B2)(B1)(0))(B1))(IF(NOT((>(T)(10))))
(IF(B2)(B1)(0))(IF(NOT((<(T)(10)))))(B3)(B3)))))
30 (IF((<(T)(20)))(IF(NOT(AND(NOT(1))(AND((=(T)(5)))(OR(AND((>(7)(5)))(1))(AND(B3)(B2))))))
(IF(B3)(IF((>(6)(5)))(0)(B2))(B1))(B2))(IF(B2)(IF(B2)(B1)(0))(IF(B2)(B1)(0))))

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

```
 1   if offer at t-2 > offer at t-3, repeat action a t -2
     else if accepted at t-1, repeat action at t-2
     else repeat action at t-1 (reject)
10   if offer at t = 7
          if offer at t-3 < 4, repeat action at t-1
          else repeat action at t-3
     else if accepted at t-2, repeat action at t-1
     else if t < 5, reject
     else accept
20   if offer at t-3 = 8, repeat action at t-1
     else if accepted at t-2, repeat action at t-1
     else if t <= 10
          if accepted at t-2, repeat action at t-1
          else reject
     else repeat action at t-3
30   if t < 20
          if accepted at t-3, reject
          else repeat action at t-1
     else if accepted at t-2, repeat action at t-1
     else reject
```

Results of expanded search with population size 500

no improvement over original search

Example 7

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	54786696779777627788757788770877877878877		
B	11101101110111111100111100111011011010111	best rule fitness	mean fitness
rule			
1	11111111111111111111111111111111111111	26	20.39
10	11111111111111111111111111111111111111	26	24.26
20	11111111111111101111101111111111111111	28	25.12
30	11111111111111111111111111111111101111	27	25.28

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

1 (IF(0)(B1)(1))
10 (IF(1)(1)(1))
20 (IF((<(A0)(8)))(1)(IF((<(8)(5)))(1)(IF(NOT((=(5)(8))))(B1)(B1))))
30 (IF(1)(IF(AND(NOT(NOT(AND(NOT(NOT(AND((=(T)(35)))(B1))))(B1))))((>(T)(5))))
(IF((<(A3)(A1)))(1)(IF(NOT((<(A0)(4))))(IF((<(8)(5)))(B1)(IF((<(T)(15)))(B1)(B3)))(B2)))(1))
(IF(AND(NOT(AND(1)(NOT((<(T)(10))))))(NOT((<(8)(5))))(1)(B3)))

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

1 always accept
10 always accept
20 if offer at t < 8, accept
else repeat action at t-1
30 if t = 35 and accepted at t-1
if offer at t-3 < offer at t-1, accept
else if offer at t >= 5, repeat action at t-3
else repeat offer at t-2
else accept

Results of expanded search with population size 500

(if((>(a0)(7)))(if((<(7)(a0)))(0)(b1))(if(0)(b1)(1))), fitness = 37, generation 6

Example 8

Data/rule comparisons in generations 1, 10, 20, & 30

subject			
A	5696746586546556565756767666666665666666		
B	10011111011101111111011010111100011001 00	best rule fitness	mean fitness
rule			
1	11111111111111111111111111111000000000	28	20.28
10	11111111111111111111111111111111111100	27	23.60
20	11111111111111111111111111111111110000	27	23.82
30	11111111111111111111111111111111110000	27	24.22

Rule expressions (minimum depth among best fit in generations 1, 10, 20, & 30)

```
 1 (IF((>(T)(30)))(0)(1))
10 (IF((>(T)(35)))(B3)(1))
20 (IF((>(T)(35)))(0)(1))
30 (IF((>(T)(35)))(IF(OR(NOT(NOT((<(T)(5)))))((<(A3)(7))))
   (IF(NOT(OR(NOT(NOT(1)))(OR(NOT((>(8)(A1))))(OR(NOT(NOT((>(7)(4)))))
   (OR(1)((<(A3)(7)))))))(1)(0))(1))(1))
```

Minimum depth best fit rule interpretations in generations 1, 10, 20, & 30

```
 1   if t > 30, reject
     else accept
10   if t > 35, repeat action from t-3
     else accept
20   if t > 35, reject
     else accept
30   if t > 35
          if offer at t-3 < 7, reject
          else accept
     else accept
```

Results of expanded search with population size 500

no improvement over original search