Improving Modularity in Genetic Programming Using Graph-Based Data Mining

Istvan Jonyer and Akiko Himes

Oklahoma State University
700 N Greenwood Ave
Tulsa, OK 74106
jonyer@cs.okstate.edu

Abstract

We propose to improve the efficiency of genetic programming, a method to automatically evolve computer programs. We use graph-based data mining to identify common aspects of highly fit individuals and modularizing them by creating functions out of the subprograms identified. Empirical evaluation on the lawnmower problem shows that our approach is successful in reducing the number of generations needed to find target programs. Even though the graph-based data mining system requires additional processing time, the number of individuals required in a generation can also be greatly reduced, resulting in an overall speed-up.

Introduction

Genetic programming (GP) is a machine learning technique that evolves computer programs to solve specific problems. During a run of GP, a population of programs is randomly generated and transformed into a new generation using evolutionary operations. Genetic programming require substantial computational effort for solving any realistic problem, which is why most of the research in the field is concentrated on reducing the required computational effort.

One natural direction of research is towards modular programs. This has been explored mostly in the context of automatically defined functions (ADF) (Koza, 1994). An ADF is a subroutine which is evolved during the evolutionary run. The number of ADFs can be either predetermined by the user, or evolved along with the program itself. The success of ADFs—whether they will actually help to evolve a solution faster—depends on the user-defined parameters, such as the number of ADFs, the number of arguments to ADFs, and the function set. As such, there exist optimal and suboptimal choices for parameters. This typically leads to approaching a problem using different parameter settings in a trial and error fashion. Reducing the number of user specifiable parameters while speeding up GP would be highly desirable.

We hypothesize that subroutines that are common in highly fit individuals would be good candidates for ADFs. We propose to dynamically identify ADFs using a graph-based data mining technique that finds substructures in graphs (or parse trees of computer programs in this case). We evaluate the success of this approach empirically.

Related Work

Many approaches are studied for improving the overall efficiency of genetic programming. Of these, we are most concerned with those approaches that seek improvement through increased modularity, since this is the subject of our study. An object oriented approach to combined learning of decision trees and ADFs in genetic programming was proposed by Niimi and Tazaki (1999). While GP with ADFs gives expected solutions, its learning speed is slow. Decision tree learners, on the other hand, construct trees rapidly, but may not categorize correctly when the input data has noise. Combining these two methods makes up for the disadvantages of each technique.

Jassadapakorn and Chongstitvatana (1998) proposed an approach in which the number of ADFs is not pre-specified. The extension of ADF (ADFX) allows each individual to have any number of ADFs within a range, which gives flexibility in the number of ADFs while maintaining efficiency. The ADFX-approach finds the desirable number of ADFs automatically.

Background

In this section we give an overview of genetic-programming, and graph-based data mining, the two main components of the proposed system.

Genetic Programming

Genetic programming is an extension of genetic algorithms (GA) (Koza, 1994). GP is the technique of evolving computer programs using operations such as crossover, reproduction, and mutation. One popular way to represent programs in genetic programming is by rooted parse trees. For example, the program "(2+4)/3" can be represented as shown in figure 1.

An invocation of GP requires five aspects of the problem to be specified: (1) terminal set, (2) function set, (3) fitness

measure, (4) parameters for controlling the run, and (5) the method for designating a result and the criterion for termination of a run.

The terminal set may include constants and variables. The function set must be specified by the user, and may include mathematical functions (+, -, *, /, log, sin, …), logical functions (AND, OR, NOT), or other domain specific functions. If an individual is found that solves problems perfectly, the run is terminated.

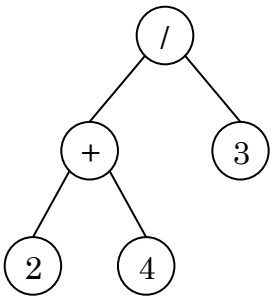

Figure 1 Parse tree for the program (2+4)/3

A run of GP starts by creating an initial population of computer programs that is a random composition of available functions and terminals. Each individual is represented as a tree structure. GP evaluates all individuals in the population based on a fitness measure. A new generation of individuals is created using the evolutionary operators of crossover, mutation and reproduction with differing probabilities. Crossover swaps random subtrees of two programs; mutation randomly deletes a subtree, and randomly grows a new subtree at that location; reproduction copies the individual into the new generation without modification.

When GP is invoked with ADFs, each individual has one result-producing tree ("main()") and one or more ADF trees. ADFs trees are not shared with other individuals. An example individual is shown in figure 2.

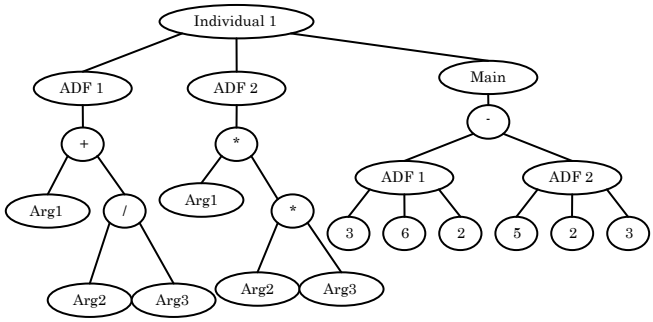

Figure 2 Individual with 2 ADFs

ADFs enable GP to automatically find useful subroutines during a run. Keeping good aspects of the program as subroutines could lead to faster convergence to the desirable solution through modularity. It is also known that an ADF becomes more effective when a problem is relatively complex and the ADF is used frequently (Koza, 1994).

Graph-Based Data Mining

Mining data represented in graph format is useful for many structurally complex domains. One of the most successful general-purpose systems is Subdue (Cook and Holder, 2000). Subdue discovers commonly occurring substructures in the input graph. The minimum description length principle (MDL) drives the search, according to which the best substructure is the one that compresses the input graph the best. Compression is defined as extracting all occurrences of a substructure from the input graph and replacing them by a single vertex. The description length of the resulting encoding is the description length of the compressed graph plus the description length of the definition of the substructure. The MDL heuristic drives the search towards large, frequently occurring substructures.

The input to Subdue must be represented as a graph, which is usually accomplished by representing objects in the data as vertices, and relationships between objects as edges. For a detailed description of the system the reader is referred to (Cook and Holder, 2000).

Here we give an example of Subdue's operation on a domain that translates to a tree structure, as shown in figure 3.

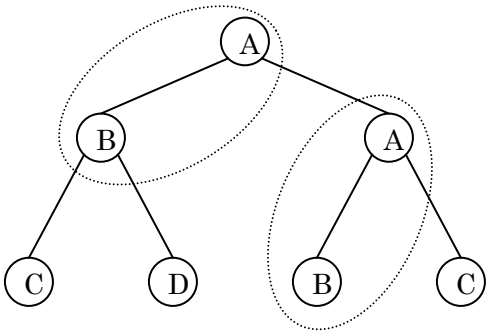

Figure 3 Input graph

The largest, most commonly occurring substructure in this example is the substructure [A-B], circled in the figure. This substructure is identified by Subdue, and is used to compress the input graph, as shown in figure 4.

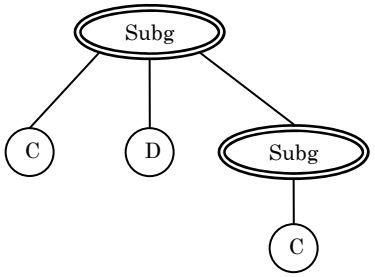

Figure 4 Compressed graph

Genetic Programming Using Graph-Based Data Mining

Our hypothesis is that if the same code segment occurs in the result-producing branch of a number of well-performing individuals in a population of programs, this code may be responsible for the good performance. Such code segments could be abstracted out and turned into an ADF for reuse in multiple locations in the program. To test our hypothesis, we modified an existing GP system, ECJ (Luke et al, 2004) , to include a data mining step that identifies potentially useful subprograms. We describe the resulting system in the following subsections.

Extraction of Useful Functions

After each individual is evaluated, a certain number of highly fit individuals are sent to Subdue for identification of commonly occurring substructures, which are returned to ECJ. We use the best substructure discovered to create an ADF, which we will call “Subdue-ADF”, or SADF for short. Those individuals that have this piece of code in their main result-producing branch will have that part of the code removed, and a call to the new SADF is inserted.

Parameters of the new system

The proposed system requires additional parameters which must be tuned for optimal performance. While adding parameters is not desirable it will be compensated by the fact that some existing parameters will be rendered obsolete. It is also hoped that values for the new parameters can be fixed across domains. The new parameters will be the number of individuals to send to Subdue for analysis, the number of individuals in the population that should receive the new ADF, how often the analysis by Subdue should be performed, and what the acceptable size and frequency of functions returned by Subdue are. In this section we examine general considerations for these parameters. We arrive to recommended values through empirical investigation in the next section.

The number of individuals to send to Subdue for analysis must be carefully examined. First of all, only highly fit individuals should be considered as those are the ones containing successful program segments. Also, sending too many individuals to Subdue will have an adverse impact on the running time. On the other hand, too few individuals may not contain enough repeating patterns.

We must also consider the number of individuals that should receive the new SADF. One possible choice is to insert the new ADF only to the individuals that were sent to Subdue for analysis, since those are the ones in which the functions are found. Alternatively, we may add the new function to all the individuals in the population, or only to a certain percentage of individuals in the population. If we decide not to add the new function to all individuals, the question arises of what to do with the rest of the individuals. We can simply not add a function, or add a randomly generated function.

We had a number of considerations for each of these choices. First of all, only individuals that were sent to Subdue will have an immediate use for the newly defined functions. However, these individuals will not immediately benefit from the new functions, since no new functionality will be added: a piece of code is simply removed from the main branch and is now called as a function. For an immediate benefit of the newly discovered function, it would make sense to add this function to the rest of the individuals in the population. The problem here, however, is that if the same function is defined for each individual, the crossover operator will introduce very little diversity among these ADFs. To promote diversity, some randomly generated ADFs could be added to some individuals in the population instead of the ADF returned by Subdue.

Another consideration is when and how often Subdue should be invoked. If it is invoked in every generation each individual in the population will grow in size by one ADF. This may add an unnecessary number of ADFs while also increasing the time needed to evaluate individuals. It may be advantageous not to invoke Subdue in every generation.

The substructures returned by Subdue may be very small or have very few occurrences. Reusable code that is too small in size or is used infrequently may not turn out to be useful as a function. By requiring newly discovered ADFs to be of certain size and/or frequency, we might be able to further increase the effectiveness of our approach.

Empirical Evaluation

As most initial advances in evolutionary algorithms, our analysis is also experimental. In this section we evaluate our proposed system and possible values of the proposed parameters. While he performed experiments on multiple domains, here we use the Lawnmower problem (Koza, 1992), which is representative of our results. We kept most of default system parameters constant so we can observe the effects of the improvements to the system. We used a crossover rate of 90%, reproduction rate of 10%, and mutation rate of 0% for all our experiments.

The Lawnmower problem

The lawnmower problem is about finding an algorithm for the movement of a lawnmower that mows all the grass in the yard. The yard is modeled as a discrete 8-by-8 square. The lawnmower is capable of moving forward one square in which it is currently facing and mow the grass if any (mow), rotate left (left), and to jump to the new location that is specified as an argument with facing the same direction and mow the grass if any (frog). The operations *mow* and *left* do not take any arguments, therefore *mow* and *left* are terminals. The operation *frog* requires one argument that indicates the new location.

Two more operations are provided. One is named v8a and has two arguments that are summed, modulo 8. For example, (v8a, (3,4),(2,6)) returns (5,2). The operation, progn2, is the sequencing operation that returns the second argument's value. The complete function set for this problem is F = {frog, progn2, v8a} and the terminal set T = {(i,j), left, mow}.

A human programmer developing a program to solve this problem may decompose the entire problem into several subproblems, which are used repeatedly used to solve the entire problem. For example, a subprogram could be developed for mowing a row of grass, which could be invoked for each row.

Empirical evaluation

In this section we describe the empirical evaluation of the proposed system, including experiments used to arrive at a recommended set of parameters. Again, the new parameters of the system are

- The number of individuals in a generation to send to Subdue for evaluation
- The number of individuals that should receive the new ADF
- The frequency with which Subdue should be invoked
- The number of individuals in a generation

Because of the randomness involved in GP, we used the average number of generations it took the system to find a solution over 30 runs in the following comparisons.

Our first concern is the number of top ranking individuals to be sent to Subdue for an analysis. We experimented with sending 2%, 3%, 5%, 10%, 25%, and 50% of the individuals. Surprisingly enough, 2% and 3% worked best across a number of domains. In addition, when sending substantially more individuals to Subdue, not only did the performance gain diminish but Subdue's processing time dramatically increased.`

The number of individuals in the generation to receive the new ADF returned by Subdue was our next concern. By default, only the top individuals that were sent to Subdue always received the new SADF. In addition, we also experimented with adding the new SADF to an additional 30%, 50% and to all of the individuals, in the hopes that these individuals can also make use of the SADF. This might only occur, however, in subsequent generations as the evolutionary operators insert references to the new SADF. It might be beneficial if we forced low performing individuals to take advantage of the new SADF by randomly inserting a reference to it in the main branch.

Figure 5 shows the results, where the SADF is extracted from the top 3% of 1024 individuals, and inserted back into the percentage of individuals indicated on the horizontal axis (dashed line). In addition, we experimented with adding references to the new SADFs randomly in the main branch (solid line). As I can see, forcing low performing individuals to reference the SADF resulted in a performance increase of about two generations.

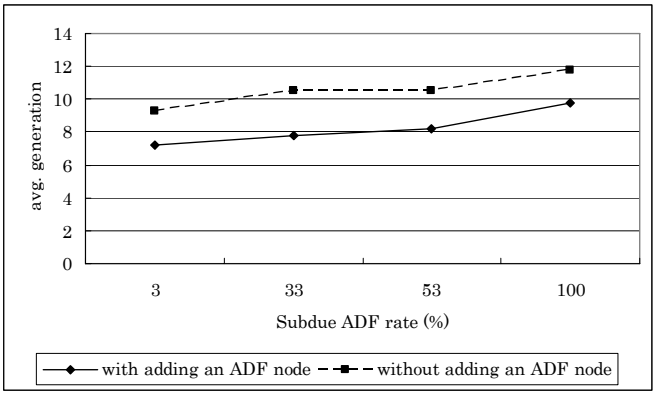

Figure 5 Average generation with various Subdue ADF rates (top 3%)

Next, we tested our hypothesis that very small ADFs, or ADFs with extremely few occurrences, may not help improve performance. We restricted the number of instances (number of times the subprogram is found in the top 2% of the population) to be >=0 (no restriction), >=5, >=10, >=20, and >=30. We restricted the number of vertices in Subdue-ADF >=0 (no restriction), >=4, and >=8. Figure 6 shows the average generation in which the ideal solution is found for each case discussed above.

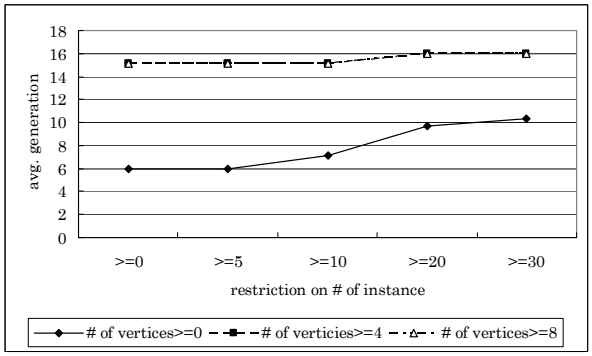

Figure 6 Average generation with restriction on size and frequency

As we can see, the best result occurs with no restriction on the number of vertices or the number of instances.

When the number of vertices was restricted, the results were very similar, and the two lines are impossible to distinguish on the plot. When no restrictions were imposed on the number of vertices, the performance increase was between six and nine generations. Apparently, even small and infrequent subprograms can have a big impact.

Next, we investigated the impact of not adding an SADF in each generation. We compared to the performance of the system when adding SADFs in every generation, every other generation and every third generation. Keeping the other parameters constant (2% passed to Subdue, 1024 individuals), we observed the following. As expected, the performance of the system decreased with skipping generations. The system converged in 7.5, 7.8 and 8.15 generations on average when adding SADFs in every generation, every other generation and every third generation, respectively. However, the amount of decrease was very small, especially when compared to the savings in processing time, which runs from about nine seconds to about four seconds. There was no difference in processing time when skipping one or two generations to

To test the effect the population size has on finding a solution, we tested our system with 32, 64, 128, 256, 512, and 1024 individuals. Our initial experiment, sending 2% of the top individuals to Subdue, did not work well for the lowest population sizes. Since 2% of 32 is less than 1, we had to adjust our approach about the number of individuals to be sent for analysis to Subdue. We have decided to fix the number of individuals at 20, which is 2% of 1024. As we can see in figure 7, this setting worked well for all population sizes. The figure also shows the average number of generations needed to find an optimal solution when no ADFs are used (original problem specifications by Koza, 1992).

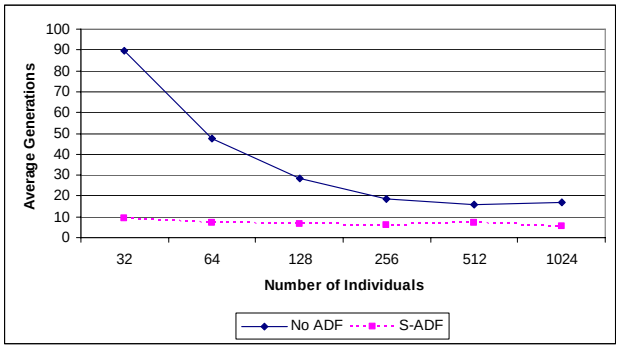

Figure 7 Performance with various population sizes

Interestingly enough our system's performance changes little with the number of individuals in the population. This is very encouraging because our system was able to converge to an optimal solution in 10 generations on average, even with only 32 individuals in the generation. This means that our system evaluated a significantly lower number of individuals compared to the original system (320 vs. 2880).

Conclusions and discussion

In this research we have demonstrated the validity of our hypothesis that extract useful code segments from successful individuals can improve genetic-programming. We performed a large number of experiments to empirically investigate recommended settings for the system, and have shown here a representative sample. The experiments are encouraging, and confirm our hypothesis.

It was interesting to see that although a decreased population size usually deteriorates the efficiency of GP, GP with Subdue-ADFs performs well even for a low number of individuals. We plan to investigate this finding further, and describe exactly the phenomena at work that makes this possible.

While the success of this approach will definitely vary across domains, the power of the approach comes from its ability to converge fast even with a small population size. Even though the execution of Subdue requires computation time, the additional processing time is made up many times over by having to evaluate fewer individuals in fewer generations. Therefore our approach has highly desirable characteristics, especially for complex problems that require a lot of time for the evaluation of individuals.

Even though we introduce additional parameters to the system, we eliminate the need for specifying the number of ADFs, and the overall program architecture, which in turn actually makes the system easier to use. Problem setup is the same setup as with no ADFs.

The experiments performed so far are definitely encouraging. Our plan is to pursue a theoretical evaluation to describe the conditions under which performance improvements will occur and explore the boundary conditions where gains will diminish. We also plan to investigate the use of tree mining algorithms instead of a graph mining algorithm.

References

Chittimoori, R.N., L.B. Holder, and D.J. Cook, “Applying the Subdue Substructure Discovery system to the Chemical Toxicity Domain”, University of Texas at Arlington, Arlington, TX, 1999.

Cook, D.J., and L.B. Holder, “Graph-Based Data Mining,” IEEE Educational Activities Department, IEEE Intelligent Systems March/ April 2000, pp.32-41.

Cook, D.J., L.B. Holder, S. Su, R. Maglothin, and I. Jonyer. “Structual Minining of Molecular Biology Data,” IEEE Engineering in Medicine and Biology, Special issue on Advances in Genomics, Vol. 20, No. 4, pp. 67-74. 2001.

Jassadapakorn, C., and P. Chongstitvatana, “Reduction of

Computational Effort in Genetic Programming by Subroutines", Chulalongkorn University, Bangkok, Thailand, 1998.

Jonyer, I., L.B. Holder, and D.J. Cook, "MDL-Based Context-Free Graph Grammar Induction," Proceedings of the Sixteenth Annual Florida AI Research Society, 2003.

Koza, J.R., "Genetic Programming," The MIT Press, Cambridge, MA, 1992.

Koza, J.R., "Genetic Programming II: Automatic Discovery of Reusable Programs," The MIT Press, Cambridge, MA, 1994.

Koza, J.R., "www.genetic-programming.org (a source of information about the field of genetic programming and the field of genetic and evolutionary computation)," 16 Sep. 2004. Retrieved 2 Feb. 2005 <http://genetic-programming.org>.

Luke, S., L. Panait, G. Balan, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop, "ECJ 12: A Java-based Evolutionary Computation and Genetic Programming Research System," Retrieved 14 Sep. 2004, <http://cs.gmu.edu/~eclab/projects/ecj/>

Nanduri, D.T., "Comparison of the Effectiveness of Decimation and Automatically Defined Functions," RMIT University, Melbourne, Australia, 2005.

Niimi, A., and E. Tazaki, "Object Oriented Approach to Combined Learning of Decision Tree and ADF GP," Toin University of Yokohama, Yokohama, Japan, 1999.

Srinivasa, S., S. Acharya, H. Agrawal, and R. Khare, "Vectorization of Structure to Index Graph Databases," Indian Institue of Information Technology, Bangalore, India, 2004.

Vanneschi, L., "Theory and Practice for Efficient Genetic Programming," Doctoral Dissertation. University of Lausannne, Italy. July 2004.