VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Genetic Programming for Evolving Programs with Recursive Structures

Tessa Phillips

Supervisors: Mengjie Zhang, Bing Xue

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

In human written computer programs, loops and recursion are very important structures. Many real-world applications require recursion and loops. Loops and recursion can also be applied to genetic programming (GP). There has been a lot of work on loops in GP but not much on recursion. We did some initial work in summer 2015 on GP with recursion, which resulted in three recursive methods, but was specifically designed for certain problems. This project aims to extend this work to develop two new GP methods that are more general. The two new methods are tested on symbolic regression problems, binary classification problems and artificial ant problems. They are compared to methods using loops, traditional GP, and the methods developed in our summer project 2015. The results show that the new methods have improved the accuracy and increased the range of symbolic regression problems the methods can perfectly solve. The new methods can also solve classification and artificial ant problems, and have better performance than loops on many of these tasks. This project is the first work using recursion for classification problems, and is the first design of a generic recursive method for GP.

Acknowledgments

I would like to thank my supervisors Mengjie Zhang and Bing Xue for the countless hours poured into helping me with this project. Their contributions and extensive knowledge on the topic has been vital in improving the quality in writing and research in this project.

Contents

Figures

Chapter 1

Introduction

1.1 Motivation

Loops and recursion are very important in human written computer programs. Iterative structures allow human written computer programs to solve repetitive tasks. Without iterative structures, human written programs would need to repeat the same code many times resulting in a program that is unnecessarily long.

Genetic Programming (GP) [1] has been used to solve a wide range of problems. A limitation with traditional GP however is that, solving large repetitive tasks is often difficult. Many real-world applications such as image classification or navigation tasks require iterations. Including iterative structures such as loops and recursion in GP could have the potential to work well for some of these tasks and solve this issue [2].

Recursion and Loops are very similar, but some problems are better suited to either recursion or loops [3]. For example traversing a tree or linked list data structure is simpler when using recursion. Iterating through all the elements of an array could be more understandable using a loop structure. It is desirable that programs evolved by GP are simple and easy to understand.

There has been a lot of work in using loops in GP to solve a wide range of tasks including classification tasks and navigation type problems which are real-world tasks. Loops have had very good performance on these difficult tasks [4][5][6][7].

There has not been much work done on recursion in GP, and it certainly has not been applied to any real-world tasks. The work that has been done has used a simple problem like a symbolic regression task to illustrate the method performance. Recursion has the potential to perform better than loops for many problems, or provide a more understandable solution. It is important to include this structure in GP to allow GP to better perform on a wide range of difficult problems.

Early work on recursion was done by Koza [8]. It involved using automatically defined recursions(ADR), which were a special function to provide recursion. Yu and Clark [9] created recursion in GP with lambda abstractions. Alexander and Zacher [10] created recursion by evolving recursive call trees. Moraglio [11] used non recursive fitness functions to evolve recursive functions.

We have done initial work on GP with recursion [12], developing three new recursive methods for GP (R1, R2, and R3), which had promising results on simple problems. R1 created recursion by including a range of pre-defined recursive functions in its function set. R2 was a template function for recursion, and R3 used an Automatically Defined Function (ADF) that included itself in its function set, which resulted in a recursive ADF tree. The results of these methods showed that in all of the symbolic regression tasks and artificial ant tasks that were tested, two of our methods performed better than loops.

In previous work, the recursive methods provided a recursive solution to solving simple well known problems. The main limitation however is that these methods were problem specific. In addition, the previous work does not provide a solution to solving difficult or unknown tasks such as classification or image recognition with GP using recursion.

1.2 Aim and Objectives

The goal of this project is to develop a new general GP approach to solving problems with recursive nature. We expect the new extended approach to achieve better performance and obtain understandable solution programs for the problems to be solved. This works towards increasing the performance and understandability of GP on very difficult problems.

1.2.1 Objectives

A new GP method for solving common recursive problems (GPCR)

The purpose of this method is to be an approach to simple problems that might be similar to known functions. This method aims to automatically select and combine known recursive functions to solve an unknown recursive function.

The new method aims to extend the R1 method [12]. The R1 method was able to solve some simple symbolic regression problems as well as artificial ant problems, using different structures. There were not many functions given in the function set, and many common recursive functions could not be solved by this method. This will be extended by increasing the size of the function set to include other known recursive functions. This will increase the range of problems that can be solved by automatically selecting and/or combining the given recursive functions from the function set.

A new GP method for solving generic problems with recursive structures (GPGR)

The purpose of this method is to have a general template recursive function which provides the structure for recursion but is not specific to any particular recursive problem. This will allow the method to apply across a wide range of different problems and allow it to solve some difficult problems such as classification and image recognition using a generic structure.

The design for this method is an extension to the R2 method [12]. The R2 method was a template recursive function and could solve simple symbolic regression and Artificial Ant problems significantly better than loops in most cases. The limitation with this method was that it could not solve all types of recursive symbolic regression problems and also was not easily applicable to more difficult tasks due to it being very restricted in structure. The new approach aims to solve these issues by providing a generic recursive structure that is less restricted and will apply to more types of problems.

The performance of these new methods will be measured by the accuracy of the evolved programs, how understandable the solutions are, and if the new methods are able to perform well on a wider range of tasks than R1 and R2. The results for symbolic regression problems will be compared with while loop 1 [2], while loop 2 [2], GP with no loops, and the R2 method. For classification tasks, the results will be compared to GP with no loops and a range of specific loops used for classification tasks [4][5]. The results for Artificial Ant problems will be compared against GP with no loops, while loop 1, while loop 2, R1, R2, and R3. The benchmark methods and problems are explained in detail in chapter 5

1.3 Challenges

Evolving recursive programs is not simple, and there are a number of challenges that can affect performance and training time.

A major challenge that has become apparent in this work is that a small change in the recursive body can cause a very large change in the overall solution. In the symbolic regression problems, the models often grow very quickly such as in the Factorial Problem. This makes it hard to evolve a program to fit the data, as there are only a few functions that fit this data. This means that a function could look very similar to the perfect known solution, but would possibly have very low fitness because the result from the recursive function would be far from the desired result. This means the recursive problems often have a non-smooth fitness landscape which makes it difficult for GP to find a good solution using the crossover operator. In this project, this problem is addressed in the binomial problem (which is particularly hard to solve) by increasing the mutation rate. This solution was able to slightly improve the results, however does not completely solve this challenge.

Another challenge evolving recursive programs is that they can sometimes evolve infinite solutions, which would halt the evolutionary process completely. This problem also occurs with loops, and the common solution is to limit the number of iterations that can happen within a loop. This means that if a loop is going to be infinite, it will stop early and the evolutionary process is able to continue. We handled this similarly in this project which has worked well to keep the training time low, and the evolved programs finite.

It is also challenging to design a generic structure for different types of problems. This is because different problems require different data types. Artificial Ant problems are very different to symbolic regression problems, as the final result is a path rather than a numeric function.

1.4 Contributions

1.4.1 GPCR

This new method has successfully increased the range of problems that could be solved by R1. The method can now solve the Fibonacci problem which is a very common recursive sequence in nature, as well as other different recursive problems. GPCR also performs better than the benchmark loops and standard GP, as well as the other recursive methods.

The downside is that the performance on very simple problems (Factorial and Sum) has decreased. This is due to a large number of other recursive functions being included in the function set. These extra functions can cause the GP process to get stuck in a local minimum.

This method avoids the infinite recursion challenge completely as the recursion is given by combining or selecting a pre-defined recursive function which cannot be infinite. The challenge of having a non-smooth fitness landscape was addressed by increasing the mutation rate for particularly the most difficult problem, which was the Binomial problem.

1.4.2 GPGR

This more general method was applied to four different types of problems (symbolic regression, Artificial Ant, Image Classification, and Non-Image Classification) and achieved results comparable or sometimes much better than loops for all types of problems.

This method is a much more general approach than the R2 method developed in summer [12] but is not so general that it has poor performance which was the case for the R3 method. The way this method was made more general is by adjusting the structure of the recursion

so it was not as restricted. The new approach can evolve recursive programs that are very complex in nature, or programs that have multiple recursions.

Similarly to GPCR, GPGR has had a decrease in performance on simple problems like Factorial and Sum. This is again due to the added generality of the method which can make it hard to evolve the solution. The R2 method would require a program tree of depth 1 to solve the Factorial or Sum problems, where the new method would require a depth of 2. This added complexity is needed to make the method more general to different types of problems. The results on these problems are still better than that of loops, and the method can find many perfect solutions.

1.5 Organisation

Chapter 2 describes the background and related work, this describes GP in detail as well as previous work for GP with loops and GP with recursion including the R1, R2, and R3 methods.

The design of GPCR is detailed in chapter 3 and GPGR in chapter 4. These sections describe the motivations, design, and implementation for these two new recursive methods.

Chapter 5 describes the experiment design, this gives details about the benchmark problems and benchmark methods used as well as the design of the parameter settings.

The results of this are given in chapter 6 which provides tabular results as well as analysis.

Chapter 7 gives the conclusions and future work.

Chapter 2

Background

2.1 Human Written Programs vs Automatic programming

One definition of Automatic programming is that:

> "An automatic programming system allows a computationally naive user to describe problems using the natural terms and concepts of a domain with informality, imprecision and omission of details. An automatic programming system produces programs that run on real data to effect useful computations and that are reliable and efficient enough for routine use. Automatic programming allows a program to be generated to solve a specified problem without human intervention."[13]

This definition focuses around automatic programming to be useful for solving real-world problems, which requires programs to be efficient and also accurate. The benefit of using automatic programming would be to speed up the software development process [14]. There are a number of limitations in automatic programming which are mainly around translating high-level requirements into lower level programming requirements [15]. When compared to human written programming, automatic programming can generate programs very quickly and they can often be more efficient than when a written by a human.

2.2 Recursions in programs

Recursion is a common technique used in human written programs which allows the result of a program to depend on previous results of that program. This allows for simpler code [3], as it means code does not have to be repeated. Recursive functions are functions that depend on some previous value of the function. This is the only constraint to making a recursive function. An issue in human written programs with recursion is that if the recursion goes on for too long without a stopping condition, there can be a stack overflow error [16]. This error means that there are too many programs stored in the stack waiting to be completed. This error can be avoided by ensuring that the recursive program ends within a reasonable number of recursions.

2.3 Symbolic Regression Problems

Symbolic regression problems have a numeric input and a specified numeric output, and the task is to find an equation, including a model structure and a set of coefficients, that correctly maps the input to the output. This type of problems will typically be solved by using the numeric data available as input to the learning algorithm and using the learning algorithm

to minimize the mean squared error between the true output and the learned output. This is different to traditional statistical regression, where a model is chosen based on the data distribution, and the task is to find a set of parameter values that can fit the model [17].

2.4 Classification Problems

Classification tasks are to determine which category or class an example belongs to [18]. Often this classification is between two classes, called Binary Classification. Classification is a supervised learning problem, and uses a labelled training set to learn the classifier and then applies this classifier to an unseen test set to determine how well the classifier has generalized. A type of classification problem, image classification is where the examples to be classified are images.

2.5 Artificial Ant Problems

Artificial Ant problems are provided with a trail which is the location of food. The ant starts at the beginning of the trail and the aim is for the ant to eat as much food as possible. Machine learning algorithms are used to develop strategies for the ants to follow which try to maximize the amount of food the ant eats [19][20].

2.6 Genetic Programming

GP is a common Evolutionary Computation (EC) and Machine learning approach. GP is used to automatically evolve solutions, in the form of program trees, to solve a particular problem [1].

EC is a type of computing where the algorithms are inspired by biological evolution. GP and other EC algorithms follow the evolutionary process. This begins with a population of individuals usually randomly generated, these individuals are then selected based on their survival fitness, and then are used in producing the next generation. The idea of this is based on the "survival of the fittest" idea in biology. EC algorithms including GP can be very useful in solving a very wide range of problems. This is due to the Evolutionary Process being very general, to apply this approach to a particular problem, all that needs to be modified is how the fitness of an individual will be measured. Having general methods is good, particularly when solving a problem that is not well understood [21][22].

2.6.1 GP Process

GP works by first initializing a population of random individuals. These individuals are evaluated based on the fitness function, then individuals are selected to be used in the next generation based on their fitness. These selected individuals are put through Genetic Operators (Crossover, mutation, and reproduction) and then the modified individuals are put into the next generation. GP will continue to evolve the population until the maximum number of generations is reached, or a satisfactory solution has been found [1].

2.6.2 Representation

The structure of a typical GP individual is given in figure 2.1. The representation is a program tree [23], where the nodes are either functions (internal nodes) or terminals (leaf

nodes). The functions such as mathematical operators $(+, -, \times, /)$ are defined in the function set, where the terminals, corresponding to the inputs from the environment, are defined in the terminal set.

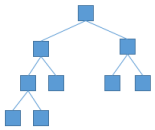


Figure 2.1: A typical GP individual.

2.6.3 Genetic Operators

The three main genetic operators used in GP are crossover, mutation and reproduction [24][25]. These operators are used on selected individuals from one generation to form the next generation of individuals.

Figure 2.2 shows how the crossover operator works. Crossover works by first selecting two individuals to cross over, and then replacing a subtree of one individual with a subtree of the other.

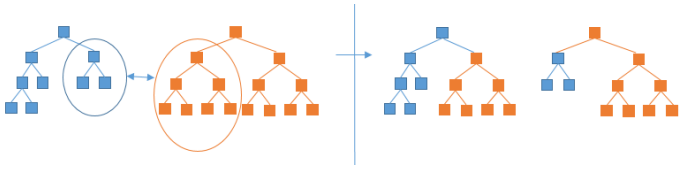


Figure 2.2: Illustration of the crossover operator. The two trees are being combined to form two different trees in the next generation.

Figure 2.3 shows how the mutation operator works. Mutation is useful for introducing and maintaining diversity in the population [26], as it randomly changes a subtree of an individual's tree with a random new subtree.

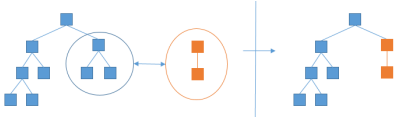


Figure 2.3: Illustration of the mutation operator. This shows how a subtree of the GP tree can be replaced by a different subtree to form a different tree in the next generation.

Reproduction is another common Genetic Operator, which works in GP by simply making an exact copy of the selected individual to put in the next generation. This operator is used to keep good individuals in the population.

2.6.4 Fitness Function

In GP the fitness function is the measure that is used to evaluate the programs [24]. For classification problems, this is often the number of examples classified correctly. For regression problems this can be the Mean Squared Error, or the number of fitness cases correct.

2.6.5 Termination Criteria

The evolutionary process is terminated when the number of generations is reached or the fittest individual is of high enough fitness. Some problems may only require a certain fitness

of solutions, which means that the evolutionary process can be stopped early if an individual is good enough. Often this criterion is that the individual is a perfect solution, which means a solution with 100% accuracy, or zero errors, for relatively easy or medium difficulty problems.

2.7 Strongly Typed Genetic Programming

Strongly Typed GP (STGP) [27] is used to ensure that programs remain syntactically correct. This is important when there are functions and terminals that need different types as inputs or return different types. For instance, if a function set contained an 'if statement' function, this would require one child of type Boolean and two children of other type. When these types of functions are included in the function set, using STGP is important because it enforces that the correct type of node is given for each child. Figure 2.4 shows an example of what this 'if statement' function would look like.

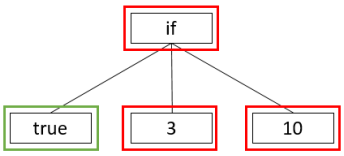


Figure 2.4: If statement example. Green box indicates Boolean type, red boxes indicate a numeric type. Here the function would return the value of the second child, 3, if the first child is true. If the first child was false, then the function would return the value of the second child, 10.

2.8 Automatically Defined Functions in GP

Automatically Defined Functions (ADF) [27][28] work by evolving their own tree, which means that the structure of the GP individual is changed. When an ADF is used, the individuals must have two program trees, one for the main program and one for the ADF. The two trees have separate function sets, so they can contain very different types of programs. ADFs allow for modularity and code reuse as they create a sub-program that can be used many times by the main GP program. To reuse code in standard GP, the same subtree must be evolved and used multiple times.

2.9 GP with Loops

The first work on GP with loops was done by Koza [1] where he developed a *do-until* loop. Chen and Zhang [2] showed that GP with *while* loop structures was better than traditional GP for solving the Factorial and a modified ant problem. Li [29] has used *for* loops for solving ant problems and binary classification problems on image datasets. Abdulhamid [4] [5] used GP with loops for image and non-image binary classification problems with good results.

Li and Ciesielski [30], [31] investigated restricted and unrestricted loops in GP, where restricted loops are limited such that there cannot be loops within the loop body. The results were that for simple problems the restricted loops had better fitness but as problems got more difficult, the unrestricted loops performed better. The reason for this is that the restricted loops have a smaller search space, so if a simple solution is correct then it can find it easier. For more complicated problems the restricted loops may not be complex enough

to solve them perfectly so unrestricted loops tend to perform better. Larres [6] used unrestricted loops for image classification problems. The results fit the idea that unrestricted loops are better than restricted loops on very complicated problems.

One issue with using loops in GP is that a single mutation in the loop can completely change the result of that individual and may even make that program an infinite loop which is invalid. A similar problem occurs when using recursion. A program with an infinite loop or infinite recursion means that evaluating the program would never stop. A typical way of dealing with this problem is to set a maximum number of iterations. Once the program has reached this number of iterations it will terminate and be given a very poor fitness value as it is an invalid solution. This means that there can never be a loop that runs for more iterations than this limit. Restricting this avoids infinite loops, and speeds up the evolutionary process.

Loops have worked well in GP to solve a wide range of problems. We know from human written programming that some problems are easier to understand or simpler when using recursive structures than when using loops. Many problems solved by GP have a simpler program when using recursive structures instead of loops. This indicates that many of the problems solved by GP with loops may perform even better by using recursion as well as having programs that are easier for humans to understand.

2.10 Genetic Programming with Recursion

There have been a very limited number of papers in GP for evolving programs with recursion to date.

Yu and Clark [9] worked on implicit recursion where they used lambda abstractions and list inputs to create recursion. Using implicit recursion meant that there was no possibility for infinite loops to occur, as the lambda abstractions worked on finite sized lists. This method had good results on the N parity problem.

Koza [8] developed Automatically Defined Recursion(ADR), which essentially defined a recursive template function that could be used to create recursive functions. This is a very useful idea, however the approach in [8] was very complicated and the function itself was not easy to understand. The approach was only applied to very simple problems such as a parity problem which is a type of symbolic regression task. This work would be difficult to extend to further problem types like classification or artificial ant problems and even some types of symbolic regression problems like the Fibonacci problem.

Alexander and Zacher [10] have done some work with evolving recursive structures using call trees. The way this approach works is by first evolving the structure, the number of base cases required and where the recursive calls are in the program, and then after user confirmation of the structure, they evolve solutions within the structure. This makes the search space limited for the main evolution. This has shown to speed up the process, and had good results on a parity problem. The main limitation of this is that it requires human intervention to confirm the structure of the call tree. The problems this method can solve are therefore limited to problems where the correct recursive structure is known. This means that any real world problem where the ideal program is unknown would be difficult to solve using this method.

Moraglio [11] has used non-recursive fitness cases to evolve recursive structures with GP. The approach they used was to replace the recursive call at the start of the training process with a non-recursive scaffolding function. This scaffolding function is a function that can evolve some parts of the recursive call, such as the base cases. The recursive structure is slowly built up and eventually the scaffolding is replaced by the full recursive call. Recursive and loop structures are often difficult to evolve due to the fact that if one part of

the function was wrong, it would dramatically change the result. This approach works very well for known problems such as sorting a list of elements. There are properties that the solution must have, and these are used in the non-recursive scaffolding fitness function. This is limited to problems that have well known solutions and that a non-recursive scaffolding function can be designed for. It would be very difficult to apply this method to classification or ant problems because they usually do not have well defined perfect solutions to use for the scaffolding function.

We developed three recursive methods R1, R2 and R3 [12] which had varying generality (in our summer project).

2.10.1 R1

The first method R1 was very specific to the problems to be solved. This method had a function set that contained the model, or part of the model that was being evaluated. This method used a lot of domain knowledge as it requires at least part of the function be known and included in the function set. The way this method worked was that the GP tree would evolve to use or combine different recursive functions for each problem. This was particularly useful for the Binomial problem as this problem is a combination of factorials. The method could simply use the function that was already included in the function set to solve this more complex problem.

The problems which this method can reliably solve are problems that are simple combinations of the recursive functions from the function set. This greatly limits the range of problems that this method can solve as there are only a small number of functions included in the function set. However an example of a function that can be reliably solved is given in figure 2.5 which is the factorial problem. This is clearly a very easy problem for this method to solve. Another example is given in figure 2.6 which shows a solution to the Binomial problem. This problem had to combine multiple factorial functions which makes it more difficult in nature.

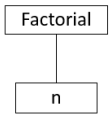


Figure 2.5: Example of the Factorial problem with the R1 method, which shows how simple common recursive functions are easily evolved with the R1 method.

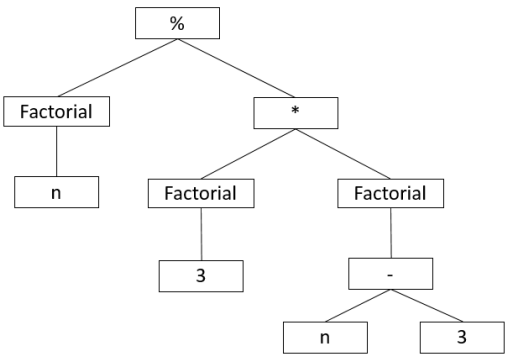


Figure 2.6: Example of the Binomial problem with the R1 method, which shows how quite complex recursive structures can be evolved using this simple R1 method.

The R1 method will be extended in this project, and chapter 3 describes the design of the

new method GPCR. R1 is designed to solve some specific problems. This means that it is not an issue that it cannot solve all recursive problems, but it would be better to have a wider range than the current design.

2.10.2 R2

The second method, R2 was about as general as loops as it defined a template function that was recursive, but the specific operators and terminals needed to be defined. This gives the recursive structure, but GP must evolve the details of the function itself.

The recursive template function is given by figure 2.7, which shows the structure of this function. This shows that the structure is limited to functions of the form $f(i) = i * f(i-1)$ where $*$ is a mathematical operator $(+, -, \times, /)$. This easily solves problems like Factorial, Sum, and combinations of these problems.

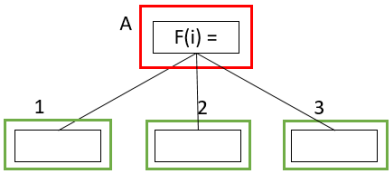


Figure 2.7: General recursive structure for the R2 method. A indicates the recursive function. Input 1 is the operator for recursion $(+, -, \times, /)$. Input 2 is the base case, and input 3 is the number to evaluate the function at.

An example of this function being used is given in figure 2.8, which shows how for this problem, it is very simple to find a solution with the R2 function.

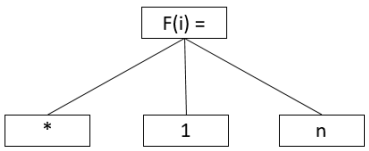


Figure 2.8: Example of the R2 method for the Factorial problem. The recursive operator is $\times$, the base case is 1 and the number evaluating at is n.

The results of this method were promising, and better than the benchmark loops for most problems. R2 was able to find many perfect solutions to the Factorial and Sum problems which had particularly good results. The method was also able to find perfect solutions to one of the ant problems, and performed better than loops for the other more difficult ant problems.

This method performed well for the problems tested in our summer work, however there are some limitations. One limitation is that it is only for symbolic regression and artificial ant problems. The method is not general to other types of problems such as classification problems.

This method is also limited in the types of symbolic regression problems that it can solve, due to restrictions in its structure. A problem that cannot be solved by this method is the Fibonacci problem. This is a common recursive sequence [32] which is defined by $f(i) = f(i-1)f(i-2)$. The R2 method cannot solve this problem, as it does not allow for $f(i-2)$ to be used.

Another limitation is that, the method is not consistently designed across the two types of problems. For Ant problems the method uses whether there is food ahead to determine if it should continue recursion, where for the symbolic regression problems it uses a count based recursion. It is desired to have a generic approach across all of the problems.

2.10.3 R3

The last method, R3 was very general, and recursion was implemented by the use of an ADF. This method was designed to be intentionally general, so it could solve a very wide range of recursive problems.

This method worked by having the ADF include itself in its function set, which allowed the ADF to be called recursively. This meant that the recursive function did not have any restrictions in its structure.

An example of the Binomial problem being solved with R3 is given in figure 2.9. This shows the generality of the R3 method, as there is no recursive structure fixed, it must evolve all of the structure and parameters. However this also shows that the R3 method is much more difficult to evolve correct programs, as it must develop a very large recursive ADF tree as well as the correct main program tree.

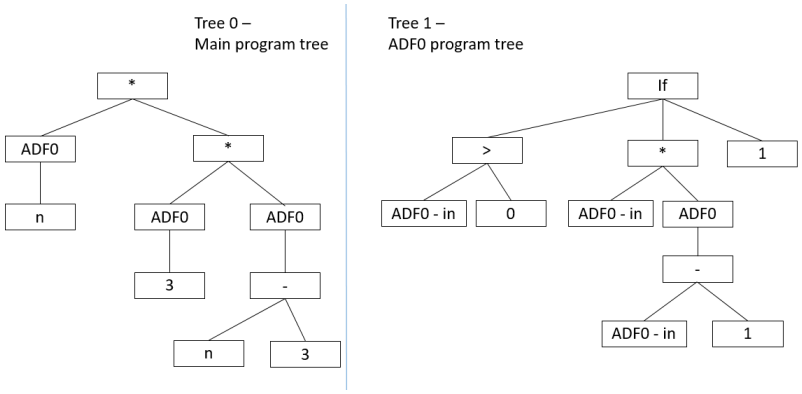


Figure 2.9: Example of the Binomial problem with the R3 method, this shows how complex evolved programs are when using this more general method. This also shows how the R3 method can have a type of code reuse.

This method did not perform as well as the other methods, and not as well as loops. The method showed some success as it was able to evolve perfect solutions on the Factorial and Sum problems, however it was difficult to evolve good solutions. The problems that this method was tested on did not require the level of generality that this method has, so it had a large search space which would mean a lower performance.

The main limitation with R3 is that it failed to perform as well as the other methods, particularly on very simple problems. The reason for this is that the R3 method is too general, using R3 it is harder for GP to evolve good solutions, than the other recursive methods, and loops. R3 does not provide the structure for the recursive program, so it must evolve this alongside the rest of the program.

2.11 Summary

Most work on GP with recursion before ours [12] are focused on using GP to solve a particular task or a specific type of problems. The methods R1, R2 and R3 aim to solve a much wider range of problems than what has previously been solved using GP with Recursion. The range of problems solved by these methods has previously been limited to just symbolic regression problems and Artificial Ant problems. This project aims to extend these methods by developing two new recursive methods, GPCR and GPGR. The new methods aim to be more general and to solve classification problems, artificial ant problems, and many different types of symbolic regression problems.

Chapter 3

A New GP Method for Evolving Common Recursive Programs (GPCR)

This section describes the design and implementation of the new GPCR method. GPCR aims to solve a wide range of common recursive symbolic regression problems.

3.1 Motivations

R1 is a problem specific method that aimed to solve a range of recursive problems by combining pre-existing recursive functions. The limitation of R1 was that it only solved very simple symbolic regression problems and artificial ant problems. The previous R1 method was not consistently designed for different types of problems. The R1 method on the ant problems was not general with the design for regression problems due to the types of problems being very different.

There were still some very common symbolic regression problems that R1 could not solve. This includes the Fibonacci problem that occurs very often in nature as well as being a common example of recursion in mathematics. This problem is of a different form to the ones included in the function set of R1 in [12].

3.2 Design

GPCR aims to solve recursive problems that are either well known problems or similar to well known recursive problems. It also aims to produce very understandable solutions as the recursion is given by the existing library of recursive-functions.

The method works by using the evolutionary process to automatically search for the correct recursive functions and operators which make up the recursive program. The recursive functions are given in a pre-existing library which is included in the function set.

This new method uses a pre-existing library of 20 recursive functions to include in the function set. The initial R1 approach used only 6 recursive functions. The types of function included are also different, which makes this method more general than the original.

The new GPCR approach is designed to be limited to just symbolic regression problems. Symbolic Regression problems have well defined perfect solutions, so it is easy to find common recursive functions that are solutions for this kind of task. Other types of problems including ant problems do not have well defined perfect solutions, so this method is intentionally excluding those types of problems.

3.2.1 Recursive functions

Table 3.1 shows the recursive functions included in the pre-existing library for GPCR. This includes a range of recursive functions that depend only on $f(n-1)$, and functions that depend on both $f(n-1)$ and $f(n-2)$. This means that this method should be able to solve more complex problems, such as the Fibonacci problem as this function and some similar functions are included in the library.

The function set for all problems with the GPCR method contains the 20 recursive functions given in table 3.1 and common mathematical operators (+, - , $\times$, %), where % is protected division. The terminal set contains (n, integers{0-15}).

Name	f(0)	f(1)	f(n)
F1	1	1	$f(n-1) \times n$
F2	0	1	$f(n-1) + n$
F3	0	1	$f(n-1) + f(n-2)$
F4	3	6	$f(n-1) \times 2$
F5	0	1	$f(n-1) + 2n - 1$
F6	2	4	$f(n-1) \times 2$
F7	2	2	$f(n-1)^n$
F8	1	2	$f(n-1) \times f(n-2)$
F9	5	7	$f(n-1) + 2$
F10	2	4	$f(n-1)^2$
F11	n	n^2	$f(n-1)^2$
F12	3	9	$f(n-1)^2$
F13	2	8	$f(n-1)^3$
F14	n	n^3	$f(n-1)^3$
F15	3	27	$f(n-1)^3$
F16	1	2	$f(n-2) \times n$
F17	1	2	$f(n-1) + f(n-2) + n$
F18	1	2	$f(n-1)^2 + f(n-2)$
F19	1	2	$f(n-1)^2 f(n-2)$
F20	1	1	$f(n-1) f(n-2) \times n$

Table 3.1: Recursive functions to use for GPCR on the symbolic regression problems.

3.3 Implementation

This method uses the standard GP process as given in algorithm 1 [33].

This method uses recursion when evaluating individuals that use the pre-defined recursive functions. The way the functions were implemented is by evaluating the recursive function at the specified number (given as the input to the function). This was done so that they could evaluate any integer number for each function. The functions were set up with a recursive method to evaluate and a base case to return when evaluated at 0, which is where the recursion stops. Because the recursive programs that are being evaluated are pre-defined, they are restricted so that infinite recursion never occurs. This means that all the programs evolved are valid and never infinite given a finite input, so there is no need to restrict the number of iterations.

3.4 Summary

The GPCR method is expected to have a good performance on common recursive functions. The method should have a very short training time and high accuracy on these functions

Algorithm 1 GP algorithm

```
1: procedure GP PROCESS
2: initialize population
3:     for All individuals in population do
4: Evaluate individual
5:     for i = 0; i < No. Generations; i++ do
6: Select individuals for breeding
7: Apply cross over, mutation, reproduction to selected individuals
8: Copy new individuals to next generation
9:         for All individuals in next generation of the population do
10: Evaluate individual
11:
            Find best individual for that generation
12:         if Best individuals fitness is 0 (perfect individual) then
13: Break
14:
    Set best individual to the best individual from last generation.
15: return the best individual's program tree and fitness.
16:
17:
```

when compared to benchmarks which are described in chapter 5. The new GPCR method is expected to solve more problems than R1, as it now contains a larger number of common recursive problems.

Chapter 4

A New GP Method for Solving Generic Problems with Recursive Structures (GPGR)

This chapter describes the motivation, design, and implementation of the GPGR method. The design will be contrasted with that of the R2 method in our recent work [12]. This method is designed to use a generic template recursive function which with the correct subtrees and parameters can solve a wide range of recursive problems. The level of domain knowledge included is comparable with the loops that have been developed previously for GP.

4.1 Motivations

The main motivation behind this method is that, the R2 method was too restricted in the problems it could solve. The new GPGR approach aims to be a very general recursive function for GP such that it can solve any recursive based problem. This meant that the R2 function needed major changes to its structure. The R2 method was too specific and would only be able to solve a small set of recursive tasks.

The aim of this method is to have a general template function allows for any specific recursive function to be developed depending on the problem specified. The function has no restrictions other than a maximum number of iterations which is used to stop the evolutionary process from being too slow.

This means that in contrast to GPCR, the GPGR method should not be limited to a predefined range of problems it can solve. In general recursive functions can be described by any mathematical function which includes a number of recursive calls. This is all that restricts a general recursive function.

The design of the R2 method as discussed in chapter 2 did not have the same generality. R2 was restricted such that it could deal with functions that were defined by $F(y) = y * f(y-1)$ where y is restricted to being the number the function is evaluated at and $*$ is any mathematical operator $(\times, +, -, /)$. This means that the depth in the recursive function was restricted to 2. The function could only use one recursive call which had to be $f(y-1)$, general recursive functions can call any $f(y-n)$ as the recursive call. It was shown in chapter 2 that the R2 method could not solve the Fibonacci problem, which is a common recursive sequence.

This new design works towards creating a general recursive template function, which aims to solve many difficult symbolic regression problems, classification problems, and Ar-

tificial Ant problems. By having a function that can be applied to all of these tasks, GP can perform well on a wide range of problems using very little domain knowledge. This is in contrast to R1 which would need to have the function or some part of the function included in the library to have good performance.

4.2 Design of Recursive Template Function

The recursive template function aims to be a generic function that can solve any recursive based problem. This is designed such that it is not restricted in structure. The template function can evolve a recursive program of any depth and use any number of recursions within the program.

Figure 4.2 shows how the GPGR method works to find a solution to the factorial problem, and a more general example of how the recursive function is used is shown in Figure 4.1. The recursive function **A** has three inputs shown in green boxes in figure 4.1. The first input is the body of recursion. The second input is the base case, which defines the output at i = 0. The third input is the number that the function is being evaluated to. If input 3 had value 4 for the factorial problem shown in Figure 4.2, then the function would be $Factorial(4) = 4 \times 3 \times 2 \times 1$.

Inside the body of recursion, there is always a recursive terminal **B**. This terminal returns the value of a previous recursion. In the function set, two of these special terminals are included, one as in the figure that returns $F(i-1)$, and one that returns $F(i-2)$.

Another special terminal can occur within the body of recursion, which is the terminal "i". This terminal returns the current count of the recursive function. This is used in Figure 4.2 as the factorial function requires $Factorial(i) = Factorial(i-1) \times i$.

Strongly typed GP is used to ensure that the programs evolved are syntactically correct. This is particularly important for the recursive function and recursive terminal.

It is very important to only ever have a recursive terminal ($f(i-n)$) within the body of a recursive function, or else it would not be obvious what it should evaluate to. If this is not the case, there will be a run time exception thrown, but strongly typed GP can take care of this constraint. It is also important to ensure that every recursive function has at least one recursive terminal, because if it did not then it would not be recursive, and would be a waste of processing time evaluating it at all of the different values. The special terminal i is also restricted to only occur within the recursive function, because this is the only place where this count value would be well defined.

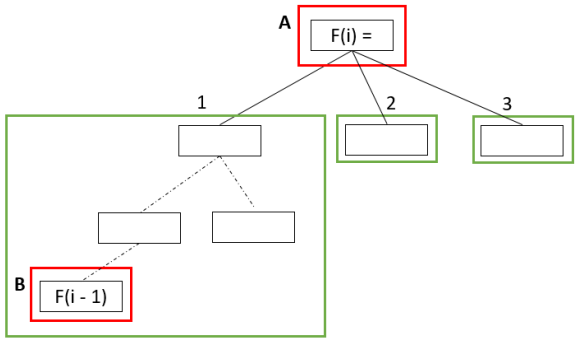


Figure 4.1: General recursive structure for the GPGR method. A indicates the recursive function from the R2 method, B indicates the recursive terminal. The inputs of the recursive function are shown in green boxes and numbered from one to three.

By including the recursive terminal $f(i-2)$, this method is able to solve problems like the Fibonacci problem, which is a symbolic regression problem that the previous R2 method

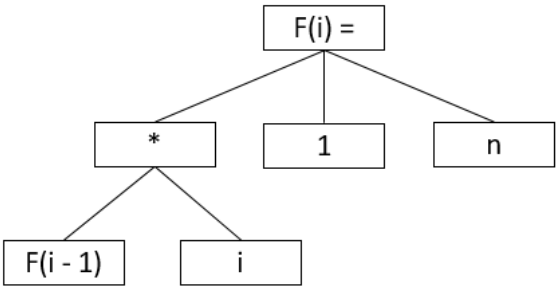


Figure 4.2: Factorial(n) function as an example of the R2 structure.

could not solve. This method is also more flexible than the previous R2 method because the body of recursion is not restricted to any specific depth, whereas the previous R2 method only allowed functions of depth two within the recursive body. In theory this method could include any number of recursive terminals $f(i-n)$ which could be needed for more complex problems.

4.3 Implementation Details and Constraints

There were three different types of problems used to evaluate this method. All three types of problems were different in how the input data was represented which meant they had to be implemented slightly differently. The idea of the template function remains the same, and the difference is mostly in extra functions for the classification problems, and a different implementation for Artificial Ant problems as they use a different kind of data type.

4.3.1 Symbolic Regression Problems

Symbolic regression problems are the simplest design the recursive template function for. The way the template recursion function has been implemented was done to make the evaluation of recursive programs as efficient as possible. The data type of regression problems is numeric, so the program uses numeric values entirely with no need for other types of data. There are no extra special functions for the regression problems as they should have solutions by simply using the recursive template function.

The implementation used a list, which starting from 0 the function would evaluate at each possible input number and store the result in the list. The function's recursive calls were evaluated by taking the correct previous result from the list. This was a simple and efficient way to evaluate symbolic regression problems where the values for each input number remain constant for all time. Algorithm 2 details how this approach works.

4.3.2 Classification Problems

The implementation for classification problems was the same as the regression problems in terms of the recursive function would store the results in a list and use these values to evaluate the recursive calls. This efficient implementation where no result is evaluated more than once is much more important in classification problems as there are **500** training examples that each individual in each generation needs to be evaluated with.

To keep consistency with previous work on classification tasks with GP with loops [4], [5] some special functions for combining pixel values must be included in the function set for these tasks. These functions are Sum Mode which returns the sum of all the modes of the input data, Sum which returns the sum of all the input data, and Standard Deviation

Algorithm 2 GPGR - Recursive template function for Regression problems

```
1: procedure GPGR
2:     Body ← children[0]
3:     End ← children[1]
4:     Base ← children[2]
5:     CurrentCount ← End
6:     for i = 0; i < End; i++ do
7:         Values[i] ← Body.evaluate()
8: The recursive body will then be evaluated as follows
9:
10: procedure Body.evaluate(i)
11: This evaluates the GP subtree that is set as the recursive body
12: After evaluating other mathematical operators eventually there will be a recursive terminal
13: return ... RecursiveTerminalN.evaluate(i) ...
14: The recursive terminal is then what provides the recursion
15:
16: procedure RecursiveTermianalN.evaluate(i)
17:     if i is equal to 0 then
18: return Base.evaluate()
19:     else
20: return Values(i - N)
21: By returning the previous value, this is providing recursion
```

which returns the standard deviation of the input data. Each special function can only occur within the recursive function, and takes in two arguments. The first argument to the special functions is the recursive terminal, which gives part of the recursive structure for the recursive function. The second argument is the pixel or attribute information to be used in the function. The recursive function evaluates all the pixel values used in the special function and combines and returns the result of this function.

Another change to the function for classification tasks was that the special count terminal "i" returns the data of the current pixel or attribute instead of the current count value. The base case, or third argument is also a pixel or attribute number and the function returns the data of in the base case. Unlike regression tasks where the data is only concerned with numeric types, the input of this task is pixel or attribute data, so the recursive functions will need to work with this data instead of a single number.

4.3.3 Artificial Ant Problems

The implementation had to change for the artificial ant problems. This is due to the fact that these problems have solutions that create a path. While the individual is being evaluated it will move along its path, so the result of a recursive call will be a set of instructions which will need to be evaluated at the time to reposition the ant. Algorithm 3 shows how this approach was implemented and further highlights some of the issues this method has in terms of training time.

This caused some difficulties as now the implementation has the potential to be very very slow especially when including multiple recursive calls or a nested recursion. This highlighted the need for more constraints around the program structure for this particular type of problems.

The solution to these issues was that the programs were restricted to only include one

Algorithm 3 GPGR - Recursive template function for Ant problems

```
1: procedure GPGR
2:     Body ← children[0]
3:     End ← children[1]
4:     Base ← children[2]
5:     i ← End
6:     if End > 0 then
7: return Body.evaluate(i)
8:     else
9: return Base.evaluate()
10: Body.evaluate is then called
11:
12: procedure Body.evaluate(i)
13: Somewhere within the body, there will be a recursive terminal, which provides recursion
14: return... RecursiveTerminal.evaluate(i) ...
15: The recursive terminal is then called, which invokes a recursive call on Body.evaluate
16:
17: procedure RecursiveTerminal.evaluate(i)
18:     if i > 0 then
19: returnBody.evaluate(i − 1)
20:     else
21: return Base.evaluate()
```

recursive terminal, which was beneficial to the results and particularly the training time. The fact that this improved the results shows that perhaps this level of generality is not always needed for some problems and by restricting the search space, good solutions were discovered much sooner in the evolutionary process.

This method is still the same design as the other types of problems, although it is more restricted. The evolved program tree can still make a recursive call at any depth, and uses a count based system to determine how many recursions should occur. The R2 method used whether there was food ahead to determine if it continued to execute the recursive function.

Overall it would have been good to not restrict the number of recursive terminals, to keep the GPGR method generic. This was unfeasible in terms of training time due to implementation constraints, and from test experiments appeared to be detrimental to performance.

4.4 Summary

The GPGR method is a much more general approach to solving recursive problems than the R2 method developed over summer. The new approach is designed to solve symbolic regression, classification, and artificial ant problems. The R2 approach only solved symbolic regression and artificial ant problems. The new approach is also more general in its recursive structure and should now solve a wider range of symbolic regression problems including the Fibonacci problem which is very common.

Chapter 5

Experiment Design

This chapter describes the benchmark problems and methods as well as the parameter settings used in the experiments. Each GP method will be tested over 50 independent runs with 50 different random seeds. The experiments will then be compared with the benchmark methods and tested for statistical significance using the Wilcoxin Rank Sum test.

5.1 Benchmark Problems

In order to measure the performance of the improved methods in terms of accuracy and speed, they need to be tested across a wide range of benchmark problems. These problems will include Symbolic Regression problems similar to those used in [12], Binary Classification problems, and Artificial Ant problems.

5.1.1 Symbolic Regression

Factorial Problem

This function was used as a benchmark in [2] [12]. It is a relatively easy problem for loops and recursion to solve. The factorial function can be defined recursively as $Factorial(x) = x \times Factorial(x-1)$ with base case $Factorial(0) = 1$.

Sum Problem

This function was used in [34][12] as a benchmark for loops and recursion. It is another relatively easy problem for loops and recursion to solve. It is defined recursively as $Sum(x) = x + Sum(x-1)$ with base case $Sum(0) = 0$.

Binomial Problem

The Binomial problem was used as a benchmark for our recursive methods in [12]. This problem is quite hard for even loops and recursion to solve as it requires multiple recursive functions. The Binomial function is defined as $Binomial(x) = \frac{Factorial(x)}{Factorial(x-k) \times Factorial(k)}$. To solve this problem, GP must find three slightly different factorial problems and then combine them correctly.

Factorial + Sum Problem

This is a problem of moderate difficulty, as GP must combine the two recursive functions, Factorial and Sum. This makes it more difficult than the simple Factorial or Sum problems on their own, but less difficult than the Binomial problem. The function can be defined as $Factorial + Sum(x) = Factorial(x) + Sum(x)$.

Fibonacci Problem

This problem is a different type of recursive function from the others. The above problems have all only used recursion by the previous value. The Fibonacci problem is defined as $Fibonacci(x) = Fibonacci(x-1) + Fibonacci(x-2)$ [1]. This function uses two recursions, one by x-1 and one by x-2. The base cases for this function are $Fibonacci(0) = 1$, $Fibonacci(-1) = 0$.

5.1.2 Binary Image Classification

This set of benchmarks are image classification, where the Cell and MNIST datasets have been used in [5] for GP with loops and achieved good results. All tasks use **500** training instances and **500** testing instances for classification between 2 classes.

Cell

The Microscopical Cell Image Database (Serous cytology) [35][36] contains 18 different classes of cells. This task will focus on classifying between Lymphocytes non actives and Mesotheliales. The images have been resized to be 25 $\times$ 25 pixels and the pixel values are integers between 0 and 255 to give a grey scale image. The content inside the cells is relatively plain and is similar across the two classes. In order to perform accurate classification, GP will need to use the shape of the cells making it a well suited task to loops or recursion. Loops and recursion can find the shape of the cell by using a large block of pixels. Without loops or recursion, GP would select a few specific pixels to base the classification on.

MNIST

The MNIST dataset of handwritten digits is a dataset that contains 28$\times$28 pixels [37]. This is a multi-class dataset, but the digits 6 and 9 were chosen to be used for a binary classification task. These classes were chosen as they were used as a benchmark for GP with loops in [5] and had good performance using these loop structures. The GP systems must be able to determine between the shape of the two digits.

Face

The face dataset is a collection of 19$\times$19 pixels, grey scale images of faces and non-faces [38]. This is a binary classification task. The GP system must be able to recognize the key features of the face.

Pedestrian

The pedestrian dataset is a more challenging dataset, it contains images of pedestrians crossing roads as well as images without pedestrians [39]. The task is difficult because the images can contain background objects which can confuse the classifier. The images have been taken at different locations so do not have a consistent background, and different pedestrians are

used in the images. A very general solution is needed for this task as the training and testing data has a lot of variation.

5.1.3 Binary Non-Image Classification

These tasks were used in [4],[5] and had promising results with several specific loops. Some of these problems, particularly the poker hand problem, can be very difficult for standard GP with no loops or recursions to solve. All of these tasks use **500** training instances and **500** testing instances for classification between 2 classes.

Poker Hand

This is a classification task to classify between single pair or an empty hand in poker [40]. The dataset has 10 attributes, which give the value and suit of each card. The attributes are ordered alternating between value and suit. This is a particularly challenging task, the step loop in [5] however has managed to achieve nearly perfect results for it. The recursive method should be able to find a good solution as it uses a similar approach. The reason this is a challenging task is that the relevant attributes are only every second attribute, as the suit does not matter. This requires the function to ignore the irrelevant suit attributes but still consider all of the value attributes as the pair can occur in any two positions of the hand.

Forest Cover

The forest cover dataset is used to predict the forest cover type from 53 cartographic attributes [41]. The full dataset contains seven different classes. For this work only two classes, Cottonwood/Willow and Douglas-fir, will be used for Binary Classification.

MiniBooNE

The MiniBooNE dataset is used to classify between particles being electron neutrinos and muon neutrinos [40]. This is a way of classifying a signal from background. Each entry in this dataset has 50 numerical attributes.

Spam

The spambase dataset is used to classify emails as spam or not-spam [40]. The dataset has 57 floating point attributes that characterise the different emails.

5.1.4 Artificial Ant

The goal of the Artificial Ant Problems is too eat all the food indicated by a '#' character on the trail. The ant eats food when it moves onto a square that has food on it. Gaps in the trail, where there is no food, are indicated by a '.' character.

Three Artificial Ant tasks were used to test the initial R1, R2 and R3 methods. The modified 1 trail [2] shown in table 5.1, modified 2 trail [29] shown in table 5.2, and the Santa Fe trail [29] shown in table 5.3 are the three different ant trails used, and are of increasing difficulty. Modified 1 had very good results with the R1 and R2 methods, but the Santa Fe trail and modified 2 trail were more challenging and only the R1 method achieved good results.

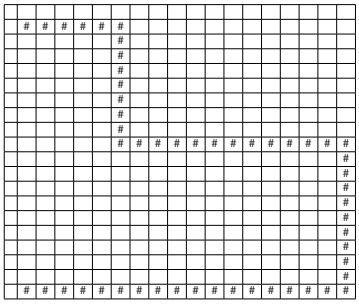


Table 5.1: Modified 1 trail.

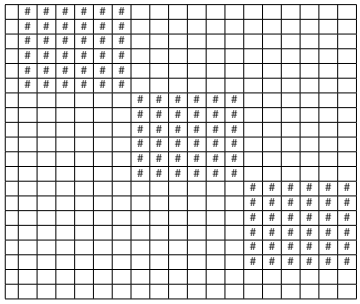


Table 5.2: Modified 2 trail.

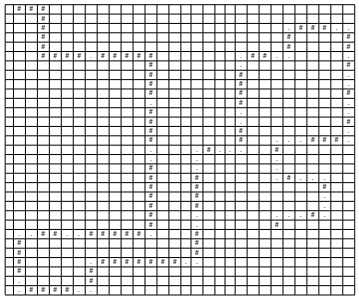


Table 5.3: Santa Fe trail.

5.2 Benchmark GP Methods

To measure the performance of the new recursive methods, they will be compared to standard GP, GP with loops, and the R2 method, for the symbolic regression tasks. For the classification tasks the new methods will be compared to GP with loops that were specifically designed for classification tasks, as well as standard GP with no loops. For the Artificial Ant problems, GPGR will be compared against standard GP, GP with loops, R1, R2, and R3.

5.2.1 Standard GP - No Loops

This is the standard GP method, not using any loops or recursions. On most of the problems used, this is not expected to do as well as loops or recursion, because the problems are very complex. The functions and terminals used for this method are {+, -, *, %, 1, 0, ERC(1-15)} for symbolic regression problems, and {+, -, Pixel() (or input() for non-image tasks), ERC(1-100)} for classification problems. ERC(1 - N) generates a random value between 1 and N, Pixel() returns the pixel or value where the argument(s determine which value. If the task is an image classification task pixel() will have two arguments which determine the x and y coordinate for the pixel, if it is a non-image task, then input() will be used instead, which has one argument to determine which attribute to be returned.

For the Ant problems, the function set was $\{prog2(), prog3(), ifFoodAhead()\}$ and the terminal set was $\{move, right, left\}$

5.2.2 Count controlled Loop - WhileLoop1

As described in [2] while loop 1 is a count controlled loop. This loop has three arguments (start, end, body). Start is the value that the loop counter starts at, end is the value it ends at, and body is the program to be executed within the loop.

Some other special terminals and functions need to be included for this loop to be useful. The first is a terminal i, which returns the current count value of the loop, i starts at the value of start and increases as the loop is executed until the counter is equal to the value of end.

Another special function and terminal used is SetY and Y. This is a variable used in the program that can be set with SetY and used with the terminal Y. This is useful inside the loop as a way to store the previous value of the loop body. SetY takes in one argument and sets the value of Y to this argument.

5.2.3 Event controlled Loop - WhileLoop2

WhileLoop2 is an event controlled loop described in [2]. The loop takes two arguments the first is a condition which is of Boolean type (true or false), and the second is the loop body. The loop works by first checking if the condition is true and then executing the loop body.

This loop again uses special functions and terminals i, SetY and Y. The terminal i starts at zero and increases by one every time the loop is executed.

5.2.4 R2

This is the R2 method developed in our previous work [12]. It was important to compare this method for the symbolic regression problems to determine if the new design of the R2 method shows any improvement.

The R2 function R2() has three inputs, the first, y, is the function operator (one of {+, -, *, %}) to use, the second is the base case, z, which is what the function returns when x < 1, and the third, x, is the value to use in the function. The function takes the form $f(x) = if(x < 1, xyf(x-1), z)$ where x, y, z are described above. To solve a simple example of the factorial(n) problem, the inputs to the function would be x = n, y = *, z = 1.

The function set used is the same as [12] for the Factorial, Sum and Binomial problems, where a minimal function set is used. For the new symbolic regression problems, Fibonacci and Factorial + Sum, the function and terminal sets include {+, -, *, %, 1, 0, n, R2}. These functions are not trivially solved for the previous R2 method, particularly the Fibonacci problem as it requires a function $f(x) = if(x > 1, f(x-2) * f(x-1), 0)$ which is not possible to be solved by the previous R2 method due to it needing a terminal f(x-2).

5.2.5 Classification Loops

For classification tasks, many different and often specific loops have been used. The classification tasks are the same as used in [4],[5], so will be compared to the results of loops in these papers as well as GP with no loops. The loops used in these two papers are Li's loop, EB-Loop, Step-Loop, Sort-Loop, Bar-Loop and Seam-Loop. The Bar-Loop and Seam-Loop were specific to image problems so are not being used for comparisons.

In this work, special functions are used in order to aggregate the image and attribute data. These functions are sum-mode, sum and standard deviation. These functions are used inside the loops to combine the pixel data found with the loops. Another special function used is pixel(x, y) for the image problems which gets the pixel value at position (x,y). For the non-image classification tasks a function input(x) was used to get attribute data at attribute x.

Some of the loops used have very good performance on one or two particular tasks, but not the others, whereas some other loops perform moderately across many of the problems as they are a more general solution. All of these loops will be used as comparisons to R2 in this work to compare how the recursive method does in comparison to the loops. GPGR is not expected to do as well as the more specific loops for some of the tasks, but is expected to perform reasonably across all of the classification problems.

5.3 Parameter settings

The parameters used for this work were kept similar to what was done in the previous work for R1,R2,R3 in [12][4][5][29]. The typical parameters used in our work are given in table 5.4, although there are some exceptions to these which are mentioned below.

An exception to these basic parameters was the Binomial problem, which used a population size of 2000 and a mutation rate of 0.5, cross over rate of 0.48. The change in parameters for the Binomial problem was because this problem is much harder to evolve solutions for than the others. The Binomial problem also used 150 generations. The Factorial Plus Sum problem used 150 generations, as it was a difficult problem to solve. The For classification

Parameter	Value
No. Generation	50
Population size	1024
Initial depth	2
Maximum depth	6
Crossover rate	0.7
Mutation rate	0.28
Reproduction rate	0.02

Table 5.4: Showing the parameters used in the majority of my experiments.

problems, a maximum depth of 8, an initial depth between 2 and 3 was used as the tasks often required a more complicated solution. The ant problems used 100 generations to account for them being more difficult than the other problems.

5.4 Summary

The new methods will be compared against 10 different state of the art algorithms on four types of 16 problems. The problems used were chosen because they are known to have an iterative nature.

Chapter 6

Results

This section describes the results obtained from running the experiments. Each method will be evaluated individually against the benchmarks.

Table 6.1, table 6.2 and table 6.3 show the results on the symbolic regression, classification and artificial ant problems respectively. The results of the symbolic regression problems are compared to NoLoops, WhileLoop1, WhileLoop2 and R2. The new methods tested on these problems are both GPCR and GPGR. The classification problems are tested against NoLoops, and a range of loops for classification problems, and the only new method used for classification is GPGR. The artificial ant problems results use the new method GPGR, and are compared to R1, R2, R3, as well as while loop1, while loop2 and no loops. Statistical significance was tested using a Wilcoxon rank-sum test for a significance level of 95%. The fitness is given by the average number of hits that the best program of each run achieves. The statistical significance is based on this fitness value.

6.1 GPCR

The results for the GPCR method in Table 6.1 show that as expected, it was able to find very good results for the regression tasks that were directly included in the function set. The results were not as good as the previous R1 results, but this is due to the size of the function set increasing, making the search space larger. The GPCR method can also now solve problems like the Fibonacci problem, which is a common recursive sequence that appears in many real-world problems.

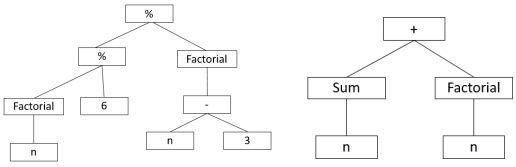
Problem	Measures	No-Loop	Loop1	Loop2	GPCR	R2	GPGR
Factorial	Hits (15)	2.02±0.14	8.94±6.32[1]	7.48±6.4[1]	15.0±0.0[123]	15.0±0.0[123]	14.44±2.4[123]
	Time (ms)	184.18±21.19	331.22±1329.4	256.66±84.8	35.88±7.84	7.88±1.24	68.96±130.79
	No. Perfect (50)	0	26	21	50	50	47
Sum	Hits (15)	1.74±0.44	10.86±6.05[1]	8.56±6.21[1]	15.0±0.0[123]	15.0±0.0[123]	14.76±1.68[123]
	Time (ms)	120.82±18.68	403.04±940.71	443.92±453.69	34.68±5.9	7.62±0.77	158.08±461.09
	No. Perfect (50)	0	34	24	50	50	49
Binomial	Hits (15)	1.58±0.6	3.08±0.59[1]	2.66±0.65[1]	5.82±2.9[123]	4.54±2.77[123]	5.44±3.3[123*]
	Time (ms)	316.86±62.47	256141.16±1145089.66	78094.54±149008.64	1596.78±571.63	630.66±169.43	3468.86±2199.75
	No. Perfect (50)	0	0	0	4	3	3
Factorial + Sum	Hits (15)	2.16±0.37	2.44±0.5[1]	2.18±0.38	5.84±5.74[13]	2.26±0.44	3.64±3.38[123*]
	Time (ms)	481.46±58.22	390508.14±853500.48	789.56±230.41	1173.88±410.51	446.14±123.94	585.0±183.47
	No. Perfect (50)	0	0	0	14	0	4
Fibonacci	Hits (15)	4.52±0.54	4.74±1.56	4.04±0.2	13.24±4.03[123]	4.68±0.9[3]	8.18±4.74[123*]
	Time (ms)	495.32±53.83	218894.8±874728.91	804.2±600.02	252.94±306.81	468.04±115.23	514.32±284.79
	No. Perfect (50)	0	1	0	42	0	16

Table 6.1: Table showing results on **Symbolic Regression** problems. [1] indicates significantly better than no loops, [2] indicates significantly better than WhileLoop1, [3] indicates significantly better than whileLoop2. * indicates that the GPGR method is significantly better than the R2 method.

The GPCR method when compared to the benchmark loops and no loops was signifi-

cantly better than them on all problems. The only exception to this was that for the Factorial + Sum problem, GPCR was similar to loop1. However GPCR had 14 perfect solutions on this method compared with none for either of the loops or no loops. There was a lot of variance in the fitness of the evolved programs for this problem with the GPCR method, which is why it was not able to be significantly better than loop 1 which was the best performing benchmark.

The evolved programs for the tasks were also very simple and easy to understand using knowledge about the recursive functions defined in the function set. Figure 6.1 shows the Binomial problem evolved program structure which is one of the more complex tasks. Figure 6.1 also shows the evolved program for the Factorial Plus Sum problem which is another combination of functions included in the function set. In both problems, the evolved programs are very simple, which indicates why this method performs better than benchmarks for these difficult methods, as the evolved programs are less complicated.



(a) Binomial (b) Factorial Plus Sum

Figure 6.1: Best evolved programs for the Binomial and Factorial Plus Sum problems using GPCR method.

6.2 GPGR

6.2.1 Symbolic Regression Problems

Table 6.1 shows that for the factorial and sum problems. The results are similar across all the recursive methods, and the loops also perform well. The Binomial problem proves to be difficult for all the methods, but the GPGR method is significantly better than the loops, no loops and the R2 method. The Fibonacci and Factorial + Sum problems are of medium difficulty and have good results with the GPGR method, where GPGR is significantly better than the benchmarks for the two problems. The Fibonacci problem did not get any perfect solutions with the R2 method, which is not surprising as the method cannot handle two recursive calls within the same function, which is what the solution requires.

The results show that GPGR has improved in performance over the R2 method on the more difficult symbolic regression tasks. The performance has decreased on the simple (Factorial and Sum) symbolic regression problems. This is due to the new method being more general than the previous. The R2 method was very well suited to the Factorial and Sum problems so it had nearly perfect performance on them. The new method requires a larger program tree to perfectly solve these tasks which makes it more difficult. The GPGR method still performs well on these simple problems, as it is still able to find perfect solutions in most cases. The new method however can now solve tasks like the Fibonacci problem and other problems of that form. The results for the Binomial problem show that the new method is an improvement over the previous, however it is still only able to get a few perfect solutions on this problem. A possible reason for this is that the search space is not very smooth. The way that the fitness function works for these tasks is that it counts the number of hits the problem is getting with a maximum of 15. There are about four fitness cases that usually hit, but the other 11 are more difficult. This causes GP to evolve many programs that are far from the perfect solution but that hit 4 or 5 of the fitness cases, meaning that perhaps

solutions that are closer to the true solution are not being carried over to the next generation as they might not have as many hits.

6.2.2 Classification Problems

GPCR was also used for classification tasks. The results were compared against that of GP with loops that were developed specifically for classification tasks, these results are shown in table 6.2. The results show that the R2 recursive method was significantly better than many of the loops on some of the tasks, and at least better than one of the loops on all tasks. A problem that was well suited to this recursive structure was the Poker Hand problem. The method however did no perform as well as the step loop for this problem. The reason this was suited to the GPGR method was the $f(i-2)$ recursive terminal. The solution was able to skip over every other attribute, as the poker hand task has attributes representing the value and then the suit of each card in a poker hand. The task was to classify between a single pair and an empty hand, therefore the suit attributes were irrelevant, so the GPGR method skipped over these. This can be seen in the evolved program for this task in figure 6.2. The step loop also has this property of being able to skip over every other attribute, but requires a slightly shallower GP tree which is probably why the performance was slightly better for this loop than our recursive GPGR method. The best program for the poker hand dataset had 99.8% training accuracy, and 99.6% testing accuracy. This is a very good performance, however the step loop was able to find many perfect solutions with 100% testing and training accuracy. This indicates that there is likely some added complexity in the GPGR method that makes it harder to find a good solution quickly.

Dataset	Measures	No-Loop	EB-Loop	Li's Loop	Step - Loops	Sort-Loop	GPGR
Forest	Training accuracy %	78.35±2.4	77.05±2.44	80.15±2.02	84.08±2.54	66.85±1.56	82.69±2.6^{1235}
	Testing accuracy %	67.93±4.64	69.69±5.66	74.96±6.23	79.19±6.95	62.62±2.81	75.22±7.17^{125}
	Time (s)	4.49	47.58	50.59	144.53	65.03	164.48
Poker	Training accuracy %	57.55±0.82	56.69±0.83	71.11±2.14	100.0±0.0	84.95±0.76	94.42±3.84^{1235}
	Testing accuracy %	52.64±2.6	52.65±1.87	66.14±2.01	99.93±0.18	87.56±0.96	92.22±4.91^{1235}
	Time (s)	4.32	24.14	59.69	12.77	46.26	188.78
MiniBooNE	Training accuracy %	82.96±0.22	82.7±0.53	84.71±1.43	88.72±1.95	87.85±1.54	88.32±1.21^{123}
	Testing accuracy %	75.86±0.8	75.89±1.13	78.71±1.34	81.8±2.24	83.68±1.93	81.54±1.76^{123}
	Time (s)	5.08	37.73	62.92	149.83	76.8	261.91
Spam	Training accuracy %	87.65±4.92	88.31±2.13	91.41±1.47	91.2±1.63	72.78±1.56	85.25±4.21^{5}
	Testing accuracy %	86.58±4.09	86.88±2.49	89.43±1.35	89.19±1.9	74.31±1.58	85.04±3.65^{5}
	Time (s)	4.72	42.04	46.46	67.66	57.14	92.42
Mnist	Training accuracy %	99.26±0.47	98.83±0.85	99.52±0.14	99.46±0.21	66.37±0.64	98.97±0.37^{5}
	Testing accuracy %	96.66±0.73	97.95±1.3	98.87±0.42	98.46±0.63	61.68±1.03	96.19±0.9^{5}
	Time (s)	11.11	155.46	73.32	434.82	376.3	82.1
Pedestrian	Training accuracy %	83.08±1.04	80.6±1.6	84.58±2.29	82.05±1.5	82.35±2.58	82.8±0.87^{24}
	Testing accuracy %	77.3±1.85	76.75±1.95	78.12±3.02	76.56±2.26	78.38±3.28	77.06±2.02
	Time (s)	8.55	169.63	169.63	363.34	268.28	91.55
Face	Training accuracy %	96.36±0.91	93.5±2.23	96.11±1.65	95.77±2.06	84.99±2.82	97.89±1.06^{12345}
	Testing accuracy %	91.1±1.69	87.86±2.74	90.69±2.7	91.56±3.07	86.71±2.43	94.4±1.53^{12345}
	Time (s)	7.8	95.08	31.38	300.28	285.27	451.06
Cell	Training accuracy %	95.85±0.43	95.17±2.06	96.14±0.59	95.92±1.26	89.95±0.86	96.74±0.55^{12345}
	Testing accuracy %	93.84±1.25	92.68±2.41	94.02±0.91	93.42±1.96	86.58±1.39	94.3±0.49^{2345}
	Time (s)	9.48	106.59	50.17	437.03	243.77	202.01

Table 6.2: **Classification** results showing the GPGR method against the benchmarks. Statistical significance of the GPGR method is indicated with ^{1} for statistically better than no loops, ^{2} for statistically better than EB-Loop, ^{3} for statistically better than Li's Loop, ^{4} for statistically better than the Step-Loop and ^{5} for statistically better than the Sort-Loop

GPGR had very good performance on two of the image problems when compared to the other methods, the GPGR method was significantly better than all of the benchmarks on the Face dataset, and GPGR was significantly better than all but one benchmark for the Cell dataset. The best evolved programs of these are shown in Appendix A.

It is known that for a good classification accuracy on the cell dataset, the classifier needs to evaluate the shape of the cells as opposed to the individual pixels. It was expected to have high performance with recursion on the cell problem, as it has had good performance with loops in prior work. Loops and recursion allow a classifier to use large blocks of pixels as well as single pixels. The trees for the best programs on these two problems were very large and complex and using many of the recursive functions. The program tree for the GPGR method on the Face dataset uses five recursive functions as well as a large number of individual pixel values. This might be because the face images have multiple key features that need to be discovered by the recursive functions and then used for classification. The program tree for the Cell problem used seven recursive functions, making it a very complex program. It is known that the Cell data is mainly classified based on the shape of the cells, which means it should use large groups of pixels to determine the class. By using many recursive functions, the evolved program tree is able to use large groups of pixels.

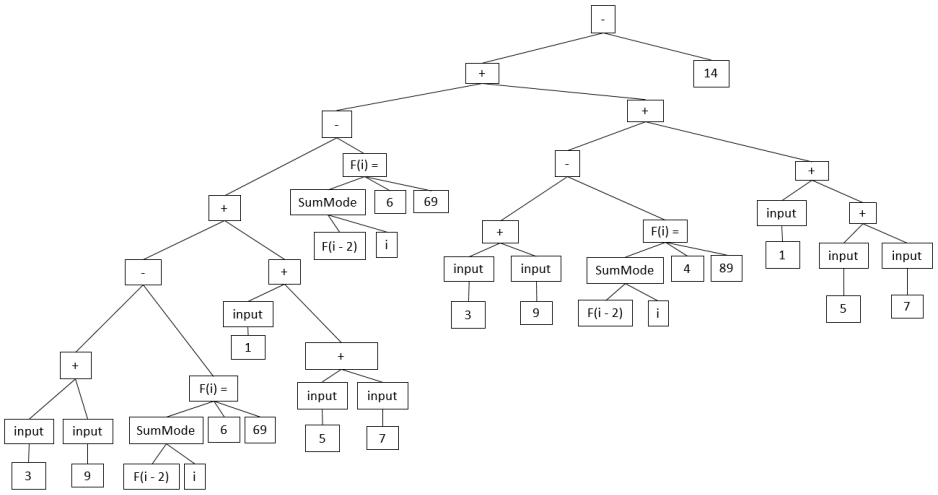


Figure 6.2: Best program evolved by the GPGR method for the poker hand classification problem.

6.2.3 Artificial Ant Problems

The results for GPCR on Ant problems are given in table 6.3. These results show that for the Artificial Ant problems. The new GPGR approach has significantly better performance on the difficult trails, modified 2 and Santa Fe. The results on the simplest trail, modified 1, show that the R2 method performed better than the new approach. The new approach however, was still able to find many perfect solutions to this simple trail.

Problem		No-Loops	Count loop	Event Loop	R1	R2	R3	GPGR
Modified1	Food Eaten (53)	21.84±5.71	39.06±14.23[1]	45.34±10.16[1]	52.9±0.7[123]	53.0±0.0[123]	24.3±17.43	43.54 ± 12.07 [126]
	Time (ms)	330.2±52.59	223.3±112.59	270.32±171.43	96.0±158.37	26.68±22.33	370.62±109.36	1328.56 ± 1458.63
	No. Perfect (50)	0	21	27	49	50	9	28
Modified2	Food Eaten (108)	26.18±5.32	41.92±16.78[1]	36.56±9.43[1]	108.0±0.0[123]	36.84±5.87[1]	31.12±17.62	46.74 ± 16.83[1356]
	Time (ms)	360.76±70.89	358.6±81.59	2060.16±6391.75	39.2±44.39	415.42±57.71	451.68±64.41	2765.02 ± 2144.97
	No. Perfect (50)	0	0	0	50	0	0	0
SantaFe	Food Eaten (89)	13.2±4.13	16.5±7.5[1]	15.44±5.94[1]	89.0±0.0[123]	13.04±8.78	14.64±8.66	18.62 ± 7.38[1356]
	Time (ms)	362.58±68.12	306.22±39.06	343.74±51.52	147.64±120.72	440.74±100.78	393.86±58.42	11440.8 ± 45968.02
	No. Perfect (50)	0	0	0	50	0	0	0

Table 6.3: **Artificial Ant** results show the GPGR method against the benchmarks. Statistical significance is indicated with [1] for significantly better than no loops, [2] significantly better than loop1, [3] significantly better than loop2, [4] significantly better than R1, [5] significantly better than R2, [6] significantly better than R3.

For the ant problems, the recursive template function had to be slightly more restricted in structure than for other problems, due to constraints in implementing that type of problem.

This meant that the biggest change from the R2 method is that the template function now uses a count based recursion.

The results for the GPGR method showed that, the performance was not as good as R2, and not as good as whileloop2. The reason for this is that with the event based loop or recursion like in R2, the ant is able to follow the trail of food. It does this by moving within the loop, and then adjusting direction so that the ant is always facing in the direction of food. With a count based approach, there would need to first be a check for food ahead and then the same loop as in the event based loop. This means that the program is more complicated. For the other two problems, the results are better than R2, however they still are not able to find perfect solutions which is likely due to the function being too restricted. These problems require very complex solutions to solve them, including nested recursion for modified 2. Overall the GPGR method has improved results on the two most difficult out of three tasks. The GPGR method is more consistent with design for classification tasks and regression tasks than the previous, which works towards the aim of having a general recursive method for a wide range of problems.

Figure 6.3 shows the best evolved programs trails for the GPGR method. This shows that for the modified 1 problem, the ant follows the correct path entirely. The modified 2 problem, the best trail (with 95/108 food eaten) traverses most of the available grid and still misses some food. The Santa Fe trail was only able to get 45/89 of the food for that trail, but this result shows that it is following a large portion of the trail, however travels a lot of unnecessary paths.

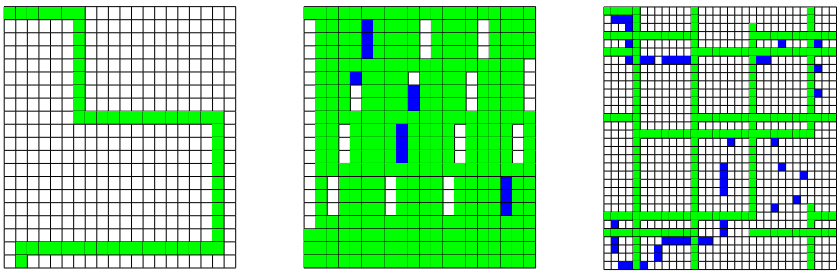


Figure 6.3: Trails of best ant problems from left to right, modified 1, modified 2, Santa Fe. Blue indicates the original trail that the ant has missed, green indicates where the ant has travelled.

6.3 Analysis of Evolutionary Process

The average of the best-of-generation fitness from 50 runs was also examined for each of the symbolic regression and Artificial Ant problems. The classification problems were not tested for this, as the benchmark loop results were taken from previous work instead of evaluated in this project.

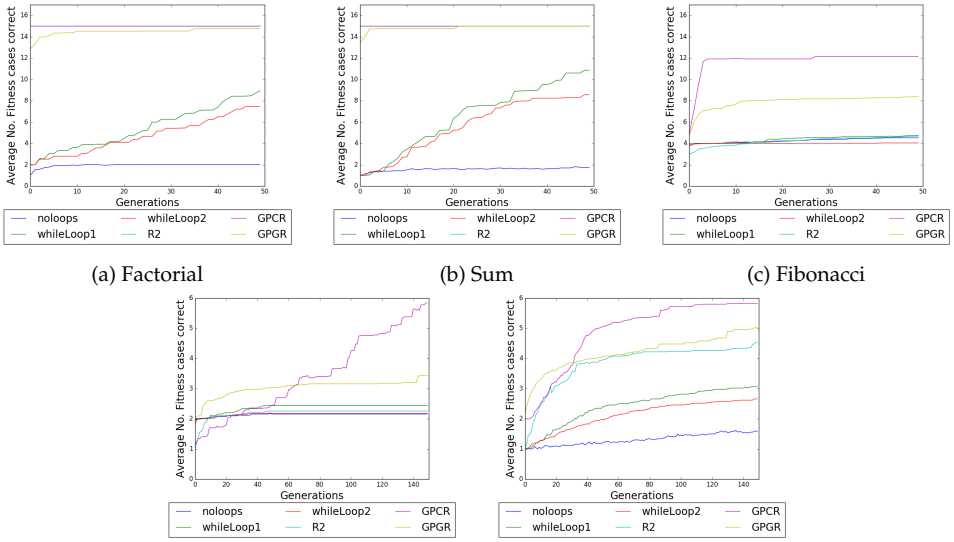
Figure 6.4 shows the evolutionary process for the five symbolic regression problems. This shows that for the Factorial and Sum problems, GPCR and the R2 benchmark had perfect performance from the first generation, meaning that the solutions for these methods must be very simple and of a small depth. GPGR was able to find mostly perfect solutions before 20 generations, and then did not improve much for these two problems. GPGR never quite reached all perfect solutions, but got very close especially in the Sum problem.

For the Fibonacci problem GPCR did the best, and had very high performance after just a few generations, which shows that it was easy for this problem to be solved by the GPCR method. GPGR on this problem did much better than the benchmarks, and although the performance was not as good as GPCR, it still was able to be much better than the

benchmarks rather quickly. The benchmarks for this problem all perform much worse than the two new methods and do not show much of an increase over the generations, but rather stay at a constant fitness of about 4 hits. This shows that this problem is far too complicated for the existing benchmark methods to solve, and they tend to converge to a solution of fairly poor fitness.

The Factorial + Sum problem was surprisingly difficult for all methods, however it can be seen that the two new methods, GPCR and GPGR, perform the best on this problem after 40 generations.

The Binomial problem also had much lower fitness than expected across all methods. It can be clearly seen that GPCR had the best performance, although it only converged at about 100 generations. GPGR was the next best, and seemed to still be growing at 150 generations which is where the evolutionary process was stopped for this problem. The R2 method was the best out of the existing benchmarks, and had performance similar to GPGR.



(a) Factorial (b) Sum (c) Fibonacci

(d) Factorial + Sum (e) Binomial

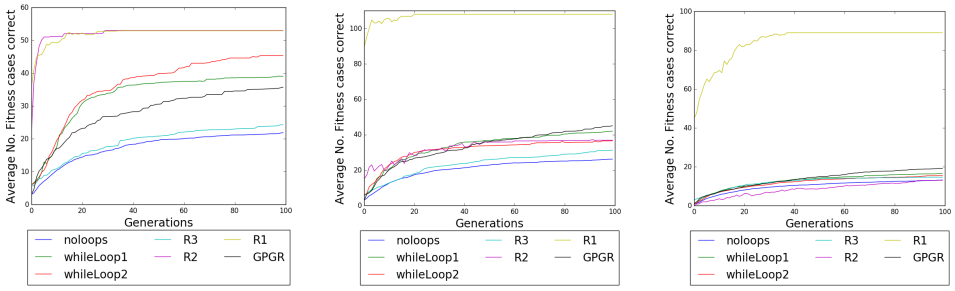
Figure 6.4: Evolutionary Process for the Symbolic Regression problems.

The evolutionary process on Artificial Ant problems was also analysed and is given in figure 6.5. This shows that for the Modified 1 problem, the GPGR method does worse than the benchmarks, which is because it is too general when compared to the benchmarks so cannot easily and quickly find a good solution. The GPGR method however is increasing much faster than the loop methods when it is approaching the end of the process, so perhaps with more generations, the performance would be better than loops.

For Modified 2, with the exception of R1 which was very problem specific, GPGR performs the best, and is still increasing when it reaches 100 generations. Similarly for the SantaFe problem. This shows that for these harder problems, GPGR is able to perform better than the benchmark loops and recursive methods. Overall for the Ant problems, GPGR appears to be improving still at 100 generations, so increasing the number of generations may help to improve performance on this method. GPGR is a very general method so it might take some time for GP to evolve a good solution.

6.4 Analysis of Hits for Regression Problems

The graphs in figures 6.6, 6.7, 6.8, 6.9 and 6.10 show the comparisons between all the methods on the regression problems. These graphs show the distribution of hits, showing the

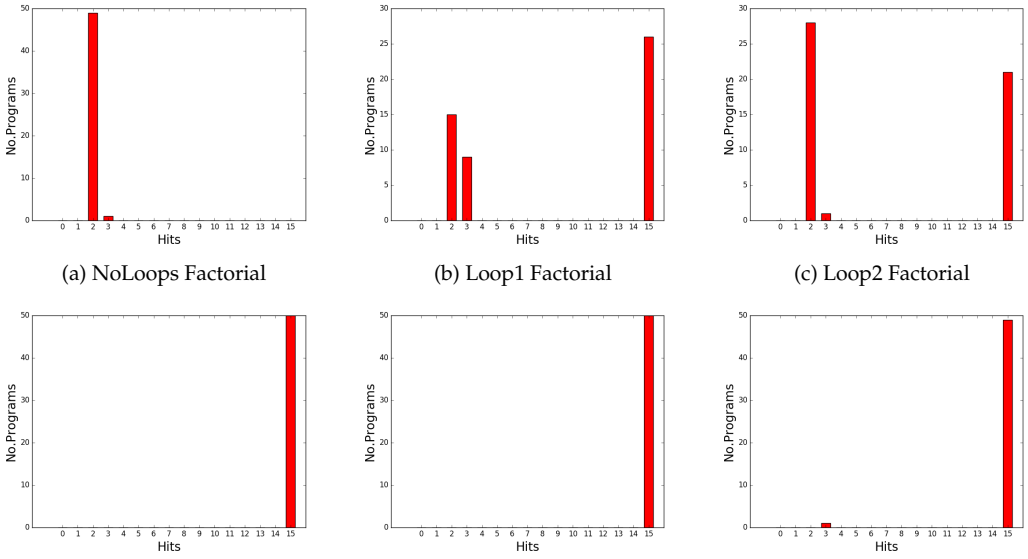


(a) Modified1 (b) Modified2 (c) SantaFe

Figure 6.5: Evolutionary Process for the Artificial Ant problems.

number of hits on the x axis and the number of best programs getting that number of hits on the y axis. The new methods show that they although may have fewer perfect solutions in the easy Factorial and Sum problems. For the other problems they have more good solutions and in most cases more perfect solutions.

The results showing the number of hits for the Factorial problem are given in figure 6.6. These show how the distribution of hits looks for the 50 runs with each method. This shows that GPCR and R2 were able to get completely perfect solutions (15 hits) in all runs. GPGR was able to get mostly perfect solutions, except for two solutions with three hits and one solution with 11 hits. This implies that there may some local minimum with this number of hits for this problem. The solutions without any loops or recursions only once had three hits, and otherwise had two. This shows that it is very likely that there are two hits that are very easy to get for this problem. The results with loops also had many solutions with only two hits as well as some with three hits, so there is likely a local minima with two hits that is very easy for GP to find. The loops also find many perfect solutions, although much less than the recursive methods. Overall, the results show that the recursive methods tend to perform much better than the loops on this problem.



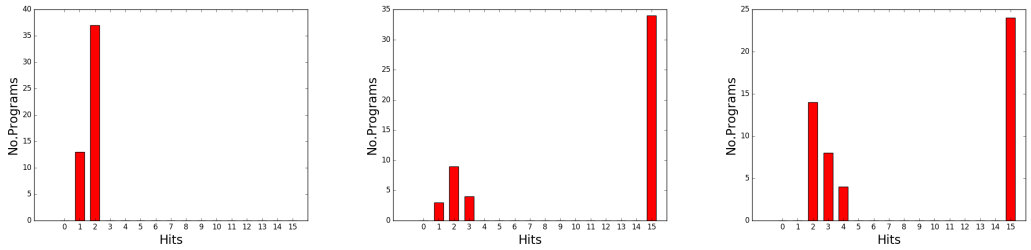
(a) NoLoops Factorial (b) Loop1 Factorial (c) Loop2 Factorial

(d) GPCR Factorial (e) R2 Factorial (f) GPGR Factorial

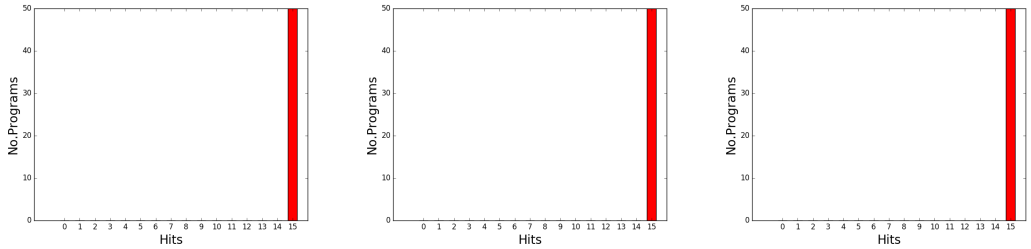
Figure 6.6: Graphs showing number of hits for the factorial problem

The hits for each method on the Sum problem are given in figure 6.7. This shows that the Sum problem is very similar to the Factorial problem, which is not surprising as they

have a very similar structure. The GPGR method was able to perform very well on this problem which is not surprising, and had all perfect solutions except for one program, which had only three hits. The GPCR method also unsurprisingly had all perfect solutions. The benchmark R2 method also had all perfect solutions, which is expected as the structure is very suited to this problem. The no loops method had solutions with mostly two hits, but some solutions with only one hit. The two loops had mostly perfect solutions, but also had a few cases where there were very few hits. This shows that there may be local minima in the solutions, where the solutions can get two or three hits but cannot easily evolve into the correct solution. It seems that for this problem and the factorial problem that there are rarely solutions with a moderate number of hits, it is typically between one and four hits or a perfect solution. This could be because the larger the numbers get in the fitness cases, the fewer programs will actually match them, meaning if a program is getting at least 10 hits it is quite likely to get the other five and be a perfect solution.



(a) NoLoops Sum (b) Loop1 Sum (c) Loop2 Sum

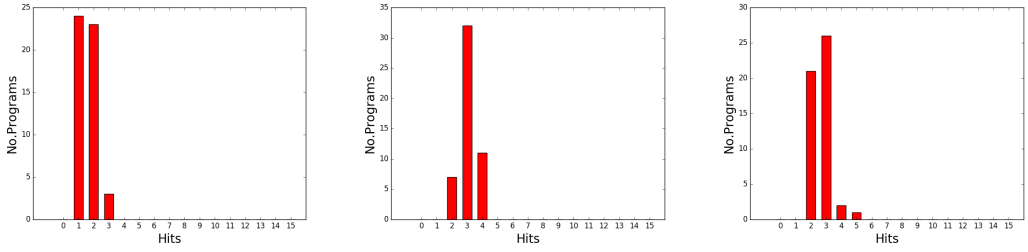


(d) GPCR Sum (e) R2 Sum (f) GPGR Sum

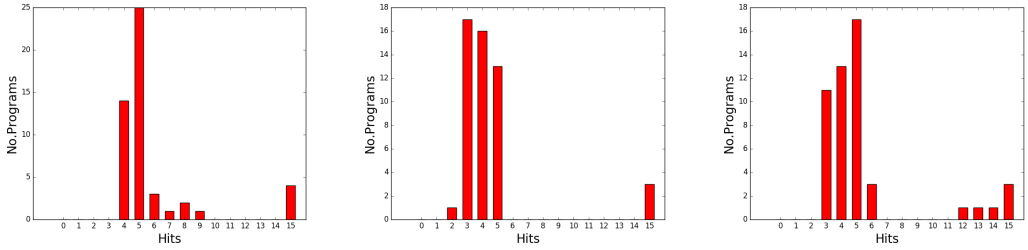
Figure 6.7: Graphs showing number of hits for the sum problem

The distribution of hits for the Binomial problem is shown in figure 6.8. This figure shows that for the binomial problem, the distribution of hits is much more complex than the factorial and sum problems. None of the methods are able to get all perfect solutions like in the previous problems. The three recursive methods are each able to get 3 perfect solutions. The two new methods GPCR and GPGR are mostly getting solutions between 3 and 6 with the GPCR having a few solutions between seven and nine hits. GPGR was able to find solutions that got 12, 13, and 14 hits, which indicates that there is perhaps a local minima, as there were functions with high performance that were not able to evolve into perfect solutions. The methods with loops and no loops did not perform as well as recursion, and did not have any solutions that had more than five hits. This shows that this method is very complicated for these methods to solve.

The distribution of hits for the Fibonacci problem is given in figure 6.9. This shows that the new methods, GPCR and GPGR performed best on this problem. This is not surprising as the R2 method should not be able to solve this problem. The count based loop, loop 1, was able to find a perfect solution for this problem. This is surprising as the problem is very difficult to solve with that type of structure. Other than this one perfect solution for Loop 1, the loops and no loops and R2 all performed very similarly on this problem, mostly getting



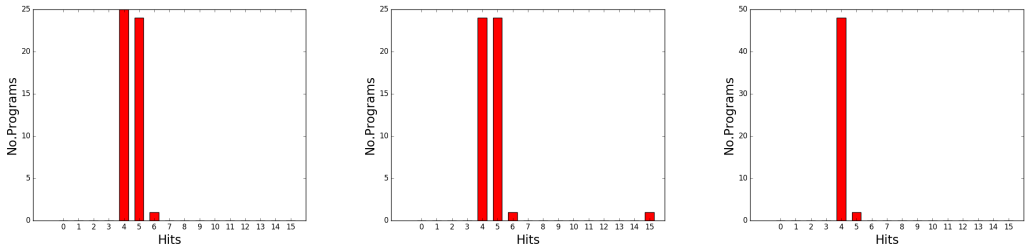
(a) NoLoops Binomial (b) Loop1 Binomial (c) Loop2 Binomial



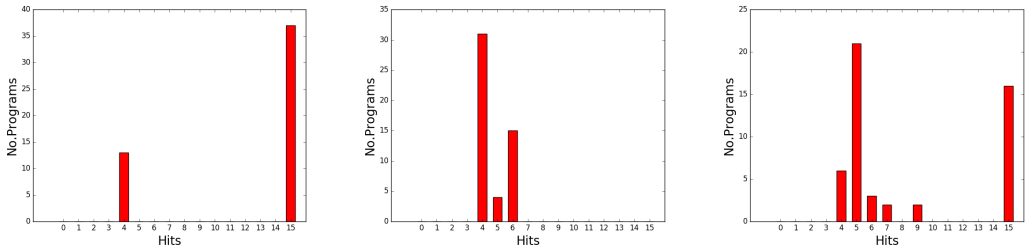
(d) GPCR Binomial (e) R2 Binomial (f) GPGR Binomial

Figure 6.8: Graphs showing number of hits for the binomial problem

solutions with four to six hits. This highlights the need for the new GPGR method, as clearly the R2 method has not performed well on this type of problem.



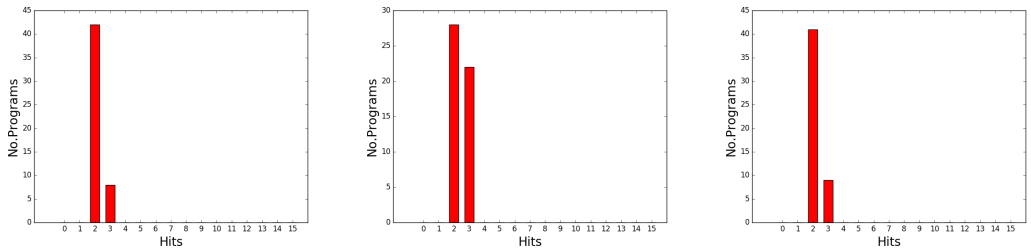
(a) NoLoops Fibonacci (b) Loop1 Fibonacci (c) Loop2 Fibonacci



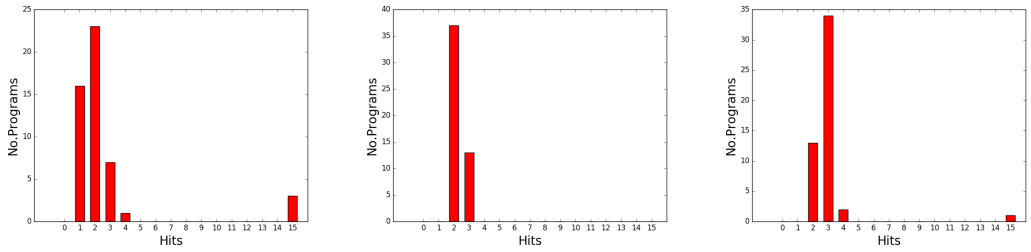
(d) GPCR Fibonacci (e) R2 Fibonacci (f) GPGR Fibonacci

Figure 6.9: Graphs showing number of hits for the fibonacci problem

The distribution for the factorial + sum problem is given in figure 6.10. These results are interesting, as it sounds like a very simple problem to solve, but has proven to be quite difficult for all the methods. The loop and noloops methods could not find any perfect solutions on it, and only had solutions with two or three hits. The R2 method had similar performance to the loops. The two new methods did the best, with GPCR doing better than GPGR. GPCR found a range of solutions with one to four hits, and then 14 perfect solutions. GPGR mostly found solutions with two or three hits which is consistent with the benchmarks, but also found 4 perfect solutions.



(a) NoLoops FactorialPlusSum (b) Loop1 FactorialPlusSum (c) Loop2 FactorialPlusSum



(d) GPCR FactorialPlusSum (e) R2 FactorialPlusSum (f) GPGR FactorialPlusSum

Figure 6.10: Graphs showing number of hits for the factorial + sum problem

6.5 Summary

The results show that overall, the performance has improved from the loops for a wide range of the benchmark problems. GPCR is able to perform very well on the common recursive problems it is tested for, this method out-performs the loops and standard GP. GPGR is the first generic recursive method designed for a wide range of problems of different types. GPGR manages to out-perform many of the loops on most problems, some of the classification loops out-perform it, however those loops are designed specifically for classification and do not provide a generic approach like the GPGR method does. The results on Ant problems show that GPGR is performing better than the benchmark methods for the two more difficult tasks, and still finding many perfect solutions on the easiest task. The evolved programs are understandable for the regression tasks, however for classification are still hard to interpret. This is due to these classification tasks being rather complicated. The recursive programs are no more complicated than those evolved by standard GP, but have better performance.

Chapter 7

Conclusions

This project aimed to develop a new general GP approach to solving problems with recursive nature. This goal was successfully achieved by significantly extending and improving R1 and R2 to develop two new methods, GPCR and GPGP. The new methods were evaluated against 10 state of the art algorithms on 16 different problems of four different types.

GPCR was intended as a recursive method to solve common recursive symbolic regression problems. This method extended R1 by increasing the size of the function set to include a wider range of common recursive problems. GPCR should be used when solving a problem that looks similar to some common recursive functions. The results of this work showed that by including a wider range of functions, GPCR was able to perform very well on a range of common recursive problems which meets the objective for this method. GPCR performed significantly better than loops and no loops in all cases.

GPGR was intended as a generic method for solving recursive problems. GPGR extended the R2 method by making the structure much more general. GPGR now allows for multiple recursive calls and a recursive body of an arbitrary depth instead of restricted to a depth of 1 in the original R2 design. The results show that for symbolic regression problems, the GPGR method was able to find perfect solutions to all problems tested, including problems that the R2 method could not solve. GPGR performed significantly better than loops for all the symbolic regression problems. For classification problems, GPGR was able to perform consistently well across a wide range of image and non-image binary classification problems. GPGR was better than the majority of the benchmark loops for classification. The step-loop was the only loop than consistently outperformed GPGR on classification problems, which is because it is slightly simpler in structure than the recursive method. The step-loop was not as general as GPGR, as it was specific to classification problems, whereas GPGR is a generic method for symbolic regression, classification, and artificial ant problems. GPGR was also implemented for Artificial Ant problems, for which the performance was significantly better than the R2 design for the two harder trails, and still able to find many perfect solutions for the easiest trail.

The results of this work show that the new methods can evolve recursive functions to solve a wide range of different tasks. The performance of the new methods was often better than state of the art GP methods. The evolved programs were of a similar complexity to those of loops for the classification problems and simpler than loops for the Symbolic Regression tasks. The programs evolved for both the recursive methods and loops are much simpler than programs evolved using standard GP, as they can use iteration instead of having to repeat a subtree many times.

7.1 Future work

This project is still very early work on GP with recursion. The new methods work as well as if not better than state-of-the art loop methods for most of the tasks, but it would be beneficial to improve upon some of the results.

In the symbolic regression problems, some of the problems like Factorial + Sum and Binomial have poor results across all of the methods. It would be good to figure out exactly why this is, and create a recursive method to handle problems like this.

It would also be interesting to improve the methods for the Artificial Ant problems. This might involve adapting the recursive function, so it can be based on whether there is food ahead. It would also be interesting to try running the Artificial Ant problems for a longer number of generations to see if the new GPGR method could ever find perfect solutions to the more difficult Modified 2 and Santa Fe trails.

Finally, it is important to continue testing these recursive methods on more types of problems, so that a completely generic recursive approach can be found.

Bibliography

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[2] G. Chen and M. Zhang, "Evolving while-loop structures in genetic programming for factorial and ant problems," *AI 2005: Advances in Artificial Intelligence*, pp. 1079–1085, 2005.

[3] R. Sedgewick and K. Wayne, *Introduction to programming in Java: an interdisciplinary approach*. Addison-Wesley Publishing Company, 2007.

[4] F. Abdulhamid, S. Andy, K. Neshatian, and M. Zhang, "Evolving genetic programming classifiers with loop structures," *2012 IEEE Congress on Evolutionary Computation*, 2012.

[5] F. Abdulhamid, K. Neshatian, and M. Zhang, "Genetic programming for evolving programs with loop structures for classification tasks," *The 5th International Conference on Automation, Robotics and Applications*, 2011.

[6] J. Larres, M. Zhang, and W. N. Browne, "Using unrestricted loops in genetic programming for image classification," *IEEE Congress on Evolutionary Computation*, 2010.

[7] D. MCGAUGHRAN and M. ZHANG, "Evolving more representative programs with genetic programming," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 01, pp. 1–22, 2009.

[8] J. R. Koza, *Genetic programming III: Darwinian invention and problem solving*, vol. 3. Morgan Kaufmann, 1999.

[9] T. Yu and C. Clark, "Recursion, lambda-abstractions and genetic programming," *Cognitive Science Research Papers-University Of Birmingham CSRP*, pp. 26–30, 1998.

[10] B. Alexander and B. Zacher, *Parallel Problem Solving from Nature – PPSN XIII: 13th International Conference, Ljubljana, Slovenia, September 13-17, 2014. Proceedings*, ch. Boosting Search for Recursive Functions Using Partial Call-Trees, pp. 384–393. Cham: Springer International Publishing, 2014.

[11] A. Moraglio, F. E. Otero, C. G. Johnson, S. Thompson, and A. A. Freitas, "Evolving recursive programs using non-recursive scaffolding," in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pp. 1–8, IEEE, 2012.

[12] T. Phillips, M. Zhang, and B. Xue, "Genetic programming for evolving programs with recursive structures," *To appear in: IEEE Congress on Evolutionary Computation*, 2016.

[13] D. R. Barstow, "Domain-specific automatic programming," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1321–1336, Nov 1985.

[14] R. Balzer, "A 15 year perspective on automatic programming," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1257–1268, 11 1985. Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) Nov 1985; Last updated - 2011-07-20; CODEN - IESEDJ.

[15] C. Rich and R. C. Waters, "Automatic programming: Myths and prospects," *IEEE Computer*, vol. 21, no. 8, pp. 40–51, 1988.

[16] E. W. Dijkstra, "Recursive programming," *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960.

[17] J. R. Koza, *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

[18] J. Finlay and A. Dix, *An introduction to artificial intelligence*. Crc Press, 1996.

[19] W. B. Langdon and R. Poli, "Why ants are hard," 1998.

[20] W. B. Langdon and R. Poli, "Better trained ants for genetic programming," *SCHOOL OF COMPUTER SCIENCE RESEARCH REPORTS-UNIVERSITY OF BIRMINGHAM CSR*, 1998.

[21] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*, vol. 53. Springer, 2003.

[22] T. Bäck, D. B. Fogel, and Z. Michalewicz, *Evolutionary computation 1: basic algorithms and operators*, vol. 1. CRC Press, 2000.

[23] R. Poli and J. Koza, "Genetic programming," in *Search Methodologies*, pp. 143–185, Springer, 2014.

[24] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic programming: an introduction*. Morgan Kaufmann Publishers San Francisco, 1998.

[25] J. R. Koza, "Survey of genetic algorithms and genetic programming," in *Wescon Conference Record*, pp. 589–594, WESTERN PERIODICALS COMPANY, 1995.

[26] K. E. Kinnear, *Advances in genetic programming*, vol. 1. MIT press, 1994.

[27] J. R. Koza, "Genetic programming ii: Automatic discovery of reusable subprograms," *Cambridge, MA, USA*, 1994.

[28] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[29] X. Li, *Utilising restricted for-loops in genetic programming*. PhD thesis, School of Computer Science and Information Technology, Faculty of Applied Science, Royal Melbourne Institute of Technology, Melbourne, 2007.

[30] X. Li and V. Ciesielski, "An analysis of explicit loops in genetic programming," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 3, pp. 2522–2529, IEEE, 2005.

[31] V. Ciesielski and L. Xiang, "Experiments with explicit for-loops in genetic programming," *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, 2004.

[32] G. Darvas, "Fibonacci numbers in nature," *Symmetry: Cultural-historical and ontological aspects of science-arts relations The natural and man-made world in an interdisciplinary approach*, pp. 109–130, 2007.

[33] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu. com, 2008.

[34] S. Silva, J. Foster, M. Nicolau, P. Machado, and M. Giacobini, *Genetic Programming: 14th European Conference, EuroGP 2011, Torino, Italy, April 27-29, 2011, Proceedings*. LNCS sublibrary: Theoretical computer science and general issues, Springer, 2011.

[35] O. Lezoray, A. Elmoataz, and H. Cardot, "A color object recognition scheme: application to cellular sorting," *Machine Vision and Applications*, vol. 14, pp. 166–171, 2003.

[36] O. Lezoray, "Microscopical cell image database (serous cytology)." [Online]. Available: https://lezoray.users.greyc.fr/researchDatabasesSerousCells.php.

[37] Y. LeCum and C. Cortes, "The mnist database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[38] M. cbcl, "Face data," 2000. [Online]. Available: http://cbcl.mit.edu/software-datasets/FaceData2.html.

[39] S. Munder and D. M. Gavrila, "An experimental study on pedestrian classification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 11, pp. 1863–1868, 2006.

[40] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml.

[41] S. Hettich and S. D. Bay, "The uci kdd archive," 1999. [Online]. Available: http://kdd.ics.uci.edu.

Appendix A

Best Evolved programs for classification problems

This section shows the program trees as lisp expressions for selected classification problems for GPGR and noloops methods.

A.0.1 PokerHand

Figure A.1 show the best evolved program on the poker hand problem using GP with no loops. The training accuracy was 59%, testing accuracy was 54.4%

```
(+ (+ (+ 0.96 (input 9 )) (- (+ (+ (+ (+ (input 9 ) 3.4) (- (input 7 ) (input 0 )))
3.4) (- (+ 0.96 (input 9 )) (+ (input 8 ) (input 5 )))) (+ (input 8 ) (+ (input 9 )
3.4)))) (- (+ (+ (input 9 ) 3.4) (- (+ (+ (input 9 ) 3.4) (- (input 4 ) (input 3 )))
(+ (input 8 ) (input 5 )))) (+ (input 8 ) (input 5 ))))
```

Figure A.1: PokerHand Problem best evolved program with the No Loops method.

A.0.2 Cell

FigureA.2 shows the best evolved program for the Cell problem using the GPGR method. This program had This program had 97.6% training accuracy and 95.2% testing accuracy. FigureA.3 shows the best evolved program for the Cell problem using the GP with no loops method. This program had 96.6% training accuracy, and 95% testing accuracy.

```
(+ (+ (f = f (Sum f'(i - 1) i) (pixels 11 10 ) (- (f = f (Sum f'(i - 1) i) (+ (- 97 64)
(pixels 24, 7 )) 43) 15)) (+ (f = f (Sum f'(i - 2) i) (pixels 11 10 ) (pixels 1 6 )) (-
(f = f (Sum f'(i - 1) i) (pixels 8 13 ) (f = f (St.Dev f'(i - 2) i) 38 (- 97 64))) (+ (-
24 (pixels 9 15 )) (+ (pixels 7 10 ) (pixels 5 8 )))))) (- (f = f (Sum f'(i - 1) i) 13
35) (+ (f = f (Sum f'(i - 2) i) 38 26) (pixels 6 8 ))))
```

Figure A.2: Best program evolved by the GPGR method for the Cell image classification problem.

A.0.3 Face

FigureA.4 shows the best evolved program for the Face problem using the GPGR method. This program had 99.4% training accuracy and 96.8% testing accuracy. FigureA.5 shows

```
(+ (+ (+ (+ (pixels 7 3 ) (+ (+ (pixels 15 12 ) (- (pixels 13 2 ) (pixels 11 20 )))
(- 25.1 (- 65.2 (pixels 8 14 ))))) (pixels 2 11 )) (+ (+ (pixels 22 5 ) (pixels 1 12
)) (- (+ (pixels 12 1 ) (pixels 1 12 )) 72.9))) (+ (+ (+ (pixels 11 2 ) (+ (pixels
12 1 ) (pixels 10 15 ))) (+ (pixels 1 16 ) (+ (pixels 16 9 ) (+ (- (pixels 23 12 )
74.7) (pixels 1 19 ))))) (+ (+ (- (pixels 18 3 ) (- 65.2 (pixels 8 14 ))) (- (pixels
14 1 ) 50.5)) (- (+ (pixels 6 2 ) 36.2) (+ (pixels 15 5 ) (pixels 18 24 ))))))
```

Figure A.3: Best program evolved by the NoLoops method for the Cell image classification problem.

the best evolved program for the face problem using the GP with no loops method. This program had 98% training accuracy and 92.6% testing accuracy.

```
(+ (+ (- (f = f (St.Dev f(i - 1) i) (- (- 98 (- (pixels 15 6 ) (pixels 6 8 )))(f = f
(St.Dev f(i - 1) i) (pixels 0 5 ) (pixels 6 15 ))) (f = f (SumMode f(i - 1) i) (+ 51
(pixels 5 6 ))(- 77 (pixels 4 14 )))) (- (pixels 6 10 ) (pixels 17 2 ))) (+ (- (pixels
6 11 ) (pixels 0 6 ))(- (pixels 6 14 ) (- (- (pixels 2 8 ) 30) (f = f (St.Dev f(i - 1)
i) (pixels 0 5 ) (pixels 6 15 ))))))(- (+ (+ (f = f (St.Dev f(i - 2) i) (pixels 7 18 )
51) (- (- (pixels 0 3 ) (- (pixels 10 8 ) (pixels 5 5 ))) (pixels 9 6 ))) (pixels 4 12
)) (pixels 5 9 )))
```

Figure A.4: Best program evolved by the GPGR method for the face image classification problem.

```
(+ (+ (+ (- (- (pixels 13 2 ) (+ (- (pixels 8 6 ) (pixels 6 3 )) 31.7)) (+ (- (pixels
9 15 ) (+ (pixels 16 16 ) 12.5)) 31.7)) (+ (- (- (pixels 12 7 ) (pixels 1 11 )) (-
(pixels 11 2 ) 29.4)) (pixels 6 14 ))) (- (pixels 6 3 ) (+ (- (pixels 9 15 ) (+ (pixels
6 17 ) 10.2)) 29.4))) (+ (- (- (pixels 6 3 ) (- (pixels 9 15 ) (+ (- (pixels 6 14 )
(pixels 10 15 )) (pixels 2 18 )))) (- (pixels 11 2 ) 29.4)) (pixels 6 14 )))
```

Figure A.5: Best program evolved by the GP with no loops method for the face image classification problem.