“Implementation of a Genetic Programming System in a Game-Theoretic Context”

95-01-02

Garett Dworman, Steven Kimbrough and James Laing
IMPLEMENTATION OF A
GENETIC PROGRAMMING SYSTEM
IN A GAME-THEORETIC CONTEXT

Garett Dworman
Steven O. Kimbrough
James D. Laing

95-01-02

Department of Operations and Information Management
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104-6366
Implementation of a Genetic Programming System in a Game-Theoretic Context

Garett Dworman
dworman@opim...

Steven O. Kimbrough
kimbrough@

James D. Laing
laing@

University of Pennsylvania
The Wharton School
3620 Locust Walk, Suite 1300
Philadelphia, PA 19104-6366
...@wharton.upenn.edu

I. Overview

This working paper records the high level design and certain implementation details the research and software described in Dworman, Kimbrough, and Laing (1995). This system is a revision and extension, in the context of a cooperative game with sidepayments, of the software originally developed by James Rice and John R. Koza (Koza, 1992). We assume that the reader is familiar with genetic programming.

The next section discusses the environment for which we evolve decision strategies. This environment is a cooperative game with sidepayments modeled by a characteristic function. The third section describes the representation of the agents that play this game. The fourth and fifth section describes how the agents’ strategies are created and then evolved. Section six discusses how we evaluate strategies in a competition amongst the different agents. Section seven covers the reports produced by our code and the statistics presented in them. Finally, section eight discusses how we edit strategies when reporting them. An appendix summarizes the parameters discussed in the paper.

II. Simulation of the agents’ environment

Our research investigates a cooperative game with sidepayments. This type of game may be represented by a characteristic function, \( \nu \), that specifies for each coalition \( S \) of agents the total payoff, \( \nu(S) \), that is to be divided among members of that coalition if it forms. In the game addressed below, all payoffs are integers (points), just three agents (denoted A, B, and C) are involved, and only two-agent coalitions have nonzero value: \( \nu(AB)=18 \), \( \nu(AC)=24 \), and \( \nu(BC)=30 \). The game involves negotiations among the three agents through which an agreement is reached by two of them on a two-way split of their coalition’s value. The third agent wins zero.

The process of the negotiation games is governed by formal rules embodied in the following mechanism. The mechanism awards the floor to one of the three agents to initiate negotiations. Technically, the mechanism makes this agent a null offer. If the agent accepts, the

---

1 We gratefully acknowledge the Research Foundation of the University of Pennsylvania, which supported this work in part. File: GPWP01.MAC. Revision: 18 January 1994. This paper is accessible on the WWW at http://opim.wharton.upenn.edu/users/sok/comprats/ourdocs.html
game ends and all three agents win zero payoff (this situation is dealt with specially when calculating statistics – see Data report later). Otherwise, the initiator selects a potential coalition partner as responder and proposes a two-way split of their coalition’s value. If the responder accepts, the game ends, and the coalition partners win the payoffs as specified in the agreement, while the third agent wins zero. If the responder does not accept, then it becomes the initiator and makes a proposal, tacitly rejecting the previous offer.² The game continues in this fashion until an agreement is reached (i.e., an agent returns ACCEPT – see Representation of strategies later), or until a specified maximum number of offers have been rejected, in which case the mechanism ends the game by default, and awards all three agents a payoff of zero.

The parameter max-round-num specifies the maximum number of offers that can be rejected before the game ends by default. Currently, it is set to 6. The restriction to 6 offers per game is not very limiting. In our simulations, almost all agreements are reached within 3 offers. Initially, we limited games to 10 offers, but discovered that any game that went beyond 5 offers continued until the mechanism ended the game by default. Therefore, we lowered the limit to 6 to speed up the simulations.

All offers are recorded in a log. When an agent attains the floor, the most recent offer on the log is copied into that agent’s message board. This message board contains all the information available to strategies. We chose this mechanism for its generality. If we wish to change the information available to the agents, we only need to change what is transcribed to the message board. In the current simulations the only information in the message board, and therefore the only information available to the agents’ strategies, is the most recent offer made to the agent. The message board and the log are empty for the first round initiator.

Offers in the log and message board consist of a Lisp list structure. Each element in the list is a cons whose car indicates an agent in the proposed coalition and whose cdr indicates the amount that that agent would receive in the proposal. The ordering of the list elements is important. The first element is the initiator’s and the second is the responder’s. For example, if agent A proposed the coalition {A, C} with payoffs (20, 4), then this offer would be represented on the log and message board as follows:

( (A 20) (C 4) )

The agent who is not included in the offer’s proposed coalition does not have a corresponding element in the offer’s list. Agents who are part of the proposed coalition but are offered zero have an element whose cdr is zero. This representation of an offer distinguishes between an agent who is not made an offer (and would thus receive zero) and an agent who is made an offer of zero.

III. Representation of agents’ strategies

An agent is represented by a population of complete strategies. Each strategy completely determines an agent’s behavior in all situations. A strategy is a genetic programming Lisp tree (s-expression). As per any genetic programming system, strategies are determined by three components: a terminal set to define the symbols that comprise the leaf nodes of the strategy trees,
thereby determining how strategies assess the state of the game and what output they can produce: a function set to specify the class of operators to be permitted as nonterminal nodes in the program trees, thus determining which functions the strategies can employ; and, a program syntax to constrain (but not determine) the structure of the program trees to assure meaningful, and executable, strategies.

A. The terminal set

Terminal set = { <condition>, <offer>, ACCEPT }.

The ACCEPT symbol signals an agent’s acceptance of an offer. The <condition> and <offer> symbols are complex data structures (CDS) that have multiple fields. The offer CDS specifies a proposal to another agent. The condition CDS represents a set of states that may have been proposed to the agent. When evaluated, a strategy returns either the ACCEPT symbol or an offer CDS.

The condition and offer CDSs have the following fields:

Condition CDS:
  Agent = specifies an agent
  Lower bound = specifies a lower bound on the specified agent’s payoff
  Upper bound = specifies an upper bound on the specified agent’s payoff

Offer CDS:
  Agent = specifies an agent
  Amount = specifies an amount being offered to the specified agent.

The fields in the condition and offer CDSs may only have explicit values; don’t cares are not used. However, note that a value of 0 in the lower bound field of a condition CDS acts like a don’t care. Similarly, a value of \( v_{ij} \) in the upper bound field of a condition CDS also acts like a don’t care. The lower bound, upper bound, and the amount fields are represented in unsigned binary code.

A.1 The condition CDS

The condition CDS terminals specify a condition against which the current offer is tested. A condition CDS evaluates to TRUE iff, in the current proposal, the agent specified by the agent field would receive a payoff between lower bound and upper bound (inclusive). Note that agents who are not in the current proposal are implicitly offered 0. Hence, a condition CDS that specifies such an agent with a lower bound of 0 would evaluate to TRUE.

For example, suppose B offers 5 points to A. Taking \( v(AB) = 18 \) into account, the mechanism would record this in the log as ((B 13) (A 5)). Then the following conditional CDSs in A’s rules would evaluate as follows:

(A 5 14) = TRUE: A’s proposed payoff of 5 points lies in the closed interval [5, 14].
(B 2 10) = FALSE: B’s payoff of 13 would exceed the Upper Bound.
(C 0 12) = TRUE: C would get 0 (implicitly).

On the first-round initiation the message board consists of an empty list. Therefore, all agents are assumed to be receiving zero. Therefore, only those condition CDSs that specify lower bounds of zero will match. All other condition CDSs will fail.
A.2 The offer CDS

When a strategy returns an offer CDS, it is tacitly rejecting the previous offer and making a counteroffer to the agent indicated in the agent field. The amount field of the offer CDS indicates the amount the other agent would receive if the counteroffer is accepted. The agent's own amount is the rest of the coalition value. Therefore, if agent C makes the offer \{A 10\}, then the mechanism records this as the proposal ((C 14)(A 10)) in the log.

The agent field must specify one of the other agents in the game; it cannot be the agent's own ID. Agents can not make offers to themselves.

B. The function set

\(Fs = \{\) IF-THEN-ELSE (condition-arg, then-arg, else-arg) (action node)
AND (arg1, arg2) (conditional node),
OR (arg1, arg2) (conditional node),
NOT (arg1, arg2) (conditional node)\}

The IF-THEN-ELSE function first evaluates its condition-arg. If this returns TRUE the then-arg is evaluated; otherwise the else-arg is evaluated. The IF-THEN-ELSE function returns whatever its then-arg or its else-arg returns.

The AND function evaluates its arguments in left-to-right order. As soon as any argument returns FALSE the AND function returns FALSE; otherwise the AND function returns TRUE.

The OR function also evaluates its arguments in left-to-right order. As soon as any argument returns TRUE the OR function returns TRUE; otherwise the OR function returns FALSE.

The NOT function inverts its first argument. If the first argument is TRUE, NOT returns FALSE; if the first argument is FALSE, NOT returns TRUE. The second argument is ignored; it is provided only to provide closure under the mutation operator.

C. Program syntax

Strategy trees consist of three different node types. A node type is defined by the symbols (functional and terminal) that may be assigned to that position. The different node types and the symbols they allow are summarized in the table below. Maintenance of node types guarantees closure (under crossover and mutation) even though different data types are used: subjecting legal strategies to genetic operators always yield legal strategies if node types are preserved.\(^3\)

<table>
<thead>
<tr>
<th>Node type</th>
<th>Accepted function symbols</th>
<th>Accepted terminal symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>IF-THEN-ELSE</td>
<td>(none)</td>
</tr>
<tr>
<td>Conditional</td>
<td>AND, OR, NOT</td>
<td>condition</td>
</tr>
<tr>
<td>Action</td>
<td>IF-THEN-ELSE</td>
<td>offer, ACCEPT</td>
</tr>
</tbody>
</table>

\(^3\) For a discussion of the need to preserve closure see Koza (1992, p. 81).
The root node of any tree is a root node type (i.e., it must be the IF-THEN-ELSE function). The node types of the rest of the tree are determined by the figures below. That is, the conditional-arg of the IF-THEN-ELSE function is a conditional node and the then-arg and the else-arg are action nodes. The arguments for the AND, OR, and NOT functions are conditional nodes.

Below are some examples of valid strategies:

(IF-THEN-ELSE \{A 10 13\} ACCEPT\{B 5\})
(IF-THEN-ELSE (OR \{A 10 13\} \{B 5 10\}) ACCEPT \{B 5\})
(IF-THEN-ELSE \{A 4 4\}
  (IF-THEN-ELSE \{A 13 18\} \{C 7\} \{B 9\})
  ACCEPT)
(((IF-THEN-ELSE (AND \{C 8 16\} \{B 0 7\})
  ACCEPT
  (IF-THEN-ELSE \{A 23 23\} \{C 14\} ACCEPT)))
(IF-THEN-ELSE (AND (AND \{A 15 18\} \{C 16 18\})
  \{B 0 16\})
  (IF-THEN-ELSE \{A 5 18\}
    (IF-THEN-ELSE \{A 23 24\} \{C 5\} \{C 1\})
    (IF-THEN-ELSE \{A 8 23\} \{C 17\} ACCEPT))
  (IF-THEN-ELSE (OR \{C 1 13\} \{C 21 21\})
    (IF-THEN-ELSE \{A 0 19\} ACCEPT\{C 19\})
    (IF-THEN-ELSE \{A 0 8\} \{B 0\} ACCEPT))

(IF-THEN-ELSE (NOT (AND \{C 9 17\} \{A 19 22\})
  (OR \{B 1 14\} \{A 19 24\}))
  (IF-THEN-ELSE (NOT \{A 21 23\} \{C 4 5\})
    (IF-THEN-ELSE \{A 20 21\} ACCEPT \{C 3\})
    (IF-THEN-ELSE \{A 3 13\} \{B 9\} \{B 18\}))
  (IF-THEN-ELSE \{A 4 4\}
    (IF-THEN-ELSE \{A 13 18\} \{C 7\} \{B 9\})
    ACCEPT))

D. Tree validity

In addition to proper syntactic structure, a tree must have valid values in the CDSs. There are three validity checks that must be maintained. These checks are listed below.

In the statement of these validity checks (and in the rest of this document) the term \(v_{ij}\) denotes the value of the coalition between agents i and j. Agent i is the agent for whom the rule has been created. Agent j is the other agent in the coalition. For a condition CDS, agent j is the agent who made the offer to agent i. For an offer CDS, agent j is the one to whom agent i is initiating a proposal.
1) In condition CDS, the lower and upper bound fields must satisfy $0 \leq \text{lower bound} \leq \text{upper bound} \leq v_{ij}$.

2) In offer CDSs, amounts must be non-negative and must not exceed the $v_{ij}$ of the proposed coalition.

3) In offer CDSs, the agent field cannot be the agent itself — i.e., An agent may not propose to itself.

These checks are performed automatically during the creation and breeding of new strategies.

**IV. Creation of strategies**

**A. Tree creation**

When creating a new tree, the creation mechanism is careful to use only symbols that are legal for each node according to the node's type. This is done generically so the creation mechanism will work if we redefine the terminal set, function set or the node type definitions. Within the constraints of the program structure, strategy trees are created randomly using the ramped-half-and-half method in the code by James Rice and John Koza (Koza, 1992, p. 93).

**B. Leaf creation**

Leaf creation consists of either selecting a terminal symbol (e.g., ACCEPT) or constructing an appropriate CDS and filling in the fields with random legal values. The table below summarizes the legal values for each field.

<table>
<thead>
<tr>
<th>Field</th>
<th>Legal values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Condition CDS</strong></td>
<td></td>
</tr>
<tr>
<td>Agent field</td>
<td>{A,B,C}</td>
</tr>
<tr>
<td>Lower bound field</td>
<td>$\text{Integer} \in [0, v_{ij}]$</td>
</tr>
<tr>
<td>Upper bound field</td>
<td>$\text{Integer} \in [\text{lower bound}, v_{ij}]$</td>
</tr>
<tr>
<td><strong>Offer CDS</strong></td>
<td></td>
</tr>
<tr>
<td>Agent field</td>
<td>{A, B, C} (but may not be the agent's own id.)</td>
</tr>
<tr>
<td>Amount field</td>
<td>$\text{Integer} \in [0, v_{ij}]$</td>
</tr>
</tbody>
</table>

**V. Breeding strategies**

The breeding algorithm is described in pseudocode below. Note that we have introduced two new genetic operations: Leaf-crossover and leaf-mutation. The normal crossover and
mutation operators treat nodes as wholes and cannot manipulate the fields inside a CDS. The leaf-crossover and leaf-mutation operations were created to evolve the CDS’ internal values. System parameters are summarized in the appendix.

**Breed-population (old-population)**

number-reproduced = 0
repeat until now population is full
iC = a copy of a strategy selected probabilistically based upon fitness
if (number-reproduced < *crossover-fraction*)
then /* perform reproduction */
  if (random() > *mutation-probability*)
  then mutate(iC)
  if (random() > *leaf-mutation-probability*)
  then leaf-mutate(a random cds in iC)
pout iC into new population
increment number-reproduced
else /* perform crossover */
  jC = a copy of a selected individual
  subtreei = any subtree from iC
  subtreej = any subtree from jC of the same node type as subtreei
  crossover(iC, jC, subtreei, subtreej)
  if (random() > *mutation-probability*)
  then mutate(subtreei)
  if (random() > *mutation-probability*)
  then mutate(subtreej)
  cdsi = a random CDS in subtreei
  cdsj = a random CDS in subtreej of the same node type as cdsi
  if (random() > *leaf-crossover-probability*)
  then leaf-crossover(cdsi, cdsj)
  if (random() > *leaf-mutation-probability*)
  then leaf-mutate(cdsi)
  if (random() > *leaf-mutation-probability*)
  then leaf-mutate(cdsj)
pout iC into new population
if room, put jC into new population
end if
end repeat
end Breed-population

We have also introduced a new tree mutation mechanism. The mechanism in the code by James Rice and John Koza (Koza, 1992, p. 105) simply picks a node, destroys the subtree at that node, and inserts a randomly created subtree. Our mutation is less destructive; it changes only the symbol at the selected node, not the entire subtree under the node.
Mutate(tree)
node = a random node in tree
if (node is a non-terminal node)
    then change the function symbol at the node preserving the
    original node type
else change the terminal node preserving the original node
type. When changing a terminal node into a CDS, then
randomly create a new CDS from scratch.
return the altered tree
end Mutate

The leaf-crossover and leaf-mutate procedures assume that appropriate CDSs have already
been selected and the decision to perform leaf-crossover or leaf-mutation has been made.

Leaf-crossover (cds1, cds2)
repeat for *max-number-of-leaf-xover-attempts*
    Choose a random field to be altered
    Choose a random binary crossover point in that field
    Swap the bits to the right of the crossover point of the
    CDS's chosen field
    if (new CDSs have legal values) then return new CDSs
    else restore original values
end repeat
return original CDSs.
end Leaf-crossover.

Leaf-mutation (cds)
repeat for *max-number-of-leaf-mutation-attempts*
    Choose a random field to be altered
    Toggle a random bit in that field
    if (new CDS has legal values) then return new CDS
    else restore original values
end repeat
return original CDS
end Leaf-mutation

VI. Fitness evaluation

The fitness of each strategy is determined through a round robin tournament. Ideally,
every strategy of each agent would play every combination of the other two agents' strategies.
Moreover, each triad of strategies would be played three times to let each agent begin negotiations
in one game. Unfortunately, unless the populations are small, this ideal is not practical.
Therefore, we must compromise. Two parameters determine the round robin's structure:

*play-triad-three-times* If t then each selected triad will play three times,
allowing each agent to be the first-round initiator for
one game. Nil means play each triad only once,
choosing the first-round initiator randomly.

*number-of-trials* This parameter determines the exact number of
opponent strategy combinations each strategy is to
play. A value of 0 means all combinations are to be
played. NB: the number of combinations need not
equal the number of games. If *play-triad-three-times* is t then the number of games is three times the number of combinations.

The tournament is performed as follows: If *number-of-trials* is 0, then play every possible combination of the three agents' strategies. Otherwise, each strategy must play against exactly *number-of-trials* combinations of opponents. The combinations are selected by choosing strategies without replacement from simulated urns. Each urn represents a different agent and contains *number-of-trials* representations of each strategy. A combination is then a set of three strategies, each picked from a different urn. This technique guarantees that every strategy of every agent plays exactly *number-of-trials* combinations. Unfortunately, this technique does not guarantee that each combination is unique. However, the odds of any combination being repeated are very small – with a population size of 50 strategies per agent, there are $50^3$ different combinations.

If *play-triad-three-times* is t, then each chosen combination is played three times allowing each agent to begin negotiations. Otherwise, play each combination only once, selecting a random agent to initiate first. Note that if *play-triad-three-times* is t, then each strategy plays three times *number-of-trials* games.

**VII. Reports**

There are three reports produced by our system. Each report is a pure ascii text file.

**A. Data report**

The data report prints out the run's statistics every n generations (n = *num-generations-between-data-reports*) and when the simulation ends. If n = 0 (i.e., *num-generations-between-data-reports* = 0), then the population report is printed out every generation. Statistics are not calculated every generation, but only when the data report is printed. The first line of the data report contains headings for the statistics contained in the file. Each subsequent line represents the statistics for one generation.

There is one special situation that must be accounted for when calculating statistics. Often a game is played in which an agent never gets the floor so that the agent's strategy is never evaluated during the game. There are two situations in which this can happen. First, when the first-round initiator 'accepts' rather than makes an offer, the game ends by default, and all agents receive 0 for that game. In this case the other two agents never have a chance to respond. Second, two agents may negotiate with each other and completely ignore the third agent.

If a strategy is not evaluated during a game, then that game's outcome conveys no information about the strategy. Therefore, we do not count that game when calculating statistics for the strategies of the non-participating agents. Consider the following extreme (but not entirely unlikely) scenario. Agents A and B accept any offer. Agent C always requests 15 points for itself. In this scenario, whenever agents A and B initiate in the first round the game ends in default with all agents receiving 0. Whenever agent C initiates, it receives 15 points. What should C's average points per game be? If all games are counted, then C receives 15 in a third of the games and its average is 5 points. However, C never has a chance to play in the other 2/3 of the games. In all the games in which C actually participates it receives 15 points. Hence, C's average should be 15.

Before listing the statistics, we provide the following definitions:
An agent’s **best** strategy is that strategy that attained the highest average payoff value during the current generation.

An agent's **median** strategy is that strategy whose average payoff value during the current generation was the median of all that agent’s strategies.

**Average payoff value**: The average payoff in all the games in which the strategy’s agent participated.

**Average coalition value**: The sum of all points the strategy attained during those games in which it was a member of the final coalition, divided by the number of games in which it participated.

**Will-accept values**: Each strategy has two will-accept values: one for each of the opposing agents. These values are the most an opponent can get from the agent when proposing to the agent. For example, assume a strategy for agent A has the following will-accept values: (B 12) and (C 15). This means that this strategy will not accept any offers made to agent A by agent B where agent B receives more than 12 points; and this strategy will not accept any offers made to agent A by agent C where agent C receives more than 15 points. The will-accept values are used to calculate the opportunity and coalition opportunity costs.

**Opportunity cost**: The maximum number of points the opposing agents would have granted the agent by accepting an offer minus the number of points the agent actually received, averaged across all games in which the agent participated. Note that opportunity cost can be negative. This occurs when an opponent’s will-accept value for the agent is less than the amount the opponent actually offers the agent (who accepts). For example, agent B may offer 15 to agent C but would never accept more than 12 if C initiated to B. Agent C’s opportunity cost for accepting agent B’s offer is 12 - 15 = -3.

**Coalition opportunity cost**: The same as opportunity cost except it is only calculated for those games in which the agent was a member of the final coalition.

The following statistics are printed once each time statistics are reported.

- Current generation number.
- Average length of each game in the current generation (e.g., the number of offers made before the game ended).
- % of games that resulted in a coalition.

The following statistic is printed three times, once for each agent.

- % of games that resulted in a coalition involving agent x.

The following statistics are printed for each agent’s best strategy, median strategy, and the average over all the agent’s strategies. Hence, each of the following statistics are printed a total of 9 times whenever statistics are reported.

- Average payoff value over all games.
- Average coalition value over all games.
- Opportunity cost incurred over all games.
- Coalition opportunity cost incurred over all games.
- Will-accept values.
B. Population report

The population report prints out the strategies every n generations ($n = \ast\text{num-generations-between-pop-reports}\ast$) and when the simulation ends. If $n = 0$ (i.e., $\ast\text{num-generations-between-pop-reports}\ast = 0$), then the population report is printed out every generation.

For each generation, the population report produces the following:

- Results of the tournament of champions (see below).
- Results of the tournament of the mediocre (see below).

The following information is repeated for each agent.

- The best m strategies ($m = \ast\text{print-percent-of-sexps}\ast \times \text{number of strategies in a population}$) for agent x.
- The median strategy for agent x.

Strategies are printed in their original form, edited form or both forms depending upon the values of $\ast\text{print-unedited-population-report}\ast$ and $\ast\text{print-edited-population-report}\ast$. If neither parameter is 1, then the population report is not printed.

The tournament of champions consists of three special games played by the best strategy of each agent. Three games are played to allow each agent to begin negotiations. The tournament of the mediocre is identical except that the median strategy of each agent is used. The results of these games do not affect the statistics calculated by the system. The population report prints out the entire log of each game played in these tournaments.

C. Debugging log

The system can produce a log file for debugging. If the parameter $\ast\text{DBG}\ast$ is 1, then a log file is created that contains data about every game played. The data reported in the log file are:

- Generation
- Number of games played so far
- ID of agent A's strategy
- ID of agent B's strategy
- ID of agent C's strategy
- Game length
- Average game length of all games played so far
- Number of coalitions that include
  - agent A
  - agent B
  - agent C
- The game's log

VIII. Tree editor

It is often hard for the reader to decipher the trees because of their complexity. Indeed, trees of depth four are challenging to read and understand, yet our simulations allow trees to
achieve depths of eight. Therefore, we use an editor to simplify the trees. The rules of the editor are designed to assure that the edited trees are logically and behaviorally equivalent to the original trees.

The editor is a modified version of the editor provided in the code by James Rice and John Koza (Koza, 1992, p. 108). The modification permits us to allow destructive editing if we choose. Destructive editing changes the original population. Non-destructive editing leaves the original population unchanged and returns new trees. If the parameter "allow-destructive-editing" is t, then destructive editing is used. Currently, we do not employ destructive editing because we wish to preserve the genetic material that destructive editing throws away.

There are two sets of editing rules. The first set of rules simplifies complex conditions. These rules may introduce the symbols TRUE and FALSE into a tree. The second set of rules simplifies the IF-THEN-ELSE subtrees. These rules remove any TRUE or FALSE symbols introduced by the first set of rules.

A. Condition reducing rules
- (OR blah blah) or (AND blah blah) ==> blah
- (NOT (NOT blah)) ==> blah
- (AND condition1 condition2) with overlapping conditions
  ==> condition3 = condition1 \cap condition2
  Example: (AND {A 10 15} {A 12 18}) ==> {A 12 15}
- (AND condition1 condition2) with non-overlapping conditions ==> FALSE
- (OR condition1 condition2) with overlapping conditions
  ==> condition3 = condition1 \cup condition2
  Example: (OR {A 10 15} {A 12 18}) ==> {A 10 18}
- (OR TRUE blah) or (OR blah TRUE) ==> TRUE
- (OR FALSE blah) or (OR blah FALSE) ==> blah
- (AND TRUE blah) or (AND blah TRUE) ==> blah
- (AND FALSE blah) or (AND blah FALSE) ==> FALSE
- (NOT TRUE) ==> FALSE
- (NOT FALSE) ==> TRUE

A.1 IF-THEN-ELSE reducing rules
- (IF-THEN-ELSE TRUE foo bar) ==> foo
- (IF-THEN-ELSE FALSE foo bar) ==> bar
- (IF-THEN-ELSE condition blah blah) ==> blah

IX. Appendix: Summary of parameters

- crossover-fraction* Fraction of the population to be crossed over.

---

4 Indeed, our earlier runs allowed sexpressions to achieve depths of 18.
*mutation-probability*
Probability of performing mutation.

*leaf-crossover-probability*
Probability that leaf-crossover is performed.

*leaf-mutation-probability*
Probability that leaf-mutation is performed.

*max-number-of-leaf-xover-attempts*
Maximum permissible number of attempts to produce a legal new leaf via leaf-crossover.

*max-number-of-leaf-mutation-attempts*
Maximum permissible number of attempts to produce a legal new leaf via leaf-mutation.

*max-round-num*
Maximum number of rejected offers permitted before we end the game in default and award all agents 0 points.

*num-generations-between-pop-reports*
Number of generations simulated between printing each of the population reports (sexp files).

*num-generations-between-data-reports*
Number of generations simulated between printing each of the data reports (data files).

*num-generations-between-saves*
Number of generations simulated between saving the simulation (save files).

*print-unedited-population-report*
If t, then the original strategies are printed in the population report.

*print-edited-population-report*
If t, then the edited strategies are printed in the population report.

*play-triad-three-times*
t means each selected triad will play three games, allowing each agent to be the first-round initiator in one game. nil means play each triad only once selecting the first-round initiator randomly.

*number-of-trials*
This parameter determines the exact number of opponent strategy combinations each strategy is to play. A value of 0 means all combinations are to be played.

*allow-destructive-editing*
If t then populations are edited destructively.

*DBG*
If t the system produces many messages to help us determine whether the system is running properly. See Debugging log above.

X. Appendix: Summary of files

Note: in the following descriptions, "original code" refers to the genetic programming code developed by James Rice and John R. Koza (Koza, 1992, Appendix C).

cds.lisp
Defines the complex data structures, implements support functions and the genetic operators for them. This is all new code.

CFGeval.lisp
Implements the tournaments by which fitness is evaluated. This is all new code.

CFGrep.lisp
Defines the representation of the strategies and the game's structure. This is all new code.

CFGedit.lisp
Defines the s-expression editor rules.

compile-gp.lisp
Compiles and loads the system.
**XI. References**
