Enhanced Generalized Ant Programming (EGAP)

Amirali Salehi-Abari
School of Computer Science
Carleton University
Ottawa, Canada
asabari@scs.carleton.ca

Tony White
School of Computer Science
Carleton University
Ottawa, Canada
arpwhite@scs.carleton.ca

ABSTRACT

This paper begins by reviewing different methods of automatic programming while emphasizing the technique of Ant Programming (AP). AP uses an ant foraging metaphor in which ants generate a program by moving through a graph. Generalized Ant Programming (GAP) uses a context-free grammar and an Ant Colony System (ACS) to guide the program generation search process. There are two enhancements to GAP that are proposed in this paper. These are: providing a heuristic for path termination inspired by building construction and a novel pheromone placement algorithm. Three well-known problems -- Quartic symbolic regression, multiplexer, and an ant trail problem -- are experimentally compared using enhanced GAP (EGAP) and GAP. The results of the experiments show the statistically significant advantage of using this heuristic function and pheromone placement algorithm of EGAP over GAP.

Categories and Subject Descriptors

I.2.2 [**Computing Methodologies**]: Artificial Intelligence – *program modification, program synthesis.*

I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence – *Intelligent agents.*

General Terms

Algorithms, Performance, and Design.

Keywords

Automatic Programming, Ant Programming, Heuristic, Generalized Ant Programming, Enhanced Generalized Ant Programming.

1. INTRODUCTION

Automatic programming is an active research area that has stimulated by the Genetic Programming (GP) technique. In automatic programming, the goal of the desired program is first specified; then, based upon this goal, programs are generated according to an algorithm and tested to demonstrate to what extent they satisfy the desired goal. Genetic programming (GP), a method of automatic programming, was proposed by Koza [9-11]. GP utilizes an idea similar to that of a genetic algorithm (GA) but with representational and operator differences. GP represents genes in a tree structure as opposed to an array of numbers typically used in a GA. As a consequence of this representation, there are some other differences in the mutation and crossover operator of GP in comparison to GA.

According to Koza [9], there are five preparatory steps which should be completed before searching for a program: selecting of terminal symbols, choice of functions, fitness function specification, selection of certain parameters for controlling the run, and defining the termination criteria. Hence, an automatic programming approach can be any search method which has the ability to do these five steps before searching.

While search algorithms inspired by evolution have demonstrated considerable utility, other learning models are attracting increasing interest. One model of social learning recently attracting increasing attention is Swarm Intelligence (SI). There are two main classes of algorithm in this field: ant colony system (ACS) and particle swarm optimization (PSO) [1]. The former is inspired by the collaborative behavior of ants in finding food. The latter is derived from the flocking behavior of birds and fish and is often utilized in optimization problems. Both ACS and PSO exhibit flexibility, robustness and self-organization [1].

ACS and PSO have been used in Automatic Programming. O'Neill and Ryan present an automatic programming model called Grammatical Swarm (GS) [12, 13]. In this model, each particle or real value vector represents choices of program construction rules specified as production rules of a Backus-Naur Form (BNF) grammar. In other words, each particle shows the sequence of rule numbers by applying which a program can be constructed from the starting symbol of the grammar. GS is based on the linear Genetic Programming representation adopted in Grammatical Evolution (GE) [14] that uses grammars to guide the construction of syntactically correct programs.

O'Neill and Ryan describe several advantages of a grammatical approach to genetic programming [14], including the ability to encode multiple data types into the solutions generated as opposed to tree-based GP in which type information is typically absent. That is, the non-terminal symbols within a grammatical approach provide the capability for developing programs with multiple data types like strings, integers, booleans and so on. Moreover, it is simply possible to encode the knowledge domain into the grammar which can be employed to bias the construction of solutions and also any changes to the language of constructed program easily can be made by modifying the grammar. Furthermore, this approach is language independent.

Other researchers have used ACS for automatic programming. Roux and Fonlupt [15] use randomly generated tree-based programs. A table of program elements and corresponding values of pheromone for these elements is stored at each node. Each ant builds and modifies the programs according to the quantity of an element's pheromone at each node. The higher concentration of pheromone one element has, the higher probability it has for selection. This approach met with limited success.

Boryczka and Czech have presented two other models of Ant programming [3, 4, 5 and 6]. They used their model only for symbolic regression. In the first approach called the expression approach, they search for an approximating function in the form of an arithmetic expression written in Polish (Prefix) notation. They create a graph whose nodes are either an arithmetic operator or a variable. When an ant goes through these nodes and edges, the expression is constructed by selecting the nodes visited. Ants put pheromone on the edges based on the fitness of expression in order to lead other ants to the specific solutions. In the second approach, the desired approximating approach is built as a sequence of assignment instructions which evaluates the function. In other words, there is a set of assignment instruction defined by the user; each of these assignment instructions is placed on a node of graph. Then, ants build their program by selecting the sequence of these instructions while passing through the graph.

Both expression and instruction approaches showed promise but they are only applicable to regression or function approximation, they cannot generate more general types of program. Another drawback of these approaches is the existence of introns in the resulting program. Some solutions for these introns are presented by Boryczka [7].

Keber and Schuster offer a new AP model using a context-free grammar and an ant colony system. They use this model in function approximation for the purpose of option valuation [8]. They called it Generalized Ant Programming (GAP) because they believe that it is applicable to all problems in which the search space of solutions consists of a computer program. In spite of their claim, they did not test this approach for anything except symbolic regression. The lack of termination condition for generating the path by each ant and generating paths with non-terminal components cause GAP to have the weak performance in some domains.

The main contributions of this paper are the introduction of a new heuristic function for program generation and a different method of pheromone placement for GAP. We have compared the performance of the new algorithm with GAP on 3 problems: Quartic symbolic regression, multiplexer and Santa Fe ant trail. The results obtained demonstrate a statistically significant improvement.

The remainder of the paper is structured as follows. In section 2, the GAP algorithm is presented in some detail. Section 3 highlights areas for improvement in GAP and describes the enhanced GAP (EGAP) algorithm. Section 4 details the experimental approach adopted and results. Finally, Section 5 provides conclusions and opportunities for future work.

2. GENERALIZED ANT PROGRAMMING

2.1 Introduction

Generalized Ant Programming (GAP), introduced by Keber and Schuster, is a new method of Automatic Programming. This method is inspired by GP and ACS. GAP is an approach designed to generate computer program by simulating the behavior of ant colonies; specifically, reinforcement through pheromone deposition. That is, when ants forage for food they lay pheromone on the ground that affects the choices they make. Ants have a tendency to choose steps that have a high concentration of pheromone. Pheromone trails can be seen as common information that is modified by ants to show their experience while solving a given problem.

2.2 Methodology

GAP uses artificial ants to automatically generate computer programs. By analogy to real ants, artificial ants explore a search space including the set of all feasible computer programs. The ant generates a program by moving along a specific path in the graph. The amount of pheromone deposited by an ant is proportional to the quality of the solution found by that ant. The quality of a path is measured using the fitness function which will be described in the next few paragraphs.

All computer programs are written in a well defined programming language. In GAP, $\mathcal{L}(\mathcal{G})$ is the programming language in which an automatically generated program is written and it is specified by the context-free grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}erm, \mathcal{R}, \mathcal{S})$. In other words, $\mathcal{L}(\mathcal{G})$ is a set of all expressions that can be produced from a start symbol $\mathcal{S}$ under application of R rules, a set of non-terminal symbols $\mathcal{N}$, and a finite set of terminal symbols, $\mathcal{T}$. Thus,

$$\mathcal{L}(\mathcal{G}) = \{\mathcal{P} \mid \mathcal{S} \Rightarrow \mathcal{P} \wedge \mathcal{P} \in \mathcal{T}erm^*\} \qquad (1)$$

Where $\mathcal{T}erm^*$ represents the set of all expressions that can be produced from the $\mathcal{T}$erm symbol set. Given the grammar $\mathcal{G}$, a derivation of expression $\mathcal{P} \in \mathcal{L}(\mathcal{G})$ consists of a sequence of $t_1, t_2, \dots, t_p$ of terminal symbols generated from the sequence of derivation steps. This derivation is denoted by

$$\mathcal{S} \overset{*}{\Rightarrow} \mathcal{P} \qquad (2)$$

Assume the following $\mathcal{G}$

$$\mathcal{G} = (\mathcal{N} = \{S, T, F\},$$
$$\mathcal{T}erm = \{a, +, *, (,)\},$$
$$R = \{S \to S + T | T, T \to T * F | F, F \to (S) | a\},$$
$$\mathcal{S} = \{S\})$$

Each derivation in this grammar represents a simple arithmetic expression including the symbols $a, +, *, ($, and $)$. The simple derivation of this grammar is presented below:

$$S \Rightarrow S + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F$$
$$\Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a$$

In GAP, $\mathcal{L}(\mathcal{G})$ is the search space of all possible expressions (programs) that can be generated by the grammar $\mathcal{G}$, $\mathcal{P} \in \mathcal{L}(\mathcal{G})$ is a path which can be visited by one ant and it is an expression (a program) and $\mathcal{J}(t) \subset \mathcal{L}$ is a set of all paths already visited at time t. Furthermore, each path $\mathcal{P} \in \mathcal{L}(\mathcal{G})$ consists of a sequence of terminal symbols $t_1, t_2, \dots, t_p$ and the corresponding derivation step $t_i \to t_{i+1}$ $(for\ i = 1, \dots, p-1)$ that cause the generation of the terminal symbols $t_1, t_2, \dots, t_p$ from the start symbol S. Thus, each path $p_i \in \mathcal{J}(t)$ can be seen as a derivation:

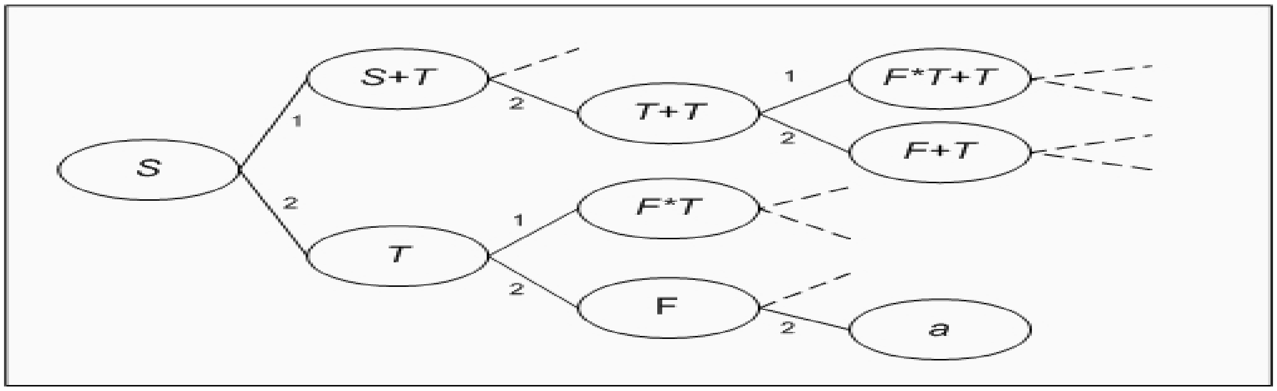

Figure 1. Part of the tree which ants explore to generate the expression (program)

$$S \overset{*}{\Rightarrow} p_i \qquad (3)$$

Let us return to the previous example to illustrate these explanations further. Suppose that an ant is in the starting symbol S, then it will have two choices, either select the first rule $S \to S + T$ or the second rule $S \to T$. By selecting the first rule, the ant‘s expression will be $S + T$ and the ant should now seek to make a substitution for S again; this process continues until the complete path is formed in the graph. It is worth noting that the graph in this model is a tree. Figure 1 shows the starting part of this tree.

In Figure 1, the text inside each oval shows the current expression which ants have generated by moving through the tree and selecting the corresponding rules. The numbers on the edges show the rule's number which is applied for the first non-terminal symbol in the previous oval. For instance, to create the expression a , the ant should first select the second rule of S then the resulting expression will be T and then by selecting the second rule of T, i.e. $T \to F$, the expression is changed to F and finally the second rule of F $(F \to a)$ will result in the expression a. Considering another example, the sequence rules of "1-2-1" can lead the expression of an ant to become $F * T + T$. Thus, the sequence of numbers determines the final expression of an ant.

Ants select their path in this tree according to the amounts of pheromone deposited by the other ants on the tree. In GAP, all derivation steps in the path get an equal amount of pheromone while an ant puts pheromone on the path.[1]

The amount of deposited pheromone is stored in the T table. The T table is a hash table that consists of the string key indicating the rule numbers that have been selected by ants to reach this edge and also consist of the amount of pheromone deposited on this edge. For instance, in Figure 1, for the edge between oval F and oval a, the "2-2-2" key can be used. To lookup the amount of pheromone for one edge, if the key of that edge exists in the T Table, the corresponding value will be returned otherwise the initial value of pheromone (T_0) will be returned.[2]

The amount of pheromone in T table at time t is update by:

[1] In section 3, the proposed method discriminates the amount of pheromone in each edge.

[2] These explanations about T table are not mentioned by the GAP algorithm.

$$T(t) = (1 - p) * T(t - 1) + \Delta T(t) \qquad (4)$$

Where $0 < p \leq 1$ is the coefficient representing pheromone evaporation, and

$$\Delta T(t) = \sum_{k=1}^{K} \Delta T_k(t)$$

is the pheromone increase obtained by accumulating the contributions $\Delta T_k(t)$ of each ant $k = 1, \ldots, K$. In other words, this is the amount of pheromone deposited on some edges of tree by k ants at time t. This quantity of pheromone is given by:

$$\Delta T_k(t) = \begin{cases} Q.L_k(t,p) & if \text{ k}^{\text{th}} \text{ ant pass edge e} \\ 0 & otherwise \end{cases}$$

Where Q is an experimental constant and $L_k(t,p)$ is the value of the objective function obtained by ant k at time t. The expression found by ants can be seen as a function $p: \mathcal{E} \to A$ transforming input data $\mathcal{E}$ into a solution or output data A. Therefore, the function $L_k: A \to \mathbb{R}$ is defined in a way that it awards higher values to those paths (programs) that represent a good solution to the task, and lower values to a less suitable program.

As mentioned previously, ants choose their path in the tree based on the amount of pheromone deposited on the tree, the formula[3] below gives us the probability of each edge to select:

$$P_e^k(t) = \frac{[T_e(t)]^\alpha . [\eta_e]^\beta}{\sum_{e \epsilon C(n')} [T_e(t)]^\alpha . [\eta_e]^\beta} \qquad (5)$$

Where $P_e^k(t)$ is the probability of selecting the edge e and $T_e(t)$ is the amount of pheromone deposited on the edge e and η_e is a heuristic value related to the selection of the edge e. The $C(n')$ is the candidate set, the edges which can be selected when the ant is on the node n'. The experimental parameters α and β control the relative importance of pheromone trail versus heuristic function.

The pseudo code for GAP can be described as follows:

```
[0] **Program** Generalized Ant Programming
[1]     t = 0;
[2]     Initialization ();
[3]     **repeat**
[4]         t = t + 1;
```

[3] This formula is different with the one GAP present. They didn't explain enough about their formula.

```
[5]         for each ant k do
[6]             Build a path p;
[7]             Calculate $L_k(t,p)$
[8]         end;
[9]         Save the best solution found so far;
[10]        Update trail levels $T(t)$;
[11]        Shrink $T(t)$;
[12]        Perform global shaking on $T(t)$;
[13]    until termination;
[14] end.
```

While path creation and pheromone level updates have been discussed, the Shrink $T(t)$ and Shaking $T(t)$ procedures require explanation. Shrink $T(t)$ is used to decrease the size of the T table. In this phase, all the keys that have a value less than the initial value of T_0 ($T_0 \geq 0$) are eliminated. The reason that these edges get values less than T_0 is the existence of evaporation. Shaking $T(t)$ is used to normalize the amount of pheromone on the edges. This method is used because the pheromone on a derivation step (an edge of tree) becomes higher than all others, then the other will not be selected by ants so the ants will not explore properly the search space. The formula GAP is using in this regard is logarithmic one and is given by:

$$T(t) = T_0 * \left[1 + \ln\left(\frac{T(t)}{T_0}\right)\right]$$

Where T_0 is the minimum value for $T(t)$.

3. Enhanced GAP (EGAP)

3.1 Introduction

Although section 2 represents a GAP algorithm that is more easily implementable, there is an important issue that is not addressed by the GAP algorithm [8]. This is, what is the termination condition for generating the path by each ant or when the path is complete? The simple solution to this problem is that the path is complete if it doesn't have any non-terminal symbol in its expression. The other question is under what conditions can an ant seek a path that hasn't any non-terminal symbol in it? The simple answer to this latter question is providing an upper bound on path length. However, this is not the end of the problem as we need to assign a fitness value for the paths which have a non-terminal symbol in their expression and are not still a complete computer program. When there are recursions in the rules of grammar, this issue becomes more significant. Consider the following grammar rules:

$$< expr > \rightarrow < expr >< op >< expr > \mid \; < var >$$
$$< op > \rightarrow * \mid - \mid + \mid / \qquad (6)$$
$$< var > \rightarrow a$$

The starting symbol in this grammar is $< expr >$ and the first rules of this symbol ($< expr > \rightarrow < expr >< op >< expr >$) generate an expression that has two recursions to $< expr >$. Using GAP for this grammar will produce poor results[4]. We observed that the expressions usually generated are either just 'a' or very long, incomplete and unexecutable expressions.

[4] The experiment described in section four demonstrates the weakness of GAP when using this grammar.

This observation prompts us find a fundamental remedy for this issue. The basic idea of this work described in this paper is to encourage ants to build their path in more well-structured way.

In order to motivate the proposed GAP enhancements, consider building construction. First, civil engineers plan the foundation, cornerstone and structure of building rather than the internal decoration and appearance of the building. After that, architects will consider and think what kind of internal decoration, texture and lighting is suitable for this structure in order to achieve an aesthetic goal. As a result of this cooperation, the building will be well-structured and well-designed.

Drawing an analogy to building construction, the research described here introduces a heuristic function and a new ant pheromone placement method in order to encourage ants to first *build* a good solution structure and then *tune* it. Similar to real programming, programmers exhibit the same behavior; they first design the schema of their program; for instance, they first consider where they will have a loop, if or switch structure and then they think about the conditions and parameters of these structures.

3.2 Heuristic Function

The heuristic function is designed to have ants expand the expression for a fraction of the maximum number of allowed rules and then select completion rules for the remainder. *Maximum number of using rules* is a constant specified by the user to limit the total number of rules which an ant can select to generate its own expression (program). Expression construction has two phases: expanding the expression (similar the task which civil engineers do) and completing and tuning the expression (similar to the task which an architect does). The first phase will be performed in a fraction of *maximum number of using rules* and the second phase will be done in the remainder.

From this perspective, the rules of a grammar fall into two categories: expanding rules and finishing rules. Expanding rules tend to expand the expression by producing some other non-terminal symbol as opposed to finishing rules which have a tendency to replace the non-terminal symbols of the expression with terminal ones. We present an expanding factor (f_e) that shows to what extent a rule is an expanding rule. High values of f_e demonstrate the high probability of being an expanding rule while low values shows the high probability of being finishing rule.

In order to clarify the above statements, in the grammar shown in (6), $< expr > \rightarrow < expr >< op >< expr >$ has a high value of (f_e) as it has more tendency to expand the expression than the others whereas $< var > \rightarrow a$ has a low value of f_e and it is more a finishing rule. Not only can the rules have an expanding factor but also the non-terminal symbols have expanding factor related to their rules' expanding factors.

To calculate f_e for all the rules, we suggest an iterative algorithm. This algorithm first initializes the expanding factor (f_e) of all the rules and non-terminal variables with a large value. Then it updates the expanding factor of each rule during every iteration. Each rule adds together the expanding factor of the non-terminal symbols that it generates and finally adds them with 1. Each non-terminal variable updates its expanding factor by calculating the average over all of its rules. The update formula is:

$$f_e(x,i) = \left[\sum_{y \in nt} f_e(y,0)\right] + 1 \qquad i = 1 \dots N \qquad (7)$$

$$f_e(x,0) = mean\, f_e(x,i) \qquad i = 1 \dots N$$

Where $f_e(x,i)$ is the expanding factor of non-terminal symbol *x* for its i^{th} rule and nt is the set of all non-terminal symbols included in that specific rule (i^{th} rule). $f_e(x,0)$ is the expanding factor of the non-terminal symbol *x*. For example, suppose:

$$F \rightarrow aaSabT \mid aa \mid Fa$$

In this rule, S and T are non-terminal symbols while a and b are terminal symbols. Then, f_e for rules related to F is updated by:

$f_e(\text{"F"},1) = f_e(\text{"S"},0) + f_e(\text{"}T\text{"},0) + 1$

$f_e(\text{"F"},2) = 1$

$f_e(\text{"F"},3) = f_e(\text{"F"},0) + 1$

And f_e of F is updated by:

$f_e(\text{"F"},0) = \text{mean}\,(f_e(\text{"F"},1), f_e(\text{"F"},2), f_e(\text{"F"},3))$

The pseudo code of assignment of f_e is presented below:

```
[0]  Initialize $f_e(x,i)$ with a high value for all x and i.
[1]  for $i = 1$ to number of non-terminal symbols
[2]      for each x belonging to non-terminal symbol set
[5]          Update $f_e(x,i)$ according to (7);
[6]      end
[7]  end
```

Returning to the heuristic function explanation, the heuristic function should be proportional to f_e when the ant is in the first phase of path generation (the expanding phase) and should be reversely proportional to f_e when the ant is in the second phase (the finishing phase). Then

$$H(x,i,n) \propto f_e(x,i) \quad \text{if } n < t_n$$
$$H(x,i,n) \propto 1/f_e(x,i) \quad \text{if } t_n < n < \text{maxN} \qquad (8)$$

Where *x* is a non-terminal symbol and *i* is an index of *x*'s rules. Furthermore, n is the number of rules that an ant has applied so far to reach its current expression and t_n is the constant threshold related to changing the phase of the construction (from expanding phase to finishing phase). Finally, $maxN$ is the *maximum number of using rules* for all ants.

The following function has the characteristics defined in (8):

$$H(x,i,n) = e^{\frac{t_n - n}{t_n} * (\log(f_e(x,i) + 1)} \qquad (9)$$

Where *x* is a non-terminal symbol and *i* is an index of *x*'s rules. Furthermore, n is the number of rules that an ant has applied so far to reach its current expression and t_n is the constant threshold related to changing the phase of the construction ($1 < t_n < maxN$) while $maxN$ is the maximum number of using rules for ants. In the case of $n < t_n$, the fraction $\frac{t_n - n}{t_n}$ is positive then H and $f_e(x,i)$ has a direct relationship. The higher value of $f_e(x,i)$ results the higher value of H but, when $n > t_n$, H is reversely proportional to $f_e(x,i)$. Thus, the higher value of $f_e(x,i)$ results the lower value of H.

3.3 The Pheromone Placement Method

In the GAP model, the amount of pheromone deposited on the graph by an ant depends on the fitness value $L_k(t,p)$ and the constant Q. This paper presents another method of pheromone placement. In this method, in addition to $L_k(t,p)$, the rank of the ants as well as the number of the rules used to reach that edge is considered.

This method of pheromone placement tends to put more pheromone in the derivation steps, the steps made at the beginning of the path. This is based upon the hypothesis that if the fitness of a path is better than others this path is likely to have good structure and putting more pheromone on the early selections can encourage other ants to build this (or similar) structure(s). A small amount of pheromone in the latter edges of a path provides this opportunity for ants to explore final edges better. This is because these edges with a small amount of pheromone are biased less than the beginning edges.

The total amount of pheromone ant k places on the trail is:

$$\Theta_k \coloneqq f(\text{rank}(k)).\ L_k(t,p) \qquad (10)$$

Where $L_k(t,p)$ is the value of the objective function obtained by ant k at time t and $f(\text{rank}(k))$ is a factor that depends on the rank of the path (program) found by ant k. Note that ranking is done with respect to the $L_k(t,p)$ of ants.

The contribution of ant k to the update of a trail is computed as follows. As argued above, the intent of this method is to put more pheromone at the beginning of the path than at the end of the path.

$$\Delta T_k(t) = \Theta_k\ .\ 2\ .\frac{L - n + 1}{L^2 + L} \qquad (11)$$

Where L is the total number of rules which ant k has used to generate its program (path) and n is the number of rules used by ant k to reach this specific edge whose pheromone is being updated.

Note that, since $\sum_{n=1}^{L}(L - n + 1) = \frac{L^2 + L}{2}$, it is easy to verify that the total amount of pheromone placed on the trail by ant k is Θ_k.

4. EXPERIMENTAL RESULTS

In this section, the performance of GAP and EGAP will be compared in three experiments: Quartic Symbolic Regressions, Multiplexer, and Santa Fe ant trail.

GAP and EGAP have been run with the same parameters. The evaporation rate p is 0.5 and α and β are considered 2 and 1 respectively. The initial pheromone concentration, T_0, is 10^{-6} and *maxN* is 100. For each algorithm, 10 simulations are run with 100 iterations and, in each iteration, 20 ants have passed through the graph.

For both of the algorithms (EGAP, GAP), the number of generated individual is equal. In both EGAP and GAP, 100 iterations for 20 ants (100*20 = 2000) have been considered.

4.1 Quartic Symbolic Regression

The target function is defined as $f(a) = a + a^2 + a^3 + a^4$, and 200 numbers randomly generated in the range of $[-10,10]$ are used as the input for this function and the corresponding output of them is found. Therefore, the desired output for these 200 input numbers will be these outputs called y vector. The objective of this experiment is that these two algorithms (EGAP and GAP) find the expression that has the nearest output to y for x input vector. The fitness function for all the two algorithms is defined as follows:

$$f(p,x,y) = \frac{1}{N}\sum_{n=1}^{N} |p(x(n)) - y(n)| \qquad (12)$$
$$Fitness(p,x,y) = \frac{1}{1+f(p,x,y)}$$

Where p is the expression generated by the automatic programming algorithm; x and y are the input vector and desired output vector respectively. Finally, N is the number of the elements of x. The grammar used in this experiment for EGAP and GAP is given by

$< expr > \rightarrow < expr >< op >< expr > \mid < var >$

$< op > \rightarrow * \mid - \mid + \mid /$

$< var > \rightarrow a$

In Figure 2, the plot of the mean best fitness over 10 runs can be seen. EGAP clearly outperforms the GAP in this experiment. A t-test comparing these two methods in this experiment gives a score of 7.562 in the favor of EGAP—significant at the 99% confidence level.

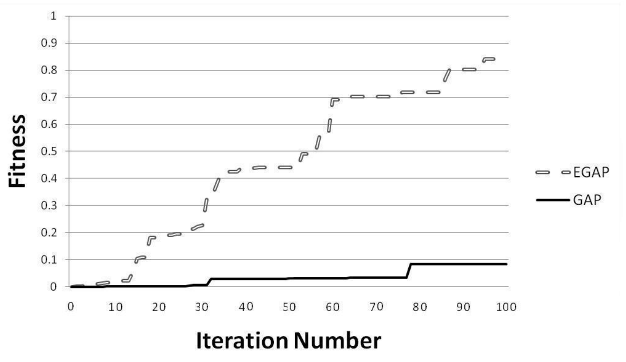

Figure 2. Plot of the mean of the best fitness on quartic symbolic regression problem during the 100 iterations

In Figure 3, the average node branching factor of all the nodes have visited in 10 runs each of which has 100 iterations can be observed. By inspection, although we use a heuristic method in EGAP, the branching factor in both methods looks similar. Superficially at least, it appears that EGAP has the same ability of exploration of search space as GAP although it uses the heuristic.

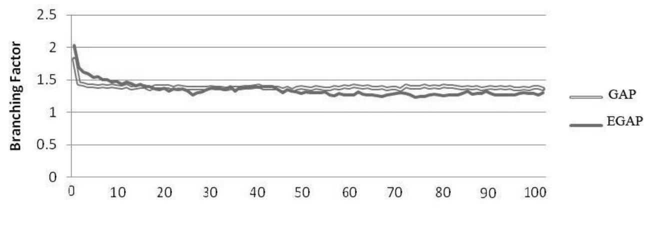

Figure 3. The plot of average node branching factor over 10 runs during 100 iterations

4.2 4-to-1 Multiplexer

The goal of this problem is to find a boolean expression that behaves as a 4-to-1 Multiplexer. There are 64 fitness cases for the 4-to-1 Multiplexer, representing all possible input-output pairs. Program fitness is the percentage of input cases for which the generated boolean expression returns the correct output. The grammar adopted for this problem is as follows:

$< mexpr > \rightarrow < mexpr >< op2 >< mexpr > \mid < op1 >< mexpr > \mid < input >$
$< op1 > \rightarrow and \mid or$
$< op2 > \rightarrow not$
$< input > \rightarrow$
$input0 \mid input1 \mid input2 \mid input3 \mid input4 \mid input5$

A plot of the mean best fitness over 10 runs of 100 iterations for these two algorithms is illustrated in Figure 4. As shown, EGAP had the better performance compared to GAP. It is interesting to note that this problem is not hard and both of these algorithms could usually find the simple boolean expression generating 40 correct answers out of 64 possible correct results in their first iteration. Despite the fact that this problem is not a good benchmark for these two algorithms because of its simplicity, EGAP represents a statistically significant improvement over GAP for this problem. A t-test comparing these two methods gives a score of 3.621 in the favor of EGAP—significant at the 95% confidence level.

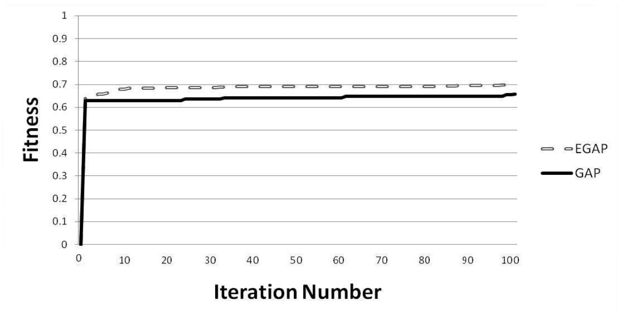

Figure 4. Plot of the mean of the best fitness on the multiplexer problem during the 100 iterations

4.3 Santa Fe ant trail

The Santa Fe ant trail is a standard problem in the area of GP. The objective of this problem is to find a computer program to control an artificial ant in such a way that it finds all 89 pieces of food that are located on the discrete trail. Furthermore, the ant is limited to find the food in a maximum number of time steps and the trail is located on the 32x32 grid. The ant can only turn left, right, or move one square ahead. Also, it can check one square ahead in the direction facing in order to recognize whether there is a food in that square or not. All actions, except checking the food, take one step for the ant to execute. The ant starts its foraging in the top-left corner of the grid. The grammar used in this experiment is:

$< code > \rightarrow < line > \mid < code >< line >$

$< line > \rightarrow < condition > \mid < op >$

$< condition > \rightarrow if(food\_ahead())$
$\{< line >\}$
$else$
$\{< line >\}$

$< op > \rightarrow left(); \mid right(); \mid move();$

The fitness function for both algorithms is the number of food items found by ant over the total number of food items, which is equal to 89.

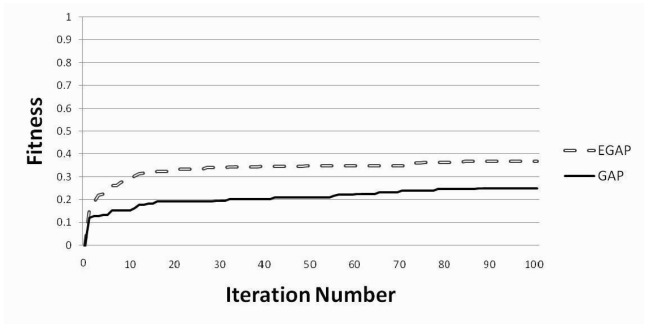

Figure 5. Plot of the mean of the best fitness on ant trail problem during the 100 iterations

In Figure 5, the plot of the mean best fitness over 10 runs for ant trail problem can be seen. EGAP outperforms the GAP in this experiment. A t-test comparing these two methods gives a score of 2.195 in the favor of EGAP—significant at the 95% confidence level.

4.4 Discussion

The main contributions of this paper are the introduction of a new heuristic function for program generation and a different method of pheromone placement for GAP. As shown in previous sections, the results of the experiments reveal the advantage of using this heuristic function and pheromone placement over GAP. An intriguing question is that to what extent each of these two suggested modifications (Heuristic and Pheromone Placement) contributes to the strength of EGAP. In order to understand this we devised another experiment, separating the two modifications. We chose the same parameter setting of quartic symbolic regression and run two more algorithms, GAP with the proposed heuristic function (GAP + H) and GAP with the proposed pheromone placement method (GAP + PP) to solve the quartic symbolic regression. The result of these two new algorithms and previous results of GAP and EGAP are shown in Figure 6.

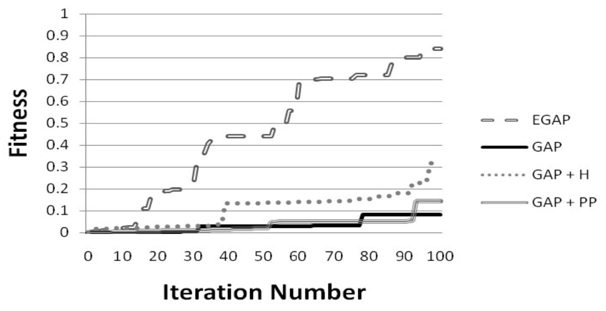

Figure 6. Plot of the mean of the best fitness on quartic symbolic regression problem during the 100 iterations

As illustrated in Figure 6, both GAP + H and GAP + PP outperform GAP in 100 iterations in this experiment; although the performance of GAP + H was better than the result of GAP + PP. Interestingly, this experiment shows that although EGAP uses both heuristic function and pheromone placement methods, its performance is considerably better than the performance of GAP + H and GAP + PP. In other words, the strength of EGAP is not only because of using these two heuristic function and pheromone placement but also because of the interaction of these two methods.

5. CONCLUSIONS AND FUTURE WORK

This paper proposes a novel automatic programming technique based upon the use of ACS and context free grammars. The research reported identifies weaknesses in GAP in the areas of path termination and pheromone placement. To solve these problems, a new heuristic function which is inspired from the building construction is presented in this work as well as the new method of pheromone placement. The heuristic function guides the ants to first construct a good structure for solution which is analogy to two phases of building construction, constructing the structure and decoration. This kind of the thinking exists in other aspects of life in that we first consider about the whole of object then we considerate on details and parts.

The results of the experiments reveal the advantage of using this heuristic function and pheromone placement over GAP. Furthermore, the results empirically demonstrate that EGAP and GAP have almost the same node branching factor, and then the heuristic did not affect the exploration ability of the algorithm in the solution space.

We plan to compare the result of EGAP with GP in our future work in order to understand better the performance of EGAP.

We believe that the heuristics proposed in this paper can be extended and used in other automatic programming algorithms. Furthermore, extending the heuristic function so that it has an adaptive phase duration should be investigated. In this way, based on the grammar and the fitness of generated program, ants will decide about the duration of each phase of construction.

The other suggestion for future work is in the use of heterogeneous ants in EGAP. Agent heterogeneity has been found to be useful in other search algorithms, including ACS and PSO. In the context of EGAP, ants would have variability in the duration of their construction phases. Hence, these ants could produce programs with different levels of complexity and length.

6. REFERENCES

[1] Bonabeau, E., Dorigo, M. and Theraulaz G. Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press, Oxford.1999.

[2] Boryczka, M. and Wiezorek, W. Solving approximation problems using ant colony programming. In Proceedings of AI-METH 2003, pages 55-60, 2003.

[3] Boryczka, M. Ant Colony Programming for Approximation Problems, Proceedings of the Eleventh International Symposium on Intelligent Information Systems, Sopot, Poland, June 3–6 2002.

[4] Boryczka, M., Czech, Z. J. Solving Approximation Problems By Ant Colony Programming, GECCO–2002: Proceedings

of the Genetic and Evolutionary Computation Conference (W. B. Langdon, E. Cant´u-Paz, K. Mathias, al., Eds.), Morgan Kaufmann Publishers, New York, 9–13 July 2002, ISBN 1–55860–878–8.

[5] Boryczka, M., Czech, Z. J. Solving Approximation Problems by Ant Colony Programming, Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO–2002) (E. Cant´u-Paz, Ed.), AAAI, New York, NY, 9–13 July 2002.

[6] Boryczka, M., Czech, Z. J. and Wieczorek, W. Ant Colony Programming for Approximation Problems, Genetic and Evolutionary Computation— GECCO–2003, Lecture Notes in Computer Science 2723–2724 (E. Cant´u- Paz, al., Eds.), Springer–Verlag, Berlin Heidelberg, 2003. Fundamenta Informaticae 68 (2005) 1–191.

[7] Boryczka, M. Eliminating Introns in Ant Colony Programming, Fundam. Inform. 68(1-2): 1-19 (2005).

[8] Keber, C. and Schuster, M. G. Option valuation with generalized ant programming. In W. B. Langdon and E. Cantii-Paz et al., editors, GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 74-8 1, New York, 9- 13 July 2002. Morgan Kaufmann Publishers.

[9] Koza, John R. Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, MA, 1992.

[10] Koza, John R. Genetic Programming II: Automatic Discovery of Reusable Programs.MIT Press, 1994.

[11] Koza, J. R., Bennet III, F. H., Andre D., and Keane, D.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, 1999.

[12] O'Neill M., and Brabazon A. Grammatical Swarm. In: LNCS 3102 Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2004. Seattle, WA, USA, pp. 163–174, Springer, Berlin, 2004.

[13] O'Neill, M., and Brabazon, A. Grammatical Swarm: The generation of programs by social programming. Natural Computing: an international journal 5, 4 (Nov. 2006).

[14] O'Neill, M., and Ryan, C. Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language, Kluwer Academic Publishers, 2003.

[15] Roux, O., and Fonlupt, C., 2000, Ant Programming: Or How to Use Ants for Automatic Programming, in Proceedings of ANTS' 2000, ed. By M. Dorigo et al. pp. 121-129.