Genetic Programming Using VGP

by Will Smart
smartwill@mcs.vuw.ac.nz
14 January, 2004

Contents

Chapter 1

Genetic Programming using VGP

1.1 Introduction

The intent of this document is to provide the reader with enough information to both use and extend the VGP package for experimentation. A moderate background in GP and comptence in ANSI C is assumed.

VGP is in the very earliest stages of development, and will change markedly by use and modification.

VGP was to built to be both light-weight and fast. Being written entirely in C means faster throughput, and being of such a small size means that the program is easily understood. Also VGP has some useful features for doing large amounts of research, the experimentation can be networked in a client/server fashion, and results collected into a binary database for extraction of the final data in any textual format.

1.2 Genetic Programming

This is a very quick run-down of GP as a process:

Genetic Programming is an evolutionary algorithm that performs a search for a *program* that can perform a task.

Programs are based on LISP-S expressions, and can be visualised as trees. Each program is made up of terminal nodes, which form leaves of the tree, and function nodes, which form the internal nodes. An example is shown in figure 1.1.

The fitness of a program is dependent on how well it can perform the task assigned. In the case of regression problems the fitness could be a MSE value between the desired function and the program. In the case of classification the fitness could be the accuracy of the program at predicting the class of a pattern.

In GP programs with better fitness are more likely to be selected to contribute to the next generation.

The first population of programs is generated randomly. Subsequent populations are made of the best of the previous population simply copied (elitism), programs altered

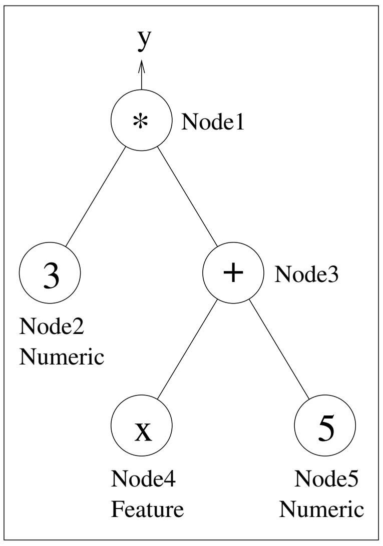

Figure 1.1: An example program encoding the expression $3(x+5)$

randomly and then copied (mutation), and programs made with material from two parents (crossover).

In general the fitness of the best individual in the population will increase from generation to generation.

For a thorough discussion of GP, the books by John Koza provide a useful backbone, and numerous papers are available online.

1.3 General Use of VGP

VGP is very flexible, and there is no guarantee that the method used in any specialized implementation is described here.

The base system of VGP currently performs one evolution per run, with options specified on the command line. For a list of options, run the program with any garbage as an option.

The base system is set up to perform classification of patterns, that are read in from a pattern file. The pattern file is of a (fairly contrived) format used for object classification, examples are given in the "`Classify/pats`" directory.

For example, to run the base system, first make it ("`make`" in a terminal), then type:

```
./vgp patfn:../Classify/pats/ce_4_pat.txt
```

in a terminal in `src`.

This should perform a run of GP, classifying feature vectors of 10 and 5 cent coins, using accuracy as the fitness. At the end the various accuracies will be printed, as well as the best program.

Running the program again produces the same results, however try:

```
./vgp patfn:../Classify/pats/ce_4_pat.txt seed:200
```

This produces different results. In fact, currently the seed (of the random number generator) is set by default to 100.

There are many options available to the user, even in the base system. To display them (and default values), get the help screen by typing some garbage like:

```
./vgp blah
```

Either all arguments are understood, or the run is halted and the help screen shown.

All arguments are of the format `<key>:<value>` and value can be an integer, floating point number, or string depending on the key. So far no multi-word strings have been required, so strings are without quotation marks.

In order to display more or less information about the run (eg. fitnesses of individual generations), use the verbose key (`v:<value>`). Smaller values print less.

Chapter 2

VGP Structure

Although VGP is very small in code-size, its structure is fairly complex. In this chapter the structure will be described, anyone who wishes to extend VGP should become familiar with the contents of this chapter.

The layout of the global functions used in the base system of VGP are shown in figure 2.1.

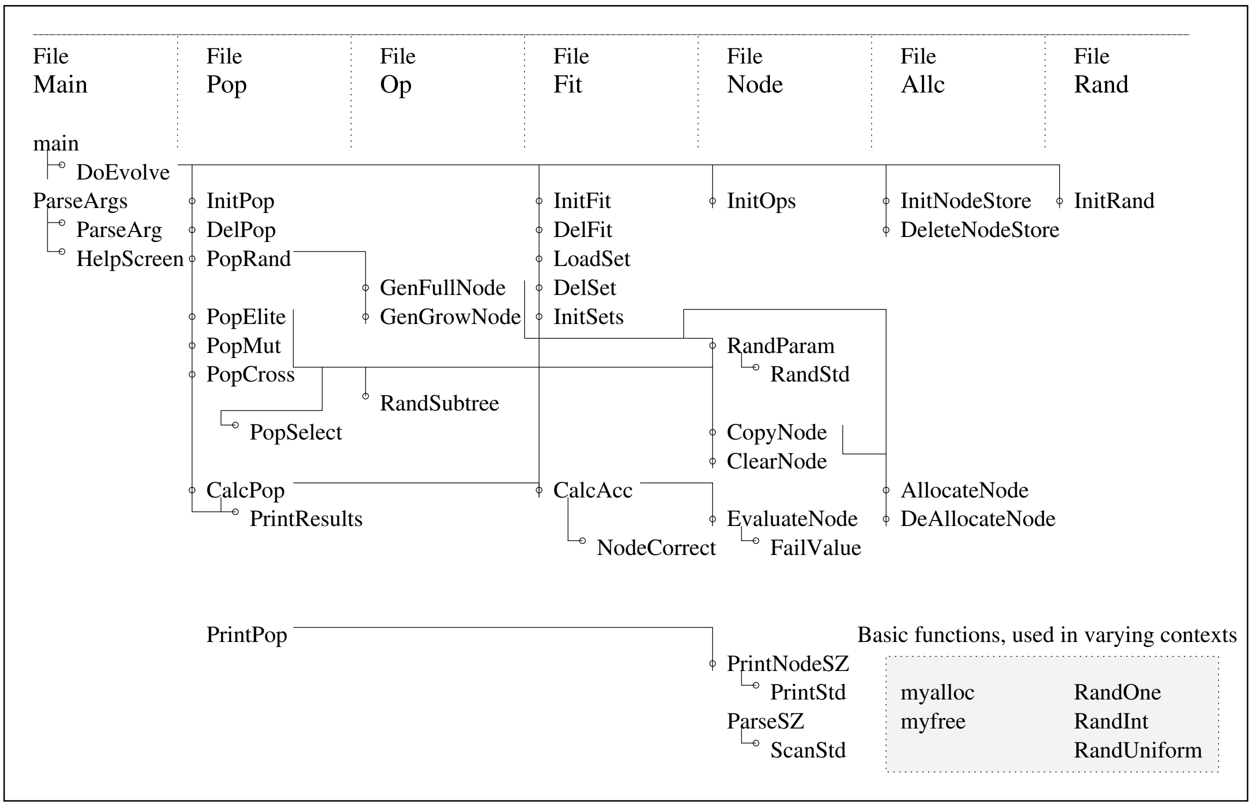

Figure 2.1: Layout of functions in the VGP base system.

2.1 Base System Files

VGP is written entirely in ANSI C, so there are no classes, but the files possess domains of utility as follows:

allc	: Allocation of nodes for use in programs, also the allocation of general memory, with new functions allowing memory leaks to be quickly exposed.
common.h	: Common settings amongst all files, also includes options in specialized versions via defines.
fit	: Operations on programs such as calculating accuracy and fitness, and dealing with sets.
main.c	: Parsing the arguments to the program, and performing the loop for evolution.
node	: Elementary operations on the nodes of programs. All functions and terminals share the same Node structure.
op	: Operations on programs such as generation and selection of nodes.
pop	: Operations on populations, such as selection, elitism, mutation and crossover. Also maintanance of populations.
rand	: Random number functions.

The following sections examine the fuctions more closely:

2.1.1 main.c

main.c contains five functions: A function called ParseArgs simply sets global variables from command-line like arguments. ParseArgs uses ParseArg anf HelpScreen, these will need to be remade if more options are to be parsed, for example in the Classify extension.

The main function performs initialization of the system, and does a single evolution loop. In the srcSock extension the main loop is remade to be far more functional. main uses DoEvolve, which does the actual evolution.

2.1.2 common.h

common.h is the header file common to all implementation files. In it there are defines for the specialized extensions, allowing the extensions to be included, and so control the basic types of the base sytem.

One global is declared, g_verbose, and macros for using it are also included.

2.1.3 pop.h, pop.c

In the pop files there are various functions for dealing with populations.

Structures

The Pop structure is declared, containing an array of programs, and fitnesses.

Variables

- g_minDepth and g_maxDepth alter the depths allowed in future modifications to, and generation of, programs.
- g_fullRate affects the proportion of programs generated by full and grow. It is between 0 and 1.
- g_numTourn affects the number of individuals used in tournament selection.

Functions

- InitPop and DelPop allow populations (Pop structures) to be allocated and deallocated.
- PopRand allows an initialized population to be filled with random programs. The programs are generated by either grow or full, with proportions determined by a global variable (g_fullRate). The rules of maximum and minimum depth are enforced by two global variables (g_maxDepth, g_minDepth).
- PopElite will take some number of the best (not just selected programs, but the actual best) programs in one population, and copy them to another population. Each program is only copied once.
- PopMut takes some number of programs selected from a population, and copies mutated copies to another population. For each selected program, a random node is found (so long as its not too deep), and a random subtree is generated from the node. The rules of maximum and minimum depth are enforced by two global variables (g_maxDepth, g_minDepth). The new subtrees are generated as in PopRand.
- PopCross takes some number of pairs of programs from a population, and copies crossed copies into another population. For each selected pair of programs, both have random nodes found (so long as the combination will pass depth tests), and the subtree at one of the nodes is copied to the other node. The new tree is copied to the other population. The rules of maximum and minimum depth are enforced by two global variables (g_maxDepth, g_minDepth).
- CalcPop does any work required to be done to a population in each generation. In the base system, this includes finding the fitness of the population, and printing using PrintResults.
- PrintResults prints results for either the end of the run, or some interrim generation.
- PrintPop simply prints the LISP S-expressions for all programs in the population.

2.1.4 op.h, op.c

In the op files are the complex operations on programs, quite seperate from populations.

Functions

- GenFullNode generates a program with only functions for some depth, then one layer of terminals to terminate the program's ends.
- GenGrowNode generates a program with all nodes randomly assigned as functions or terminals, so long as all depths conform to the arguments (eg not too shallow or deep).
- RandSubTree picks a node at random from all conforming nodes in a program. Conforming nodes are within bounds of depth in program and depth of subtree. The bounds are arguments to the function. This function also returns the depth in program and depth of subtree of the selected node.

2.1.5 fit.h, fit.c

In the fit files are the functions for dealing with sets (Set structure) and patterns (Pat structure).

Structures

- The Fit structure defines the fitness of a single program.
- The Pat structure defines all information required for a pattern.
- A Set is really just an array of Pat's, but allows easier access, and allocation/deallocation, by the main procedure.

Variables

- g_numCls and g_numFeat are set in LoadSet, and refer to the number of classes and features for classification.
- g_evolDone can be set to 1 in order to terminate evolution (it is checked in the main loop).
- g_slotSize affects the width of the slots used in SRS (classification strategy).

Functions

- InitFit and DelFit allocate and deallocate any structures to be used by fit. These are called for each evolution.
- LoadSet and DelSet allocate and deallocate Set structures. LoadSet in the base system loads a pattern file of a certain format, and is nearly always changed in extensions.
- InitSets takes an array of sets, and does something with them that cannot be done in LoadSet (individually). In the base system this scales the patterns' feature values to be within a range (-1 to 1).
- CalcAcc calculates the accuracy of a program on a set. This is used for classification, for the test and validation accuracies of the best programs.
- NodeCorrect determines whether a program correctly guesses the class of a pattern. It is called by CalcFit and CalcAcc.

2.1.6 node.h, node.c

In the node files are the functions used to perform simple operations on nodes and subtrees.

Structures

- The Feat structure defines the value of a feature.
- The FeatNI structure defines refers to a feature, and is used in feature nodes (NI = Node Information).
- The Node structure is used by all nodes to store their information.
- The Op structure is used by operations (primitive types) to store the functions and/or macros used to evaluate, print and parse nodes of that type. The macros are defines in node.h, and just speed evaluation for types that can fit in a macro.

Variables

The only globals are the operation array and count.

Functions

- InitOps initializes the operations (primitives) used in EvaluateNode, including the macros or functions used to evaluate a subtree. InitOps will need to be overridden if new functions or terminals are required. For an example of this see the SymRegression extension.

- RandParam is called by RandSubtree to allocate random parameters to a node (note that this matters only for nodes that have paramters, such as a value of feature number, most functions do not have such parameters). RandParam calls RandStd if it is used in InitOps.
- CopyNode duplicates a subtree (node) from one node to another (which is assumed to be an allocated terminal).
- ClearNode deallocates a subtree (node), reducing it to a single terminal (of operation number 0).
- EvaluateNode fills a subtree with the return values of each node (put into the value field) and returns the return value of the subtree. EvaluateNode depends heavily on the operations set in InitOps, and shouldn't need changing in most circumstances.
- PrintNodeSZ prints a subtree, as a LISP S-expression, to a character buffer. The functions used to print the nodes are set in InitOps, and standard operators use PrintStd.
- ParseSZ is intended to parse a string to produce a subtree. No use has been found for this yet, so it is not implemented.

2.1.7 allc.h, allc.c

The allc files contain the functions used to allocate and deallocate nodes and general memory.

Structures

NodeStore contains a single block of nodes.

Variables

g_totMalloc can be used to determine the total number of bytes of memory allocated by mymalloc, and not freed by myfree.

Functions

- InitNodeStore allocates memory in one big block, which is then used for node allocation. Currently only one block can be allocated, which sits on a global variable.
- DeleteNodeStore deallocates the memory used for the nodes.
- AllocNode and DeAllocNode (quickly) allocate and deallocate nodes, using the NodeStore initialized by InitNodeStore. The nodes allocated by AllocNode are initialized only in that they are set as terminals.

- mymalloc (declared in common.h) simply allocates memory using the c system function malloc, and has the same parameter. However myalloc also maintains a count of the total memory allocated using it (g_totMalloc).
- myfree (declared in common.h) calls free in the same way as mymalloc calls malloc. However, myfree requires an additional parameter, the block size, and subtracts this from the total allocated (g_totMalloc).

2.1.8 rand.h, rand.c

The rand files contain functions used to produce random numbers, including:

- InitRand sets the seed of the random number generator (a `long`).
- RandOne and RandUniform return random reals between [-1, 1) and [0, 1) respectively.
- RandInt returns a random integer between 0 and $n - 1$ inclusive, for some given n.

Chapter 3

VGP Specializations

3.1 How to Extend VGP

In order to do anything much with VGP, it must be extended. The base system will perform classification of object feature vectors using static range selection (SRS). However, if you want to (for instance) use a different file format or classification strategy, the system needs extending.

VGP is written entirely in C, and is designed to be very fast, so the method of extending the program is different to many other programs. The premise is that the extension be *no slower to run* than the original program. This is achieved by selective compilation of functions, so any function in the base system can be overridden by a new function. Along with the ability to add completely new functions, this satisfies the premise and allows the system to be extended in almost any direction.

The base setup of VGP serves as a set of functions, many of which will still be useful for the new task. The extension method is to replace the functions in the base that are incorrect for the new task with new functions, and to write any other procedures desired.

All procedures in the base can be masked to a different name using compiler defines. The defines are simply a 'D' then the name of the procedure. If the define exists, the base procedure is renamed as with a Base prefix, so the method to extend a file is to create the following file:

- Starts with any headers required by the new procedures.
- If desired one may define for the preprocessor `Headers`, which will supress the headers in the base file.
- Include the new procedures, with a define for each new procedure that is replacing a base one, of the form `D<proc_name>`. The base procedure is still available with a Base prefix.
- Include the base file using `#include "../src/<file>.c"`

The whole base system of VGP currently has only 54k of code. Emphasis has been placed of the inteligibility of what is going on in the base system, so as to make extension easier.

There are some example extensions of the system included in the package. These cover a wide range of requirements, and go to show that this method of extension is not difficult to use.

3.2 Extension Cookbook

This section is intended to give simple instructions to extend the base system in certain directions. The directions include:

- Changing the datatype returned by nodes to integer. See section 3.2.1.
- Add a terminal type. See section 3.2.2.
- Add a function type. See section 3.2.3.
- Remove a terminal/function type. See section 3.2.4.
- Add a test set. See section 3.2.5.
- Add a test and validation set. See section 3.2.6.
- Add a parameter to be parsed on the command line. See section 3.2.7.
- Change to regression. See section 3.2.8.

3.2.1 Changing the Datatype Returned by Nodes to integer

Looking at `node.h` in the `src` directory, the datatype is the Ret macro. To change it in an extension, have a `common.h` file in the extension, included by the `common.h` file in `src`. In the `common.h` file some macros must be defined (note that these will override the macros in the base system, see `node.h`). The macros to be defined are:

- `Ret`: define to the base type, eg `long`.
- `RetPrintString`: define to the formatting string used by printf, eg `"d"`.
- `RetParseString`: define to the formatting string used by scanf, eg `"d"`.
- `NOFEATSCALE`: define to anything. it just stops the features from being scaled to withn [-1,1).

Next, alter the way that numeric terminals are assigned, by making another `RandStd` procedure (called something else). In the procedure, set the value of the node passed to some random integer. Link the procedure to the numeric terminal operator by extending `InitOps` (in `node.c`) in a similar way to the removal of a terminal type, Example code is as follows:

```
#define DInitOps
void BaseInitOps(void);
void InitOps(void) {
  long l;
  BaseInitOps();
  for (l=0;l<g_numOps;l++) if (g_ops[l].type=='n') {
    g_ops[l].rand=<new_proc>;
    break;
  }
}
```

3.2.2 Adding a terminal type

In the `InitOps` procedure (`node.h`), the terminal type should be added as another operator.

A procedure should be written that prints the type and value of a node of the new type (see `PrintStd` in `node.c`). A procedure should be written that sets the value of a node of the new type randomly (see `RandStd` in `node.c`).

A procedure should be written that returns the value of a node of the new type (see `Div` in `node.c`). Alternately a macro can be set to evaluate the terminal. Find an unused macro number (look in `node.c` for the `EVALMACRO_<num>` macros. Set the macro in a way similar to the numeric and feature macros already made, but utilising the new terminal's properties.

Link the new terminal operator by extending `InitOps` (in `node.c`) as follows:

```
#define DInitOps
void BaseInitOps(void);
void InitOps(void) {
  long l;
  BaseInitOps();
  l=g_numOps;
  g_ops[l].rand=<new_proc>;
  g_ops[l].macro=<macro number or 0>;
  g_ops[l].type=<char_type>;
  g_ops[l].print=<PrintProc>;
  g_ops[l].funcN=<new_evaluate_proc>;
  l++;
  <any other functions/terminals>
  g_numOps=l;
}
```

If a macro is used, there is no need for the line setting the funcN property.

3.2.3 Adding a function type

For a concrete example, see the symbolic regression extension. In it the sin function is added.

In the InitOps procedure (node.h), the function type should be added as another operator.

A procedure should be written that returns the value of a node of the new type (see Div in node.c). Alternately a macro can be set to evaluate the terminal. Find an unused macro number (look in node.c for the EVALMACRO_<num> macros. Set the macro in a way similar to the addition macro already made, but utilising the new function's operation.

Link the new function operator by extending InitOps (in node.c) as follows:

```
#define DInitOps
void BaseInitOps(void);
void InitOps(void) {
  long l;
  BaseInitOps();
  l=g_numOps;
  g_ops[l].macro=<macro number or 0>;
  g_ops[l].type=<char_type>;
  g_ops[l].nargs=<num_args>;
  g_ops[l].print=<PrintProc>;
  g_ops[l].funcN or func2 or func3=<new_evaluate_proc>;
  l++;
  <any other functions/terminals>
  g_numOps=l;
}
```

If a macro is used, there is no need for the line setting a function property.

3.2.4 Removing a terminal or function type

For a concrete example, see the XOR extension. In it the protected division (%) function is removed.

Extend the InitOps procedure as follows:

```
#define DInitOps
void BaseInitOps(void);
void InitOps(void) {
  long l;
  BaseInitOps();
  for (l=0;l<g_numOps;l++) if (g_ops[l].type=='<char_type>') {
    g_ops[l].nargs=-1; // this means it will never be assigned
    break;
  }
}
```

where `<char_type>` is the character assigned in the base's `InitOps`.

3.2.5 Adding a Test Set

In order to add a test set some extension files must be written to augment the base. The following sections describe the changes.

Changes to `fit.c`

The new set must be declared at the top of the extension `fit.c` file. The base file declares `mainSet`, however for the extension it might be easier to declare two new sets `trainSet` and `testSet`.

Each set is given a character that is used to refer to it. This is defined in a macro called `SETFROMCHAR`. The base simply ignores the character, assuming it refers to the `mainSet` (the macro evaluates to a reference to `mainSet`). To add another set try:

```
#define SETFROMCHAR(ch) ((ch=='s')?&testSet:&trainSet)
```

This uses the character 's' to refer to the test set, and anything else refers to the training set.

The next change in `fit.c` is to `InitFit`. This looks very much like the base InitFit, but loads two sets instead of one. The base `LoadSet` takes as an argument a string that describes the layout of the sets in the file. This could be set at the command line (see the SimpleClassify example) or could be hard coded to "rs" to divide the pattern file in half (first half training, second half test).

The final change to `fit.c` is to cleanup the sets in `DelFit`.

Changes to `pop.c`

The only change to `pop.c` is to print the test accuracy at the end of the run. This is done by augmenting `PrintResults`. Put the printing before the calling the base procedure, and only print if verbose is sufficient (use POUT(3)) and the evolution is done (unless test accuracy is desired for each generation). See `SimpleClassify/pop.c` for some examples (note that it also uses a validation set).

3.2.6 Adding a Test and Validation Set

In order to add a test set some extension files must be written to augment the base. See the `SimpleClassify` files for a concrete example.

The changes to `fit.c` are similar to those in the previous section for adding a test set, just add a validation set as well as the test set. Also the string handed to `LoadSet` for the layout of the sets in the pattern file may be set on the command line. To do this see section 3.2.7.

The changes to `pop.c` include changing `CalcPop` to look for a peak in the validation set accuracy, taking the test set accuracy when it happens. `PrintResults` prints this new information at the end of evolution.

3.2.7 Adding a Command Line Parameter

In order to add parameters to be parsed on the command line, two procedures in `main.c` must be augmented.

`ParseArg` should be made as follows:

```
#define DParseArg
long BaseParseArg(char *arg);
long ParseArg(char *arg) {
  if (STRSTARTSWITH(arg, "<key>:")) strcpy(<string_arg>,arg + <length_of_key>);
  if (STRSTARTSWITH(arg, "<key>:")) <long_arg> = atoi(arg + <length_of_key>);
  if (STRSTARTSWITH(arg, "<key>:")) <double_arg> = atof(arg + <length_of_key>);
  else if (!BaseParseArg(arg)) return(0);
  return(1);
}
```

where `<key>` is a string like `rate:` of length `<length_of_key>`. `<string_arg>` is a string to be set. `<long_arg>` is a long to be set. `<double_arg>` is a double to be set. These variables need to be declared using `extern` so the procedure can find them (or they could be defined in `main.c`).

`HelpScreen` is a procedure that is called if a parameter is not understood. It should be augmented as follows:

```
#define DHelpScreen
void BaseHelpScreen();
void HelpScreen() {
  BaseHelpScreen();
  if (POUT(1)) fprintf(stderr,"    <key>:<format>          <description> (default : <d
}
```

3.2.8 Changing to Regression

To change from classification to regression atleast four files should be written as follows:

`common.h`

In the extension `common.h` (which should be included in the base `common.h` via a define, as for the other extensions), the following defines should be defined:

```
#define FITBETTER(fa,fb) (fa.fitness<fb.fitness)
#define CLEARFIT(fa) fa.fitness=<very_high_error>;
#define BESTFIT 0.0
#define AIMMSE <stop_when_mse_reaches_this>
```

These change the fitness from accuracy to error.

Changes to `fit.c`

Two functions are changed in `fit.c`. CalcFit is made to calculate the MSE between the program and the target function (see `SymbolicRegression/fit.c`). LoadSet is made to simply ignore the call, setting the set to empty and setting the number of features (see `SymbolicRegression/fit.c`).

Changes to `pop.c`

In `pop.c` the `CalcPop` procedure is remade to better suit regression, by ending evolution when the error gets small enough, and clipping the fitness to a finite range (see `SymbolicRegression/pop.c`).

3.3 Example Specializations

3.3.1 XOR

In the XOR directory. Type “`make`” to make.

To run the program type “`../vgp`”.

This example uses classification to solve the XOR problem, that is to create a program that can discern between two boolean features being the same or different. As such there are two classes, and four patterns.

In order to extend the base to do this, three files were used:

- node.c: One function was changed. RandStd is the function which randomly assigns internal parameters to a node. The change was to the distribution of the numeric terminal, it is just set to 1.0, instead of ±1.0 as in the base.
- fit.c: Two functions changed. NodeCorrect was changed, as in the base system a return value equal to a boundary is assigned the class above the boundary. For the purposes of XOR it is nicer if 0 is classified as class 0 (below the boundary). LoadSet was changed to hard code the patterns (though this is not strictly neccesary if we made a pattern file).

- common.h: This file is used to alter the basic types of the base system, and is included at the top of all base files (via the common.h in the base). The change here is profound, from an floating-point node return type to integer.

3.3.2 Symbolic Regression

In the SymRegression directory. Type "`make`" to make.

To run the program type "`../vgp`".

This example uses regression to approximate a simple function.

The function is contained in a macro at the top of `fit.c`.

In order to extend the base to do this, three files were used:

- node.c: One function was changed, and one added. InitOps is the function which forms the table of operations (primitives) used as nodes. The function is similar to the base function, but has another primitive, sin, added. The added function simply prints the word "sin" to a string, and is referenced in the InitOps.
- fit.c: CalcFit was added to perform a MSE calculation for the fitness of the program. LoadSet was changed to operate without a pattern file (obviously none is needed).
- pop.c: The CalcPop procedure was modified to suit regression.
- common.h: This file is used to alter the basic types of the base system, and is included at the top of all base files (via the common.h in the base). Here the direction of fitness is changed, macros are changed to make 0 the best fitness, and greater fitnesses worse.

3.3.3 Classification

In the Classify directory. Type "`make`" to make.

To run the program type "`../vgp patfn:pats/<pattern file>`".

This example has many options for object classification. These options are available on the command line (type "`./vgp h`" to list the options.

Current new methods supported by Classify include

- Classification strategies including SRS, CDRS, and SDRS (option "`clstype:`")
- Online error propagation for the SRS, CDRS and SDRS strategies (option "`rate:`")
- Use of CBD for classification (option "`clstype:com`")

Many base functions and types have been changed for use in Classify, including the following:

- common.h: The Node structure has the gradient added to it, allowing fast gradient-descent. The Op structure has fields added to it, allowing for calculation of derivitives of the various functions, these are set in node.c.

- fit.c: Functions are changed in three directions: To allow gradient-descent (such as IdealVal, PropNode). To perform communal binary decomposition (such as CalcComFit, CalcAcc, GetNorms, InitFit, DelFit). To perform the new classification strategies (such as ReClassify, NodeCorrect). To add test and validation sets.
- main.c: Additional parameters are parsed in ParseArgs.
- node.c: An additional function calculates the derivative of the program output with respect to the numeric terminals' values, used for gradient-descent. Additional functions are referenced in the InitOps function, allowing the calculation of the derivitives of functions.
- pop.c: CalcPop is augmented to perform a gradient-descent step, and deal with three sets.
- stats: Header and implementation files, giving statistical functions for the communal binary decomposition technique.

This example shows an extremely extended system, able to produce results on some very interesting GP developments.

As for the base system, type garbage as an argument in order to display the help screen.

The makefile for Classify actually makes two programs, vgp and vgps. vgps is a socketted version of the same program, for use with the srcSock extension.

3.3.4 srcSock : Networking

In the Classify directory. Type "`make`" to make all the required programs.

This extension goes beyond the simple, command-line argument, interface that the base system provides.

The same functionality is provided remotely using a kind of client-server relationship.

The makefile will produce five programs, and in the Classify directory vgps is made:

- vgps: This is simply a socket enabled version of the Classiy extension. This is done using the main.c and pop.c files in the srcSock directory to further extend the system. The main procedure is replaced with a loop to continually receive instructions from a socket (GetInstructions) and perform a run of the GP system using the parameters set (DoEvolve).

  Results are sent back over the remote link, dependent of the third decimal place of the verbose value. i.e. `v:501` displays nothing on the console, but provides full results over the link.

  The program starts by appending a file in the srcSock directory with the host name and port on which it can be contacted. This is a simple text file called "machines.txt", and is intended to be manually altered as workers are dismissed.

- vgpcntrl: This is the other end of the remote link, and is intended to run on the user's local PC (though this is not neccesary). This program refers to a single worker server, and keeps it fed with experiments to do.

  In order to run vgpcntrl, type "`./vgpcntrl <db> <hostname>:<index>`", where `<db>` is the prefix of the experiment database file being used, (see the use of the addplan program for more about database files). `<hostname>` is the name of the host, as written in machines.txt (may be just the first word). `<index>` is the index of the entry in the machines.txt file. i.e. to use the third entry of "`cuba.mcs.vuw.ac.nz`" in the machines.txt file, type:

  ```
  ./vgpcntrl <db> cuba:3
  ```

  To use the first entry, leave off the "`:3`" altogether.

- addplan: This allows experimental databases to be made or added to.

  To run the program type:

  ```
  ./addplan [new] <db> <planfile>
  ```

  The "`new`" command is optional, and indicates the database should be cleared, or created. "`<db>`" is the prefix of the files used for the database. "`<planfile>`" is the plan file to use in adding the experiments. Using a planfile allows experiments involving several variables to be exhaustively searched, with minimal instructions. Some examples are included, such as "`plan_fst.txt`". The exact format is left to an appendix.

  The database system used in srcSock involves to data files: The experiment file contains text listing all the experiments planned. The first character of each line can be an 'e' to indicate the experiment is yet to be done, a '+' indicating the experiment is being done, or a '*', indicating the experiment has been done. If the experiment has been done, two numbers follow the '*', the first indicates the offset of the output data for the experiment in the binary file, the second is the length of the block.

  The binary file is thus built upon as experiments are done, with the result data of the experiments. The data is stored nearly exactly as it is recieved from the worker running vgps, so the best place to look for the exact format of the binary file is main.c in srcSock. The format is not examined here.

  The files are protected by a semaphore, so that access by multiple programs at once is not a problem (for example, more than one vgpcntrl can write to the same database. To keep this semaphore intact, the programs that write to the database files should not be halted using ctrl-c, doing so imposes the risk that if the program held the semaphore, all other programs will have to wait forever. As such, to quit the vgpcntrl program, type `q<enter>`.

- resetsem: resetsem is to be used if this semaphore is abused, such as if a program is halted while it held the semaphore. It sets the semaphore value to 1, and should be used only when no programs are using the database, or when programs using the database are being starved (i.e. there is clearly a problem with the semaphore).

  To use the program type:

  ```
  ./resetsem <db>
  ```

  Where "`<db>`" is the database prefix as used elsewhere.

- removesem: removesem is to be used if a semaphore is to be removed, this is easiest before the database files are removed. If this is the case, just use the program as for `resetsem`.

  If the semaphore is not found (such as if the file is not there), run "`ipcs -s`" to display the current semaphores, and remove using:

  ```
  ./removesem i<id>
  ```

  Where "`<id>`" is the id value printed for the semaphore (note that removing a semaphore for an existing database should not cause a problem, unless it is being used by a program at the time).

- ext: ext is a program to be used to extract average results from the results database. The format of the output is very flexible using text files for formatting. An example is tab.txt. The exact format is left to an appendix.

- exttab: exttab is a program to be used to extract average results from the results database. The output is made into a latex table, using a template file. The exact format of the template is left to an appendix.

Appendix A

Plan File Format

Plan files are used by addplan to exhaustively produce experiments in the experimental database.

An example is “plan_fst.txt”, which explores some mutation rates, and elitism rates. The lines starting with “`#˜` ” indicate possible values for variables. For example the line:

“`#˜ dataset:s1 patfn:../../se_3_021_pat.txt`”

indicates a variable called “`dataset:s1`” with the value “`patfn:../../se_3_021_pat.txt`”

The lines starting with “`#˜` ” indicate experiments to be carried out.

So the line:

“`##stdcom dataset:  mut:  elite:  seed:`”

exhaustively tries all combinations of variables that begin with the words `stdcom`, `dataset:`, `mut:`, `elite:` and `seed:`.

In all this line will produce 450 experiments (`dataset:` can be 5 values, `mut:` can be 3 values, `elite:` can be 3 values, and `seed:` can be 10 values. $5 \times 3 \times 3 \times 10 = 450$).

Other options are available for contructing names, built from the different parameters, for use as the values. For information on how to use these I suggest you contact the author, or look at the code.

Appendix B

Experimental-output Format File Format

Format files are used by ext to output values of experiments. Lines can be of the following formats:
`printns <str>` : prints a string.
`print <str>` : prints a string with a trailing space.
`new line` : prints a new line
`exp <str>` : Set the current experiment string, the string may contain variables. To include a variable us a string like `%1%` for variable 1. there are 10 variables, one per digit.
`var <digit> <str>` : set a variable string, expanded in the experiment string
`string for print <str>` : The string used by printf for floating point numbers (default is `%f`
`final time`, `final test acc %`, `final gen`, : print data, from the final generation, averaged over all experiments (that are complete) that fit the experiment string.
`best val %`, `time at best val`, `test acc at best val %`, `gen at best val`, : print data, from the generation that gave best validation set accuracy, averaged from all experiments (that are complete) that fit the experiment string.
`total number` : prints the total number of experiments referenced so far.

This list is bound to change markedly over time.

Appendix C

Tabular Template File Format

Template files are used by ext to output values of experiments using exttab.

Look to the example template files (such as `db1_tab_rate.txt`) for the exact format of template files. In short:

A table environment is started by using the keyword `table` on a line, followed by the label and caption of the table.

The character '|' is used to delimit table entries, and may be followed immediately by a digit to indicate multiple column entries.

The character after the '|' or digit may be '_' to indicate that the entry is to be underlined. A double '_' after the last '|' indicates to double-underline the row.

In an entry: Any text inside '%' characters is used for variables. A single digit, or a digit followed by an '=' indicates the variable to change. Any text after the '=', or text that contains something other than a digit followed by a '=', indicates the value to change it to. Variables not set somewhere in the current column, are set to the most recent set value (i.e. in the current or previous row(s)).

Any text inside '~' characters is used for commands. Commands include: bestgens, besttime, besttest, finalgens, finaltime and finaltest. The command may be prefixed with a digit to indicate the experiment string to use. The commands will perform a search in the database for any experiments matching the experiment string, returning the appropriate averages. The command may be appended with a number in paretheses, in which case a warning is printed if the number of experiments retrieved is different from the number appended. In order to simply perform the same command as performed most recently in the same column, type two '~' characters together. Only one command is allowed per table entry.

All other text is simply printed. In order to print some special character use a '\' prefix.

Appendix D

Examples

There are currently two sets of example files, one for a quick introduction, and one that performs a far more lengthy series of experiments.

D.1 Example: fst

fst is the simple example, that produces a table in about three minutes. Instructions follow:

In a terminal in the `Classify` directory, type:

```
rm ../srcSock/machines.txt
./vgps
```

to start a worker thread.

In a terminal in the `srcSock` directory, type:

```
./addplan new dbs/fst toolfiles/plan_fst_rate.txt
```

to make an experiment database with 50 experiments. Have a look in the plan and experiment files to see what this did.

Type `./vgpcntrl dbs/fst <first name of machine as in machines.txt>`

to run a client thread, attached to the worker. Experiments will appear on the terminal, as they are done. In about three minutes all the experiments should be done.

Type

```
./exttab dbs/fst toolfiles/tab_fst_rate.txt
```

to display the latex table (included as table D.1).

D.2 Example: db1

db1 is a far more involved example, including 12500 experiments, with the advanced features of the Classify extension.

In order to build the experiment database, type:

```
./addplan check:50 new dbs/db1 toolfiles/plan_db1.txt
```

This will build fill the database files with yet to be done experiments. The `check:50` argument causes the program to check for duplicate experiments (every 50th experiment),

Table D.1: Results of using error propagation on the shape datasets using SRS as the classification strategy.

Dataset	Classes	Prop. Rate	Generations		Time (s)		Accuracy	
			Bio off	Bio on	Bio off	Bio on	Bio off	Bio on
Shape	4	off	13.20	—	6.85	—	100.00 %	—
		0.4	0.20	0.20	1.23	1.22	100.00 %	100.00 %
		1.0	0.20	0.00	1.23	0.99	99.88 %	99.81 %
Coin	5	off	27.60	—	9.15	—	97.50 %	—
		0.4	6.60	4.00	5.56	3.39	99.00 %	100.00 %
		1.0	5.00	4.40	4.55	3.72	99.25 %	98.50 %

which are not included more than once. The output of this command should be several lines of dots, and an indication that there were 12500 experiments added (note that without the `check:50` argument this would be 15000, check the plan file to see why).

In order to run the experiments, do something along the lines of the previous example. However, due to the large number of experiments, run some number of machines in parallel. To do this: clear the machines.txt file, start a worker on each machine, start a terminal on the home machine for each worker and start vgpcntrl as before, with the name of a different worker in each. They can all write to the same database. The experiments took me about six hours to do using four machines.

The work is already done however. To display some of the data in the database, type:

```
./exttab dbs/db1good toolfiles/tab_db1.txt
```

which displays some latex tables (included as tables D.2, D.3, D.4 and D.5).

Table D.2: Results of using error propagation on the shape datasets using SRS as the classification strategy.

Dataset	Classes	Prop. Rate	Generations		Time (s)		Accuracy (%)	
			Bio off	Bio on	Bio off	Bio on	Bio off	Bio on
Shape	3	off	2.58	—	0.89	—	99.88	—
		0.2	0.00	0.00	0.19	0.20	99.93	99.95
		0.4	0.00	0.00	0.20	0.20	99.94	99.99
		0.7	0.00	0.00	0.20	0.19	99.88	99.88
		1.0	0.00	0.00	0.19	0.19	99.88	99.88
		1.4	0.00	0.00	0.20	0.19	99.93	99.91
	4	off	14.38	—	6.09	—	99.66	—
		0.2	0.12	0.12	0.39	0.40	99.94	99.92
		0.4	0.14	0.08	0.42	0.35	99.94	99.91
		0.7	0.06	0.14	0.34	0.41	99.93	99.93
		1.0	0.10	0.06	0.37	0.33	99.91	99.92
		1.4	0.16	0.24	0.44	0.54	99.90	99.87
Coin	4	off	20.26	—	3.94	—	98.17	—
		0.2	1.38	1.76	0.75	0.97	99.48	99.31
		0.4	1.02	1.36	0.60	0.79	99.38	99.45
		0.7	1.14	1.52	0.69	0.88	98.89	98.97
		1.0	1.38	1.70	0.81	0.96	98.92	99.05
		1.4	2.30	2.60	1.32	1.47	99.30	99.05
	5	off	28.34	—	7.80	—	97.79	—
		0.2	4.04	6.42	3.03	4.97	99.05	99.46
		0.4	4.38	5.12	3.13	3.70	99.21	99.52
		0.7	4.28	5.20	3.33	4.75	98.91	99.20
		1.0	4.40	5.54	3.43	4.36	99.12	99.15
		1.4	6.60	7.08	5.87	5.85	98.96	99.04
Coin (hard)	5	off	32.40	—	3.44	—	82.53	—
		0.2	16.28	16.00	5.62	5.05	89.58	87.17
		0.4	14.56	14.62	4.68	4.60	88.75	88.91
		0.7	10.96	15.50	3.39	4.93	89.43	90.64
		1.0	12.68	14.74	4.01	4.48	90.64	90.72
		1.4	18.82	17.32	6.48	6.29	89.40	89.96

Table D.3: Results of using error propagation on the shape datasets using SDRS as the classification strategy.

Dataset	Classes	Prop. Rate	Generations		Time (s)		Accuracy (%)	
			Bio off	Bio on	Bio off	Bio on	Bio off	Bio on
Shape	3	off	1.02	—	0.52	—	99.91	—
		0.2	0.02	0.02	0.21	0.22	99.84	99.94
		0.4	0.00	0.00	0.20	0.20	99.93	99.88
		0.7	0.00	0.00	0.20	0.20	99.77	99.89
		1.0	0.00	0.00	0.20	0.20	99.90	99.94
		1.4	0.00	0.02	0.20	0.21	99.84	99.83
	4	off	3.90	—	2.68	—	99.88	—
		0.2	0.52	0.84	0.83	1.28	99.81	99.90
		0.4	0.76	0.44	1.17	0.78	99.86	99.80
		0.7	0.26	0.58	0.59	0.93	99.83	99.87
		1.0	0.56	0.74	0.90	1.15	99.83	99.88
		1.4	0.38	0.90	0.70	1.30	99.84	99.83
Coin	4	off	13.64	—	4.44	—	97.69	—
		0.2	7.66	8.96	5.23	6.02	98.56	98.88
		0.4	6.20	9.46	4.07	6.81	98.84	98.72
		0.7	6.14	7.90	4.26	5.51	98.97	99.08
		1.0	5.96	7.68	3.95	5.39	99.08	98.88
		1.4	5.92	7.34	3.94	5.12	98.55	99.16
	5	off	20.06	—	8.04	—	98.42	—
		0.2	12.80	13.22	12.34	12.46	98.92	99.05
		0.4	9.12	10.98	8.93	10.04	98.74	98.82
		0.7	8.94	10.76	8.25	9.95	98.88	98.87
		1.0	11.56	10.44	11.45	10.19	98.66	98.85
		1.4	9.98	9.86	10.14	10.35	99.13	98.90
Coin (hard)	5	off	17.98	—	2.75	—	84.23	—
		0.2	14.54	18.88	4.46	6.34	90.87	89.51
		0.4	10.22	18.38	3.22	6.62	90.19	88.34
		0.7	12.90	16.20	4.27	5.35	90.38	90.64
		1.0	12.16	15.56	4.37	5.41	90.19	89.74
		1.4	10.24	12.10	3.60	4.18	89.51	90.15

Table D.4: Results of using error propagation on the shape datasets using CDRS as the classification strategy.

Dataset	Classes	Prop. Rate	Generations		Time (s)		Accuracy (%)	
			Bio off	Bio on	Bio off	Bio on	Bio off	Bio on
Shape	3	off	0.48	—	0.34	—	99.93	—
		0.2	0.00	0.00	0.20	0.20	99.90	99.91
		0.4	0.00	0.00	0.19	0.19	99.84	99.96
		0.7	0.00	0.00	0.19	0.19	99.88	99.91
		1.0	0.00	0.00	0.19	0.19	99.96	99.99
		1.4	0.00	0.00	0.19	0.19	99.90	99.87
	4	off	6.48	—	4.44	—	99.79	—
		0.2	0.06	0.12	0.32	0.39	99.90	99.91
		0.4	0.02	0.08	0.29	0.36	99.91	99.92
		0.7	0.08	0.06	0.35	0.32	99.86	99.89
		1.0	0.06	0.04	0.32	0.31	99.91	99.82
		1.4	0.04	0.02	0.30	0.28	99.87	99.91
Coin	4	off	13.40	—	4.46	—	98.30	—
		0.2	1.48	2.60	0.94	1.76	99.08	98.97
		0.4	1.64	2.36	1.04	1.50	98.94	99.16
		0.7	1.58	2.26	1.06	1.52	98.66	99.08
		1.0	1.58	1.58	1.00	1.03	98.83	98.95
		1.4	2.32	2.50	1.62	1.79	98.89	98.94
	5	off	24.88	—	11.71	—	98.49	—
		0.2	7.58	5.76	7.83	5.20	98.85	99.09
		0.4	4.72	6.02	4.60	5.66	98.85	99.04
		0.7	4.78	5.66	4.49	5.49	98.79	99.02
		1.0	4.14	5.74	3.99	5.54	98.86	98.97
		1.4	5.50	5.50	5.50	5.81	98.84	98.93
Coin (hard)	5	off	23.72	—	4.03	—	88.19	—
		0.2	8.42	12.26	2.83	4.53	91.09	90.53
		0.4	7.36	8.46	2.45	3.03	91.96	89.62
		0.7	6.10	7.50	1.91	2.46	90.91	91.62
		1.0	6.64	9.32	2.20	3.16	90.00	89.85
		1.4	6.96	9.56	2.35	3.14	90.26	90.08

Table D.5: Results of different classification strategies on all datasets, without error propagation.

Dataset	Classes	Time (s)			Accuracy (%)		
		SRS	SDRS	CBD	SRS	SDRS	CBD
Shape	3	0.89	0.52	0.50	99.88	99.91	99.98
	4	6.09	2.68	0.70	99.66	99.88	99.90
Coin	4	3.94	4.44	0.31	98.17	97.69	99.53
	5(easy)	7.80	8.04	0.41	97.79	98.42	99.14
	5(hard)	3.44	2.75	0.22	82.53	84.23	96.30