GAPS: a Genetic Programming System

Michael D. Kramer and Du Zhang

Department of Computer Science
California State University
Sacramento, CA 95819-6021
U.S.A.

Abstract

Genetic programming tackles the issue of how to automatically create a working computer program for a given problem from some initial problem statement. The goal is accomplished in genetic programming by genetically breeding a population of computer programs in terms of the principles of Darwinian natural selection of the fittest and genetic operations. In this paper, we describe a genetic programming system called GAPS. GAPS has the following features: (1) It implements the prototypical generational algorithm for genetic programming with three improvements (the honor roll, improved termination criteria and the tree techniques for fitness evaluation). (2) It includes an extensible language tailored to the needs of genetic programming. And (3) it is a complete, standalone system that allows for genetic programming tasks to be carried out without requiring other tools such as compilers. Preliminary results with GAPS have been satisfactory.

1. Introduction

Evolutionary computation refers to the use of evolutionary algorithms to solve difficult computational problems [3]. Two major approaches exist in the field of evolutionary computation: genetic algorithms (GA) and genetic programming (GP) [4,5,6,7]. One important difference between GA and GP lies in the representational formalisms for hypotheses: GA uses bit strings whereas GP uses computer programs. GP deals with the issue of how to automatically create a working computer program for a given problem from some initial problem statement. The goal of GP is accomplished by genetically breeding a population of computer programs in terms of the principles of Darwinian natural selection of the fittest and genetic operations [1,2,7–10,12–16].

In this paper, we describe a complete, standalone system for GP called GAPS [11]. Preliminary results with GAPS have been satisfactory. GAPS has the following features:

(1) It implements the prototypical generational algorithm for GP with three improvements: the use of the honor roll for the top scorers throughout the entire evolution process, improved termination criteria and the tree techniques for fitness evaluation.
(2) It has an extensible language tailored to the needs of genetic programming. The language can be used for creating the structures that GAPS manipulates, and for creating the "shell" programs to evaluate the fitness of the computer programs generated. It is extensible in that it allows new functions and subroutines to be dynamically defined and used.
(3) It offers a complete, standalone development environment that allows for genetic programming tasks to be carried out without requiring other tools such as compilers.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the main tasks and issues in GP. Discussion on GAPS is provided in Section 3. Performance issue on GAPS is briefly described in Section 4. Finally, Section 5 concludes the paper with remarks on future work.

2. Genetic Programming

In GP, a computer program is often represented as a tree (a program tree)[1] where the internal nodes correspond to a set of functions used in the program and the external nodes (terminals) indicate variables and constants used as the input to functions. For a given problem, GP starts with an initial population $\wp$ of randomly generated computer programs. The evolution process of generating a final computer program that solves the given problem is defined in the following prototypical algorithm:

Define:

f: a fitness function that yields an evaluation score for a computer program $p \in \wp$;
$\Im$: a fitness threshold used as a terminating condition;
ϑ: the size of $\wp$;
γ: the fraction of $\wp$ to be replaced by crossover at each generation;

[1] The term "tree" is used to refer to a LISP style program throughout the remainder of the paper.

0-7695-0792-1/00 $10.00 © 2000 IEEE

μ: the mutation rate;

λ: the limit used as an alternative terminating condition and $\angle\lambda$ indicates that limit is not reached[2].

The algorithm ***GP*** is then given as follows:

```
GP(f, ℑ, ϑ, ϒ, μ, λ)
Step 1: Initialize population: $\wp \leftarrow \vartheta$ randomly generated
        computer programs (trees);
Step 2: Compute $f(p)$ for all $p \in \wp$;
Step 3: While $((\max(\{f(p) \mid p \in \wp\}) < \Im) \wedge (\angle\lambda))$ do
Produce a new generation of computer programs $\wp'$
  (a) Select probabilistically $(1 - \Upsilon)\wp$ members of $\wp$ to
      be included in $\wp'$. The probability $\Pr(p)$ of selecting $p$
      $\in \wp$ is defined as follows:
```

$$\Pr(p) = \frac{f(p)}{\sum_{i=1}^{\vartheta} f(p_i)}$$

```
  (b) Select probabilistically $(\Upsilon\vartheta/2)$ pairs of programs from
      $\wp$ according to $\Pr(p)$ above. For each tree in a pair, a
      crossover point is randomly chosen and two offspring
      (trees) are produced from the pair in terms of the
      crossover operation and are placed into $\wp'$.
  (c) Choose $\mu$ percent of the trees from $\wp'$ with uniform
      probability. For each, a mutation operation is
      performed.
  (d) Update $\wp$ with $\wp'$ $(\wp \leftarrow \wp')$.
  (e) Compute $f(p)$ for all $p \in \wp$.
Step 4: Return the program tree $p \in \wp$ such that $f(p)$ is
the maximal. □
```

There are a number of issues to be considered in a GP system:

(1) Definitions of functions and terminals to be used in the program trees generated.
(2) Definition of a fitness function for evaluating program trees and the way those trees are evaluated.
(3) Generation of the initial population.
(4) Selection strategies for trees to be included in a next generation population.
(5) How crossover and mutation operations are carried out and how often these operations are performed.
(6) Criteria for terminating the evolution process and the way to check if the terminating conditions are satisfied.
(7) Return of the final results.

3. GAPS

The most important difference between GAPS and the other systems we have examined is that GAPS is a complete GP environment. The other systems might be more correctly characterized as tool kits for creating programs to solve a specific problem. In order to make use of these systems, the user must have available the tools for creating programs in whatever source language that the system was written in, whether that be LISP, C, C++, or any of the other languages used in genetic programming.

To work in a specific problem domain, the current technique is:

- Write modules that will input the parameters to the problem, evaluate trees with regard to the problem domain, and handle any sort of output required during the run.
- Compile these modules.
- Link the modules with the supplied system modules.
- Execute the resulting program.

With its integrated language, GAPS is wholly self-contained. For a given problem, a problem specification (PS) file[3] is created that specifies the nature of the problem at hand as well as certain conditions for the evolution process. To attack different problems, all that is needed is to run the same GP engine with *different* PS files.

3.1. The GAPS language

There are three types of statements, functions, and commands in the language.

Meta Statements. These are used to direct the system in how to execute the run, and what data structures (seed trees, test sets, shell tree, miscellaneous parameters) to create and use during the run. The setting of the control parameters is handled through the meta statements.

Tree Functions. These are the actual executable instructions. They are all fairly standard computer instructions, and can be put together in a list structure to represent trees. Each one takes as input a specified number of values of specified or variable data types and returns a value of a specific data type.

Shell Commands. A GAPS shell is simply a special case of the general tree. The shell is responsible for evaluating the generated trees, feeding them any required inputs, and returning a meaningful fitness measure back to the GAPS system. Shell commands are similar to the tree functions, but their use is generally restricted to the job of setting up and evaluating the runs of individual trees. These commands make the job of tree evaluation, data initialization, etc., simpler. While the shell commands are designed for use only by the shell program, they can be used in generated trees by creating user-defined functions that call these commands.

All of these statements/functions/commands are contained in a text file (PS files). They can be generated either interactively within GAPS, or independently from

[2] The limit here may be defined in terms of either time or resources consumed during the evolution process, or both.

[3] This is a text file with the file extension of .GAP.

GAPS, or a combination of both. All of the files that GAPS uses are intentionally text files. This enables a user to see exactly all of the relevant information on the run that a file sets up. It also makes it possible for the output file from a run to be edited and fed back into GAPS for a modified run or to be able to suspend a run and then pick up where the run had left off.

3.2. Fitness function and tree evaluation

The fitness function describes the criterion for ranking trees. It also serves as the basis for probabilistically selecting trees for inclusion in the next generation population. It can be defined based on the user provided test sets for the target program. These test sets would consist of the input data for the program and the expected corresponding output[4]. Thus the fitness of a tree $f(p)$ for $p \in \wp$ boils down to the accuracy of p over the test sets.

To evaluate the fitness of trees in the current population, GAPS uses its own language to define an evaluation program. There are a number of advantages. First, the system is complete unto itself and does not require additional tools such as interpreters or language systems in order to work in new domains. Second, special purpose tools for performing fitness evaluations can be built into the language, making it simpler to do the evaluation.

3.3. Initial population generation

Once the size of the initial population is defined, the main task is to create enough trees to fill out the population. If any "seed" trees are specified in the PS file, these are used first, and are simply created directly from the descriptions supplied in the PS file. When there are no more seed trees, trees are created randomly, using the terminals and functions specified in the PS file as being permissible. After each tree is created, it is added to the initial population and then evaluated.

GAPS implements several standard methods for creating the initial population of program trees. The *full* method creates only fully populated trees. That is, every branch will end at the same level. Until this "maximum depth" is reached, only functions are chosen to be added to the tree. At the maximum depth, only terminals are chosen. The *grow* method creates trees that might be, but are not required to be fully populated. No restrictions are placed on the selection of functions versus terminals other than possibly enforcing a minimum depth, and for enforcing a maximum depth. The *ramped half-and-half* method combines the full and grow methods in a 50/50 split and adds the concept of a *ramp*. If the ramp is, for instance, 3..5, equal proportions of trees will use maximum depths of 3, 4, and 5.

[4] Other variables, such as degree of tolerance (how far off an answer can be and still be considered correct), the number of correct answers that will be considered sufficient for accepting a solution, can also be used in defining the fitness function.

3.4. Selection strategies

GAPS offers a number of selection strategies for use by the users. The first strategy is called *Fitness Proportionate Selection* (sometimes referred to as *Roulette Wheel Selection*). Trees are selected randomly, with the probability of $p \in \wp$ being selected proportional to p's fitness $f(p)$ divided by the total fitness of the population. The second strategy is referred to as *Tournament Selection* in which a selection of some fixed number of candidates is taken randomly from the population and the individual with the highest fitness score within this random group is chosen. This process is repeated until the required number of individuals have been selected. The third strategy is *Greedy Over-Selection* which partitions the population into two groups, a relatively small group of "elite" scorers (typically the top 15% of the population) and the remainder of the population. Proportionate selection alternates between these two groups, with the proportion for the elite group ranging from 50% to 100% of the time.

3.5. Genetic operations

The two main genetic operators in GAPS are *crossover* and *mutation*. In the crossover operation, a pair of parent trees having different sizes is probabilistically selected from the current population based on fitness. A crossover point is then determined in the two trees randomly. Finally, subtrees rooted at the crossover points of the parents are swapped, generating two offspring. Crossover operation is the predominant operation that is often performed at a high probability (85% to 90%).

In the mutation operation, a single parent tree is probabilistically selected from the current population based on fitness. A mutation point is then randomly determined. Finally the subtree rooted at the mutation point is replaced by a new subtree that is created using the same random tree generation process as in the initial population. Mutation rate is much smaller (< 5%).

In addition to crossover and mutation, there are some other operations that have been found to be useful in GAPS [11].

3.6. Terminating condition

In a genetic programming system, a set number of generations is generally used as the condition for terminating the evolution process. GAPS uses a more flexible system. As long as improvements continue to be made, the run continues. This prevents either the premature termination of avenues of search that continues to pay off, or continuing to search dead-ends when no further progress seems likely. A maximum number of *unproductive* generations is the primary criterion used to halt a run in GAPS rather than a limit on the total number of generations. This eliminates a reliance on a priori assumptions regarding the run and instead uses the results themselves to guide the run.

Specifically, as each new tree is created in GAPS, it is evaluated. During each generation in which no improvement in the top score occurs, a counter is incremented. When this counter reaches a user-defined threshold, the run is signaled to shut down. Before termination occurs, two things could happen. First, a temporary boost in the mutation rate could be triggered, in the hopes of shaking things up enough to get progress going again. Second, the scoring strategy could be altered, favoring smaller trees. This latter effect is useful for getting more generalized solutions to the given problem and for weeding out "deadwood" - branches that do not actually contribute to the problem solution.

In GAPS, there are two user-defined parameters **MaxGens** and **ExtraGens**, and two run-time variables **UnproductiveGens** and **SHRINK**. The initial values for **UnproductiveGens** and **SHRINK** are 0 and off, respectively. Let **stop** be initially false, checking for the termination condition is captured in the following procedure.

```
while (not((max({f(p) | p ∈ ℘}) ≥ 𝔖)∨stop) do
{L1: carry out the evolution process (based only on
       fitness consideration);
if (there is no improvement in current generation)
then {UnproductiveGens++;
  if (UnproductiveGens == MaxGens)
  then { SHRINK = "on";
         UnproductiveGens = 0; }
  else goto L1;
  L2: carry out the evolution process (based on both
       fitness and tree sizes consideration)
       if (there is improvement in current generation)
       then { SHRINK = "off";
              UnproductiveGens = 0; }
       else {UnproductiveGens =
                 UnproductiveGens + 1;
            if (UnproductiveGens==ExtraGens )
            then { stop = true;}
            else goto L2; }}}
```

If the user elects to cancel a run, an option is presented to save all of the trees from the final generation. This allows the run to be resumed later where it left off.

3.7. Result return

In general, the final result of a GP run is considered to be the top scoring individual from the final generation. GAPS extends this with a structure called *honor roll*, which contains the top scorers from all generations of the run. The size of the honor roll is user-definable, with a minimum size of one. Even with this minimum size, the honor roll provides a better result return mechanism than the standard method. Because in the standard method, there is no guarantee that the best individual in a run will be found in the final generation, but it is guaranteed that the best individual from a run *will* be found at the top of the honor roll.

The advantage of larger honor roll sizes is that in trying to solve a particular problem, a human can often see advantages to various individuals that were not the highest scorers. The human can then manually combine features of various high scoring individuals, or can use these high scoring individuals as seeds for a new run. It can also be beneficial to combine the honor rolls from multiple runs to seed a new run, giving the benefit of traits evolved in different evolutionary pathways in different runs.

The honor roll is printed in descending order of fitness scores. Each member of the honor roll contains the following information:

- The numerical index of the tree. This provides information on how long it took to evolve this particular tree.
- The fitness score of the tree.
- The size of the tree.
- The tree itself, in list notation.

GAPS produces a log file at the end of the run. The log file can include various types of information. At a minimum, it will include a copy of the PS file, progress summaries for each generation, and a listing of the honor roll at completion of the run. To change the amount of information provided, several "diagnostic levels" can be specified in the PS file. The commands and their associated information are:

- Print-Trees: Amount of information to be printed for each generated tree.
- Print-Generations: Amount of summary information for each completed generation.
- Print-Deletions: Amount of information to be printed regarding deleted trees.
- Print-Tests: Level of tracing to be done during the evaluation of each tree.

In all cases, a numeric parameter ranging from 0 for no information, to 5 for maximum information, must be provided.

The log file can be very useful for generating input files for new runs. If a run was cut short, the population at that time, or the honor roll at that time could be used as seed trees when restarting the run later. Or the honor rolls from several runs could be combined in a new run to get the benefits of separate evolutionary pathways being combined. And of course the log can be useful in understanding just how a resulting tree was arrived at.

3.8. How GAPS operates

GAPS can be used in three different modes:

(1) Interactive Mode. In the interactive mode, a PS file can be either created using built-in editing functions, or read from disk and then modified and saved to disk. Files can be executed, generating populations of trees that are then evolved in an effort to solve the defined problem. An initial PS file to be opened can be specified on the command line.
(2) Pseudo-batch Mode. In pseudo-batch mode, the editing functions are not used, and the PS files are opened and executed from within the GAPS environment.
(3) True batch mode. In true batch mode, a file containing the names of PS files to be run is specified on the command line and the GAPS environment is never accessed.

GAPS uses a fairly standard Windows interface, so that anyone having experience with other Windows programs should be able to use GAPS.

The basic sequence of events when running in the GAPS environment is:

(1) Open a PS file if one has previously been created.
(2) Edit the file to customize it to the particular problem at hand.
(3) Save the file with the changes.
(4) Run the GP engine with the file.

In the pull-down menu of Edit, a user may edit any of the following items in either a PS file or the system environment parameters for the run:

- Test sets (to be used for fitness evaluation of generated trees).
- Seed trees (to be included as part of the initial population for the evolution process).
- User defined functions (to be used in generated trees).
- Built-in functions (to be used in generated trees).
- Shell tree (to define how to carry out the fitness evaluation).
- Target tree (to define what the target tree looks like).
- Miscellaneous parameters (parameters for the evolution run).

Any of the tree functions defined in the GAPS language can be used as part of the target trees generated. Changing the built-in functions used in the evolution run would result in different trees being created.

When "Run" is selected from the GAPS main window, a whole sequence of events occurs. It starts with verification of the control parameter values that were supplied in the PS file or via the editing functions. Once this is done, several run-time variables are initialized. This includes setting a pointer to the first test set if test sets are being used, clearing out any existing trees, and zeroing out the generation count. Then the actual work begins.

While the run is occurring, a dialog box is displayed to keep the user apprised of the progress of the run. Included in the dialog are:

- The number of the current tree.
- The number of the current generation.
- The number of generations that have passed with no improvement and the maximum number of generations with no improvement allowed.
- The average and best fitness so far.
- The average and best number of "hits" so far.
- Buttons to either pause the run or end it immediately.

Watching the numbers in the dialog box can give you an interesting view into the dynamics of evolution, the best scores varying between short spurts of sudden progress and relatively long periods of apparent equilibrium.

Finally, when the run finishes, GAPS outputs the following information in a window, in addition to creating the log file:

- Total number of generations created in the evolution run.
- Total number of trees generated.
- The best fitness score.
- The finish time.

4. Performance

GAPS is developed in the windows environment using C++. It has been found to be an effective GP system for a number of different categories of problems such as sequence induction, symbolic regression, pattern recognition, optimal control, automatic programming, and many others. Due to space limit, we only briefly mention the result of an example run for the *sequence induction* problem as follows.

The log file contains various information as promised. The lines before END_DATA are essentially a copy of the PS file for the given problem. The entire run results in 490 trees being created in 14 generations. The honor roll contains the top ten scorers from all generations. It was not until the 469th tree that a perfect score was achieved and the correct program was determined.

5. Conclusion

GAPS does indeed seem to be a viable way to work with the GP paradigm. The ability to change problem domains quickly and easily through the GAPS language seems to have been achieved. The GAPS language is a complete one, including all the basic computer constructs such as selection, iteration, sequential execution, and even recursion. Extensions to the language can be added fairly easily, either by an end user adding new keywords along with the trees to implement those keywords, or by a programmer adding functionality for new keywords to a few well confined areas of the GAPS computer code. The system is written in an object-oriented fashion to make changes to the system relatively painless.

The honor roll is a useful extension to GP. It has proven invaluable in our work with GP. Likewise for the termination strategy of using continuing progress as the criteria rather than a set number of generations. This eliminates a reliance on a priori assumptions regarding the run and instead uses the results themselves to guide the run.

Allowing trees as the means of evaluating other trees does make it much easier to jump between different problem domains, only requiring one executable program to be used for solving any GP problem.

Future work and enhancements to GAPS can be pursued in the following directions:

- Conversion routines for automatically creating program code in C, C++, or Java from successful trees. This would be extremely useful for making GP a truly useful tool for creating solutions that can then be immediately put into standard computer programs.
- Multiple populations. Currently the groundwork is in place for creating and working with multiple populations in GAPS. Very little more work would be required to allow this. However, some of the benefits of multiple populations, such as independent evolutionary pathways, already exist in GAPS through the honor roll and the ability to mix populations from various runs into a new run.
- Competitive game playing. Here too, the groundwork is in place to allow trees to compete against each other to generate optimal competitive strategies. When multiple populations have been implemented, the GAPS language will easily support evaluating trees from different populations for purposes of head-to-head competition.
- Faster tree evaluation. There has been much work done in the field of optimizing tree evaluation, and more work still to be done. Of course the faster that trees can be evaluated, the more useful and the more powerful GP becomes.
- Multiple data types. GAPS can theoretically support multiple data types. All that is really needed is code to enforce that crossover only occurs at appropriate locations so that closure is not compromised.
- ADF. ADF's, or automatically defined functions, are a method for increasing the power of GP by providing a mechanism for creating subroutines within a tree. This would be another useful addition to GAPS.
- Age-based death. One of the primary facets of biological genetics that is not mirrored in GAPS (or in other GP systems that we have seen) is age-based death. With the honor roll preventing the total loss of exceptional genetic specimens, this might be a useful addition to the GP repertoire. It would be interesting to see the effects this has on the speed of evolutionary progress.
- Evolutionary throwbacks from the honor roll. With the honor roll, high scoring individuals could be randomly put back into the population some time after they had been removed. This would also be an interesting area to study for its effects on evolutionary progress.
- More extensive input/output facilities in the GAPS language. One of its major shortcomings as a complete language is the limited I/O capabilities. Further work in this area, along with increases in the speed of tree evaluation, would eliminate most of the reasons for wanting to use the standard genetic programming method of recompiling the entire system for each new problem.

References

1. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming: An Introduction*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998.
2. J. Cona, Developing a Genetic Programming System, AI Expert, No.2, 1995.
3. K. A. De Jong, Evolutionary Computation for Discovery, *Communications of the ACM*, Vol.42, No.11, November 1999, pp.51-53.
4. D. E. Goldberg, Genetic and Evolutionary Algorithms Come of Age, *Communications of the ACM*, Vol.37, No.3, March 1994, pp.113-119.
5. J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975 (Second edition, MIT Press, Cambridge, MA, 1992).
6. J. H. Holland, *Hidden Order: How Adaptation Builds Complexity*, Addison-Wesley Publishing Company, Reading, MA, 1995.
7. J. R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*, MIT Press, Cambridge, MA, 1992
8. J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, 1994.
9. J. R. Koza et al, *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.
10. K. E. Kinnear, (ed.), *Advances in Genetic Programming*, MIT Press, Cambridge, MA, 1994.
11. M. D. Kramer. *GAPS: The Genetic Algorithm Programming System*, MS degree thesis, Department of Computer Science, California State University, Sacramento, May 1996.
12. C. G. Langton (ed.), *Artificial Life II*, Santa Fe Institute, Addison-Wesley, 1991.
13. T. M. Mitchell, *Machine Learning*, WCB/McGraw-Hill, Boston, MA, 1997.
14. A. Singleton, Genetic Programming With C++, Byte, No.2, 1994.
15. L. Spector et al (ed.), *Advances in Genetic Programming* Volume 3, MIT Press, Cambridge, MA, 1999.
16. Special essays on "Genetic programming", *IEEE Intelligent Systems*, Vol.15, No.3, May/June 2000, pp.74-84.