Training Function Stacks to play the Iterated Prisoner’s Dilemma.

Daniel Ashlock
Department of Mathematics and Statistics
University of Guelph
Guelph, Ontario, Canada, N1G 2W1
dashlock@uoguelph.ca

ABSTRACT
Cartesian genetic programming uses a directed acyclic graph structure rather than a tree structure for its representation of evolvable programs or formulas. In this paper a derivative of Cartesian genetic programming called a function stack is introduced and trained to play the iterated prisoner’s dilemma with an evolutionary algorithm. Function stacks differ from Cartesian genetic programming in that (i) they use a crossover operator and (ii) they have a form of memory or recurrence that permits the use of internal state information. Several properties of function stacks are developed and compared with other representations for the iterated prisoner’s dilemma. Function stacks are proved to encode the same strategy space as finite state machines but to explore that strategy space in a significantly different manner. A technique called fingerprinting is used to automatically classify the evolved strategies. Function stacks are shown to produce a significantly different distribution of strategies from those found when evolution is used to train finite state machines. Function stacks are shown to be different from many other representations studied for the iterated prisoner’s dilemma. They are relatively prone to cooperation and encode a rich space of strategies.

I. INTRODUCTION
The prisoner’s dilemma [4] is a widely known abstraction of the tension between cooperation and conflict. In the prisoner’s dilemma two agents each decide simultaneously, without communication, whether to cooperate (C) or defect (D). One situation modeled by the prisoner’s dilemma is that of two suspected criminals accused of the same serious crime, say burglary, and placed in separate interrogation rooms. The sheriff has evidence that can be used to convict both suspects of some minor crime, say trespassing. He offers each suspect lenient treatment in return for testifying against his accomplice. In this case cooperation consists of maintaining silence, while defection is embodied by testifying. There are four possible outcomes: mutual silence, the two different directions of one-way betrayal, and mutual defection. The agents receive individual payoffs depending on the actions taken. The best outcome for an individual is to be set free for unilaterally betraying his partner, yielding a score of $T$, and the worst is to be unilaterally betrayed which yields a score of $P$. In between those extremes mutual cooperation, yielding a score of $Q$, is superior to mutual defection which yields a score of $R$. These scores, together with the numerical values used in this study are shown in Figure 1. In order for a simultaneous two-player game to be prisoner’s dilemma, two conditions must hold:

$$P \leq R \leq Q \leq T \quad (1)$$

and

$$(P + T) \leq 2Q. \quad (2)$$

The first of these simply places the payoffs in their intuitive order; the second requires that the average score for both players in a unilateral defection be no better than mutual cooperation.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$C$</th>
<th>$D$</th>
<th>$S$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>0</td>
<td>1</td>
<td>$D$</td>
<td>$P$</td>
<td>$R$</td>
</tr>
<tr>
<td>$S$</td>
<td>3</td>
<td>5</td>
<td>$T$</td>
<td>$P$</td>
<td>$R$</td>
</tr>
</tbody>
</table>

Fig. 1. (1) A payoff matrix of prisoner’s dilemma – scores are earned by strategy $S$ based on its actions and those of its opponent $P$. (2) A payoff matrix of the general two player game – $P, Q, R$, and $T$ are scores given for the game.

If play is repeated many times, the game is called the iterated prisoner’s dilemma (IPD). The iterated game is very different from the one-shot game. In the one-shot prisoner’s dilemma a thoughtful player will notice that his best score results from defection no matter what his opponent does. Imagine we are using prisoner’s dilemma to model the behavior involved in the selling of illegal drugs. A dealer who knows his customer is an out-of-town businessman will sell him icing sugar instead of actual drugs as this permits him to keep his drugs and still obtain money. Likewise there is little risk to the businessman in paying with counterfeit money. If the one-shot prisoner’s dilemma were a good model for drug dealing, there would be no drug trade. The iterated prisoner’s dilemma models the situation of dealing with a regular customer. In this case, the same dealer would sell real drugs, and the customer would pay actual money. At a minimum, defection by either
party would cause the next day’s deal to go sour. This example shows that when the game is iterated even selfish agents have a motive to cooperate.

IPD is widely used to demonstrate emergent cooperative behaviors in populations of selfishly acting agents and is often used to model biological systems [21], ecological systems [16], as well as systems in sociology [12], psychology [20], and economics [11]. When evolutionary computation [6] is used to study the iterated prisoner’s dilemma, representation becomes an issue. Representation is the encoding of the game-playing agents including data structure, variation operators, and method of evaluating fitness. Perhaps the most common representation is finite state machines [17], [10], [22], [9], [3]. An indirect representation for finite state machines, in which a string of directions for how to build the finite state machines is used as the representation, appears in [13]. Another representation used is that of a fixed or variable-length look-up table [5], [14], [15]. In [8] the authors used artificial neural nets.

Nine different representations are compared in [1], and it is found that the probability of a population of evolving agents cooperating varies from 0% to over 90% based solely on the choice of representation. These representations include two kinds of artificial neural nets, Boolean formulas with and without a time-delay operation implemented via genetic programming, simple look-up tables with time depth 3, probabilistic look-up tables that are a type of Markov chain, ISAc lists [2], and finite state machines using both a direct and cellular representation. Function stacks, introduced in this study, turn out to be a highly expressive representation, able to simulate all the representations from [1] except Markov chains.

A. Examples of Prisoner’s Dilemma Strategies

The following example strategies for the iterated prisoner’s dilemma are used in the subsequent analysis of evolved agents. Always defect (AllD) and always cooperate (AllC) are self-explanatory. Tit-for-tat (TFT) cooperates initially and subsequently repeats its opponent’s last move. Psycho (PSY) defects initially and returns the opposite of its opponent’s last action thereafter. Punish once (Pun1) defects initially. Its next move is cooperation. After that, it cooperates in response to cooperation. If its opponent defects, it returns one defection and then follows that defection by a cooperation no matter what the opponent does. Pavlov (PAV) cooperates initially and cooperates thereafter if it and its opponent performed the same action on the previous time step. Tit-for-two-tats (TFT2T) defects only if its opponent has defected on the last two moves. Two-tits-for-tat (TTFT) defects on the two actions after any defection by its opponent but cooperates otherwise.

The remainder of this study is structured as follows. Section II gives a careful definition of function stacks and derives some of their properties as well as comparing them with other representations at the level of the strategy space encoded by the representation. Section III gives the experimental design for training function stacks to play the iterated prisoner’s dilemma. The results of the experiments are presented and discussed in Section IV. The results are placed in the context of other experiments with attention to the representation issue in Section V.

II. Function Stacks

A function stack is a representation derived from Cartesian Genetic Programming [18], [23]. The parse tree structure used in genetic programming is replaced with a directed acyclic graph that possesses a form of time-delayed recurrent link. The vertices of this graph are stored in a linear chromosome. Each node specifies a binary Boolean operation, an initial output value for that operation, and two arguments for the operation. The available Boolean operations are: And, Or, Nand, Nor, Xor, and Equality (not-Xor). The available arguments are: Boolean constants, the opponent’s last action, the output of any Boolean operation with a larger array index than the current one, and the output from the previous time step of any Boolean operation in the function stack. This latter type of argument is called a recurrent link. Permitting references to the current output of nodes with larger index gives function stacks a feed forward topology, a directed acyclic graph. This feed forward character is not present in the use of the recurrent link arguments, the output of every operation in the previous time step. The recurrent links give function stacks a form of short-term memory. The initial output values of each node, mentioned above, are required to give the value used for the recurrent links on the first time step. Examples of function stacks are shown in Figures 2, 3, and 4.

The action of the agents encoded by function stacks is specified by the output of the lowest index (zeroth) node, also called the output node. For prisoner’s dilemma, function stacks use the encoding: cooperate=false, defect=true. During initialization, operations are selected uniformly at random from those available, and arguments are selected according to the following scheme. One argument in ten is a constant (true or false). One quarter of all arguments are recurrent links with an index selected uniformly at random. The remainder of the arguments are either references to the output of nodes of higher index in the function stack or to input variables. The probability that an argument will be an input variable (if it is not a memory or constant link) is directly proportional to the index of the node in the stack. The zeroth node is thus most likely to reference the output of other nodes while the arguments of the last node must be input variables if they are not memory links or constants. This linear ramp-up of the probability of accessing a variable works with the “feed-forward” or directed acyclic graph nature of the function stack that encourages a larger fraction of links to other nodes, either directly or indirectly, from the output node.

The binary variation operator used on function stacks is a two-point crossover of the linear chromosome. The single-point mutation operator chooses a random operation three eighths of the time, a random argument half the time, and an initial value for a node’s memory one-eighth of the time. If an operation is selected, then it is replaced with another operation.
selected uniformly at random. If an argument is selected, then it is replaced with a valid argument selected according to the scheme used in initialization. If an initial memory value is selected, it is inverted.

A visual notation to make diagrams of function stacks is helpful in understanding them. The function stacks used in this study are composed of nodes with two inputs and two outputs. The inputs are transformed into the first output via the logic function (And, Or, Nand, Nor, Xor, or Eq) associated with the node. The second output has the value of the first output on its previous evaluation or, when no such previous evaluation is available, the initial value. Recall that the second output is called the recurrent output of the node. The inputs are shown on the left side of the node, and the outputs are on the right side and marked with small circles. The first output uses a white circle; the second uses a black circle. The logic function associated with a node is used as a label for that node. This label is super-scripted with a “+” if the initial value of the second output is true and a “−” if it is false. A single input variable is available to the function stack: its opponent’s previous move, denoted \( X_1 \).

### A. Properties of Function Stacks

An important property of an agent representation for playing prisoner’s dilemma is the degree to which it can condition its behavior on its internal state as well as its opponent’s actions. A finite state machine with \( n \) states has \( n \) states. A look-up table has \( 2^k \) “states” where \( k \) is the number of previous actions, its own and its opponents, that the look-up is conditioned on. Boolean functions of previous moves and neural nets with previous moves as inputs are alternate encodings of the same strategy space as a look-up table (this does not mean these representations will evolve the same type of agents, see [1]).

Each node in a function stack has a single binary state variable - the value it produced last time. This means that an \( n \)-node function stack has, potentially, \( 2^n \) internal states. It would take a rather remarkable function stack to fully exploit this huge amount of state information. On the other hand, a function stack with \( n \) nodes has more available state information than a finite state machine with \( n \) nodes. The strategy Pavlov (shown in Figure 2) is, minimally, a two-state finite state machine but is a one-node function stack. Nodes in a function stack contain more state information, but these “states” are difficult to access compared to those in a finite state machine. The proof of the following theorem sharpens the sense of this issue.

### Theorem 1: Function stacks and finite state machines encode the same strategy space.

**Proof:**

Suppose we have a function stack \( F \) with \( n \) nodes. The output of \( F \) depends on some set of \( k \) recurrent outputs, those that are arguments of some node in the stack. Call these the active recurrent outputs. A recurrent output can have one of two values and so there are at most \( 2^k \) sets of values that the active recurrent outputs can take on. The output of the function stack is a deterministic function of these recurrent outputs and the current input. Thus a finite state machine with \( 2^k \) states can encode the same strategy as the function stack. This shows the strategy space of function stacks is a subset of that of finite state machines.

Consider a finite state machine \( M \) with \( s \) states. A node in a function stack can be configured to simply report its last input though its recurrent link, e.g. (input OR input), creating a one-step delay line. Call such a node a one bit register. Let \( 2^r > s \) and designate \( r \) nodes to function as one bit registers. Read the output of these registers to obtain a binary encoding of the state of machine \( M \). Set the initial values of the nodes functioning as one bit registers to the initial state of \( M \). The next state of \( M \) is a deterministic function of the current state.
and the current input; use nodes to implement this function as \( r \) individual Boolean functions of the current state and input. These \( r \) Boolean functions compute the binary encoding of the next state of \( M \) from the current state and inputs -- something well within the capability of general Boolean functions. The value of this representation of the next binary state of \( M \) is simply fed into the input of the one bit registers that store the state. The output of \( M \) is also a deterministic function of current state and input and so may be computed with an additional Boolean function implemented with other nodes. Thus the strategy space for finite state machines is contained within the strategy space for function stacks. \( \square \)

![Diagram of the function stack constructed to simulate a finite state machine](image)

**Fig. 5.** This figure diagrams the method used in the proof of Theorem 1 to construct a function stack that simulates an arbitrary finite state machine. The one bit registers are OR nodes with their inputs tied together to create one-time-step delay lines. The initial values of the recurrent links of these gates store the initial state of the finite state machine. Boolean functions, constructed from the automata being simulated, compute the next state and output from the current state and input.

A diagram of the function stack constructed to simulate a finite state machine is shown in Figure 5. The theorem shows that function stacks are an alternate encoding of finite state machines. In the experimental section the evolutionary correspondence of finite state machines and function stacks will be checked in several ways.

### III. EXPERIMENTAL DESIGN

Three sets of experiments to train function stacks to play the iterated prisoner’s dilemma were performed. Each experiment consisted of 400 independent evolutionary runs. The parameter varied between experiments was the number of nodes in the function stack with 10, 20, and 40 nodes being used. Evolutionary runs continued for 250 generations. The population size in all evolutionary runs was 36. Fitness was evaluated with a round robin tournament on all possible pairs of agents within a population of 150 rounds of IPD.

An elite of the 24 highest scoring players, breaking ties uniformly at random, were copied into the next generation. Fitness proportional selection with replacement was used to choose a collection of 6 pairs of parents. These parents were copied, the copies subjected to the binary variation operator and then to a single application of the unary variation operator. The mean, variance, and maximum of both population fitness and age were recorded in each generation. An agent’s age is 0 when it is first created and increases by one each generation. An average of population average fitnesses was also recorded. The final population of each run was saved for analysis.

### IV. RESULTS AND ANALYSIS

The mean fitnesses over all populations for the three experiments are given in Figure 6. Function stacks with ten nodes gain in fitness at a lower rate than those with 20 and 40 nodes. All three populations seem to slowly become more cooperative as evolution proceeds after their initial rapid rise. The dip-and-rise curve, similar to that observed for finite state machines, has a narrower dip and sharper rise, suggesting a faster shake-out of initial randomness.

#### A. Comparison with Other Representations

As part of an ongoing project, the level of cooperativeness and of better-than-random play for many representations are recorded. This publication places function stacks into this context. Cooperative play is defined as a population average score of 2.8 or better over 150 rounds of play. This number is derived in [22] and implies at most transient defection for finite state machines that use 16 or fewer states. Better-than-random play is defined as a population average score of at least 2.25, the value a player that decides its action by flipping a coin gets when playing against itself.

The probability of cooperative behavior is shown in Figure 8 for generation 250 and in Figure 10 for generation 50. The probability of better-than-random play is shown in Figure 9 for generation 250 and Figure 11 for generation 50. The codes \( \text{F10}, \text{F20}, \text{and F40} \) denote function stacks with 10, 20, and 40 nodes respectively. The representations to which function stacks are compared are as follows: \( \text{CAT} \) are finite state machines using an indirect encoding [13]. \( \text{AUT} \) are directly encoded finite state machines with 16 states. \( \text{TRE} \) are Boolean functions with access to the opponent’s last three actions encoded via genetic programming with parse trees [7]. \( \text{MKV} \) are Markov chains implemented as a probabilistic look-up table indexed by the opponent’s last three actions. \( \text{LKT} \) are look-up tables indexed by the opponent’s last three actions. \( \text{ISC} \) are If-Skip-Action lists [2], a linear genetic programming representation acting on the opponent’s last three actions. \( \text{DEL} \) are Boolean parse trees identical to \( \text{TRE} \) save that a one-time-step delay operator is incorporated into the function set. \( \text{CNN} \) are feed-forward neural nets with a per-neuron bias in favor of the output signifying cooperation; they access the opponent’s last three actions and have a single hidden layer containing three neurons. \( \text{NNN} \) are feed-forward neural nets identical to CNN save that they have no bias in favor of cooperation or defection. Details of these representations other than function stacks, are given in [1]. The relationship among the strategy spaces for these representations is given in Figure 7.

Theorem 1 demonstrates that finite state machines and function stacks encode the same strategy space. In [13] it is
shown that the cellular encoding CAT is complete, i.e. that it encodes the same strategy space as the directly encoded finite state machines AUT. In spite of this all possible pairs of finite state machine and function stack representations shown except F20 and F40 show a statistically significant difference in their probability of cooperative or better than random play. Together with the difference in sampling of strategies shown in Section IV-B, this demonstrate that representation dominates the possession of a mutual strategy space.

B. Fingerprint Analysis for Function Stacks and Finite State Machines

Fingerprints are explained in detail in [13]. A fingerprint is a function from the unit square with corners (0,0) and (1,1) to the real numbers that is an invariant of a strategy for playing iterated prisoner’s dilemma. That the fingerprint is an invariant of a strategy, rather than its implementation, makes it convenient for cross-representation comparisons. This section uses fingerprints to compare the rate at which function stacks with various numbers of nodes and finite state machines locate some well-known strategies. The theory of fingerprints was initially developed for agents encoded by finite state machines; Theorem 1 permits us to apply it to function stacks.

The play of two finite state machines in the presence of noise can be represented as a Markov process. This allows the determination of an expected score for any pair of strategies by standard techniques in stochastic processes [19]. Fingerprints use game-playing agents with strategies that incorporate parameterized noise to assess other agents. The independent variables of a fingerprint are rates for types of noise associated with each possible move in a game. The value returned by a fingerprint is the expected score of the agent being fingerprinted against the type of noisy opponent specified by the independent variables. For iterated prisoner’s dilemma, the noise represents probabilities of cooperating and defecting. The fingerprint for an agent is thus a map from probabilities $(x, y)$ of respectively cooperating and defecting to a value $E$, the expected score of the agent being fingerprinted against the noisy agent. For the fingerprints used in this study the noisy agent is based on tit-for-tat. If the play of the noisy agent as determined by $x$ and $y$ is not noise of either type, then the noisy agent returns the last move of the strategy being fingerprinted.

**Definition 1:** If $A$ is a strategy for playing prisoner’s dilemma, then $Joss-Ann$ is defined to be

$$JA(A, x, y),$$

a strategy which has a probability $x$ of choosing the move $C$. 

Fig. 7. Relationship between the various representations in this study. Representations within a box encode the same set of strategies. Upward arrows denote containment. Containment is transitive and so, for example, ISAc lists can code any of the strategies except some that are encoded by Markov Chains.
a probability $y$ of choosing the move $D$, and otherwise uses the response appropriate to the strategy $A$.

Note that when $x + y = 1$ the strategy $A$ is not used, and the resulting behavior is a biased random strategy with play probabilities $x$ of cooperating and $y$ of defecting.

Definition 2: A fingerprint $F_A(S, x, y)$ with $0 \leq x, y \leq 1$, $x + y \leq 1$, and strategies $S$ and $A$, is the function that returns the expected score of strategy $S$ against $JA(A, x, y)$ for each possible $(x, y)$. The double fingerprint $F_{AB}(S, x, y)$ with $0 \leq x, y \leq 1$ returns the expected score of strategy $S$ against $JA(A, x, y)$ if $x + y \leq 1$ and $JA(B, 1 - y, 1 - x)$ if $x + y \geq 1$.

In this study we use the double fingerprint with $A = \text{tit-for-tat}$ and $B = \text{psycho}$ to create a set of numerical features used to classify prisoner’s dilemma strategies. A grid of 25 points in the interior of the unit square is used to sample the fingerprint. These points are all those of the form $(i/6, j/6)$ where $0 < i, j < 6$. For the reference strategies, AllD, AllC, and so on, the values at these points are determined by using the actual double fingerprint functions, computed in [13]. For the finite state machines and function stacks, the value of the fingerprint is determined by sampling play against $JA(TFT, x, y)$ at the 25 values of the noise parameters. For any particular value of the noise parameters, sets of 150 rounds of the IPD are played repeatedly until the variance of the estimate of the fingerprint value drops to 0.01. A strategy is classified as similar to a reference strategy if the distance in Euclidean 25-space between the fingerprint functions is no more than 0.08. The threshold was chosen because the closest that any two reference strategies approach is 0.17. The counts for each agent type in each representation are given in Table I.

The substantial number of agents exhibiting the AllD fingerprint in the representations other than directly encoded finite state machines requires some explanation that highlights a weakness of fingerprints for classification. The fingerprint captures the expected, asymptotic score of a strategy in the presence of noise. Thus transient states (those a strategy can leave and never return to) do not affect the fingerprint. This means that two strategies with different fingerprints are different, no question, but strategies with the same fingerprint are not identical; they can differ in their transient states which are sometimes very significant.

The strategy vengeful cooperates until its opponent’s first defection and defects thereafter. It is asymptotically indistinguishable from AllD and has the exact same fingerprint. Unlike AllD, vengeful plays pure cooperation against itself. Examination of population files showed that many of the strategies with an AllD fingerprint were, in fact, playing the strategy vengeful. This resolves an apparent contradiction between the relatively high level of cooperation exhibited by
function stacks and the substantial fraction of agents exhibiting an AllD fingerprint.

The enormously higher fraction of “other” strategies in the finite state machine populations as compared to the function stack populations suggests that finite state machines are more likely to create complex strategies. All the known strategies used for fingerprint comparison in this study use two or fewer states in a minimal finite state implementation and so may be reasonably considered simple.

V. Conclusions

This study introduces function stacks as an extension of Cartesian genetic programming. The novel features of function stacks are a binary variation operator and memory in the form of recurrent links for each Boolean operation. While function stacks are shown to encode the same set of strategies as finite state machines, they sample the space of strategies in a significantly different manner. This difference is apparent in both the level of internal cooperation exhibited by populations of function stacks as compared to other representations as well as by the frequency of simple strategies in evolved populations shown in Table I.

The far larger number of “other” strategies found in populations of finite state machines suggest that function stacks locate simpler strategies more often than finite state machines.

For the iterated prisoner’s dilemma there is some reason to suspect that simplicity is a virtue [5]. In spite of this, finite state machines and function stacks group together in Figures 8-11 which assess the degree of cooperativeness; only look-up tables are in the interior of the region spanned by finite state machines and function stacks in these figures. This suggests that these two encodings may be more similar to one another than to the other representations studied.

When using game-playing agents to simulate human or animal behavior, the issue of removing implementation bias from simulation design becomes a substantial one. This study continues the demonstration that representation is a large source of implementation bias even in very simple games. In addition to introducing and characterizing function stacks, this study demonstrates that changing the number of nodes in a function stack causes statistically significant changes in their evolutionary behavior. Changing the number of nodes is a non-trivial change in the representation. There is some good news in that 20 and 40 node function stacks behave in a similar manner.

This paper proves that finite state machines and function stacks implement the same strategy space. However given the different behavior of function stacks with different numbers of nodes, it is unlikely that this identity holds once a number of states/nodes is selected. The number of nodes required on
average to implement an $s$-state finite state machine and the number of states required for an FSM to simulate an $m$-node function stack are not known. A few hand-worked examples suggest that this equivalence is not simple; the trade-off is complex and idiosyncratic.

Fingerprinting was used to demonstrate that the five representations participating in the fingerprinting study sample the strategy space in very different ways. It also highlighted, with the high fraction of AllD fingerprints, the need to incorporate information about the transient states of a strategy into the classification process. This could be done by recording the sequence of plays a strategy makes against itself for a small number of moves (10-30) and using this self-play string as a second identifier, possible for separating the equivalence classes induced by fingerprinting. The self-play string, like the fingerprint, is implementation invariant.

This study is part of an ongoing study that seeks to understand the impact of representation on the way evolution trains game-playing agents. As part of this study, known representations are cataloged and new representations are invented. Potential collaborators with unique or interesting representations for game-playing agents are invited to contact the author.

VI. ACKNOWLEDGMENTS

The author would like to thank Julian F. Miller for the excellent idea of Cartesian genetic programming, inspirational to this study. David B. Fogel is thanked for many helpful discussions that improved the design of the experiments performed. The author would also like to thank the University of Guelph Department of Mathematics and Statistics for its support of this research.

**TABLE I**

<table>
<thead>
<tr>
<th>Strategy</th>
<th>F40</th>
<th>F20</th>
<th>F10</th>
<th>AUT</th>
<th>CAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllD</td>
<td>2836</td>
<td>3560</td>
<td>5476</td>
<td>381</td>
<td>6897</td>
</tr>
<tr>
<td>AllC</td>
<td>1155</td>
<td>1077</td>
<td>827</td>
<td>72</td>
<td>221</td>
</tr>
<tr>
<td>TFT</td>
<td>8951</td>
<td>8272</td>
<td>6606</td>
<td>2663</td>
<td>2542</td>
</tr>
<tr>
<td>PSY</td>
<td>697</td>
<td>799</td>
<td>426</td>
<td>1</td>
<td>362</td>
</tr>
<tr>
<td>PAV</td>
<td>3</td>
<td>10</td>
<td>35</td>
<td>69</td>
<td>2</td>
</tr>
<tr>
<td>TF2T</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>53</td>
<td>30</td>
</tr>
<tr>
<td>TTFT</td>
<td>559</td>
<td>569</td>
<td>713</td>
<td>6</td>
<td>44</td>
</tr>
<tr>
<td>PUN1</td>
<td>12</td>
<td>2</td>
<td>10</td>
<td>442</td>
<td>17</td>
</tr>
<tr>
<td>Other</td>
<td>146</td>
<td>108</td>
<td>1041</td>
<td>10713</td>
<td>4285</td>
</tr>
</tbody>
</table>

**REFERENCES**


