Automatically Defined Functions for Learning Classifier Systems

Muhammad Iqbal
Victoria University of Wellington
Muhammad.Iqbal@

Mengjie Zhang
Victoria University of Wellington
Mengjie.Zhang@

Will Browne
Victoria University of Wellington
Will.Browne@ecs.vuw.ac.nz

ABSTRACT

This work introduces automatically defined functions (ADFs) for learning classifier systems (LCS). ADFs had been successfully implemented in genetic programming (GP)for various domain problems such as multiplexer and even-odd parity, but they have never been attempted in LCS research field before. ADFs in GP contract program trees and shorten training times whilst providing resilience to destructive genetic operators. We have implemented ADFs in Wilson's accuracy based LCS, known as XCS [14]. This initial investigation of ADFs in LCS shows that the multiple genotypes to a phenotype issue in feature rich encodings disables the subsumption deletion function. The additional methods and increased search space also leads to much longer training times. This is compensated by the ADFs containing useful knowledge, such as the importance of the address bits in the multiplexer problem. The ADFs also create masks that autonomously subdivide the search space into areas of interest and uniquely, areas of not interest. The next stage of this work is to implement simplification methods and then determine methods by which ADFs can facilitate scaling for more complex problems within the same problem domain.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Learning Classifier Systems, Genetic Programming, CUDA, Automatically Defined Functions, Pattern Recognition

1. INTRODUCTION

Genetic programming (GP) and learning classifier systems (LCS) are two population-based evolutionary computation techniques. GP has ability to generate a computer program automatically for a given task. LCS generate a population of production rules that together address a task. Both GP and LCS have been shown to solve complex problems, but are relatively slow compared with some other artificial intelligence techniques, such as decision trees. Often they provide non-optimally compact solutions in terms of program bloat and extraneous rules, which require simplification and compaction respectively.

The GP generated computer program is normally represented as a tree, which may contain unnecessary terms (bloat) and non-optimum expressions (a phenotypic behaviour may not be represented by the most compact genotype). Partly, in order to address these problems, ADFs have been successfully implemented in GP to make the tree program simple and concise [7]. ADFs also have the benefit that the evolution of the program tree is relatively protected from crossover and mutation disruptions. Speed up of performance is also claimed compared to GP without ADFs. Higher order functions (abstraction) have been investigated in LCS, but not widely adopted [5]. Thus, the objective of this work is to introduce ADFs in LCS. Comparisons will be made with ADFs in GP on the multiplexer problem.

One possibility for speeding up the performance of evolutionary techniques is to use the recently introduced graphical processing units (GPUs) that have been programmed for general purpose problem solving in order to leverage the power of the parallel processing. An initial study of GP in GPUs utilises the multiplexer problem domain [8], which has been widely studied in LCS research. Utilising the same computational platform and problem domain is anticipated to lead to interesting insight into the scaling and operation of both techniques. The introduction of ADFs in LCS is novel and the benefit of ADFs to GP and LCS in the CUDA platform will be investigated.

The ultimate aim of this work is to utilise the parallel threaded nature of GPUs to identify the building blocks of the optimal solution during the evolutionary process such that the more compact and scalable solutions may be determined faster than previously. The aim of this exploratory work is to examine whether the ADFs created LCS provide useful building blocks of information.

The rest of the paper is organised as follows. Section 2 describes the learning classifier systems concept, tree-based genetic programming, automatically defined functions and general purpose graphics processing units. In section 3 the benchmark GP technique is detailed, which is extended to include ADFs and subsequently ADFs are introduced in

GECCO'11, July 12–16, 2011, Dublin, Ireland.

LCS. In section 4 experimental results are presented and compared after introducing the multiplexer problem domain. Section 5 is discussion comparing LCS and GP with and without ADFs on the CUDA platform. In the next sections this work is concluded and the future work is outlined.

2. BACKGROUND

2.1 Learning Classifier System

Traditionally, a learning classifier system represents an agent enacting in an unknown environment via a set of sensors for input and a set of effectors for actions. After observing the current state of the environment, the agent performs an action, and the environment provides a reward, as depicted in Figure 1. LCS can be applied to a wide range of problems including reinforcement learning problems, classification problems and function approximation. LCS have also been adapted to supervised learning where the environment also returns the 'correct' optimal action through the UCS (sUpervised Classifier System) framework [11].

XCS is a formulation of LCS that uses accuracy-based fitness to learn the problem by forming a complete mapping of states and actions to rewards. The agent has two modes of operation, explore (training) and exploit (application), that can be described as follows (for a more detailed description refer to [14]).

In the explore mode the agent attempts to obtain information about the environment and describe it by creating decision rules:

1. observes the current state of the environment, $s \in S$.
2. selects classifiers from the classifier population $[P]$ that have conditions matching the state s, to form the match set $[M]$.
3. performs covering: for every action $a_i \in A$ in the set of all possible actions, if a_i is not represented in $[M]$, a random classifier is generated that matches s and advocates a_i, and added to the population.
4. forms a system prediction array, $P(a_i)$ for every $a_i \in A$ that represents the system's best estimate of the payoff should the action a_i be performed in the current state s. Commonly, $P(a_i)$ is a fitness weighted average of the payoff predictions of all classifiers advocating a_i.
5. selects an action a to explore (probabilistically or randomly) and selects all the classifiers in $[M]$ that advocated a to form the action set $[A]$.
6. performs the action a, recording the reward from the environment, r, and uses r to update the predictions of all classifiers in $[A]$.
7. when appropriate runs a genetic algorithm (GA) to introduce new classifiers to the population. In XCS, two parent classifiers are selected from $[A]$ and two offspring are produced by applying crossover and mutation on their conditions, such that both offspring match the currently observed state.

Additionally, the explore mode may perform subsumption, to merge more specific classifiers into more general, accurate ones, and deletion, if the classifier population size grows larger than the specified limit. In contrast, in the exploit mode the agent does not attempt to learn and simply performs the action with the best predicted payoff.

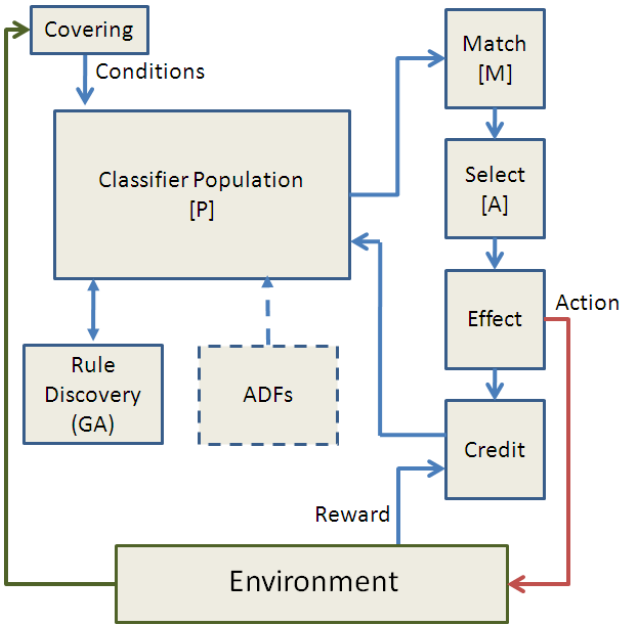

Figure 1: Schematic depicition of a learning classifier system, dashed outline block showing inclusion of ADFs.

The Markov property of the environment is assumed, meaning that the same action in the same state will result in the same reward. LCS have been shown to be robust to small amounts of noise and are often more robust than most machine learning techniques with increasing amounts of noise [3]. The generalisation property in LCS allows a single rule to cover more than one state provided that the action-reward mapping is similar. Traditionally, generalisation in LCS classifier conditions is achieved by the use of a special 'don't care' symbol (#) in the ternary representation, which matches any value of a specified attribute in the vector describing the state s. Other representations, including GP like S-Expressions and Reverse Polish Notation have also been used successfully [5, 10].

2.2 Tree-Based Genetic Programming

Genetic programming (GP) is an evolutionary approach to generate computer programs for solving a given task automatically. The task to be solved is represented by a primitive set of operations, known as the function set in evolutionary computation community, and a set of operands, known as the terminal set. The generated computer programs are commonly represented by a tree. The nodes of the tree are functions and leaves are the terminals.

To generate a computer program using GP, a set of input-output pairs is needed for training the technique along with sets of functions and terminals. GP attempts to construct a computer program that maps each of the input-output pairs correctly. For example,if the input-output pairs set is {(0,1), (1,3), (2,7), (3,13), (4,21), (5,31), ... } and {+,-,*,/} and {x,1} are the function set and terminal set respectively

then the optimal corresponding GP generated program is as shown in Figure 2.

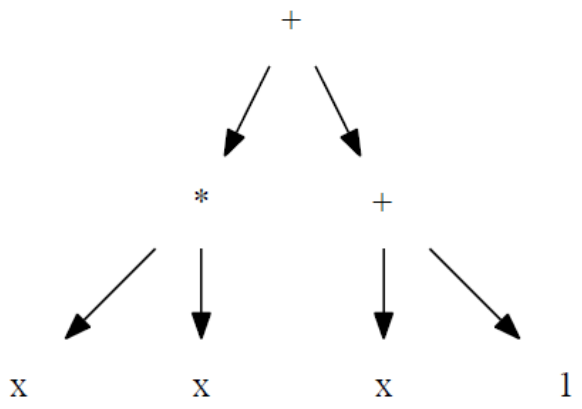

Figure 2: Tree GP Example.

Initially a population of random programs is created. Then each program from this population is evaluated to determine how many pairs it can match correctly from the complete set. The two best programs are selected using a selection method such as tournament selection or roulette wheel selection. Using crossover operator two offspring are produced from these two selected parents. Mutation may be applied to the offspring. Commonly, the two worst fit individuals from the population are replaced by these two created children. This process is repeated for a fixed number of times or until an ending criterion is met.

GP is a technique that can produce a computer program automatically to maximally map a input-output pairs set. But to generate this program it needs many CPU cycles and a lot of memory space [13].

2.3 Automatically Defined Functions

The computer program represented as a tree in tree based genetic programming often have repeated patterns in it that can be treated as separate modules. The concept of automatically defined functions (ADFs) was introduced by Koza in 1994 [7]. The main GP program has one or more function defining branches along with a result producing branch. These function defining branches evolve simultaneously with the result producing branch. The result producing branch can call any of the ADFs in its associated function defining branches, but an order is specified for calling a function within a function defining branch. Otherwise it is possible that recursive function calls can result in infinite loop.

The ADFs make the tree program more readable and concise. Thus the relatively simple and reduced size program is evolved more easily than a large program without ADFs. The crossover and mutation operators can disrupt useful patterns in large trees, but the inclusion of ADFs can protect the module from these disruptions. When ADFs are being used then a constrained crossover and mutation is applied. Result producing branches are crossed over with only result producing branches in other individuals in the population and similarly the function defining branches are crossed over only with function defining branches. The ADFs evolved in one task can be reused for any other similar task.

2.4 General Purpose Graphics Processing Units

Graphics processing units (GPUs) were originally designed for graphics processing application and the gaming industry. GPUs have orders of magnitude more computational power as compared to CPUs. The one reason for this speed up is that GPUs devote more transistors to the processing unit. The rapid increase in the performance of GPUs, coupled with recent improvements in its programmability, have made GPUs a compelling platform for computationally demanding tasks in a wide variety of application domains [12] including evolutionary computation research. The genetic programming and learning classifier systems are computationally intensive methodologies so they are prime candidates for using GPUs. There are a number of tool kits available for programming GPUs such as CUDA, MS Accelerator, Rapid Mind and Shader Programming. CUDA is an archticture introduced by NVIDIA to enable software developers to code general purpose applications that run on the massively parallel hardware on GPUs. CUDA-C is a C-like programming language, which is becoming a common platform for evolutionary computation research [4, 9].

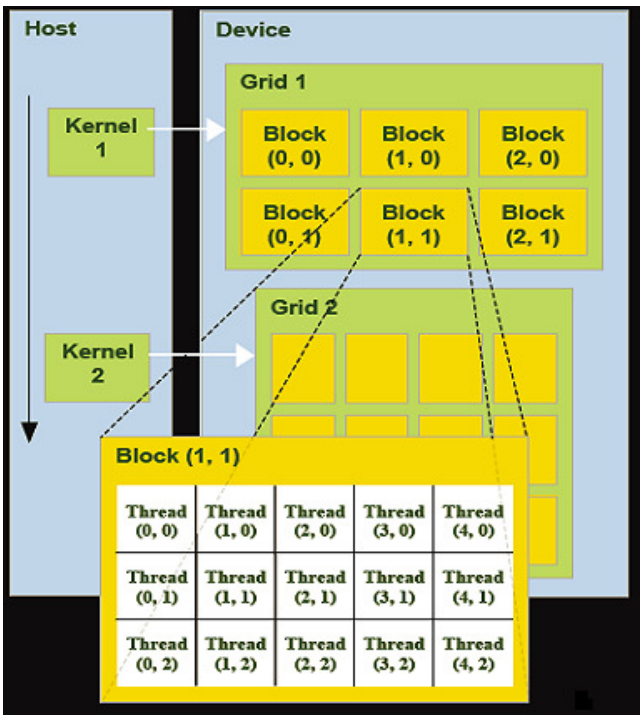

Figure 3: CUDA Programming Model[1].

CUDA is an heterogeneous serial - parallel (CPU + GPU) programming model, shown in Figure 3, that lets programmers focus on parallel algorithms instead of focusing on the mechanics of a parallel programming language. Serial code in an application executes in a host thread whereas parallel portions of the application are executed on the device (GPU) as kernels (A kernel is a function that runs on a GPU). One kernel is executed at a time and many threads execute each kernel. Each thread executes the same code on different data based on its threadID. CUDA programming uses a massive number of light-weight threads to exploit the parallelism provided by GPUs. Threads are grouped into

thread blocks. Each thread has a unique ID within a block. Thread blocks are grouped into a grid. Each block within a grid has unique block ID. Grids are executed on a GPU device.

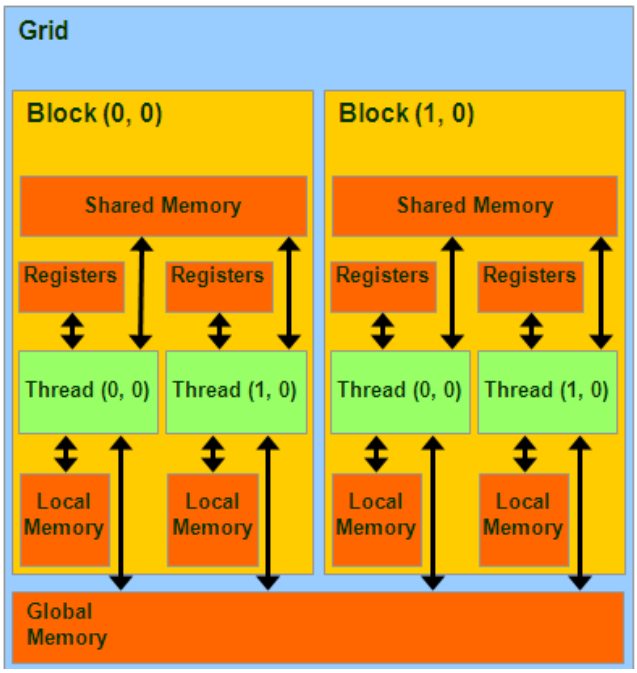

Figure 4: CUDA Memory Model[1].

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 4. Registers are the fastest form of memory on the GPU and are only accessible by individual threads and have the lifetime of a thread. Each thread also has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block through which they can communicate. Finally, all threads have access to the same global memory. Data to be processed must be copied to GPU's global memory.

3. METHODS

The introduction of ADFs in LCS is not a straightforward representational switch from ternary to S-expression representation as in previous work [5, 10]. Instead, ADFs determine if their rule matches the message instance (state). Although there is simply the same number of ADFs as condition features, e.g. 6 for the 6-bit MUX problem, there is a decoupling between an ADF and position within the condition, i.e. the order of ADFs is unimportant. Each ADF must return true (set as logical 1) for a rule to match and be considered by the select procedure in order to effect its associated action.

To develop this novel system the results of five approaches based on either GP or LCS with and without ADFs have been investigated. All of these are implemented on TESLA GPU using CUDA-C language. The approaches are: 1) GP without ADF, 2) GP with one ADF, 3) GP with two ADFs, 4) LCS without ADF, and 5) LCS with ADFs. The example problem used in experimentation is the multiplexer problem domain good to previous investigations in both the GP and LCS community. The first benchmark technique is a standard implementation of Langdon's GPU package [8] where function set is {AND, OR, NAND, NOR, NOT} and terminal set is {D0, D1, D2, ... , Dn-1} having n as 6, 11, and 20 for 6-, 11-, and 20-bits multiplexer examples respectively.

Langdon's GPU package is then modified by including one ADF branch (approach 2). Further, the CUDA implementation is improved to reduce the computational time as compared with the benchmark approach (1). Originally in Langdon's GPU package, each GP program from the population was being interpreted by a single thread. In this case, the number of threads is equal to the population size. Here a thread block is allocated to each GP program instead of a single thread. This reduced the arithmetic intensity because now the fitness cases of a GP program have been parallelized among all the threads of a thread block.

The ADF takes three arguments, which are ARG0, ARG1, and ARG2. The motivation behind the ADFs inclusion was to generate simple and more evolvable genetic programs. Section 4.2 shows that the inclusion of just one ADF did not reduce the genetic tree down to a reasonable size, but the approach took relatively less time to solve the multiplexer problems compared with the benchmark. Therefore, an additional ADF was introduced (approach 3). The main result producing branch can call both ADFs: ADF0 and ADF1. ADF1 can call ADF0, but ADF0 can not call ADF1 to prevent infinite loops.

The fourth approach is an implementation of learning classifier systems on a TESLA GPU using CUDA-C. It is the manual conversion of the Butz's "implementation of the XCS classifer system in Java - June 2000" into CUDA-C. GPU is used only for column-based matching of ternary conditions in parallel as in work by Lanzi and Loiacono [9].

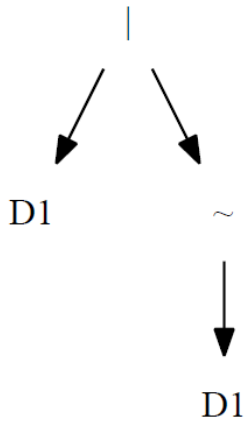

Figure 5: An ADF used as don't care symbol in the XCS's condition [Here '|' and '~' denote OR and NOT operators respecively].

The main contribution in this work is the introduction of ADFs instead of ternary conditions in Wilson's accuracy based LCS termed XCS[14](approach 5). A population of tree-based ADFs is created at the start of training. This population is not totally random; it has some manually designed ADFs that incorporate each operand alone as well as with each operator. The number of manually generated ADFs is 252, 902, and 3080 for 6-, 11-, and 20-bits multiplexer examples. The function set and terminal set are the

same as used in approach 1. As in the base LCS (approach 4), GPU is only used for condition matching. The ternary symbols in the condition are initially replaced by randomly selected ADFs from the ADF population in order to create the initial rule base. A condition is considered matched if all its ADFs return true on evaluation against the given condition from the environment. A special ADF, shown in Figure 5, is used as the don't care symbol that always returns true.

It is anticipated that if useful building blocks of information are present in the ADF population, then they will be identifiable by examining fit classifier's condition.

4. EXPERIMENTAL RESULTS

All the experiments have been repeated 10 times with a known different seed in each run. The symbols &, |, d, r, and ~ denotes AND, OR, NAND, NOR, and NOT operators respectively. The training set size for all of the three GP methods is 64, 2048, and 2048 for 6-, 11-, and 20-bits multiplexers respectively. It is to be noted that for 20-bits multiplexers a subset of the whole training set ($2^{20} = 1048756$) is being used as suggested by Langdon in [8]. Although 37-bit problems learn successfully, the time taken was too long for statistically valid repeated tests.

The system uses the following parameter values, as defined in XCS: fitness fall-off rate $\alpha = 0.1$; prediction error threshold $\epsilon_0 = 10$; fitness exponent $\nu = 5$; learning rate $\beta = 0.2$; threshold for GA application in the action set $\theta_{GA} = 25$; experience threshold for classifier deletion $\theta_{del} = 20$; fraction of mean fitness for deletion $\delta = 0.1$; classifier experience threshold for subsumption $\theta_{sub} = 20$; crossover probability $\chi = 0.8$; mutation probability $\mu = 0.04$; and the selection method is tournament selection with tournament size ratio 0.4. The number of micro classifiers is 2000 and the number of ADFs used is 2000, 5000, and 8000 for 6-, 11-, and 20-bits multiplexers respectively.

4.1 Multiplexer Problem Domain

A multiplexer is an electronic circuit that accepts n inputs and gives one output. The n inputs are divided into two groups: k address bits and the remaining n-k data bits. Actually n is of the form $k + 2^k$. Hence the data bits are n-$k = 2^k$. For example, in the case of 6-bits multiplexer there are 2 address bits and 4 data bits. If we denote address bits by A0 and A1 and data bits by D0, D1, D2, and D3 then 6-bits multiplexer works as described in Table 1. Where "#" is the don't care symbol. It can be either 0 or 1 but it has no effect on the output signal. The value of address bits is used to select the data bit to be given as output.

Table 1: 6-bits multiplexer.

Input						Output
A0	A1	D3	D2	D1	D0	
0	0	#	#	#	0	0
0	0	#	#	#	1	1
0	1	#	#	0	#	0
0	1	#	#	1	#	1
1	0	#	0	#	#	0
1	0	#	1	#	#	1
1	1	0	#	#	#	0
1	1	1	#	#	#	1

In the experimentation both address and data bits are denoted by D (instead of denoting address bits by A and data bits by D), just to simplify the programming. Lower order bits are for address and the remaining bits are for data. For example in case of 6-bits multiplexer D0 and D1 are address bits and the remaining D2 to D5 are data bits. Multiplexer problems are highly non-linear and therefore relatively difficult to learn. Multiplexer problems have been very common in GP and LCS research community to be used for testing an experiment because they contain generalizations and are suitable for examining the scalability of the algorithm.

4.2 Methods in Comparison

When comparing the different approaches, it was debatable whether to use 'CPU cycles' or the time taken as a measure of computational speed. Although the former is more robust to outside interference the latter was more illustrative of performance especially as the computations were run on a dedicated machine. Furthermore, general trends were of interest rather than fractional amounts of comparative performance improvement. Due to the lack of compaction in GP without ADFs the time taken was longer than with ADFs, although the number of generations required to reach optimum performance was actually less.

The benchmark GP program solved the multiplexer problem with increasing time as the problem domain scaled. The introduction of ADFs helped contract the overall tree as well as speeding up the performance as the complexity of the multiplexer problem increased, see Table 2.

Table 2: Comparison of GP methods with and without ADFs [Population for GP without ADF is 262144 and Population for GP with ADFs is 48640].

Multiplexer	Method	Generations Needed	Tree Length	Tree Depth	Time (seconds)
6-bits	GP without ADF	019±001	110±32	12±2	0010.82±0000.72
	GP with 1 ADF	032±004	085±35	11±1	0008.86±0001.13
	GP with 2 ADFs	030±004	078±20	10±1	0008.82±0001.38
11-bits	GP without ADF	084±011	410±54	16±0	0299.78±0052.88
	GP with 1 ADF	154±30	323±51	12±0	0079.01±0024.34
	GP with 2 ADFs	129±14	265±50	12±0	0064.17±0006.64
20-bits	GP without ADF	372±72	502±07	16±0	5638.29±2127.81
	GP with 1 ADF	759±465	486±15	12±0	0614.13±0465.21
	GP with 2 ADFs	767±376	501±13	12±0	0738.73±0435.56

Although standard XCS was generally faster than standard GP, there was a large increase in training time with the addition of ADFs in LCS, see Figure 6. There is a difference between 'generation' in GP and 'iteration' in LCS. The former is the presentation of the complete training set (or in the case here for 20-bits MUX a substantial subset) of the training data. The latter is a single instance of a training pair.

The results for time comparison as shown in Figure 7 suggest that the LCS technique exhibits better scaling as the complexity in the domain increases. XCS with ADFs is the slowest approach on the six-bit problem, but at the 20 bit problem is faster than all GP approaches, only being a bettered by the standard XCS approach, which is the fastest in all problem domains. This suggests LCS as the base platform for developing scalable learning.

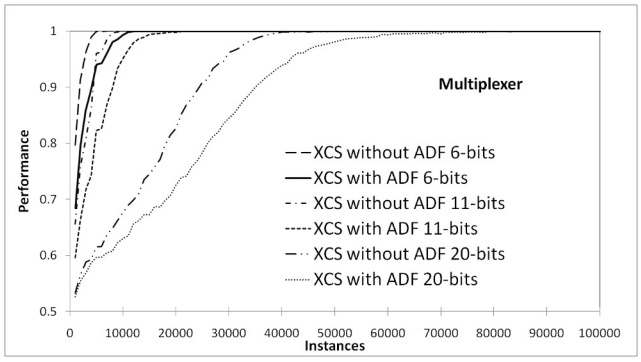

Figure 6: Comparison of XCS methods for 6-, 11-, and 20-bits multiplexers.

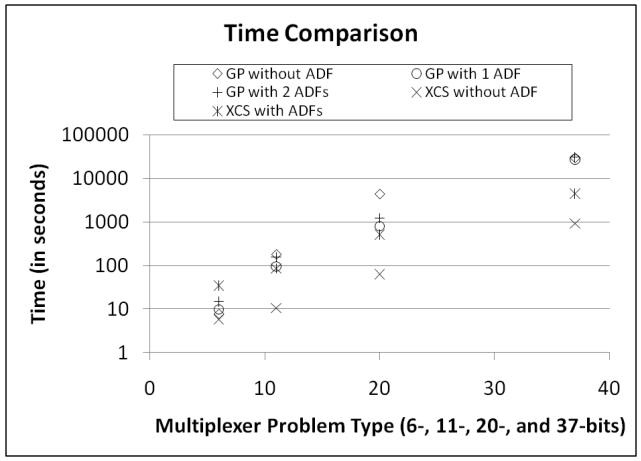

Figure 7: Scalability of GP and XCS Methods [Note: In these tests the NOT operator was not included in the function set].

4.3 Building Blocks of Information within ADFs

The final rule-base of a typical LCS run on the 6-bit MUX problem was analysed to extract the most used ADFs, see Table 3 for the top 10 ADFs out of 208 retained from the initial 2000 created. If the ADFs behaved as a standard representation, then the ADFs of length 1 and 'don't care's would be expected in rules, e.g. ~D0, D1, ~D4 : 0. However, such rules were not present, with rules containing longer ADFs preferred. These longer ADFs often contained the address bits (either D0, D1 or combination). The most common ADF (rank 1) was the 'don't care' equivalent. The rank 2 ADF is very interesting, see Figure 8 and Table 4 as it acts as a specialised 'don't care' rule. It is only not true in two situations; trivially, D0 = D1 = 0 so D3, D5 are unimportant, importantly, when D3 is 0 and D3 is addressed. It effectively acts as a mask, indicating that a rule containing it should not cover this particular situation. This is different to many standard representations that can only indicate which situations they do cover.

LCS claim that transparent learning and cooperation among rules as a strength of the technique. Considering the rule shown in Figure 9, cooperation is needed amongst both ADFs and other rules, e.g. this rule covers the case when

Table 3: Top 10 highly ranked ADFs.

Rank	ADF Function	Frequency	Length
1	D1D1~|	2121	4
2	D3D0rD1D5|d	770	7
3	D2D1r~	516	4
4	D5D1&D0~|	490	6
5	D1~D5~|	469	5
6	D1D2rD0|	455	5
7	D1D3&~	436	4
8	D4D1rD1|	435	5
9	D4	317	1
10	D0~D5D2r|	281	6

D3 = 0 and is addressed. Although still human readable, the ADF based rules currently require more interpretation than standard, say ternary encoded, rules.

It is worth considering why the simple ADFs available to the LCS were not retained, see Table 5. Address bits D0 and D1 were not used often in isolation, probably due to the need to combine with other data-bit features to be meaningful. Considering this insight, D5 was relatively lowly ranked, but this was likely due to the 'true' state being favoured when matching, i.e. D0 = D1 = 1 = true - addresses D5. The negation of D0, D1 and D5 probably did not get retained for similar reasons. Curiously, the system effectively created D0 twice, see rank 19, 34, highlighting the problem with GP like encoding in terms of multiple genotypes to a single phenotype hindering both interpretation and functionality.

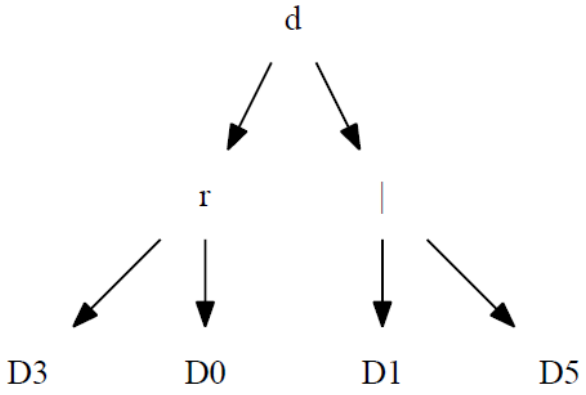

Figure 8: An ADF mask for not considering states when D3 is 0 and D3 is addressed.

Finally, as the address bits function together, would the system autonomously isolate them? Table 6 shows all be retained ADFs using just the address bits, which are few and ranked lowly. Address bits in isolation are meaningless, so they need to be combined with the data-bits as in the highly ranked ADFs.

5. DISCUSSION

This work has shown the ability of ADFs to both speed up and simplify the solutions of the GP technique, which confirms past results. It also shows that LCS are capable of functioning with ADFs, as can be seen from Figure 6, and

Table 4: An ADF mask for not considering states when D3 is 0 and D3 is addressed.

D0	D1	D3	D5	D3D0r	D1D5|	D3D0rD1D5|d
0	0	0	0	1	0	1
0	0	0	1	1	1	0
0	0	1	0	0	0	1
0	0	1	1	0	1	1
0	1	***0***	0	1	1	0
0	1	***0***	1	1	1	0
0	1	***1***	0	0	1	1
0	1	***1***	1	0	1	1
1	0	0	0	0	0	1
1	0	0	1	0	1	1
1	0	1	0	0	0	1
1	0	1	1	0	1	1
1	1	0	***0***	0	1	1
1	1	0	***1***	0	1	1
1	1	1	***0***	0	1	1
1	1	1	***1***	0	1	1

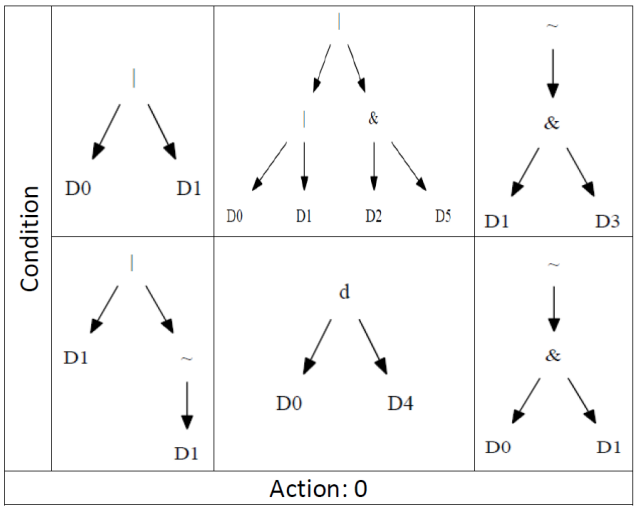

Figure 9: A classifier rule complementing a mask ADF. Consider that D0=0 and D1=1 to interpret this rule. ['&', 'd', '|' and '~' denote AND, NAND, OR and NOT operators respecively].

that in both techniques the information contained within the ADFs is useful to the learning algorithm. However, the ADFs in LCS did not compact the number of rules. Furthermore, the advantages of subsumption deletion were lost due to genotypic differences resulting in subsumption not occurring despite phenotypically similar behaviour.

Evolving one or two ADFs in LCS is unlikely to be effective compared with GP. GP tree represents a complete solution, whereas a complete solution in XCS is the population of rules. Thus more than one ADF is needed in order to fit each niche, hence why a population of ADFs was used in the LCS approaches.

It is noted that XCS is a Michigan LCS (population of individual rules), where a Pittsburgh style LCS (where the members of a population are a complete set of rules) may be more appropriate for the use of ADFs. As the purpose

Table 5: Low length ADFs.

Rank	ADF Function	Frequency
9	D4	317
27	D3	102
28	D2	100
32	D1	92
34	D0	87
39	D5	78
80	D2~	17
158	D3~	2
192	D4~	1
19	D0~~	129
87	D2~~	12
145	D4~~	3
174	D1~~	1
182	D3~~	1

Table 6: ADFs using address bits.

Rank	ADF Function	Frequency
13	D0D1&~	247
15	D0	216
30	D1D0~r	96
32	D1	94
59	D0D1|	28
89	D0~D1~r	11
149	D0D1d~	2
165	D0~D1|	1
175	D1D0&D0d	1

of this work is to discover building blocks useful across the problem domain, this was not investigated.

Time comparisons between GP and LCS for the multiplexer problem, see Figure 7, were not on a completely like for like basis. The concept of an iteration and a generation within the techniques differs. A population in GP consists of complete solutions, whereas in Michigan LCS the population is a single solution - neither population size had been optimised. The stopping criteria for LCS was arbitrary, rather than testing the complete problem set to ensure convergence. Further, the function set used can greatly decrease the time taken in GP if it produces more compact trees, e.g. the inclusion of the NOT operator would significantly reduce training times. Thus, the results only suggest LCS scales better in the multiplexer domain.

Utilising ADFs for the matching component of the LCS removes the implicit linking between the position of a condition in a rule and the corresponding feature in the problem pair. Although this could lead to compaction of a rule, it also places additional pressure on subsumption deletion as the reordering of the same conditions needs to be taken into account. The importance of the address bits and explicit links to the data bits were observed in the discovered rules.

Static, i.e. not evolved during training, ADFs in the LCS approaches were a manageable initial setup, which facilitated inspection of building blocks throughout learning in order to determine individual ADF adoption, i.e. perseverance of an ADF and the utility of an ADF as measured by the fitness of the rules in which it occurs. The next step is

to evolve the ADFs based on their utility or abstract them based on the conditions within a standard LCS.

6. CONCLUSIONS

The main objective of this work, which was to introduce ADFs in LCS, was achieved with interesting results. ADFs that linked the address to data bits were preferred over simpler representations, capturing important information. ADFs in LCS can autonomously determine which situations their associated rules cover and uniquely, do not cover.

This initial investigation of ADFs in LCS shows that the multiple genotypes to a phenotype issue in feature rich encodings disables the subsumption deletion function to the detriment of stable performance close to the optimum performance levels. The additional methods and increased search space leads to much longer training times. This is compensated by the ADFs containing useful knowledge, such as the importance of the address bits in the multiplexer problem, in a compact format.

An additional objective of this work was to compare the performance scaling using a common GPU platform (CUDA) for both GP and LCS. The introduction of ADFs in GP, on the GPU platform, improves the speed of performance for GP, but is detrimental to the speed of XCS. The results suggest that the LCS technique exhibits better scaling as the complexity in the domain increases. This suggests LCS as the base platform for developing scalable learning.

7. FUTURE WORK

Currently, GP is limited in the number of ADFs available to it and relies upon evolution to determine better functionality, whilst LCS has more potentially good ADFs available, but has no evolution. A parallel evolving population of ADFs is needed for both techniques.

The next stage is to introduce either algorithmic or numerical simplification [6] into ADFs for LCS, in order to introduce subsumption deletion. This may result in switching to a supervised learning LCS paradigm as simplification often requires a complete training set of data instead of a single instance.

Rather than evolving ADFs from a random seed, the use of evolution defined functions (EDFs) [2] will be tested where the current solutions are interrogated to determine common building blocks of information that can be extracted and introduced into the EDF population.

Ultimately, the identified fit DFs (either ADF or EDF) from a simple problem in a domain (e.g. 6-bit multiplexer) will be used to seed the DFs in a more complex problem in the same problem domain (e.g. 11-bit multiplexer) and so forth. By utilising this 'stepping-stone' approach it is hoped that eventually a problem will be solved in the domain (e.g. 1034-bit multiplexer), which had not previously been solved using the base techniques.

It is anticipated that multiple populations of DFs from different problem domains will need to be leveraged in order to assist in general problem solving.

8. REFERENCES

[1] *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 2010.

[2] M. Ahluwalia and L. Bull. Coevolving Functions in Genetic Programming. *Journal of Systems Architecture*, pages 573–585, 2001.

[3] M. V. Butz. *Rule-based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*. Springer Verlag, Berlin Heidelberg, 2006.

[4] M. Franco, N. Kransnogor, and J. Bacardit. Speeding Up the Evaluation of Evolutionary Learning Systems using GPGPUs. In *GECCO '10: Proceedings of the 12th annual conference companion on Genetic and evolutionary computation conference*, pages 1039–1046. ACM, 2010.

[5] I. Charalambos and W. N. Browne. Investigating Scaling of an Abstracted LCS Utilising Ternary and S-Expression Alphabets. In *IWLCS 2007, 10th International Workshop on Learning Classifier Systems, London, UK, July 7-11*. ACM, 2007.

[6] D. Kinzett, M. Johnston, and M. Zhang. Numerical Simplification for Bloat Control and Analysis of Building Blocks in Genetic Programming. *Evolutionary Intelligence*, 2(4):151–168, 2009.

[7] J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[8] W. B. Langdon. A Many Threaded CUDA Interpreter for Genetic Programming. In *EuroGP-2010: LNCS*, pages 146–158. Springer, 2010.

[9] P. Lanzi and D. Loiacono. Speeding Up Matching in Learning Classifier Systems Using CUDA. pages 1–20. Springer-Verlag, 2010.

[10] P. L. Lanzi and A. Perrucci. Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 345–352, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[11] A. Orriols-Puig and E. Bernadó-Mansilla. A Further Look at UCS Classifier System. In *Proceedings of the 9th International Workshop on Learning Classifier Systems - IWLCS2006*. Springer - to appear, 2006.

[12] J. D. Owens and D. Luebke. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics forum*, 26(1):80–113, 2007.

[13] D. Robilliard and V. Marion. Genetic Programming on Graphics Processing Units. In *Genetic Programming and Evolable Machines*. Springer, 2009.

[14] S. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.