Reusing Learned Functionality in XCS: Code Fragments with Constructed Functionality and Constructed Features

Isidro M. Alvarez
Victoria University of Wellington
New Zealand
isidro.alvarez@ecs.vuw.ac.nz

Will N. Browne
Victoria University of Wellington
New Zealand
will.browne@vuw.ac.nz

Mengjie Zhang
Victoria University of Wellington
New Zealand
mengjie.zhang@ecs.vuw.ac.nz

ABSTRACT

This paper expands on work previously conducted on the XCS system using code fragments, which are GP-like trees that encapsulate building blocks of knowledge. The usage of code fragments in the XCS system enabled the solution of previously intractable, complex, boolean problems, e.g. the 135 bit multiplexer domain. However, it was not previously possible to replace functionality at nodes with learned relationships, which restricted scaling to larger problems and related domains. The aim of this paper is to reuse learned rule sets as functions. The functions are to be stored along with the code fragments produced as a solution for a problem. The results show for the first time that these learned functions can be reused in the inner nodes of the code fragment trees. The results are encouraging as there was no statistically significant difference in terms of classification. For the simpler problems the new system $XCSCF^2$, required much less instances than the XCSCFC to solve the problems. However, for the more complex problems, the $XCSCF^2$ required more instances than XCSCFC; but the additional time was not prohibitive for the continued development of this approach. The main contribution of this investigation is that functions can be learned and later reused in the inner nodes of a code fragment tree. This is anticipated to lead to a reduced search space and increased performance both in terms of instances needed to solve a problem and classification accuracy.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics-Based Machine Learning, Learning Classifier Systems

General Terms

Algorithms, Performance

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2881-4/14/07:.$15.00.
http://dx.doi.org/10.1145/2598394.2611383.

Keywords

Learning Classifier Systems, XCS, Finite State Machines, Genetic Programming, Scalability, Pattern Recognition

1. INTRODUCTION

Learning may be considered a human ability that involves complex processes working together to build a repository of knowledge for later retrieval. This type of processing is very valuable but is still out of reach for today's computer systems [1].

The hypothesis is that there are a finite number of common patterns in the world [20]. Furthermore, natural, human and artificial systems will tend to fall into these aforementioned patterns. If a pattern can be recognized in one system along with the ability to solve problems related to that system or that domain, one should be able to reuse those techniques in a related domain. This is similar to how human beings learn, e.g. they tend to learn by analogy. If this approach were to be harnessed by evolutionary computation techniques, it could prove useful in solving various problems.

Learning classifier systems are one method for investigating this goal and many different types have been developed in nearly 40 years of research [14]. One such type is XCS, where the fitness is based on the accuracy of a classifier's payoff prediction instead of the prediction itself [4], [22], as opposed to ZCS which is a strength based system [21], [22], [23]. XCS is a powerful tool; nevertheless, some problems are difficult to complete because of time constraints.

With the advent of Code Fragments as a methodology within XCS, it was possible to solve until then, intractable problems in scalable boolean domains. A code fragment (CF) is a tree expression, similar to a tree generated in Genetic Programming [6]. Code Fragments create small blocks of code in binary trees up to a depth of two. This depth was chosen, based on empirical evidence, to limit bloating and the introduction of large numbers of introns. Analysis suggests that there is an implicit pressure for parsimony [10].

Reusing code fragments in terminals of trees results in large trees for large scale problems. Therefore, a computational limit in scalability will eventually be reached [8]. This is because since multiple code fragments can be used at the terminals, as the problem increases in size, then any depth of tree could be created. The concept is to learn functionality to replace nodes (sets of branches) within the tree, in order to make searching the tree tractable.

Previously, only terminals could be replaced by the constructed code fragments, which was a design choice, as the building blocks of knowledge were at the terminal level, e.g. the discovery of useful combinations of features and constants. This is analogous to feature construction using GP trees. The reason that code fragments are used only at the terminals in XCSCFC is because CFs can accept any number of arbitrary inputs where a function takes in a set number of inputs. This means that if one were to swap a function with a code fragment this might involve dissimilar objects.

A genetic programming approach may have many preprogrammed functions, but these are unrelated to each other and to useful building blocks of knowledge. In $XCSCF^2$ it is anticipated that functions and building blocks will be linked in order to guide genetic operations for improved search.

Function reuse at the inner nodes is important because by reusing the learned rule sets as Functions (termed Function-RuleSets) and their associated code fragments, it is possible to reduce the search space of the code fragment trees, or likely solutions. This will aid in the performance of the system as well as provide valuable and reusable building blocks of learned knowledge. The analogy is as follows:

$$'If < Conditions > Then < Actions >' \quad (1)$$

$$'If < Input > Then < Output >' \quad (2)$$

$$Function(Arguments < Input > Return < Output >) \quad (3)$$

Equation (1) is the standard way that a classifier would process its conditions to achieve an action, which is analogous to (2). Equation (3) is the analogy with a function. These functions will take a number of arguments as their input and will return an output.

The main aim of this paper is to determine whether it is feasible to reuse rule sets (Function-RuleSets) as functions within a code fragment based classifier system.

One way to look at this is that humans learn by accumulating knowledge learned from solving problems in one domain and then applying the new knowledge repository and functional skills to a new domain. For example, a child learns to thread a shoelace in a 3-pair hole shoe and then uses the knowledge to thread the laces of a 5-pair hole shoe. Now the child learns the function to tie a bow in a shoe lace and can then transfer that learning to tying a bow in a ribbon of a present [admittedly both examples are hugely more complex than the problems demonstrating the principles presented here].

The research objectives are as follows:

* **Develop** new methods for function reuse.
* **Determine** if learned rule sets can be used as functions in CFs.
* **Compare** performance overhead or degradation on benchmark problems with equivalent techniques.

The benefit of this research is that it is a new approach to learning where knowledge from one domain may be used to solve problems in a different but related domain. Also, this approach lends itself to expanding current scalability and helping to solve up-until-now intractable problems.

2. BACKGROUND

A learning classifier system (LCS) is an evolutionary adaptive system that learns a problem using a set of mutually cooperative rules. An LCS learns by interacting with the environment, it typically starts by covering the individual data patterns from the environment input and eventually generalizes the population by removing irrelevant information [5].

LCSs are a good method for solving certain problems because they produce a complete state-action map. Since the solution is a population of classifiers, each classifier represents a small portion of the overall solution [16]. The action of each classifier represents the decision taken by the system based on the environmental stimulus. Accuracy based LCS, e.g. XCS [16], [19], attempt to produce the final classifier population as a collection of general classifiers with optimal accuracy and fitness [3]. The concept is that the suboptimal individuals would have been eliminated by preferential selection of optimal individuals through the process of evolution. This is where the Genetic Algorithm plays a crucial part in the classifier; by preferentially selecting fitter individuals.

Although XCS brought with it many benefits, it still needed some improvements in order to have it solve more complex problems, e.g. the 135 bit multiplexer. One of the drawbacks is that although it can scale in certain domains, it still has to relearn from the start each time. Any increase in the dimensionality of the problem increases the search space, the hardware demands, and training time [5]. To address these pressures, several different flavors of code fragments have been used.

By using CFs, the system increased in scalability and was able to transfer knowledge learned in a simpler problem into a more complex one of the same domain. For example, in the 135 bit multiplexer problem, XCSCFC takes only 2 x 10^6 instances to successfully solve the problem compared to the search space of 4 x 10^{40}. The standard XCS was not able to solve the same problem [11].

After CFs were utilized in the condition of an LCS, XCSCFC, they were tested in the action, XCSCFA [11]. This method produced optimal populations in discrete domain problems as well as in continuous domain problems [11]. This however lacked scaling to large problems, even if they had repeated patterns in the data. XCSSMA [8] was introduced with the ability to generate state machines to encapsulate repeating patterns.

The first investigation involving code fragments was the introduction of GP-tree like expressions to represent condition bits in a classifier rule; this was named *code-fragment conditions* CFC [11]. In this approach the condition bit in a classifier was directly replaced with a code fragment. Initially, there was a separate population of code fragments used. Currently the code fragments are housed simply within the rules [10], [It is now the aim to once again make use of a separate code fragment population]. This means that the number of code fragments to be reused from a particular level was governed by the unique code fragments in good classifiers.

According to [11], this investigation showed that the multiple genotypes to a single phenotype issue in feature-rich encoding disabled the subsumption deletion function. Further, the additional methods and increased search space also led to much longer training times. However, this was compensated by the code fragments containing useful knowledge;

such as the importance of the address bits in the multiplexer problems.

The static binary action was replaced by a code fragment while using the ternary alphabet in the condition of the classifier rules (XCSCFA) [5]. Each code fragment was a binary tree of depth up to d. The value of d was dependent on the length of the condition length [5]. The action value of the classifier was determined by evaluating the action code tree [9]. In order to achieve this, it was necessary to replace the terminal symbols with corresponding binary bits from the associated condition in the classifier rule [5], [7].

Subsumption deletion was effectively disabled leading to increased methods and search space [11]. Whereas in standard XCS with binary action, subsumption deletion is fully enabled so the numerosity of the general classifier in a niche gets higher values as it subsumes the less general classifiers in the niche; in code fragment based XCS, the multiple genotypes to a single phenotype issue disables the subsumption deletion function, so fitness in a niche is distributed among multiple equally general classifiers, all having a relatively small fitness value as compared to the binary action-based XCS [5]. However, the lack of subsumption deletion was compensated by the autonomous separation of optimal and sub-optimal classifiers in the final population. This eventually resulted in the optimum rule set of the maximally general, compact and accurate classifiers. XCSCFA is a useful method because it produces a compact solution that can be easily converted to the optimum population [10].

The result described above is an obscure but interesting phenomenon of XCSCFA. It has to do with the fact that there are "don't cares" in the action; a highly unusual practice for a classifier system. The input message has to be converted to an output. This would be achieved by taking the environmental message and putting it into the action part. However, if one were to randomly place a '0' or '1', as part of a "don't care", an interesting effect occurs. Instead of having a classifier population with a group of optimal rules, another group of sub-optimal rules, and a spread of rules between them, in terms of fitness and numerosity; the result is a complete separation between the classifiers that are optimally fit and the newly created/sub-optimal ones. This means that it is possible to separate optimal from sub-optimal classifiers without condensation.

Presently, any new code fragments that are learned while solving a problem may be reused by a tree in any subsequent problems, however, the functions are predefined and once the current run ends, they will be "forgotten" by the system.

Some notable work that is related to the current research are Bull's and Lanzi's research using Genetic Programming within an LCS. The former studied methods for using a co-evolutionary approach for the use of automatically defined functions (ADFs) in genetic programming for classification tasks. This method involved using the main program as a binary feature selector rather than a full S-expression tree [2]. The latter introduced an extension to XCS called XCSL. In this system the usual bitstring conditions are replaced by general s-expressions. XCSL was used to learn boolean functions of increasing complexity [13],[15].

Both authors were utilizing GP as a type of filter for an additional process, e.g. classification. Their aim was to reduce the number of features and the number of conditions that go into a system and then the result would be fed to another system, such as a K Nearest Neighbor classifier. Lanzi's work centered on generating whole GP trees, where the work here is concerned with building blocks in the form of small GP trees. Also, Lanzi's approach involved replacing the entire condition part while this work is concerned with replacing individual bits of the condition with small GP trees. Finally, this work is seeking to reuse the code fragments where Lanzi was not.

3. THE METHOD

The first step is to determine the axioms and atoms that the system can assume, e.g. the hard-coded functions and terminal sets. The more of these that are included, the more domain bias is also included without the ability to learn the linkage between functions and discovered building blocks for boolean domains. For example, for boolean domains, NAND gates are building blocks with which it is possible to build other gates such as the OR, AND or XOR [18], [17].

Figure 1 provides a step by step illustration of our methodology. The top parts of the figure with the letters "FS" represent the current function set. The X and Y axes with the graphs represent the function and its expected performance during the learning phase. The X axis represents the number of instances required for the process to stabilize and fully learn the problem, while the Y axis stands for the percentage of the problem learned. The circles with the rules at the bottom of figure 1 represent the rule sets that will be learned for each of the functions. These same rule sets will be used at the inner nodes of the code fragment trees in future runs.

The system is to be initially trained with the NAND function, which is to produce the relevant rule set and the associated code fragments. Once this phase is successful, the system will have accumulated a population of rules that will solve the NAND function. The next step will be to have the system learn the OR function by using the rules belonging to the NAND function. At the end of this new run there will be the additional OR function as a rules set and also a set of code fragments. The process would then proceed to learn the AND, XOR and NOR functions in a similar manner. By the time the system attempts the multiplexer problems it should have a readily available stockpile of functions along with their rules sets and Code Fragments.

As can be surmised from figure 1, the resulting Function-RuleSets would play the role of functions in the inner nodes in the code fragment trees of any subsequent runs while the code fragments for the same function can get reused at the leaf nodes of the CF trees. It is theorized that by building upon a growing set of functions, with their respective rule sets and code fragment sets, it is possible to transfer this knowledge to a more difficult problem in the same domain and possibly into a related domain. First, it is important to mention that after the system is trained by each function, it becomes part of the function set. In other words, a nominal symbol is assigned by the system to represent the newly acquired function, so that it will be available in future runs as a component for the inner nodes.

The proposed XCS with code fragment functions, $XCSCF^2$, is based on the XCSCFC which utilizes code fragments in the condition. As can be seen in figure 2, code fragments are represented by binary trees of a determined depth. This depth is there to prevent an uncontrolled growth of trees. The inner nodes made use of predefined functions, and these were chosen randomly when the code fragment was first created. The leaf nodes contained either the address of a feature from

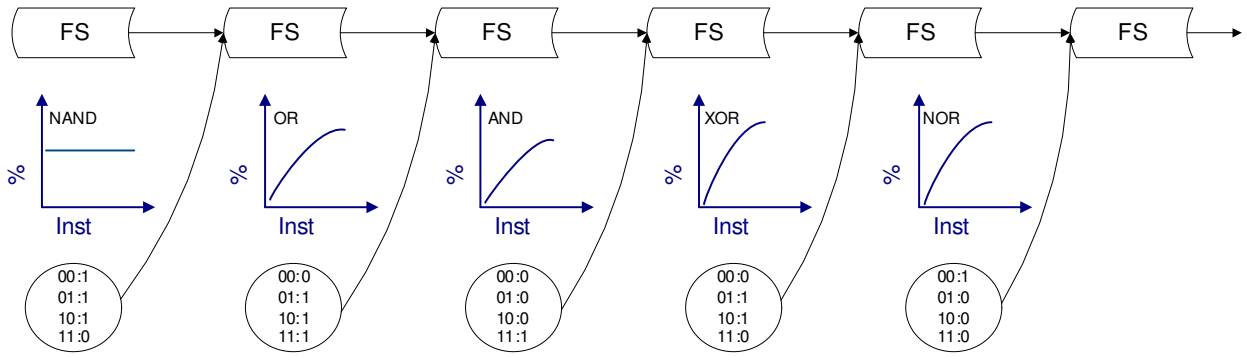

Figure 1: Code fragment and Function-RuleSet reuse - step by step.

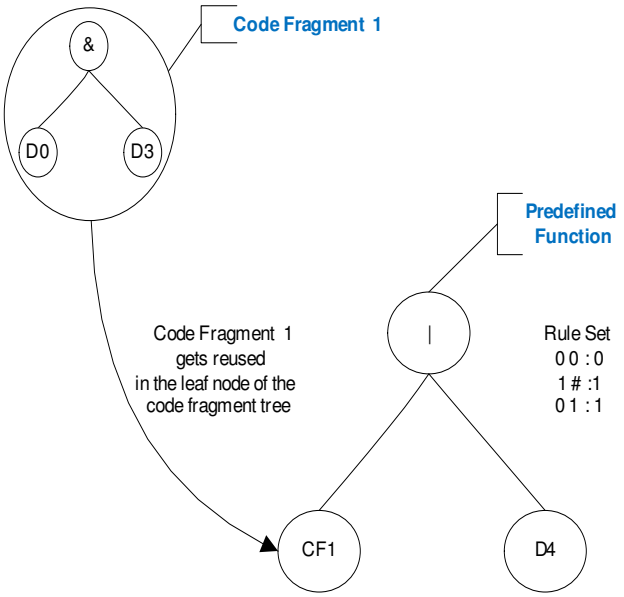

Figure 2: Code fragments - XCSCFC - Initial starting system.

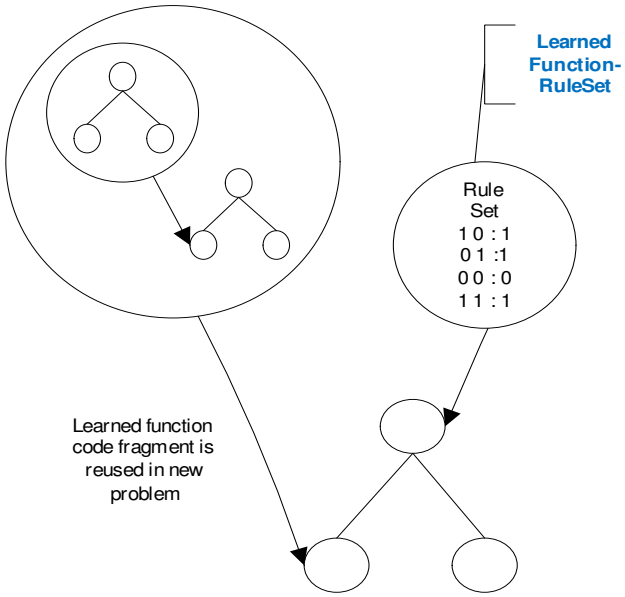

Figure 3: Code fragment and rule set reuse - goal of the proposed system.

the environmental input (message state) or a previously discovered code fragment. This was one of the limitations of the XCSCFC system that this work attempts to address. Although XCSCFC was able to solve previously intractable problems that the XCS was not able to solve, it suffered from the lack of functionality reuse at the inner nodes, as well as a large search space; in terms of the code fragments generated.

Figure 3 shows the process flow of the new method. When a problem is solved, the code fragments get stored by the system for later usage. Now the rule-sets learned by solving the problem also get saved. These rules are composed of 1's and 0's which are the result of evaluating a particular CF tree. As can be seen in figure 3, the root node of the code fragment tree is composed of a function with the rule-set containing four members; in this case the rule-set maps to the OR logic operation. In a subsequent run, while attempting to solve an even more complex problem, the learned Function-RuleSet could be utilized to construct new CFs. Therefore, when it is necessary to evaluate each of the CFs, this process will rely on the retrieved rules for each function.

The system uses the reverse polish notation method to traverse the tree, evaluating each function using the values of its children nodes. This continues recursively until the root node is reached, which produces the value to be the output of the tree.

The first set of experiments is to build the other boolean functions from NAND, which the system used as the very first function to learn. In essence, it is the bootstrap that gets the entire process on its way. As the functions were learned they were used to learn the next function. For example, after learning the NAND, its code fragments and its rule-set were used to learn the OR function and so forth. The domain used in the final experiments was the multiplexer problems. The reason for this is because these types of problems can be solved via boolean functions such as AND, OR, etc [12].

Note: Apart from the axiomatic functions, only Function-RuleSets are available for generating CFs. Thus, although predefined OR functionality is known for training the rule-sets, it is not used. Subsequently this enables previous functions and CFs to be linked to generated Function-RuleSets.

4. RESULTS

4.1 Experimental Setup

The experiments were run 30 times and compared with a XCSCFC. The settings for the single step experiments were as follows: Payoff 1,000; the learning rate $\beta = 0.2$; the Probability of applying crossover to an offspring $\chi = 0.8$; the probability of using a don't care symbol when covering $P_{don'tCare} = 0.33$; the experience required for a classifier to be a subsumer $\Theta_{sub} = 20$; the initial fitness value when generating a new classifier $F_I = 0.01$; the fraction of classifiers participating in a tournament from an action set 0.4.

4.2 Boolean Problems

The first problem presented to the system was the NAND problem. This process continued next to learning the OR, AND, XOR and NOR functions in a serial manner. By this time the system was able to draw upon a small set of learned functions. The subsequent runs were able to solve the 6, 11, 20, 37, 70, and 135 bit multiplexer problems by using this same methodology.

The population sizes used for these functions as well as the number of training instances used are listed in table 1:

Table 1: Number of classifiers and training instances

Boolean	Classifiers	Instances
OR	2,000	600,000
AND	2,000	600,000
XOR	2,000	600,000
NOR	3,000	700,000

It is notable that the population size required to learn the boolean logic problems was consistently higher than the simplest of the multiplexer problems attempted, see table 2. The number of classifiers required remain constant during the OR, AND, and XOR functions; it increased for the NOR function for both, the classifiers and instances needed to learn the function. The boolean functions required a higher number of training instances than the simplest boolean problem. This trend continued until the NOR function required slightly more than half the number of instances as the 37 bit multiplexer using $XCSCF^2$. Table 1 shows the set-up values required during the boolean function experiments, while figure 4 shows the results for said experiments. It is evident that the OR function required more instances to solve the problem than the other functions with the exception of NOR. This was expected since the system had only the NAND function in its function set at the time. The rest of the functions all required less instances with the XOR function requiring the least number of instances at approximately 21,000.

4.3 Multiplexer Problems

Table 2 depicts the number of classifiers used in the populations as well as the different training sizes for each of the multiplexer problems solved. Figure 5 shows the results for the 6-20 bit multiplexer experiments. According to the graphs, the $XCSCF^2$ system was able to solve the 6 bit multiplexer problem with approximately 15,000 instances while the regular XCSCFC system achieved this with approximately 80,000 instances. One possible explanation for this performance difference is the fact that the former system had a tool-set of learned functions by this time; the NAND, OR, AND, XOR, and the NOR function, while the latter only had hard coded boolean functions. The performances for the 11 bit multiplexer were similar as well. This was due to the fact that $XCSCF^2$ had its set of learned rule sets as well as the newly learned Code Fragments. The function reuse helped the $XCSCF^2$ system to reach maximum performance with approximately 17,000 instances while it took the XCSCFC approximately 65,000 instances. A similar outcome was observed with the 20 bit multiplexer problem.

Table 2: Number of classifiers and training instances

Multiplexer	Classifiers	Instances
6 Bit	500	500,000
11 Bit	1,000	500,000
20 Bit	2,000	500,000
37 Bit	5,000	1,000,000
70 Bit	10,000	2,000,000
135 Bit	50,000	5,000,000

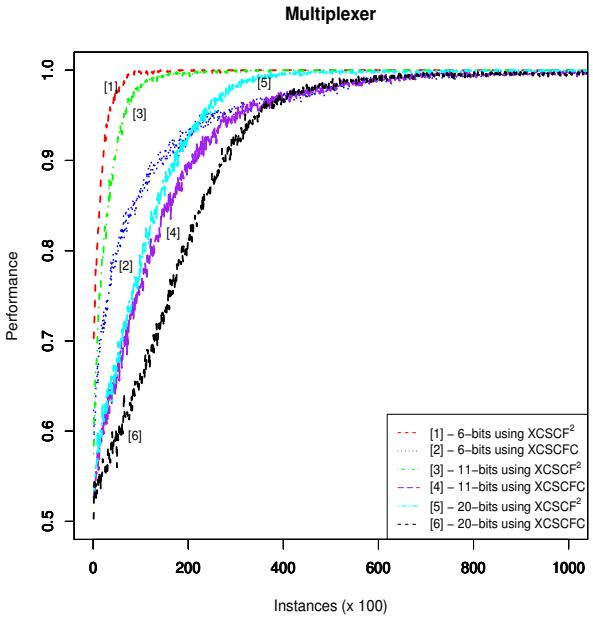

Figure 5: Results of the multiplexer problems using XCS with code fragments and XCS with function reuse.

The function reuse helped the system to learn the problem with less instances. However in this case the difference in number of instances needed to solve the problem was less than in the previous problems. It is interesting to note that with regards to time, the XCSCFC system performed better. For the 6 bit multiplexer $XCSCF^2$ took 10.1 seconds versus 3.33 seconds for XCSCFC. For the 11 bit problem the performances were 45.9 and 13.4 seconds. Likewise, for the 20 bit problem the performances were 220 and 63.6 seconds respectively.

The 37 bit multiplexer problem is depicted in figure 6. In this experiment it was the XCSCFC system which learned the problem with less instances, approximately 110,000 as compared with the $XCSCF^2$ system which needed about 170,000 instances to learn the task. In terms of time, the trend continued unabated; $XCSCF^2$ took an average of

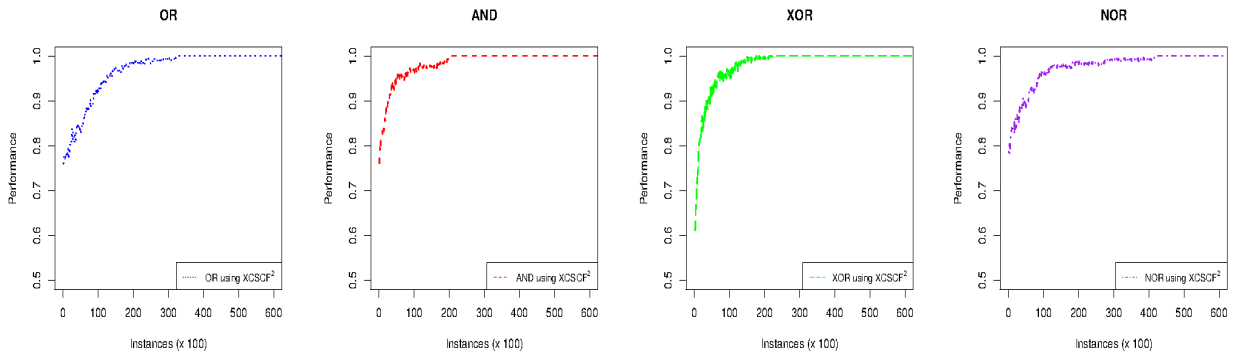

Figure 4: A sequence of boolean logic problems

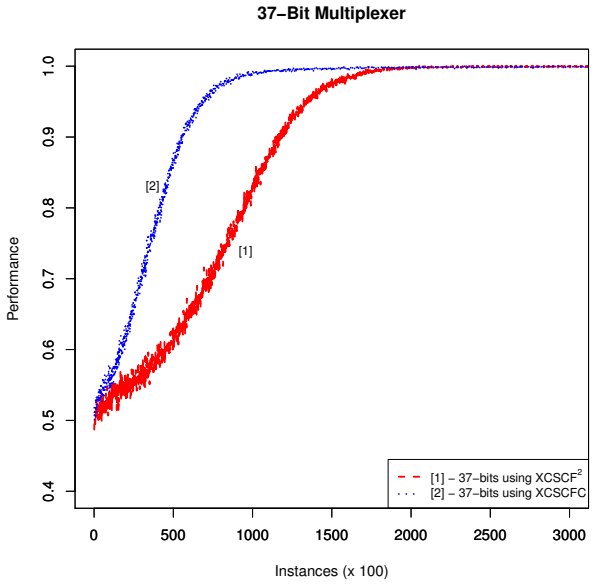

Figure 6: Results of the 37-bit multiplexer problem using XCS with code fragments and XCS with function reuse.

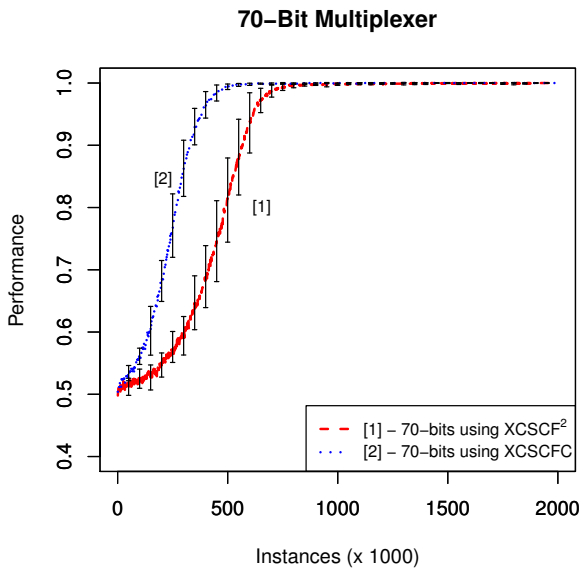

Figure 7: Results of the 70-bit multiplexer problem using XCS with code fragments and XCS with function reuse.

3136 seconds compared with the XCSCFC system which only took 681 seconds.

Figure 7 shows the results for the 70 bit multiplexer. The $XCSCF^2$ system produced a curve with values that varied much more than the ones for the regular XCSCFC system. The range of the values was ten percent of the performance while the values for the regular code fragment curve had less variation. In this experiment the fact that the regular XCSCFC system outperformed the $XCSCF^2$ in terms of required number of instances was not a surprise as this trend was already evident during the 37 bit experiment. Where the XCSCFC system stabilized at approximately 450,000 instances, the $XCSCF^2$ system achieved this with approximately 700,000 instances. In terms of time XCSCFC again performed better than $XCSCF^2$. While $XCSCF^2$ took approximately 27,600 seconds, XCSCFC took only approximately 7,800 seconds. Function reuse is likely to be slower as rule-sets need to be matched and the relevant output effected.

Figure 8 shows the results of the 135 bit multiplexer experiments. In this case the XCSCFC system also outperformed the system utilizing function reuse with respect to instances needed. The proportion of execution time remained approximately the same as for the 70 bit multiplexer. The function reuse system learned the problem with about 1,700,000 instances while the other system achieved this with about 1,300,000 instances. This is most likely because at this time both systems reused the entire population of code fragments and these were not limited to a particular function. In addition to this, $XCSCF^2$ was limited to only the five boolean functions learned up to that point in time, XCSCFC also relied on the same number of boolean functions with the only difference of the NOT function. It is theorized that the XOR function was instrumental in aiding $XCSCF^2$ to learn the early multiplexer problems more efficiently than XCSCFC, this advantage may have been diluted by the usage of the NOT function in the XCSCFC system.

A Wilcoxon signed rank test comparing XCSCFC with $XCSCF^2$ was performed. Both systems converged to the

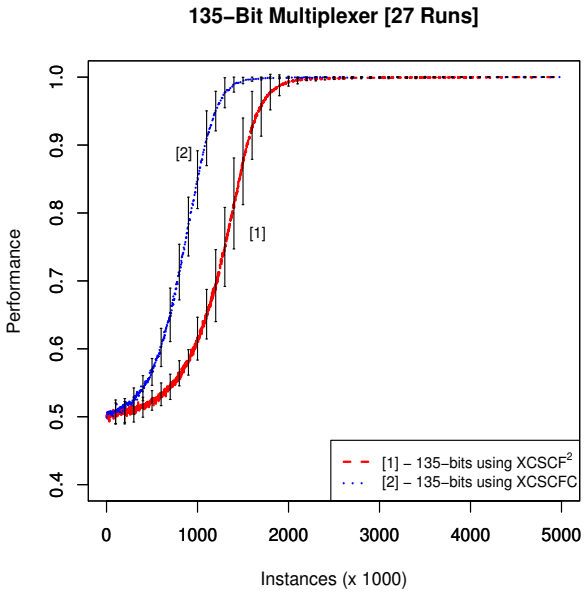

Figure 8: Results of the 135-bit multiplexer problem using XCS with code fragments and XCS with function reuse.

5 million mark and as expected, no difference was evident between both techniques.

5. DISCUSSION

Some of the results were as anticipated. For example, it was expected that function reuse would provide an improvement over the standard code fragment system because of the reduced search space; however, the magnitude of performance difference was interesting. For the 6 bit, 11 bit, and 20 bit multiplexer problems the $XCSCF^2$ system consistently solved the problems by utilizing much less instances than the XCSCFC system. One could also note, that $XCSCF^2$ was slower and as the complexity of the problems increased, so did the proportion of time required by the system.

On the contrary, when it came to the 37 bit multiplexer, the $XCSCF^2$ system took more instances and more time to solve the problem as compared with the XCSCFC system. It appears that by this time the initial efficiency on the part of the function reuse system was no longer sufficient to provide efficient performance against the competitor system. It is apparent that the initial success on the part of the function reuse system was most likely due to the XOR function. The standard XCSCFC system utilized the AND, OR, NOT, NAND, NOR hard coded in the modules while the code reuse system was trained with NAND, OR, AND, NOR, XOR and the resulting rule populations were incorporated into the newly learned functions. Functions that were different at this stage were the NOT and XOR and this might be the reason for the differing performance between the two systems. It is very possible that the XOR function leads to shallower code fragment trees hence a smaller search space.

In GP and XCSCFC a human is required to predefine the function set, whereas in $XCSCF^2$ it is anticipated to be able to efficiently select the most appropriate functions from its past experiences. This is important as a human may not know which functions are most suited to a given problem domain, e.g. whether the XOR function is more suitable than the NOR function for the multiplexer domain. Including all possible functions is currently intractable to search as there is no guidance on which functions are related to each other to aid genetic search or their suitability to a given domain. It is envisaged that $XCSCF^2$ will be able to perform a guided search for the most suited learned functions as it will know the 'family tree' for a function and which functions were utilised in certain problems. Although a human must decide upon these experiences it is anticipated that through linking rule-sets, future searches will be more efficient/effective.

For the more complex problems such as the 37, 70 and 135 bit multiplexers, there was also knowledge transfer taking place. As in the less complex cases, the code fragments learned from the previous levels were used in certain cases to create new trees. For example the following is a code fragment that was part of the solution for one of the 70 bit problems with identification (id) 73:

$$D2\ D2\ M\ D0\ D1\ N\ c\ o - - - - - - - - - - > 73$$

During the execution of the 135 bit problem this same code fragment was reassigned the id number 138:

$$D2\ D2\ M\ D0\ D1\ N\ c\ o - - - - - - - - - - > 138$$

During the execution of the same 135 bit problem, Code Fragment 138 was reused in Code Fragment with id 190.
$$D1\ CF\_155\ cD5\ CF\_138\ m\ M\ o - - - - - - - - - - > 190$$

The term "CF_138" signals to the system that it is using Code Fragment 138 in that specific node of the tree. Taking a closer look at the first line once again, the term D2 stands for the terminal value from the environment message with index 02. Similarly D0 and D1 stand for the message bits 00 and 01. The letters M, N and c stand for some previously learned function. The letter o stands for "No Operation", which in this case signifies the end of the Code Fragment. The right facing arrow is inconsequential and only points to the CF id, which in this case is 73.

The 70 bit multiplexer experiments showed that the trends in performance continued as they had been previously established in the 37 bit experiments, in addition to this, both systems demonstrated an increase in "don't care" fragments being used, however the $XCSCF^2$ system did produce more of these fragments as part of the final rules. It is believed that this was due to the fact that $XCSCF^2$ was utilizing NOR learned function where XCSCFC was using the NOT hard coded function. This led to more general Code Fragments as part of the final rules.

During the 135 bit multiplexer experiments there was also a higher performance on the part of the XCSCFC system; however, the actual ratio of performance between both systems actually remained quite similar. For the 70 bit multiplexer the ratio of instances needed to learn the problem for the XCSCFC to the $XCSCF^2$ system was approximately 0.65 while for the 135 bit multiplexer it was about 0.69.

Ultimately this approach seeks to accumulate a network of learned functions in different groupings. For example, it is envisioned that there will be a group of logic functions such

as NAND, OR, etc. There will also be a group of functions that would have proven useful for solving the multiplexer problems. It is believed that further learning of new functionality is possible and one potential candidate is a function that translates a binary input into a decimal number.

6. CONCLUSIONS

This paper showed that by using $XCSCF^2$, it is possible to learn building blocks of functionality from basic boolean rules. These Function-RuleSets were stored and used to solve the subsequent problems. In addition, it was shown that knowledge learned in the boolean logic domain could be used to solve problems in a different domain such as the multiplexer. This in fact is the crux of the argument; that it is possible to transfer knowledge learned in one domain to solve problems in a different, but related domain.

The $XCSCF^2$ system was able to achieve this for the 6-20 bit multiplexer problems with better than expected results, in terms of instances needed to learn the problems. However, it was slower in larger scale problems, but not prohibitively so.

By using the learned functions it was possible for the system to evaluate the code fragment trees at the appropriate time in the process. It is anticipated that by linking the new code fragments to their respective functions, it will make the system even more efficient in terms of instances needed for solving a particular problem. This would be a crucial step in order to fully implement cross-domain knowledge transfer that includes Function-RuleSets as well as their particular set of code fragments.

In this work it has been shown that there is potential for function re-use by transferring learned knowledge within the same domain or between related domains. It has also been shown that rule sets can be used as functions in code fragment trees.

7. REFERENCES

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming An Introduction*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[2] L. Bull and M. Ahluwalia. Coevolving functions in genetic programming: Classification using k-nearest-neighbor. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, 2:947–952, July 1999.

[3] M. V. Butz, M. Pelikan, X. Llora, and D. Goldberg. Extracted global structure makes local building block processing effective in XCS. *IlliGAL Report*, 2005011:1045–1052, February 2005.

[4] M. V. Butz and S. W. Wilson. An algorithmic description of XCS. *Soft Computing*, 6:144–153, 2002.

[5] M. Iqbal, W. N. Browne, and M. Zhang. Evolving optimum populations with XCS classifier systems. *Soft Computing*, 2013(17):503–518, September 2012.

[6] M. Iqbal, W. N. Browne, and M. Zhang. XCSR with computed continuous action. *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, pages 350–361, 2012.

[7] M. Iqbal, W. N. Browne, and M. Zhang. Comparison of two methods for computing action values in XCS with code-fragment actions. *GECCO'13 Companion*, pages 1235–1242, July 2013.

[8] M. Iqbal, W. N. Browne, and M. Zhang. Extending learning classifier system with cyclic graphs for scalability on complex, large-scale boolean problems. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1045–1052, 2013.

[9] M. Iqbal, W. N. Browne, and M. Zhang. Learning complex, overlapping and niche imbalance boolean problems using XCS-based classifier systems. *Evolutionary Intelligence*, 6(2):73–91, 2013.

[10] M. Iqbal, W. N. Browne, and M. Zhang. Learning overlapping natured and niche imbalance boolean problems using XCS classifier systems. *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1818–1825, 2013.

[11] M. Iqbal, W. N. Browne, and M. Zhang. Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems. *IEEE Transactions on Evolutionary Computation*, 2013. DOI: 10.1109/TEVC.2013.2281537.

[12] J. R. Koza. *A Hierarchical Approach to Learning the Boolean Multiplexer Function*. Morgan Kaufmann Publishers, Inc., 1990.

[13] P. Lanzi. Extending the representation of classifier conditions part i : From binary to messy coding. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, 1:337–344, July 1999.

[14] P. Lanzi. XCS with stack-based genetic programming. *Evolutionary Computation*, 2:1186–1191, 2003.

[15] P. Lanzi and A. Perrucci. Extending the representation of classifier conditions part ii : From messy coding to s-expressions. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, 1:345–352, July 1999.

[16] P. L. Lanzi. Learning classifier systems: Then and now. *Evol. Intel.*, 1:63–82, 2008.

[17] N. Nisan and S. Schocken. *The Elements of Computing Systems : Building a Modern Computer from First Principles*. MIT Press, Cambridge, Massachusetts, 2008.

[18] R. J. Tocci, N. S. Widmer, and G. L. Moss. *Digital Systems : Principles and Applications*. Prentice Hall, Upper Saddle River, New Jersey, 2011.

[19] R. J. Urbanowicz and J. H. Moore. Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, 2009(736398):1–25, June 2009.

[20] N. Wiener. *Cybernetics: or Control and Communication in the Animal and the Machine*. The MIT Press, Cambridge, Massachusetts, 2006.

[21] S. Wilson. A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

[22] S. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2), 1995.

[23] S. Wilson and O. Sigaud. Learning classifier systems: A survey. *Soft Computing*, 11(11):1065–1078, September 2007.