The 2015 IEEE RIVF International Conference on Computing & Communication Technologies
Research, Innovation, and Vision for Future (RIVF)

A New Implementation to Speed up Genetic Programming

Thi Huong Chu
Faculty of IT
Le Quy Don University
Hanoi, Vietnam
huongktqs@gmail.com

Quang Uy Nguyen
Faculty of IT
Le Quy Don University
Hanoi, Vietnam
quanguyhn@gmail.com

***Abstract*—Genetic Programming (GP) is an evolutionary algorithm inspired by the evolutionary process in biology. Although, GP has successfully applied to various problems, its major weakness lies in the slowness of the evolutionary process. This drawback may limit GP applications particularly in complex problems where the computational time required by GP often grows excessively as the problem complexity increases. In this paper, we propose a novel method to speed up GP based on a new implementation that can be implemented on the normal hardware of personal computers. The experiments were conducted on numerous regression problems drawn from UCI machine learning data set. The results were compared with standard GP (the traditional implementation) and an implementation based on subtree caching showing that the proposed method significantly reduces the computational time compared to the previous approaches, reaching a speedup of up to nearly 200 times.**

***Keywords*-Genetic Programming; Speed up; Fitness Evaluation.**

I. Introduction

Genetic programming (GP) [11], [16] is considered as a meta-heuristic based machine learning method, which induces a population of computer programs by evolutionary means. GP has successfully been used in generating computer programs for solving a numbers of problems from various fields. Specially, this method has been used to produce numerous human-competitive results [12]. However, GP evolves slowly with complex and high dimensional problems. This slowness is due to the fact that the solutions produced by GP must be evaluated on the training data using a fitness function.

In order to soften the slowness drawback of GP, many studies have been conducted to improve the execution time of this algorithm. In the early days of GP, most of the parallel approaches are based on the implementation over CPU machine clusters [5]. More recently, works about parallelization have been concentrated on using graphics processing units (GPUs) which provide fast parallel hardware for accelerating the evolutionary process of GP [8], [9]. Though these research have achieved significant success, helping to reduce GP execution time up to several hunderes times, their main shortcoming is that they require special hardware to be executed. In other words, these algorithms can only be implemented if CPU machine clusters or GPUs are available. However, this is not always true in reality.

Moreover, even when CPU machine clusters or GPUs are available, executing GP algorithm on these hardware is not a straightforward task. The reason is that GP often needs to be adapted before being able to run on these hardware systems. In some cases, the adaptation results in a much more complicated algorithm [3]. This may be an obstacle for the applicability of GP in practice. In this paper, we propose a new implementation of GP that is simple but efficient. The novel implementation is not based on the support from a special hardware. Thus, it can be implemented on standard personal computers that are by more far popular than CPU machine clusters or GPUs.

The remainder of the paper is organized as follows. In the next section, we present a background on GP model and analyzes the computational cost associated with its different phases. Section III provides an overview of previous works related to speeding up GP algorithm. Section IV presents the novel implementation to accelerate GP. The experimental settings are discribled after that. The results will be given and discussed in Section VI. Section VII concludes the paper and highlights some future work.

II. Background

In this section, we first give a brief introduction to genetic programming. The detail about GP evaluation stage is analyzed after that.

A. Genetic Programming

Genetic Programming (GP) was developed by Koza based on observations of biological systems [11]. It uses an abstraction of Darwin's natural selection mechanisms to evolve a population of solutions to problems [16]. It can also be seen as a machine learning method to optimize a population of computer programs to perform a given computational task. Following is the main steps in applying GP to solve problems:

1) Select a representation, a set of functions and terminals, and a fitness function for the problem.
2) Initialise a population of individuals.

3) Evaluate the fitness (how good) of the individuals in the population.
4) If the termination conditions have been reached, exit. Otherwise, go to step 5.
5) Choose a number of individuals (candidate solutions) using a certain selection method.
6) Apply a number of genetic operators on the selected solutions to generate a new population.
7) Repeat from step 3 to step 6.

In the first step, GP practitioners need to select an appropriate representation for individuals. Although, some other representations such as linear [14] or graph-based [13], tree-based representation is the most popular form and will be used in this paper. After that, a set of functions F and terminals T are identified. The function set $F = \{f_1, f_2, ..., f_n\}$ includes a number of functions with arity (number of possible arguments) greater than 0, whereas the terminal set $T = \{t_1, t_2, ..., t_m\}$ contains 0-arity functions or constants.

The first step in running a GP system is to generate, usually at random, an initial population of candidate solutions. This population is served as the starting point of the GP algorithm. The fitness evaluation function is then called to calculate the fitness value for each individual in the population. This fitness value presents the ability of the individual in solving the problem. The GP process is finished when the termination condition in step 4 is satisfied (e.g when a perfect solution is found or the maximum number of fitness evaluation is exceeded).

In step 5, the fitter solutions (the better solutions to the problem), based on fitness values, are selected using a selection method. Next, a new population is generated by applying a number of genetic operators to the chosen individuals. The main genetic operators are crossover, mutation and reproduction. The reproduction operation simply copies a selected individual to the next generation. The mutation operator adds new genetic material to the population by modifying the individual while the crossover operation generates two new individuals by combining two old individuals. More detail discussion about Genetic Programming can be found in [16], [11].

B. Fitness Evaluation

Although, there are several steps in running a GP system, the previous experiences have proven that on average more than 90% of the time is taken by the evaluation stage [1]. For a standard version of GP, if the population size and the number of samples in the training data set (also called the number of fitness cases) have been determined, then the evaluation stage is often implemented as the pseudo-code in Algorithm 1.

Algorithm 1: Evaluation stage of GP

```
Input: population_size, number_fitness_cases
for each individual st within the population do
    for each instance x from the dataset do
        Compute(st,x)
```

It can be seen from Algorithm 1 that this implementation requires two loops to complete the evaluation stage. In this algorithm, the produce $Compute(st,x)$ is a function that executes individual *st* with an input sample (a fitness case), *x*. This function will return an output value, t. In a real-valued symbolic regression problem, t is usually a real number. This value will be compared to the target value of the input sample in the training data set to know the quality of individual st. In most GP system, the $Compute()$ fuction has the sketch as in Algorithm 2.

Algorithm 2: Evaluation of an individual with an instance from the training data set

```
type Compute(st,x)
{
  If st is a leaf node then
  {
    return x;
  }
  Else
  {
    p=Operator(st)
    q=Compute(st.children,x)
    return p(q)
  }
}
```

Where $type$ is the data type that is returned by this function. For real-valued regression problems and in the popular languages like C and Java, this will often be the type for real numbers (double or float). Function $Operator(st)$ returns the operator at the current processing node and function $q(q)$ will apply this operator to the result obtained by the children of st. It should be noted that this is a recursive function. This recursive function for evaluating an invididual in GP has been implemented in many GP packages including a widely used one like ECJ [19].

III. Related Work

As already mentioned, one of the main shortcomings of GP is that it often requires significant amount of time to evolve on the training data. In order to extend the applicability of GP, diverse techniques have been proposed to reduce GP execution time. These techniques can roughly be categorized into three classes [16]:

1) Reducing number of fitness evaluations
2) Using caching methods
3) Paralleling GP

The first of these approaches is motivated by the fact that if a program has already shown it is inferior to the rest of the population on a fraction of the available training data, it not likely to be selected as a parent in the selection stage. Therefore, in most application, it is not necessary to evaluate a the program on all fitness cases. In other words, only a carefully selected subset of fitness cases can be used to measure the quality of each program without any major impact to the result of the whole system. This method has been used by some researchers and achieved significant success in reducing GP evolution time [17], [7].

The second approach has been to lower evaluation costs by reducing the number of node evaluations that need to be performed. In many GP systems, subtrees have no side-effects. Thus, if the inputs and output of a subtree are remembered when it was run before, they can be used to avoid re-executing code whenever the subtree is required to run again. Moreover, in GP, due to consistent copying of subtrees by the crossover operator, it is likely that many subtrees will occur multiple times in the population. Hence, if a scheme can be used to cache these subtrees to avoid re-evaluate them, it is often possible that GP speed will be increased. Some researchers have used different methods such as hash function [20] or Directed Acyclic Graph [10] for caching subtrees leading to a significant decrease in GP running time.

Perhaps, the most popular method to accelerate GP is based on paralleling techniques. These methods are supported by a fast growing in computing power of computer hardware. There are two important methods for paralleling evolutionary algorithms which are often used by GP researchers. The first is the traditional method for parallel computing. That is, an existing algorithm is sent to a supercomputer so that it runs faster [15], [5]. The second aspect comes from the biological inspiration for evolutionary computation: in nature everything happens in parallel. In other words, GP population is divided into enormous sub-populations and each of these sub-population may be evaluated in different process units such as CPU cores or graphics processing units (GPUs) [3], [8].

Due to the fast increase of computer power, these methods have been achieved a noticeable success with the speedup of up to hunderes to thousand times [4]. However, they also have some downsides such as they requires the support from special hardware and the algorithms often need to alter to be concurrently executed on different processing units. Subsequently, not every GP practitioner implements these methods in their applications. In the next section, we will propose a new implementation of GP that avoids to change the structure of the algorithm and can be implemented on normal personal computers. This means that the proposed algorithm does not require CPU clusters or GPUs to be accelerated.

IV. Methods

This section presents the new implementation proposed in this paper. The idea for this implemtation has been briefly discussed in [10] where Keijzer proposed some schemes for caching subtrees in GP. Nevertheless, there was not any concrete implementation proposed in [10]. Moreover, experiments were also not conducted to investigate the effectiveness of this method in Keijzer research [10]. In this section, we will propose a concrete implementation for this method and conducting a thorough experiment to examine its impact.

The motivation for this method is due to the fact that in most implementation of GP so far, evaluation is done on a case by case basis. In other words, each fitness case is often independently interpreted by the GP parse tree. Thus, given an individual consisting of M nodes, evaluated over N fitness cases, the number of interpretation steps for this individual is *MxN*. Since each interpretation employs a switch or jump-table to find out which function to call, the fully interpreted GP system will incur the overhead of the switch or lookup and subsequent function call for each node. This can be one of the reason that explains for the slowness of GP.

However, if the loop is reimplemented so that the evaluation works on the full array of fitness cases for each node, it is feasible to reduce the number of interpretation steps to M. This implementation is called *vectorized evaluation*. By reducing the number of interpretation steps by a factor N, the runtime of a GP system can be significantly lowered. In effect, when performing vectorized evaluation, the overhead of the evaluation stage is independent of the number of fitness cases of the problem.

In order to implement vectorized evaluation scheme, several modifications are imposed on the fitness evaluation stage of GP. First, the evaluation stage in Algorithm 1 is changed to Algorithm 3. It can be seed that in this algorithm, there is only one loop instead of two loops as in Algorithm 1. Furthermore, the input to the $Compute()$ function has only one parameter (the genome of the current individual, st). This is due to that the genome interpreter in this implementation is independent of the number of fitness cases.

Algorithm 3: Evaluation stage of GP in new implementation

```
Input: population_size
for each individual st within the population do
    Compute(st)
```

Second, the structure of the $Compute()$ function also needs to be altered. It is sketched in Algorithm 4. It can be observed that in this implementation the function has an input: the array of fitness cases (Fitness_Cases[]). Moreover, the result of this function is a vector rather than a value as in the previous implementation. Inside the function, an array result[] is allocated to contain the result of the evaluation process for all fitness cases of the problem. This implementating helps to reduce the number of genome interpretations for each individual to only 1 without depending on the number of fitness cases of the problem. In the following section, the impact of this method to the speed of GP will be experimentally examined.

V. Experimental Settings

In order to measure the impact of the vectorized evaluation method to the speed of GP, we tested this implementation on

Algorithm 4: The structure of the Compute function in the new implementation

```
Input: Fitness_Cases[]
type[] Compute(st)
{
   type result[]
   If st is a leaf node then
   {
      result[]=Fitness_Case[]
   }
   Else
   {
      p=Operator(st)
      q=Compute(st.children)
      result[]= p(q)
   }
   return result[]
}
```

TABLE II
RUN AND EVOLUTIONARY PARAMETER VALUES.

Parameter	Value
Population size	500
Generations	50
Selection	Tournament
Tournament size	3
Crossover probability	0.9
Mutation probability	0.1
Function set	+, -,*,/,sin,cos,exp,log
Terminal set	X_1,X_2,...,X_N
Initial Max depth	6
Max depth	15
Max depth of mutation tree	5
Raw fitness	mean absolute error
Trials per treatment	30 independent runs

ten multivariate regression problems. All these problems were taken from UCI machine learning dataset [1]. They are detailed in Table I.

The GP parameters used for our experiments are shown in Table II. The terminal set for each problem includes N variables corresspoding to the number of attributes of that problem. The raw fitness is the mean of absolute error on all fitness cases. Therefore, smaller values are better. For each problem and each parameter setting, 30 runs were performed.

We compared the method in this paper with the standard implementation of GP (standardGP) and a method for caching subtree semantics in GP [18]. In [18], semantics (the output) of every subtree in an individual is stored by using the attributes. These attributes are then used to speed up the evolutionary process of GP: if an individual is crossed over, only nodes from the crossover point to the root are reevaluated, other nodes are not reevaluated since they are not changed. More detailed description about this caching method can be found in [18]. The comparative results of three methods are presented in the following section.

[1]http://archive.ics.uci.edu/

VI. RESULTS AND DISCUSSION

Two metrics were used to measure the performance of the above techniques. The first metric is the average running time measured in seconds. For each run, we recored the amount of time needed to complete that run (after finishing the last generation). These values were then averaged over 30 runs and the results of three methods are shown in Table III.

TABLE III
AVERAGE RUNNING TIME IN SECONDS OF THREE METHODS

Problems	StandardGP	CachingGP	VectorizedGP
F_1	480	23	6.1
F_2	550	28	4.3
F_3	109	10	2.5
F_4	364	83	11
F_5	552	31	5.7
F_6	903	51	10
F_7	384	24	5.5
F_8	482	32	6.2
F_9	623	40	4.2
F_{10}	836	40	4.2

It can be seen from this table that both methods (caching and vectorized evaluation) help to speed up the evolutionary process of GP. However, the scale of improvement is different. While caching method only reduce the average running time of GP to about 20 times, the vectorized evaluation method decrease these values to about 100 times. Particularly, on problem F_{10}, the vectorized evaluation even helps to speed up GP to nearly 200 times. We also statistically tested the significance of the results in Table III using a Wilcoxon signed rank test with a confidence level of 95%. The statical test shows that all the different between the caching method versus standard GP and the vectorized evaluation method versus both the caching method and standard GP are significant. These results show that using the new implementation in this paper may potentially help GP practitioners safe a significant amount of time in running GP for their applications (from 100 to 200 times).

The second metric used to evaluate the effectiveness of these methods are the memory space needed to execute GP. Since, the memory space allocated for GP will vary during the evolutionary process, we only recorded the highest values (in Megabytes) of each run out of 30 runs. These values are presented in Table IV. It can be observed from this table that while the caching method needs much more memory space to

TABLE I
SYMBOLIC REGRESSION PROBLEMS.

Shorthanded	Name	Number of Fitness Cases	Number of Attributes
F_1	airfoil self noise	600	6
F_2	Housing	506	14
F_3	slump test Compressive	103	8
F_4	3D spatial network	450	4
F_5	Concrete Compressive Strength	480	9
F_6	Protein Tertiary Structure	470	10
F_7	yacht hydrodynamics	380	7
F_8	Istanbul stock exchange	536	8
F_9	wine quality white	900	12
F_{10}	wine quality red	900	12

TABLE IV
THE HIGHEST VALUE OF MEMORY SPACE OF THREE METHODS

Problems	StandardGP	CachingGP	VectorizedGP
F_1	21.5	855	21.7
F_2	21.2	790	21.5
F_3	21.4	268	21.7
F_4	30.1	1059	28.2
F_5	31.2	1027	29.3
F_6	34.2	1056	28.5
F_7	21.3	1048	32.5
F_8	26.2	986	21.6
F_9	21.5	1237	21.6
F_{10}	21.5	1259	21.9

be executed and this may risk the out of memory error when the problems become more complicated (more fitness cases and more attributes), the vectorized evaluation method does not incur the increase of the memory when running GP. In fact, the memory space allocated for the vectorized evaluation method is mostly equal to the standard GP. This is an important property that allows the proposed method can be implemented in normal personal computers.

VII. CONCLUSIONS

In this paper, we proposed a novel implementation to speed up the evolutionary process of GP. The motivation for this method is to interpret a GP genome based on an array of fitness cases rather than single case as in the traditional implementation. The proposed method was tested on a number of regression problems drawn from UCI machine learning data set. The experimental results showed that the new implementation helps to reduce the average running time of GP up to nearly 200 times. Moreover, constract to the caching method, this method does not increase the memory space when GP is executed. Thus, the new method may possibly be widely used by GP practioners on standard personal computers in solving real-world applications.

In future, we are planning to extend the work in this paper in a number of ways. First, we will test the new implementation on more complex problems especially the problems in reality like time series forecasting or network security [6], [2]. In these problems, the training data set is often much bigger. Consequently, this will take much longer for GP to find solutions for these difficult tasks. Second, it will be very interesting to measure the performance of this method on classification problems where it has been proven that the fitness evaluation process is even slower [1]. Last but not least, the proposed method does exclude other parallel GP algorithms on GPUs, therefore, in the future, we are aiming to implement this method on GPUs to see if this helps to speed up GP to a further extent.

ACKNOWLEDGMENT

The work in this paper was funded by The Vietnam National Foundation for Science and Technology Development(NAFOSTED), under grant number 102.01-2014.09.

REFERENCES

[1] A. Cano, A. Zafra, and S. Ventura. Speeding up the evaluation phase of GP classification algorithms on GPUs. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 16(2):187–202, Feb. 2012.

[2] E. Carreno Jara. Long memory time series forecasting by using genetic programming. *Genetic Programming and Evolvable Machines*, 12(4):429–456, Dec. 2012.

[3] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 2007.

[4] D. M. Chitty. Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing*, 16(10):1795–1814, Oct. 2012.

[5] B. Chopard, O. Pictet, and M. Tomassini. Parallel and distributed evolutionary computation for financial applications. *Parallel Algorithms and Applications*, 15:15–36, 2000.

[6] G. Folino, C. Pizzuti, and G. Spezzano. An ensemble-based evolutionary framework for coping with distributed intrusion detection. *Genetic Programming and Evolvable Machines*, 11(2):131–146, June 2010. Special issue on parallel and distributed evolutionary algorithms, part II.

[7] C. Gathercole and P. Ross. Dynamic training subset selection for supervised learning in genetic programming. In *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321. Springer-Verlag, 9-14 Oct. 1994.

[8] S. Harding and W. Banzhaf. Fast genetic programming on GPUs. In *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101, Valencia, Spain, 11-13 Apr. 2007. Springer.

[9] S. L. Harding and W. Banzhaf. Distributed genetic programming on GPUs using CUDA. In *Workshop on Parallel Architectures and Bioinspired Algorithms*, pages 1–10, Raleigh, NC, USA, 13 Sept. 2009. Universidad Complutense de Madrid.

[10] M. Keijzer. Alternatives in subtree caching for genetic programming. In *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 328–337. Springer-Verlag, 2004.

[11] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, MA, 1992.

[12] J. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284–, 2010.

[13] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[14] M. O'Neill. *Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution*. PhD thesis, University Of Limerick, Ireland, Aug. 2001.

[15] S. Openshaw and I. Turton. Building new spatial interaction models using genetic programming. In T. C. Fogarty, editor, *Evolutionary Computing, AISB workshop*, Leeds, UK, 11-13 Apr. 1994.

[16] R. Poli and W. L. N. McPhee. *A Field Guide to Genetic Programming*. http://lulu.com, 2008.

[17] A. Teller and D. Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328. Morgan Kaufmann, 13-16 July 1997.

[18] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and D. N. Phong. On the roles of semantic locality of crossover in genetic programming. *Information Sciences*, 235:195–213, 20 June 2013.

[19] D. R. White. Software review: the ECJ toolkit. *Genetic Programming and Evolvable Machines*, 13(1):65–67, Mar. 2012.

[20] P. Wong and M. Zhang. SCHEME: Caching subtrees in genetic programming. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 2678–2685, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.