Abstract
Widespread adoption of Genetic Programming techniques as a domain-independent problem solving tool depends on a good underlying software structure. A system is presented that mirrors the conceptual make-up of a GP system. Consisting of a loose collection of software components, each with strict interface definitions and roles, the system maximises flexibility and minimises effort when applied to a new problem domain.

Category: Genetic Programming

Introduction
Genetic Programming [Koza] systems play an important role in determining the usefulness of GP as a method of automatic programming. A proper software support structure is essential if GP is ever to become a viable alternative to manual methods in problem solving. This paper discusses GP-COM, a system developed by the author. The system includes several features to facilitate the process of solving a particular problem using genetic programming. This document is intended to highlight issues of software construction and portability in GP systems, and to propose a framework in which new techniques and methods might be incorporated into a system with minimum disruption. Such a framework might facilitate widespread uptake of these new techniques.

Our genetic programming system operates at three levels. Firstly, domain-independent software, which remains unchanged regardless of the experiment. Problem-specific software changes according to the problem domain to which GP is being applied. Run-specific software is different for each experimental run of a specific problem. This distinction is a hierarchical one: global software is used for a number of problems, each of which has a number of experimental runs active.

GP software is unusual in that it is applied to so many different problem areas. It demands the inter-operation of stable and volatile software units as the experimenter moves from one problem domain to another. The structure of the system should be such as to allow these units to cooperate with the minimum of effort.

Structure of a GP System
Genetic Programming is a cyclical process, based on the evaluation and modification of populations of potential solutions. Each component of this process implements a specific
function in the operation of the system as a whole. In a given implementation several components may be combined into a single unit, and in extreme cases the whole system may be integrated, but the GP cycle can conceptually be broken down into the following steps.

Problem-dependent components
To apply GP to a particular problem, a number of components need to be defined that are unique to the problem domain. These components constitute the definition of that particular problem, and determine the performance of the GP system with respect to that problem.

- Problem domain definition and implementation as code. This is software that models the problem in the digital domain and produces the results of applying the candidate programs to the problem data. This component requires a large amount of effort in terms of setting up an appropriate problem representation.

- Raw fitness calculation. Produces some scalar value indicating the performance of each candidate on the given problem.

- Configuration. Problem-specific and run-specific settings for the system as a whole. This includes things like the relative frequency of genetic operators to be applied to the population, selection scheme to be used, population size etc.

Problem-independent components
These components are pieces of software that do not need to be changed, regardless of which problem the system is being applied to, or which run is active. These components are the global software mentioned in the introduction.

- Population manager. This component stores candidate programs and renders them in a form suitable for execution. It also performs the necessary operations to produce a new population
from the old population. Finally this component provides permanent storage facilities, and retrieval, for recording populations.

- Standard fitness calculation. This component produces standard fitness scores from raw fitness scores. Depending on configuration, this may involve calculating the scores, or simply copying them from raw fitness.
- Adjusted fitness calculation and normalised fitness calculation. A two step process that calculates normalised fitness values from standard fitness, via adjusted fitness.
- Termination criteria. A test to decide whether a particular run has finished or not. This can be a generation count, or something based on fitness values. This component can often be drawn from a stock of standard methods.

**Structure of GP-COM**

GP-COM is a component-based system the structure of which mirrors closely the conceptual divide between the domain-dependent and domain-independent components. It was designed with the intention of minimising the amount of effort required to encode a new problem and solve it using GP. GP-COM consists of a suite of programs glued together by higher-level scripts. Each component of the generic system described earlier is represented by one or more independent programs. These programs are controlled by scripts which handle inter-process communication and sequencing of events, to implement the GP cycle. Creating a new problem involves creating new versions of those elements that are problem specific.

**The population manager**

This component is the largest in GP-COM. Its role is to maintain and manipulate populations of genetic program trees. This involves permanent storage of the trees for logging purposes and volatile storage for manipulation. It provides facilities for creating a new population from an old, by duplicating populations, moving trees from one population to another and applying genetic operators to trees. The manager is not tied to a specific function and terminal set — this information is provided at run-time.

This program is independent of the problem, and of the techniques selected for that problem. By implementing low-level operations in the population manager, high-level functions can be implemented through structured application of these operations. Consequently the population manager need not incorporate selection methods directly, nor need it implement the genetic operators itself. These roles can be taken by external programs which pass streams of commands to the population manager. A selection method, for example, need only know normalised fitness values and the population size. It is then able to generate commands to selectively copy members of an old population into a new, ‘blank’ one. Similarly, a genetic operator could take one or two individuals as input and output the offspring produced by the operator.

The population manager in GP-COM is capable of storing and manipulating arbitrary program trees. It provides checks for syntactic integrity according to information specified at run-time. It supports strongly typed primitives [Montana] and hierarchical ADFs (Automatically Defined Functions). It also provides support for using both together, i.e. it is possible to have typed ADFs. Although this is a novel feature, it has yet to be shown that the combination of these features is of value in practice, although both ADFs and strong typing have been shown to offer
improvements when used in isolation [Koza2, Haynes]. To retain the integrity of crossover and mutation operators with strong typing and ADFs, the following constraints are employed:

- Each individual in the population has an identical number of ADF trees. This ensures that any operation involving two individuals can make assumptions about those trees and their make-up.

- Each ADF must return a type that is already available and used. This is an inevitable consequence of the structure of an ADF, which is itself made up of zero or more ADFs and non-ADF primitives. The type of data returned by an ADF must therefore be based on the return type of a non-ADF primitive.

- An ADF tree of a particular index must have the same type and parameter list for all individuals in a population. So, if ADF1 for candidate 0 has return type double and takes (float, float, int) as parameters, ADF1 for all the candidates in a population must also have this prototype. This constraint guarantees correct crossover between candidates. The index of an ADF indicates which particular ADF is being referred to, i.e. in a population where individuals have 3 ADFs, such functions would be called ADF0, ADF1, ADF2.

Crossover in a strongly-typed system incorporates the constraint that the sub-trees swapped between candidates must return the same type. The operator used in GP-COM selects a tree at random from the first parent. It then searches the second parent for a sub-tree of identical type, starting from a random position in the genome. If no such sub-tree is found, a tree of different type is selected from the first parent and the process repeats until a match is found. This operator is guaranteed to succeed because eventually a tree returning the same type as the whole program tree is selected - since all candidates return the same type there will definitely be a matching typed tree in the second parent, consisting of the entire genome of the second parent. This is a worst case scenario but it can still be of use, as although whole program trees may be swapped, any ADFs will not be.

There is a chance that the crossover operator will not choose the value-returning branch of the individual at all, but will choose to operate on an ADF. In this case, the third constraint above will force the operator to use the corresponding ADF of the second candidate as the other parent. This is necessary because each ADF has its own namespace, consisting of the formal parameters to that function and any other ADFs that may be callable from inside the selected ADF. Identical parameter names (such as \texttt{p1}, \texttt{p2} etc.) in different ADFs may have different types, so crossover cannot ensure syntactic and type safety if crossing trees from ADFs with different indices.

Mutation for strongly-typed GP must produce a new sub-tree of the same type as the sub-tree it is replacing. Mutation of a strongly-typed ADF uses the namespace of that ADF to ensure type safety.

**Code Generation**

The population manager component is independent of the problem domain, so it cannot be used to execute the candidate programs it manipulates. Some external mechanism for evaluating program trees is therefore required. One popular approach is to write an interpreter and execute the trees directly, but this involves considerable coding effort to produce a pseudo-language of sufficient power, especially when supporting primitives of arbitrary type. These primitives would then have to be written in the pseudo-language also. A more powerful method is to use an
existing language as the representation for program trees, thus taking advantage of that language’s capabilities in terms of storage and execution. Many systems use LISP as it provides a unified method of execution and of manipulating program trees. GP-COM takes an alternative approach, by translating the program trees to C source code and using machine-native binaries as the execution medium.

Execution of candidate programs involves supplying a problem-specific code skeleton as a context into which the programs themselves can be inserted. GP-COM assumes that this skeleton is going to be a C or C++ program. The population manager produces a C function for each program tree, and one function for each ADF tree. This has important implications for performance of the system.

A population of 500 individuals, each with 3 ADFs, produces a total of 2000 C functions each generation. These functions are added to the problem skeleton code and compiled to produce an executable which, when run, outputs the results of one generation of the run. Compilation of 2000 functions can take a very long time depending on the equipment used, a penalty that is incurred each generation of every run. This system is obviously inappropriate for implementing steady-state genetic programming.

In terms of performance, compilation has benefits and drawbacks. Compared to systems that generate LISP functions and then evaluate those, a large compilation time is a significant performance penalty. This may be compensated, however, by much reduced run-times. Overall performance gains or losses are determined by the cost of the problem and the size of the population. A computationally expensive problem could gain considerably from being coded in C rather than LISP, and this may more than compensate for the compilation overhead. Additionally, compilation time is roughly linear in the size of the population, so growth in computational resources overall remains linear. This means that the compilation time can be viewed as a constant factor in the total execution time of one cycle of the genetic search process. Problems which are inherently expensive to run, even with the speed advantages of C, are thus affected to a lesser degree than trivial problems which execute in a few seconds. Performance gains could also be made by incorporating a “make” like system, where only candidates which have changed would need to be recompiled with the next generation.

Where compilation does score favourably is in the flexibility of the function and terminal sets. This allows the user to incorporate functionality from a huge range of software. Data types used in strongly-typed GP can map directly to types defined in C++, including of course user-defined classes. Functions and terminals in the primitive set map directly to C++ functions and variables and as such can be drawn from any library and linked in to the problem skeleton. This allows the program trees to correspond very closely to the domain representation as this is itself a model using the same set of primitives.

Parallelism in Genetic Programming

[Koza] gives three levels at which the performance of GP may be increased by parallelisation. These are:

- By fitness case. Every individual in a population is evaluated on every processor, but only a subset of the fitness cases are evaluated on each processor.
• By individual. Every fitness case is evaluated on every processor, but only for a distinct subset of the population.

• By run. A problem will most likely require several runs to produce an adequate solution. These runs can be simultaneous, with each processor evaluating the whole population (for one run) on every fitness case.

GP-COM parallelises at the run-level, using a separate CPU for each run. Runs are automatically launched on idle machines to ensure maximum performance. The run itself checks after each generation and shuts down if someone has logged on to the machine. It is then restarted automatically by a run manager on another spare machine. GP-COM supports multiple architectures and is currently used on the departmental network of Solaris and SunOS machines. This support is in part gained by the use of portable, high-level commands and languages to organise the distribution of processing.

Tree Generation
Program trees in GP-COM can be of arbitrary size and complexity. Instead of imposing a fixed depth limit on tree generation, GP-COM uses a probabilistic method to control the size of trees generated. This method uses a *terminal-probability*, a floating point value dictating the likelihood of choosing a terminal at each node during tree generation. This figure is a run-specific parameter which applies at the root node of the tree. This probability is increased additively with the depth of the current node, until at some depth it reaches 1. This increase is multiplied by the branching factor of the current node, so a function taking 5 parameters is less likely to have large sub-trees underneath it that one taking 2 parameters at the same depth.

Additionally, a separate *mutate-terminal-probability* is used to control the generation of sub-trees by mutation. This controls the way in which trees grow as the run progresses. Given that the crossover operator does not change the total size of its operands, and assuming that the genetic process does not have some in-built or fitness-inspired bias toward small trees, the mutation operator is responsible for any changes in the average size of candidate trees. Careful tuning of these probabilities allows fine control over the initial size and growth of program trees.

These two probabilities are also implemented independently for ADF tree creation and mutation.

Storage
GP-COM stores genomes as linear structures which, when combined with prototype information for the primitives, can be mapped to tree structures and back again. This representation is human-readable, which aids debugging and analysis of populations, and the writing of custom applications for problem-specific reporting. For instance, a program tree that codes a function for a problem such as symbolic regression, can easily be converted to a form suitable for visualisation by a plotting program. As an example, GP-COM includes a utility to view program trees by converting these representations to a form readable by a graph layout tool.

Run-time customisation
GP-COM is a hierarchical system. GP-COM has a default global configuration which is inherited each time a new problem is created. These configurations are used by many of the components in the system and are easily edited by the user. In turn, these problem-specific configurations are automatically inherited by each run, allowing the user to set options that apply
to all runs for a given problem, or to override them for a particular run or set of runs. Options that currently implemented are:

- mutation rate and crossover rate
- terminal-probability and mutate-terminal-probability
- ADF-terminal-probability ADF-mutate-terminal-probability
- Population size
- Selection method. These are implemented as separate programs and take inputs from normalised fitness values. They output a stream of commands which are piped into the population manager. These commands consist of instructions to place a particular member of the current population into a slot in the next population.
- Whether or not raw fitness equals standard fitness. If so, then raw fitness is simply copied to standard fitness. If not, a separate program is invoked to calculate standard fitness scores.
- Primitive set. This is a pointer to a *types file* containing the prototypes for a given set of primitives. This allows rapid changing of primitive sets with no recompilation, as each primitive set can be a subset of those functions already defined in C code. This allows quick experimentation with different primitive sets, for example to examine the effect of adding or removing a primitive from the set. Type names are identical to their representations in C, and may include pointers.
- Number of ADFs. If set to 0 no ADFs are used. If non-zero, the system will generate prototypes randomly for each ADF at generation 0 and add them to the types file. This allows each run to be different in terms of the prototypes of its ADFs. If the types file contains prototypes for ADFs already then these details are used, so runs can be forced to use the same ADF prototypes by adding them to the types file at the problem-level.
- Random seeds. These are generated using an architecture-independent generator. This allows the system as whole to be repeatable - a run started using the same configuration and same random seeds will produce the same results every time.

**System Structure**

GP-COM is a loose collection of programs, each with tightly specified input/output interfaces. With GP-COM the only programming effort required is the application of the candidate programs to produce raw fitness values. Everything else in the system is problem-independent and automatically applied. The hierarchical, inherited nature of the system saves time and effort by maximising the re-use of common software components. Three levels of control are implemented, using the Tcl scripting language and Tk interface toolkit.

The most abstract level of control is at the system level. This is a simple script that allows the user to select the problem to be worked on. Basic facilities for automatically setting up a new problem exist, but these are limited to the creation of the necessary support files etc. Problem-specific software is left to the user to implement.

The next layer of control is at the problem level. This script implements a run manager, software to control the creation, execution, and configuration of a number of runs of the same problem. Creation of a new run involves generating appropriate random seeds, and creating the file and software structure which the run will use. The code base used by each run is inherited from problem-specific software, as each run can be viewed as an instantiation of the problem. To start
a run, the run manager first searches for an available host (which for diplomatic reasons is defined as a host with no-one logged on). It then launches the run on that host via a remote shell.

Run-level control is achieved by a third script. This script implements the whole GP cycle for a given run. It supports multiple architectures, running system-specific versions of the population manager and all the support components. The problem program is always architecture specific, as it is built anew with each generation. The interface allows the user to change population size, set a number of generations to run as a batch, and to re-initialise the run. The main part of this script involves communication with the population manager. This interaction is performed using the automation tool Expect. The population manager is kept running in the background while the other parts of the cycle are executed in turn. The population is turned into C code. Building and execution of the problem program, fitness calculation and processing, population and fitness logging are performed, and the termination criterion is tested. Selection and genetic operation commands are generated and piped to the population program. The new population is written to disk, and the cycle starts again. After each generation has completed, the script checks to see if the workstation is being used. If so the run is shut down and the run manager then relaunches the run on another free machine. Communication between the runs and the run manager is achieved using Tk’s built in communication facility, with all the runs displaying on the same machine as the run manager.

Each level of control works by dividing the work up between languages according to their capabilities. Tcl is very good for high-level work, co-ordinating activity of a number of programs, and passing data between them. Tk is used to display results and enhance the interface to the system. C/C++ is good for numerical work and for storage of large data structures (such as genetic programs). A deliberate decision was made to build GP-COM out of a large number of components, so that the system would remain flexible. This allows a new selection method, for example, to be added to the system simply by writing a short program that takes normalised fitness values as input and outputs a stream of commands in a form suitable for the population manager. This new method can then be specified purely by the name of the executable file that implements the technique. No extra configuration, re-compilation or re-programming of the rest of the system is required. This allows the system to be changed and augmented quickly and with little effort.
**An example problem using GP-COM**

We used GP-COM to apply GP to an edge-detector problem [Harris]. Two components in the system, the problem program and raw fitness calculation, needed to be written from scratch. In theory these could be combined into a single program, but GP-COM assumes two programs will be used. This allows rapid substitution of a number of different fitness functions, which perform the mapping from results to raw fitness, without changing the problem definition code.

In this case, a lot of programming work was necessary because of the nature of the problem. This involved generating functions to be used in processing of 1-D signals. A suitable library of signal processing routines was developed. Due to the distributed nature of the system, and the large quantities of results produced each generation, special consideration was needed regarding performance. It was necessary to keep all intermediate results on storage local to the workstation on which the run was being executed, to avoid overloading the machine on which GP-COM was being run.

During development, the loose binding between the components helped the process of debugging and testing. By testing each component individually and independently, made possible by the discrete nature of those components, a working model was quickly built. Once all the components had been put into place, the front-end and parallelisation was automatically available through the global software components, and required no coding effort.

The GP-COM system. The problem is an edge-detector problem [Harris], and 6 runs are being displayed.
Conclusions
A generic Genetic Programming system was described. The system was split conceptually into problem-dependent and problem independent components. Based on this analysis, a compiling distributed component-based genetic programming system was presented called GP-COM. This system has a number of novel features. Possible advantages of this approach include time savings in execution, and easier problem alteration and setup for the user. The system has been used successfully with symbolic regression and edge detection problems.

Acknowledgements
Christopher Harris is a member of the Genetic Programming Group and the Machine Vision Group, in the Department of Computer Science at University College London. He is funded by the EPSRC under the Postgraduate Training Partnership, in conjunction with SIRA Ltd.

Bernard Buxton is a Professor of Computer Science at University College London, and heads the department’s Machine Vision Group.

References


