Human-inspired Scaling in Learning Classifier Systems: Case Study on the n-bit Multiplexer Problem Set

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

Learning classifier systems (LCSs) originated from artificial cognitive systems research, but migrated such that LCS became powerful classification techniques. Modern LCSs can be used to extract building blocks of knowledge in order to solve more difficult problems in the same or a related domain. The past work showed that the reuse of knowledge through the adoption of code fragments, GP-like sub-trees, into the XCS learning classifier system framework could provide advances in scaling. However, unless the pattern underlying the complete domain can be described by the LCS's representation of the problem, a limit of scaling will eventually be reached. This is due to LCSs' 'divide and conquer' approach utilizing rule-based solutions, which entails an increasing number of rules (subclauses) to describe a problem as it scales. Inspired by human problem-solving abilities, the novel work in this paper seeks to reuse learned knowledge and learned functionality to scale to complex problems by transferring them from simpler problems. Progress is demonstrated on the benchmark Multiplexer (Mux) domain, albeit the developed approach is applicable to other scalable domains. The fundamental axioms necessary for learning are proposed. The methods for transfer learning in LCSs are developed. Also, learning is recast as a decomposition into a series of sub-problems. Results show that from a conventional tabula rasa, with only a vague notion of what subordinate problems might be relevant, it is possible to learn a general solution to any n-bit Mux problem for the first time. This is verified by tests on the 264, 521 and 1034 bit Mux problems.

CCS Concepts

•Computing methodologies → Machine learning algorithms;

GECCO '16, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908812.2908813

Keywords

Learning Classifier Systems, XCS, Genetic Programming, Scalability, Transfer Learning

1. INTRODUCTION

Learning Classifier Systems (LCSs), first introduced by Holland, were originally cognitive systems designed to evolve a set of rules; LCSs were inspired by the principles of stimulus-response in cognitive psychology [3], [4], [9], [10], [24]. LCSs migrated away from being platforms to study cognition and instead have been developed to be powerful classification techniques [20]. One of the strengths of LCSs is their capability of subdividing the problem into niches that can be solved efficiently. This is accomplished by integrating generality in the rules produced. This pressure towards generality means that a single classifier could be a solution to more than one problem instance. In other words LCSs can divide the problem space into niches thereby creating smaller 'problems' from the whole [2].

Even with these advantages certain problems are difficult to solve, e.g. the multiplexer problem, due to underlying problem characteristics such as epistasis (the importance of certain bits depends on the values of other bits) and heterogeneity (when individual solutions or groups of individuals are independently predictive of the same phenotype). These phenomena were studied by Urbanowicz et al. [26] who introduced attribute tracking as a method for characterizing heterogeneity and interaction in Michigan style LCSs applied to supervised learning problems. Although this approach is useful and potentially far-reaching, it considers a complete supervised instance, whereas this work seeks to extend the reinforcement learning 'divide the problem into subordinate parts' approach.

Reusing learned knowledge has been shown to increase scalability and could provide 'shortcuts' that decrease the search space of the problem at hand [16]. Keeping scalability at the forefront of the proposed work, one of the major aims is to discover ontologies of functions that will map to numerous, heterogeneous structures. This will aid in obtaining a compact and optimal set of classifiers at each of the proposed steps [23].

In the field of Developmental Learning there is a term known as the Threshold Concept [6]. This idea conveys the fact that in human learning there exist concepts that are very influential in advocating the learning of a task. These concepts need to be learned in a certain order to enable the person to progress towards learning more difficult ideas

at a faster pace than otherwise. For instance, humans are taught mathematics in a certain progression; arithmetic is taught before trigonometry and these two are taught before calculus. The empirical evidence points to the fact that this sequence will be more effective in fostering the learning of progressively more difficult mathematics [6].

Related to the benefits of the threshold concept are Layered Learning (LL) and Transfer Learning (TL). In LL a sequence of knowledge is learned [25] in self-contained stages. In TL, learned knowledge from a source domain is transferred to a similar or related target domain [1]. LL transfers the complete solutions of sub-problems, where TL can also transfer part-solutions [7].

Modern LCSs can be used to extract building blocks of knowledge in order to solve more difficult problems in the same or a related domain. The past work showed that the reuse of knowledge through the adoption of code fragments (CFs), GP-like sub-trees, into the XCS learning classifier system framework could provide dividends in scaling.

A series of systems using CFs have been developed. XCSCFC is a system that has extended XCS by replacing the condition of the classifiers with a number of CFs. The action part of the classifiers remains the same. Although XCSCFC exhibits better scalability than XCS, eventually, a computational limit in scalability will be reached [14]. The reason for this is that multiple CFs can be used at the terminals, as the problem increases in size, then any depth of tree could be created.

Another type of CF system is XCSCFA. In this type the condition part retains its original ternary alphabet representation but the action part of the classifier is replaced by a CF. The terminals in the CF tree can be replaced with either the corresponding bits from the environment message or with bits from the classifier condition [16]. This method produced optimal populations in discrete domain problems as well as in continuous domain problems [16]. This however lacked scaling to very large problems, even if they had repeated patterns in the data. XCS with State-Machine Action (XCSSMA) was introduced with the ability to generate state machines to encapsulate repeating patterns. This was accomplished by replacing the numeric action in XCS with a Finite State Machine. XCSSMA evolved compact and easily interpretable general solutions for the even-parity and the carry problem domains, however it could not improve the generalization for the multiplexer (Mux) domain. This was because the state machines needed for this domain are more complex than the other domains mentioned above [14].

One representation (an alphabet for the condition/action encoding) for solving Mux problems is by using Disjunctive Normal Form (DNF). However DNF presents some limitations; as the problem grows, the DNF grows very large as well. The 6-bit multiplexer can be solved but is verbose, see equation 1 (the primes indicate negation and 'X' the features (bits) in a multiplexer string) [27]:

$$F6 = X0'X1'X2 + X0'X1X3 + X0X1'X4 + X0X1X5 \quad (1)$$

A human would solve the problem differently, they would realize that there is a relationship between the address bits and the data bit even if presented with just a binary input string and no prior knowledge of the underlying problem structure. It is hypothesized that humans scale to complex problems by considering the underlying patterns in such problems that are generated from previous, smaller scale problems thus transferring knowledge and skills from related problems. This learned knowledge could be from different domain(s) as some of the functionality that is needed may not exist within the problem itself. It is anticipated that when this approach is adapted into evolutionary computational approaches it will lead to advances in scalability of these techniques.

The aim of this work is to adapt the notion of threshold concepts coupled with TL into LCSs to produce general solutions to large scale problems, this will be demonstrated through the often used benchmark Multiplexer problem. The Multiplexer problem is one that lends itself for research because it is difficult, highly non-linear and has epistasis, e.g. the importance of the data bits is dependent on the address bits. A multiplexer can be viewed as a logic circuit where a certain number of bits provide the address of the output bit. If L is the length of the input then the equation:

$$L = k + 2^K \quad (2)$$

defines the relationship between the length of the input and the number of address bits required. For example, for the 6-bit multiplexer problem 110001, the number of bits that determine the output bit would be 2 and the remaining 4 are the data bits. In this case the address bits convert to 3 in decimal, which gives us the fourth data bit counting from the left, and an output of 1. Importantly, in order to compute the above formula, the software agent would need to be aware of the *power base two* function as well as *real number addition*. These functions are not part of the boolean domain, but humans have already learned these functions, therefore they can readily include them in their reasoning of this domain.

The specific research objectives are as follows:

* **Develop** methods such that learned knowledge and learned functionality can be reused for Transfer Learning of Threshold Concepts.

* **Determine** the necessary axioms of knowledge, functions and skills needed for any system from which to commence learning.

* **Demonstrate** the efficacy of the introduced methods in one complex domain, i.e. Mux. Although the underlying methods are applicable to many domains, page limitations prevent more than one domain being investigated here.

2. BACKGROUND

The original description of the LCS was of a cognitive system [9]. The implication was that this type of system could learn about its environment and about its state in order to execute beneficial actions on its environment. Cognition is a term from psychology that is defined as the things persons know about themselves, about their behavior, and about their surroundings [8]. This implies the ability to factor seemingly disparate information and then apply it towards a desired goal.

[1]Some fields define TL as transferring the underlying model [22]

Since the original inception of LCSs, the number of applicable alphabets has been expanded to include more representations such as S-Expressions and Code Fragments. S-Expressions are LISP-like expressions which have been used to express the LCS condition or action [19]. A Code Fragment (CF) is an expression, similar to a tree generated in Genetic Programming [13]. CFs generate small blocks of code in binary trees with an initial maximum depth of two. This depth was chosen, based on empirical evidence, to limit bloating caused by the introduction of large numbers of introns. Analysis suggests that there is an implicit pressure for parsimony [15].

LCSs can reuse learned knowledge to scale to problems beyond the capabilities of non-scaling techniques, i.e. those that do not reuse learned information from smaller problems. One such technique is XCSCFC [16]. This approach uses CFs to represent each condition bit enabling feature construction in the condition of rules. The action part uses the binary alphabet {0, 1}.

XCSSMA can find repetitions and loops in order to repeat useful behaviors. This type of system has been able to evolve general solutions for the n-bit parity problem and n-bit carry problem. It uses a Finite State Machine (FSM) in the action of the classifiers instead of the original alphabet [14]. Repeating patterns, which could also be represented as loops, is an important concept of this technique and will be needed for many domains suited to the proposed approach. The Mux problem does not have direct loops and hence XCSSMA is not well suited for this purpose.

It has been shown previously that rule-sets learned by an LCS system, termed XCSCF^2, can be re-used in a manner akin to functions and their parameters in a procedural programming language [1]. These learned functions, termed X, then become available to any further tasks. These are composed of learned rule-sets that map inputs to outputs, and skills, both would compute actions from given inputs and would manipulate inputs.

XCSCF^2 places the emphasis on user-specified problems, rather than user-specified instances, which is a subtle but important change in emphasis in EC approaches. This technique however, lacks a rich representation at the action part, which will need adjusting due to the different types of action values expected in this work, e.g. binary, integer, string.

The multiplexer problem is a complex and difficult problem due to epistasis and its large search space. An early attempt at scaling was the S-XCS system that utilizes optimal populations of rules, which are learned in the same way as classical XCS [12]. These optimal rules are then fed to S-XCS as messages thus enabling abstraction. The system uses human constructed functions such as Multiply, Divide, PowerOf, ValueAt, AddrOf, among others [12]. Although these key functions provide the system with the building blocks to piece together the necessary knowledge blocks, they have an inherent bias and might not be available to the system in large problem domains. It also assumes completely accurate populations whereas the system to be developed here is required to learn both the population and functionality from scratch. If supervised learning is permitted (unlike in this work), the heterogeneous approach of ExSTraCS scales well; up to the 135 Bit Mux [26].

Previous work has considered constructing general solutions to Boolean problems, e.g. Parity problem, from non-domain specific instructions (equivalent to functions here) [11]. This top-down approach predefined the instruction set, including loop mechanisms, for a virtual register machine, which successfully created general solutions occasionally with ineffectual instructions and the need for domain-specific knowledge as input. The system developed here is to be bottom-up with the ability to learn functions plus associated knowledge to improve compactness and flexibility.

3. THE METHOD

This work disrupts the standard learning paradigms in EC by aligning it with LL. Instead of the EC researcher specifying the format of an individual domain and manually adjusting the paradigms (using human knowledge to define the algorithm's parameter values, terminal set and function set), the method here is to specify the order of problems/domains (together with robust parameter values) while allowing the system to automatically adjust the terminal set through feature construction and selection, and ultimately develop the function set. This is different to the self-contained stages of LL, and closer to TL in this respect, as part-solutions (CFs) can be propagated. Similarly, complete solutions (learned functions) can form part-solutions for future problems. This is analogous to a school teacher determining the order of threshold concepts for a student in a curricula [21]. The system can use learned rule-sets as functions along with the associated building blocks, i.e. CFs, that capture any associated patterns, which is an advantage over pre-specifying functionality in EC and LL.

This method changes the fundamental problem from finding an overarching 'single' solution that covers all instances or features of a problem to finding the structure (links) of sub-problems that construct the overall solution. Learning the underlying patterns that describe the domain is anticipated to be more compact and reusable as they do not grow as the domain scales (unlike individual solutions that can grow impractically large as the problem grows, e.g. DNF solutions to the multiplexer problem).

3.1 n-bit Multiplexer Problem

In the multiplexer problems the number of address bits is a function of the length of the message string and grows along with the length. The search space of the problem is also adequate to show the benefits of the proposed work. For example, for the 135-bit multiplexer the search space consists of 2^{135} combinations, which is vastly beyond enumerated search [18].

An example of a 6 bit multiplexer is shown in Figure 1. Determining the number of address bits [2] k requires the use of the *log* function, as shown in equation 3, in this case k is 2. Then k bits must be extracted from the bit string to produce the two address bits. The next step is to convert the address bits into decimal form; this requires knowledge of the *power base 2* function as well as rudimentary *looping*, *addition* and *subtraction* functions. Depending on the approach to this step, *multiplication* may also become a requirement. The two address bits translate to 2 in decimal form, as shown in figure 1, i.e. A0 and A1. The decimal number points to the data bit D2 that contains the value to be returned. The

[2]It is possible to have a Multiplexer with address bits that are not the first k bits from the left, so the system learns this pattern without assumptions

index begins at 0 and proceeds from the left towards the right, as shown in figure 1.

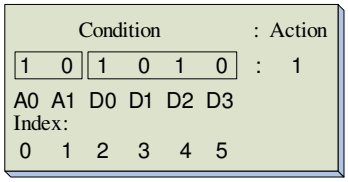


Figure 1: 6 Bit Multiplexer showing the address bits and the data bits of the condition, this distinction is not provided to the learning system.

One of the underlying reasons for choosing the multiplexer problem domain for the work is because humans can solve this type of problem by naturally combining functions from other related domains along with functionality from the Boolean domain. For instance, a human can reason that there is a correspondence between the address bits and the data bits without being given any *a priori* knowledge about the problem. Humans are also able to reason that some functions in their 'experiential toolbox' may be appropriate for solving the problem. The experiential toolbox is the totality of learned functionality for the agent. These functions include *multiplication*, *addition*, *power*, and the notion of a number line. Thus the agent here must build-up its own toolbox of functions and associated blocks of knowledge (CFs). Therefore, the agent will have to be guided in its learning so that it may have enough cross-domain functions to successfully solve the problem. It will need to perform well with more functions than necessary as the exact useful functions may not be known *a priori*, but at this stage of paradigm development is not expected to be able to adjust to fewer functions than necessary.

Besides functions, the experiential toolbox will also contain skills. These are capabilities that the agent will have learned or will have been given *a priori*; one example is the looping skill. These skills are subtly hidden from the agent and together with the functions provide a fertile ground from which to sprout useful knowledge. For example, a human understands all the operations required for counting k number of bits, starting from the left of the input string. Then the human would have to reason how to convert the address bits to decimal, which requires the ability to multiply and add. If we wanted to make it even more difficult we could have the human determine the number of k address bits required for a particular problem. In this case the formula:

$$k = \lfloor \log_2 L \rfloor \tag{3}$$

provides the functionality for determining the number of k address bits by using the length of the input. In this case the person would have to be conversant with the natural log function as well as the floor function. A human would eventually be able to determine the address bits with increasing difficulty but a software system would have to learn this functionality before even attempting to solve the n-bit multiplexer.

3.2 Proposed system

The proposed system, termed XCSCF*, consists of a number of components. Since different types of actions are expected, e.g., Binary, Integer, Real; it is proposed that the functions be created by XCSCFA based systems, although any rule production system can also be used, e.g. XCS, SC-SCFC, etc. This will facilitate the use of real and integer values for the action as well as enabling it to represent complex functionality. The proposed solution will reuse learned functionality at the terminal nodes as well as the root nodes of the CFs since this has been shown to be beneficial for scaling. XCSR would not be helpful here because on a number of the steps, the possible actions are not a number but a string e.g., kbitstring. Moreover, XCSR with Computed Continuous Action would present unnecessary complications to the work because the condition of the classifiers do not require continuous values [13].

The inputs and outputs of the overall system consist of a Mux instance (bit string) and its integer length L, which is known to standard techniques addressing the Mux problem, and an output of binary type at the very end. Base functionality will have to be provided for the system; we call these Axioms, e.g. *log*. For example, it is anticipated that basic functions like addition, subtraction, multiplication, division, natural log, power base 2 will have to be provided to the systems to bootstrap learning of the target problem.

The decomposition of the system is reminiscent of the layered learning paradigm proposed in [25]. As such, the proposed system is separated into several hierarchical parts; each part facilitates the learning of the subsequent one. This is accomplished at each layer by providing a population of previously learned CF-based rules and CFs linked to those rules. This complement constitutes one learned function.

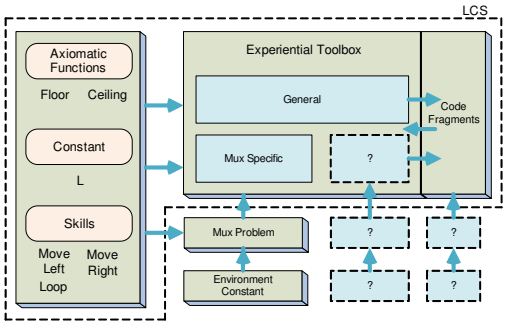


Figure 2: Training encompasses different types of functions, skills and axioms. The experiential toolbox will contain general as well as multiplexer specific learned functionality. The question marks indicate the next domain and functionality learned from it.

3.3 Individual Detailed Components

One plausible way of separating the Mux problem into subordinate problems has five parts. Each subsequent part builds upon the rules learned from the previous step as well as from the Axioms provided. Figure 2 depicts the interplay between the Axioms, skills and learned functionality and their CF representation. The figure also depicts how the type of problem tackled can feed domain specific functionality into the experiential toolbox of the system. This is shown by the arrow flowing from the Mux problem towards the Experiential Toolbox.

At each step, the system has available the environmental message, constants and hand-coded functions, as well as the learned CFs and functions. To bootstrap the learned CF functions the NAND boolean operator will be learned in a

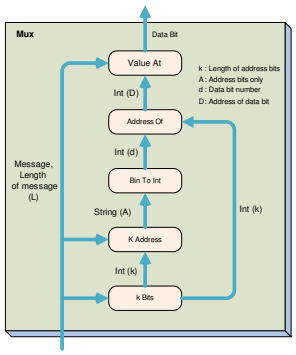


Figure 3: Multiplexer training flow - the message is utilized at three steps, while the k bits are used in two steps when learning.

standard XCS and the resulting rules will be available to the *n-Bits* system to formulate a bootstrap for the learning process. Table 1 depicts a listing of all the skills provided for the system along with their tags (used to interpret results) and their input/output data types. Table 2 shows a listing of the constant(s) provided to the system, note that these were provided in order as a curricula, but a system could be developed to address all problems in parallel, such that as each problem is solved it becomes available (CFs and function) to the remaining problems. Table 3 shows a listing of the functions to be learned.

It is important to note that the work presented here does not seek to provide a recipe for a system to follow to just arrive at the solution to a given multiplexer problem. The aim here is to facilitate learning in a series of steps, where in this case the learned functionality could potentially help a system to arrive at a general solution of *any* Multiplexer problem. In other words, it is important for the system to learn to combine the different learned functions in a way conducive to learning, a way that will produce a general solution. The number of subordinate problems can always be increased in the future, e.g. learning basic functions such as an adder or a multiplier via Boolean functions or even learning the log function from training data.

3.3.1 Sub-Problem - kBits

The first stage is to determine the number of k address bits that will contribute to the solution for the n-bit multiplexer. The constant LEN provides the system with the length of the environment message, where LEN is an environment constant that is normally hard-coded in a learning system rather than input as a constant. The training data-set used consists of instances of possible Mux lengths and the corresponding number of address bits.

3.3.2 Sub-Problem - kBit String

This part extracts the first k bits from a given input string. The data-set will be random bit strings, say length 6, and a given k length where the action is the first k bits.

3.3.3 Sub-Problem - Bin2Int

The third part entails converting a binary number to an integer. This is important because the system requires this information to be able to determine the position of the data bit. However, this is not a trivial task as the system would need to be aware of many functions that a human would

Table 1: Functionality Provided (Hard-coded functions)

Function	Tag	Input	Output
Floor	[	Float	Integer
Ceiling	]	Float	Integer
Log	{	Float	Float
BinaryString	$	Integer	String
Power 2 Loop	@	String	Variant
Add	+	Variant	Integer
Subtract	-	Float	Float
Multiply	*	Float	Integer
Divide	/	Float	Integer
ValueAt	=	Integer	Binary

Table 2: Constant(s) Provided

Constant	Tag	Input	Output
LEN	L	NA	Variant: any type

Table 3: Functions to be learned

Function	Tag	Input	Output
KBits	[	Float	Integer
KBitString	]	Float	Integer
Bin2Int	{	Float	Float
AddressOf	-	Float	Float
ValueAt	=	Integer	Binary

potentially already have in their experiential toolbox. The data-set will be random strings, say length 6, with action being the equivalent integer number.

3.3.4 Sub-Problem - AddressOf

This functionality determines the location of the data bit from an input string and known address [this is a harder problem than learning addition of address length and decoded length]. The data-set will be random strings (length 6) and decoded address with the integer action.

3.3.5 Sub-Problem - ValueAt

The functionality to be learned is to return the bit referenced from a bit string. The system is trained using a dataset of bit strings of known length (again length 6) with a reference integer and corresponding output bit.

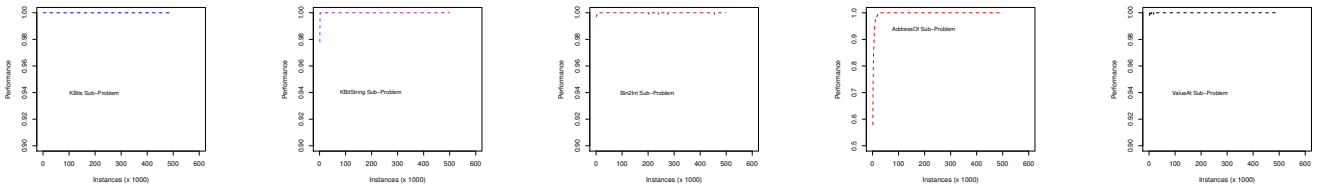
4. RESULTS

4.1 Experimental Setup

The experiments were run 30 times with each having an independent random seed. The stopping criteria was when the agent completed the allotted number of training instances, which were chosen based on preliminary empirical tests on the convergence of systems. The proposed systems were compared with XCSCFC and with XCS. The settings for the experiments are common to the LCS field. They were as follows: Payoff 1,000; the learning rate $\beta = 0.1 - 0.2$; the Probability of applying crossover to an offspring $\chi = 0.8$; the probability of using a don't care symbol when covering $P\_don'tCare = 0.33 - 0.95$; 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. In addition, error threshold ϵ_0 has been set to 10.0. This range became necessary as the problems increased in complexity.

4.2 Experimental Tests

Figures 4a - 4e, show that training was successful in Subproblems, which enabled the Mux problem to reuse the learned functionality and CFs. The number of rules and CFs generated by the fundamental parts, i.e. kBits, kBitString, Bin2Int, AddressOf and ValueAt was small. In certain cases



(a) Kbits (b) KbitString (c) Bin2Int (d) AddressOf (e) ValueAt

Figure 4: Training results for the five Mux Sub-problems.

the number of rules produced was one, as was the case with run 8 of the kBits sub-problem. This is plausible as the rule is general and will work for any length input string. This same trend continued with the more difficult problems. The condition was composed of don't cares, although it may appear counter-intuitive, this is normal as the rules were anticipated to be maximally general. It is considered that part of the reason that there were so many don't cares in the condition was that the setting for $P\_don'tCare$ for this problem was 0.95 to enable matching of conditions, plus no niching was needed with CF-based actions in this domain.

Figure 5 shows that only the proposed system XCSCF* and XCSCFC were able to solve the 135 bit Multiplexer. These experiments followed the standard explore and exploit phases as in XCS. This shows scaling by relearning, but it is the capturing of the underlying patterns without retraining that is the aim of this work.

Tests were conducted on the final rules produced by the 6 bit multiplexer to determine if they were general enough to solve more complex problems. Figure 6 shows that the rules produced by the 6 bit multiplexer were able to solve the 264 bit, 521 bit and 1034 bit multiplexer problems. The system used to test the generality of the rules was a version of XCSCFA with certain modifications; there was only the exploit phase and no covering was allowed.

Note that 2^{1034} is a vast number, meaning that testing a million instances is a fractionally small sub-sample, but will identify many deficiencies.

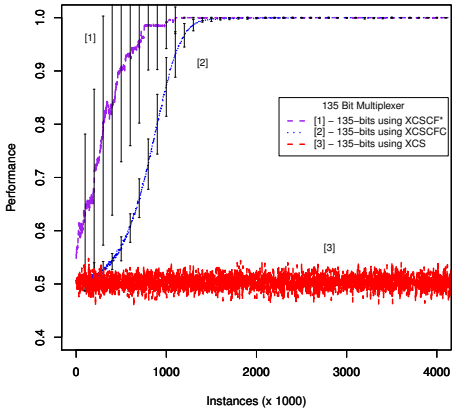


Figure 5: 135 Bit Multiplexer Solution. Note: Wilcoxon signed rank test comparing XCSCF* with XCSCFC shows no evident difference between both techniques.

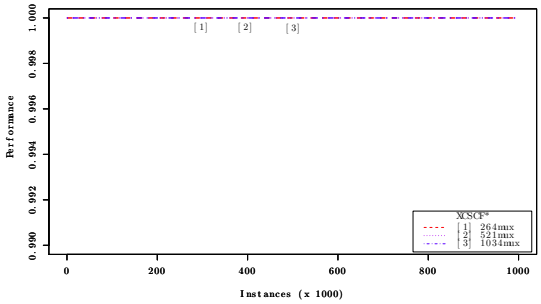


Figure 6: 264, 521 and 1034 Bit Multiplexer Solution. These results with XCSCF* did not involve any training, just the test phase.

4.3 Sub-Trees Generated

The solution tree for a 6 bit Multiplexer is shown in Figure 7. The fully expanded tree is shown in Figure 8. It is clear that the underlying chains of CFs are very long and are composed of functional blocks that are repeated in different branches. This presented an opportunity for the system to store these computed values for later usage. This also shows that while the final rules for the difficult problems can appear simple at first glance (Figure 7), the underlying functionality is quite complex and is composed of the necessary combinations of skills and learned functions that were determined by the system to provide a viable solution to the problem.

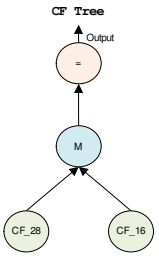


Figure 7: 6 Bit Multiplexer Solution Rule. Where '=' is the ValueAt function, 'M' is the AddressOf function and CF_16, CF_28 are Code Fragments.

5. DISCUSSION

It can be argued that the reason this approach can solve problems to a much larger scale than previously is that human knowledge separated the problem into appropriate, simpler sub-problems – noting that it is still a difficult task to

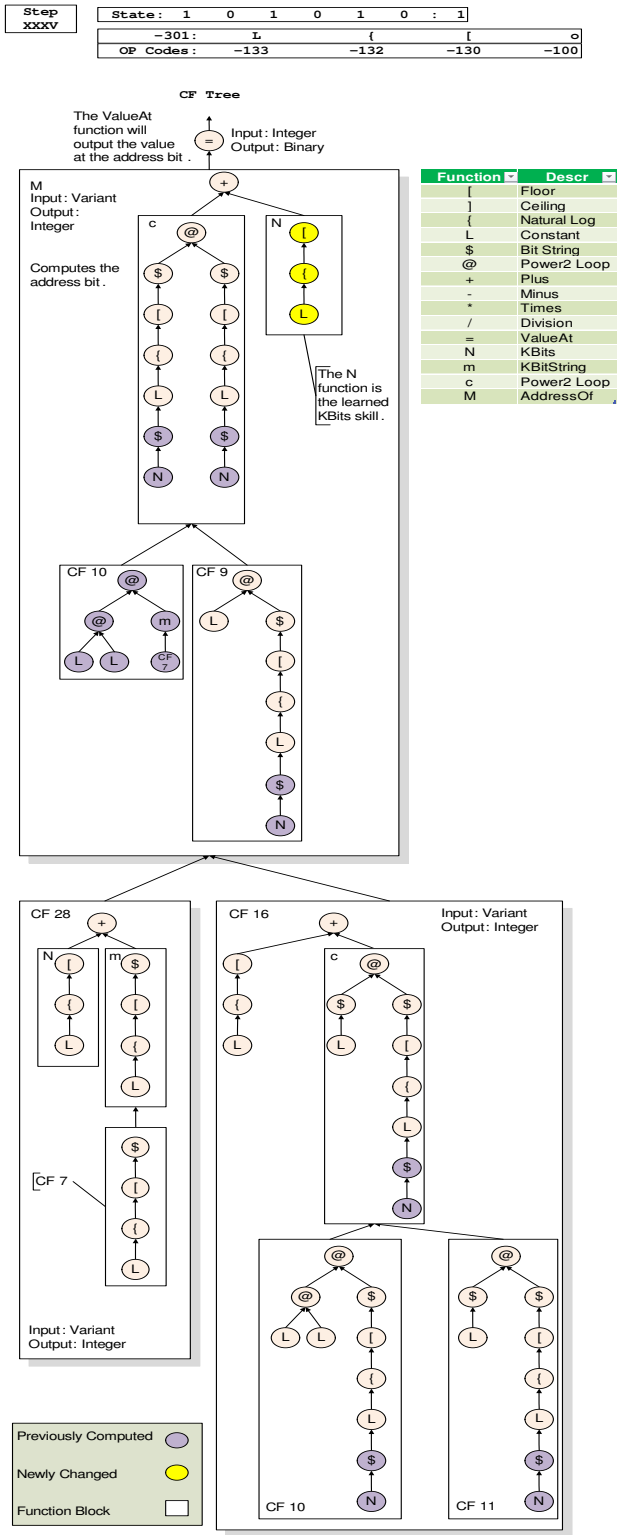


Figure 8: 6-bit Mux Solution Tree: boxes depict functions/CFs, purple circles depict previously computed subtrees, yellow circles represent the last sub-tree visited.

learn each sub-task – in such a way that learned knowledge/functionality can be transferred. However, it is important to keep in mind that the system still has to learn to combine these blocks effectively. The way that humans select sub-problems is similar to that of humans selecting function sets in standard EC approaches where too few or inappropriate selection prevents effective learning, while selecting too many unnecessary components inhibits training. In these experiments the ceiling function was available, but never used by the final solutions.

Another consideration is the impact that layered learning may have had in the performance of the proposed work. It is not clear, at this time, what benefits were provided by the decomposition of the problem versus reusing learned knowledge. It would be naive to attribute all the credit to one or the other technique without further considerations [25].

A system can be imagined where each sub-problem is addressed in parallel with the resulting functionality and building blocks becoming available in a shared repository. Simple (low-level) problems would complete first enabling higher-order functions to be consecutively solved. The links and order of solved problems would contain interesting meta-knowledge; a form of learning curricula.

It is clear that the work has benefited from the transfer of learned information from each of the components. Although a defined recipe was not given to the system, it was able to form logical determinations as to the flow of the accrued functionality, see Figure 8. This property of the system is akin to the derivation of a set of Threshold Concepts where significant learning towards the final target problem only progresses once the proper chain of functionality is formed and evaluated. Besides the structural similarities to layer learning, the solution tree shares an unexpected similarity with policy trees, such as the ones developed in [5]. The system also shares similarities with Run Transferable Libraries in the sense that each growing function represents a block of reusable code. Analogies could also be drawn to the concept of the Dynamic Linked Library (DLL), in essence all three constitute reusable program code that contains domain knowledge useful to the problem at hand [17].

The comparisons with XCS may appear unfair at face value, however they serve to highlight XCS' important role as the control system in the experiments. Furthermore, although the results are encouraging, it is not apparent that this technique will be effective in related domains such as the Hidden Multiplexer. Preliminary experiments have shown that the rules produced by the ValueAt stage can in fact solve the 18-bit Hidden Multiplexer, however they were ineffective with the 33-bit Hidden Multiplexer.

5.1 Additional Observations

Unlike the execution of XCSCFC, where the terminal nodes have a direct impact on the final answer returned by the CF sub-tree, it was discovered that for XCSCF* this does not appear to be the case. A full trace of the 6 bit multiplexer was conducted for one of the solutions, see Figure 8. It was discovered that repeating patterns of learned knowledge are interspersed throughout the chain of CFs. This suggests that it was beneficial to store the computed values of said functional blocks for later usage. It was also observed that for the general solution, the two main branches: CF_28 and CF_16 feed their outputs into the function *M* but they are not used directly when M is evaluated. The reason is that the learned function M relies on its own set of rules to compute its output, disregarding the inputs from its two child

branches. It is apparent that if both sub-branches were to be ignored, M would still furnish the correct answer as it has the relevant CF knowledge in its learned functions.

6. CONCLUSIONS

This paper has shown that starting from a conventional *tabula rasa* and using related problems, it is possible to learn a general solution to a complex problem domain, i.e. the multiplexer domain, through analogies to a human-like approach. By decomposing the problem domain into component sub-problems, providing the necessary axioms and transferring learned functionality plus knowledge, it is possible to discover general rules that can be applied to any size problem in the domain.

This was demonstrated by using XCSCF* to solve very difficult problems like the 264, 521 and 1034 bit multiplexer, which were previously insolvable by any other method. Although the aforementioned problems are comprised of a very large search space, the proposed technique successfully discovered a minimal number of general rules. One of the rules for a simple problem was traced fully and certain repeating patterns were discovered. Benefits to scaling are achieved by the system storing computed values of seen patterns, so they do not need to be recalculated. One important point to note is that it is not possible to verify each possible state of the more difficult Mux problems, such as the 1034 Mux, due to the very large search space.

Another very important observation was that not all of the available functionality was utilized in the final solutions. Thus, this style of learning system can have access to more functionality than necessary for a single problem. Future work will create a system with base axioms and a number of problems, including possible sub-problems, to be solved in a parallel architecture simultaneously. The 'toolbox' of functions (learned functions and axioms) plus the complementary knowledge (code fragments) will grow as problems become solved and will be available for addressing future problems.

7. REFERENCES

[1] I. M. Alvarez, W. N. Browne, and M. Zhang. Reusing learned functionality in XCS: code fragments with constructed functionality and constructed features. In D. V. Arnold and E. Alba, editors, *GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings*, pages 969–976. ACM, 2014.

[2] M. T. Banich and A. Belger. Interhemispheric interaction: How do the hemispheres divide and conquer a task? *Cortex*, 26(1):77–94, 1990.

[3] L. Bull. A brief history of learning classifier systems: from CS-1 to XCS and its variants. *Evolutionary Intelligence*, 8(2-3):55–70, 2015.

[4] M. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design*. Springer, Berlin, Germany, 2006.

[5] J. Doucette, P. Lichodzijewski, and M. I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In T. Soule and J. H. Moore, editors, *GECCO '12, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*, pages 97–104. ACM, 2012.

[6] N. J. G. Falkner, R. J. Vivian, and K. E. Falkner. Computer science education: The first threshold concept. In *LaTiCE*, pages 39–46. IEEE Computer Society, 2013.

[7] L. Feng, Y.-S. Ong, A.-H. Tan, and I. W. Tsang. Memes as building blocks: a case study on evolutionary optimization + transfer learning for routing problems. *Memetic Computing*, 7(3):159–180, 2015.

[8] L. Festinger. *A Theory of Cognitive Dissonance*. Stanford University Press, Stanford, California, 1957.

[9] J. Holland. Adaptation. In R. Rosen and F. M. Snell, editors, *Progress in Theoretical Biology*, volume 4, pages 263–293, New York, 1976. Academic Press.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The University of Michigan Press, Ann Arbor, 1975.

[11] L. Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Conference on Genetic Programming*, pages 158–166, July 1998.

[12] C. Ioannides and W. Browne. Investigating scaling of an abstracted LCS utilising ternary and S-expression alphabets. In J. Bacardit, E. Bernadó-Mansilla, M. Butz, T. Kovacs, X. Llorà, and K. Takadama, editors, *Learning Classifier Systems. 10th and 11th International Workshops (2006-2007)*, volume 4998/2008 of *Lecture Notes in Computer Science*, pages 46–56. Springer, 2008.

[13] 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.

[14] 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 GECCO '13*, pages 1045–1052, 2013.

[15] 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.

[16] 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.

[17] M. Keijzer, C. Ryan, and M. Cattolico. Run transferable libraries - learning functional bias in problem domains. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. K. Burke, P. J. Darwen, D. Dasgupta, D. Floreano, J. A. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. M. Tyrrell, editors, *GECCO 2004, GECCO , Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 531–542. Springer, 2004.

[18] J. R. Koza. A hierarchical approach to learning the boolean multiplexer function. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 171–192, San Mateo, 1991. Morgan Kaufmann.

[19] 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.

[20] P. L. Lanzi and R. L. Riolo. A roadmap to the last decade of learning classifier system research (from 1989 to 1999). In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *LNAI*, pages 33–62, Berlin, 2000. Springer-Verlag.

[21] J. H. Meyer and R. Land. *Overcoming barriers to student understanding: Threshold concepts and troublesome knowledge*. Routledge, New York, NY, 2006.

[22] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng*, 22(10):1345–1359, 2010.

[23] C. J. Price and K. J. Friston. Functional ontologies for cognition: The systematic definition of structure and function. *Cognitive Neuropsychology*, 22(3-4):262–275, 2007.

[24] J. D. Schaffer. Learning multiclass pattern discrimination. In J. J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA85)*, pages 74–79. Lawrence Erlbaum Associates: Pittsburgh, PA, July 1985.

[25] P. Stone and M. Veloso. Layered learning. In R. L. de Mántaras and E. Plaza, editors, *ECML*, volume 1810 of *Lecture Notes in Computer Science*, pages 369–381. Springer, 2000.

[26] R. J. Urbanowicz, A. Granizo-Mackenzie, and J. H. Moore. Instance-linked attribute tracking and feedback for michigan-style supervised learning classifier systems. In T. Soule and J. H. Moore, editors, *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*, pages 927–934. ACM, 2012.

[27] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.