Accelerating Genetic Programming by Frequent Subtree Mining

Yoshitaka Kameya
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8552, Japan
kameya@mi.cs.titech.ac.jp

Junichi Kumagai*
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8552, Japan
kumagai@mi.cs.titech.ac.jp

Yoshiaki Kurata
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8552, Japan
kurata@mi.cs.titech.ac.jp

ABSTRACT

One crucial issue in genetic programming (GP) is how to acquire promising building blocks efficiently. In this paper, we propose a GP method (called GPTM, GP with Tree Mining) which protects the subtrees repeatedly appearing in superior individuals. Currently GPTM utilizes a FREQT-like efficient data mining method to find such subtrees. GPTM is evaluated by three benchmark problems, and the results indicate that GPTM is comparable to or better than POLE, one of the most advanced probabilistic model building GP methods, and finds the optimal individual earlier than the standard GP and POLE.

Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Performance, Experimentation

Keywords

genetic programming, building blocks, frequent subtree mining, probabilistic model building genetic programming

1. INTRODUCTION

Genetic programming (GP) [10] is known as a powerful tool for optimization and problem solving, and has been applied to a wide variety of applications. One of the most crucial issues in GP is how to acquire and preserve promising building blocks efficiently [1, 8, 10, 11, 13, 16]. For this purpose, we propose GPTM (GP with Tree Mining), a GP method which first identifies the subtrees repeatedly appearing in the chromosomes of superior individuals (ones with very high fitness), and then protects them against undesirable crossover operations. To find such subtrees, we take a data mining approach where the target dataset is the chromosomes of superior individuals, and introduce an efficient subtree mining algorithm based on FREQT [3]. Using this mining algorithm, we can find subtrees like Fig. 1 (a) as well as Fig. 1 (b). Whereas some previous methods [8, 10, 11, 16] including the automatically defined functions covers the latter case, i.e. all nodes below a certain node (Y in Fig. 1 (b)) will be encapsulated, we aim to identify building blocks in a more flexible form. For instance, let us consider X_1 and X_2 in Fig. 1 (a) as IF-THEN-ELSE functions. That is, we have $X_1 = \texttt{IF\_}C_1$ and $X_2 = \texttt{IF\_}C_2$, where C_1 and C_2 refer to some conditions. Also, let x be a terminal symbol that indicates taking some action A. Then, the subtree in Fig. 1 (a) says "do A if $\neg C_1 \wedge C_2$ holds." Provided that this statement is a building block for the target problem, the GP system would be able to proceed further, delaying to think about the suitable actions for the other cases. Additionally, using subtree mining algorithms, building blocks can be identified independently of the positions in the chromosome.

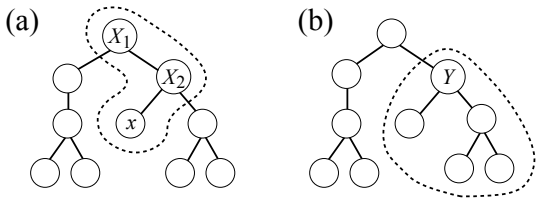

Figure 1: Examples of a subtree (surrounded by a dashed line) of a chromosome tree.

Of course, the search performance is also important. A recent approach to acquire/preserve building blocks is probabilistic modeling building GP (PMBGP). Like GPTM, PMBGP methods see the chromosomes of superior individuals as data, to which statistical techniques are applied. In this paper, we compare GPTM with the standard GP and a PMBGP method called POLE (Program Optimization with Linkage Estimation) [6], on three benchmark problems.

The rest of this paper is organized as follows. First, Section 2 describes GPTM, the proposed method. We then report the results of the comparative evaluation in Section 3. Section 4 concludes this paper with mentioning future work.

*Currently working at Fujitsu Ltd., Japan.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

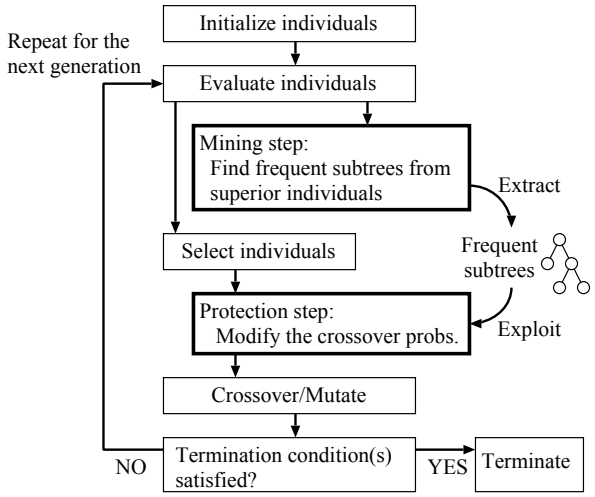

Figure 2: Outline of GPTM.

2. PROPOSED METHOD

The proposed method, GPTM, is outlined in Fig. 2. This figure indicates that GPTM basically follows the standard GP workflow, but is augmented with two steps called the *mining step* and the *protection step*. In the mining step, GPTM first selects a small fraction of individuals with higher fitness as *superior individuals*, as done in the truncate selection, and then runs a subtree mining algorithm for finding frequent subtrees (hopefully, promising building blocks) from the superior individuals. The number of superior individuals is specified in advance. Using the terminology of the data mining literature, we hereafter use the term *pattern* to refer to a subtree treated in GPTM. Then, in the protection step, to avoid undesirable crossover operations, the crossover probabilities of parents are modified based on the extracted patterns. In the sequel, we describe the two augmented steps in turn, following some preliminaries.

2.1 Preliminaries

The subtree mining algorithm used in the mining step is an adaptation of FREQT [3] to our purpose. Throughout this paper, we use F and T respectively for the set of function symbols and the set of terminal symbols in the problem domain. In the mining algorithm, the chromosome of an individual is considered as a labeled ordered tree, where nodes are labeled with the symbols from $F \cup T$, and the occurrence order among siblings cannot be ignored. The chromosome trees of superior individuals are bundled up to a single labeled ordered tree D, called the *data tree*, whose root node is labeled with a dummy symbol, say R, not included in $F \cup T$. We also give indices from 1 to $|D|$ to the nodes in D in the preorder. For example, Fig. 3 (a) shows a data tree D for two individuals, where $F = \{A, B\}$ and $T = \{x\}$, and the arity of A (resp. B) is 2 (resp. 1). In D, each of child trees of the root node, i.e. subtrees comprised of nodes $\{2, 3, \ldots, 8\}$ and nodes $\{9, 10, \ldots, 16\}$, corresponds to the chromosome of an individual.

A pattern is also a labeled ordered tree, which has labels from $F \cup T \cup \{*\}$. In Fig. 3 (a), we have two examples of a pattern, S_1 and S_2. It is seen that the pattern S_1 appears three times (i.e. as subtrees $\{2, 3, 4\}$, $\{5, 6, 7\}$ and $\{9, 10, 11\}$) in D, whereas pattern S_2 does not appear in D at all. Hereafter a node is called a *don't-care node* if it is labeled with $*$, or called a *proper node* otherwise. The size of a pattern is defined as the number of proper nodes included in the pattern.

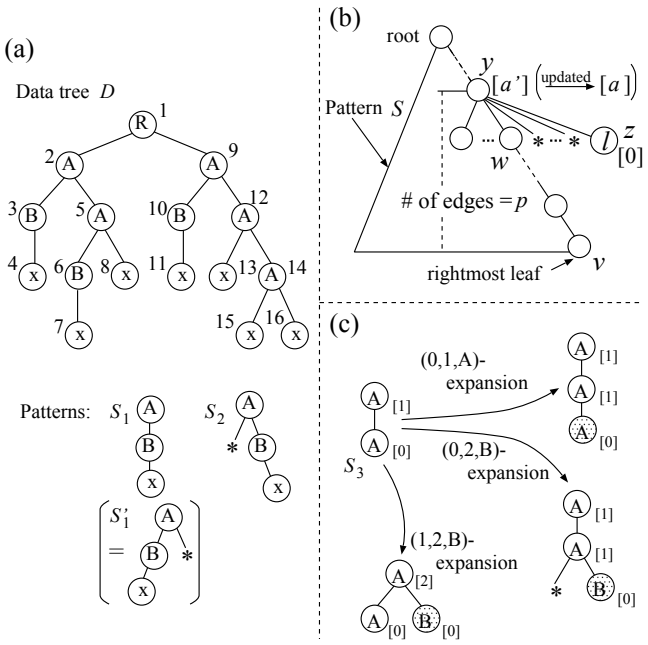

Figure 3: (a) A data tree and a pattern. (b) (p, a, ℓ)-expansion and (c) its examples.

To adapt the notions from FREQT to the GP system, two points should be taken into account. First, for example, while the original FREQT does not distinguish S_1 and S_2, we need to distinguish them. This is of course because the positions of arguments are often essential to the functions in the domain. On the other hand, to realize *rightmost expansion*, one of key notions of FREQT (described later), comprehensibly, we will use S_1, not S_1', as a representation of a pattern.[1] The second point to note is that the arity for each function is known in advance.

2.2 Finding frequent patterns

2.2.1 Rightmost expansion

In problems of finding frequent patterns from data, we usually specify σ_{min}, a positive integer often called the *minimum support*. Then, many of well-known mining algorithms try to find efficiently the patterns which appear σ times in D where $\sigma \geq \sigma_{min}$. The mining step of GPTM also does this by generating possible patterns incrementally with counting their occurrences. The incremental procedure for pattern generation is called rightmost expansion. In a rightmost expansion, for some $k \geq 1$, we grow a k-pattern (a pattern of size k) S to a $(k+1)$-pattern S' by attaching a (proper) node to the rightmost path.

[1]Precisely speaking, if a pattern S has a proper node v and v's right siblings are all don't-care nodes, then we will remove these right siblings of v from S. Also if the children of a proper node in S are all don't-care nodes, then we will remove these children from S.

To be more concrete, let us see Fig. 3 (b), where we are creating a new pattern from the pattern S by adding a node z, labeled with $\ell \in F \cup T$. S has the rightmost leaf v, and z is being added as a child of the p-th ancestor of v, say y, on the rightmost path (the path from the root to v). Besides, in the mining step, a node in the pattern S is also annotated a number of (both proper and don't-care) children included in S. In the figures, a node which has a label ℓ and has a children in the pattern of interest is denoted by $\ell_{[a]}$. As a special case, a node with a terminal label $\ell \in T$ is always considered to have 0 children. Then, let us note that the node y in Fig. 3 (b) already has a' children, and suppose that the function labeled at y is n-ary. After z added as the a-th child[2] of y ($a' < a \leq n$), the number of children of y will be updated to a, and don't-care nodes will be filled between w and z. We call this procedure a a (p, a, ℓ)-*expansion* for S. Fig. 3 (c) shows three instances of a (p, a, ℓ)-expansion for pattern S_3. Note that the newly attached node z is the rightmost leaf of the new pattern S'.

2.2.2 Counting the rightmost occurrences

Furthermore, to find frequent patterns, it is also required to count the occurrences of patterns. For each pattern S, we first compute a set $RMO(S)$ of *rightmost occurrences* of S in the given data tree D. A rightmost occurrence of S in a data tree D is referred to by the index of the node in D that matches with the rightmost leaf of S. In Fig. 3 (a), for example, the rightmost leaf of pattern S_1 is the node with label x, and hence $RMO(S_1) = \{4, 7, 11\}$. $RMO(S)$, the rightmost occurrences of a pattern S, is computed when S is created. Let us consider the case where S is created by a (p, a, ℓ)-expansion from the base pattern S_0. Then, $RMO(S)$ is computed as $\{\phi_D(i, p, a) \mid v \in RMO(S_0), L(\phi_D(i, p, a)) = \ell\}$, where $\phi_D(i, p, a)$ (resp. $L(\phi_D(i, p, a))$) is the index (resp. the label) of the a-th child node of the p-th ancestor of the node indexed by i in the data tree D. Also, for a pattern S of size 1 whose only node has a label ℓ_0, $RMO(S)$ is computed as the set of indices of the nodes labeled with ℓ_0 in the data tree D. In Fig. 3 (a), the rightmost occurrences of the pattern created by $(2, 2, A)$-expansion from S_1 is $\{5, 12\}$. The number of occurrences of S, denoted by $\sigma(S)$, is finally obtained as the size of $RMO(S)$.

2.2.3 Pattern enumeration

Fig. 4 illustrates the process of enumerating all frequent patterns appearing in the data tree D in Fig. 3 (a), given $\sigma_{min} = 2$. We start from all patterns of size 1, consisting only of one node with a function label. In the figure, a directed edge annotated with (p, a, ℓ) indicates an application of a (p, a, ℓ)-expansion. Also, for each pattern S, $RMO(S)$ and $\sigma(S)$ are written below S. We basically try all possible (p, a, ℓ)-expansions from patterns of smaller size to patterns of larger size, but for the infrequent patterns (a pattern S is said to be infrequent if $\sigma(S) < \sigma_{min}$), we can stop the further expansions.[3] In Fig. 4, these infrequent patterns are represented with shaded nodes. On the other hand, we can see that there are 12 frequent patterns, in which `(A (B x) (A * *))` is the largest pattern (the size is 4).

[2]The enumeration of children is one-based and left-to-right.
[3]Let S' be a pattern created from S by a (p, a, ℓ)-expansion. Then, it is easily seen that a larger pattern S' is also infrequent since $\sigma(S') \leq \sigma(S)$, and hence the further expansions from S will always be unfruitful.

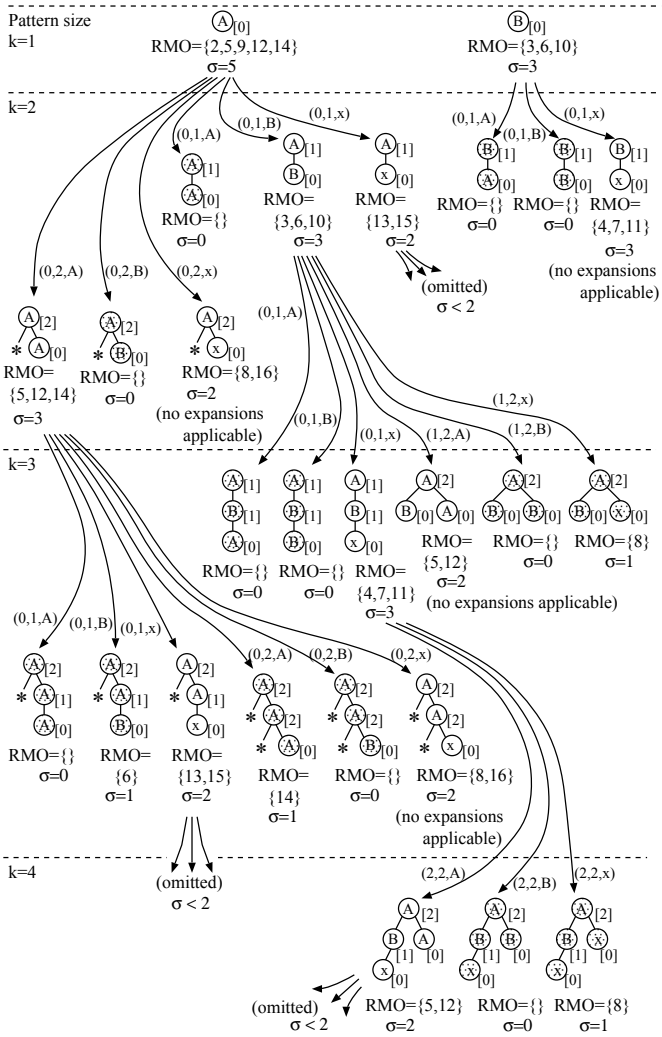

Figure 4: Frequent patterns found by GPTM.

2.2.4 Adjusting the minimal support

The last thing we consider for the mining step of GPTM is how to set a suitable value to σ_{min}, the minimum support. Unfortunately, as is well-known, it is not easy. That is, too small σ_{min} will cause a flood of frequent patterns, whereas with too large σ_{min}, we can find nothing. Besides, in considering the adaptation to the GP system, the suitable σ_{min} might be changed in the middle of the evolution. Currently, GPTM automatically finds a suitable value for σ_{min} at the cost of extra computation.

Specifically, we first prepare four control parameters — the maximum size ξ_{max} and the minimum size ξ_{min} of patterns, and the maximum number ζ_{max} and the minimum number ζ_{min} of frequent patterns at the maximum size ξ_{max}. Namely, we only try to find frequent patterns of size ξ such that $\xi_{min} \leq \xi \leq \xi_{max}$. Using ζ_{max} and ζ_{min}, we adjust the minimum support σ_{min} by the following procedure:

1. Set $\sigma_{min}^{(0)}$ as a sufficiently small number. Let $t := 0$.
2. Run the mining process described above with $\sigma_{min}^{(t)}$.
3. If the number ζ of the found frequent patterns satisfies

$\zeta_{min} \leq \zeta \leq \zeta_{max}$, then finish the whole mining step of GPTM. Otherwise, set $\sigma_{min}^{(t+1)}$ under some updating strategy, let $t := t + 1$, and go to the step 2.

In updating σ_{min}, we first repeat $\sigma_{min}^{(t+1)} := 2\sigma_{min}^{(t)}$ until $\zeta < \zeta_{min}$. Once we have $\zeta < \zeta_{min}$, we let $\sigma_{min}^{(t+1)} := (\sigma_{min}^{(t)} + \sigma_{min}^{(t-1)})/2$, and hereafter, using a binary search, we seek σ_{min} with which the number ζ of frequent patterns at maximum size falls into $[\zeta_{min}, \zeta_{max}]$. Surely this procedure would be time-consuming in that we need to run the mining process (Step 2) iteratively. However, we would like to note that the mining process is efficient itself and that the number of superior individuals is not so large.

2.3 Protecting frequent patterns

After finding frequent patterns, in the protection step, GPTM tries to protect these patterns in a soft way. That is, it modifies the crossover probabilities to make these frequent patterns less likely to be destroyed and more likely to be inherited to the next generation. Formally, the procedure is described as follows. For a (sub)tree or pattern S, we define $U(S)$ as a set of nodes in S, and $root(S)$ as the root node of S. Also, for a pattern S' of S, let $U_{anc}(S')$ be the ancestral nodes of S', and let $U_{inside}(S') \stackrel{\text{def}}{=} U(S') \setminus \{root(S')\}$. These definitions are illustrated in Fig. 5 (a), where a frequent pattern consists of shaded nodes. Then, provided that we find N distinct frequent patterns S'_1, S'_2, $\ldots$, S'_N in a chromosome tree C, we define the root node set $V_{root}(C) \stackrel{\text{def}}{=} \{root(S'_i) \mid i = 1, 2, \ldots, N\}$, the ancestral node set $V_{anc}(C) \stackrel{\text{def}}{=} \cup_{i=1}^{N} U_{anc}(S'_i)$ and the inside node set $V_{inside}(C) \stackrel{\text{def}}{=} \cup_{i=1}^{N} U_{inside}(S'_i)$.

Besides, in GPTM, we consider that a crossover operation only produces one child tree, as illustrated in Fig. 5 (b). Then, the parent tree whose root node is inherited by its child is called the maternal tree, and the other parent tree is called the paternal tree. Let us denote the crossover probability at a node v by $p_{cross}(v)$. GPTM picks up two individuals (e.g. by a tournament selection) from the population, and performs a crossover operation at a node v with the probability $p'_{cross}(v)$, which has been modified as follows:

For the maternal tree C_m:

- Let V_1 be a set of nodes to be protected, and V_2 be a set of nodes to be unprotected. That is, $V_1 \stackrel{\text{def}}{=} V_{root}(C_m) \cup V_{inside}(C_m) \cup V_{anc}(C_m)$, and $V_2 \stackrel{\text{def}}{=} U(C_m) \setminus V_1(C_m)$.
- For a node $v_1 \in V_1$, discount the crossover probability using $p'_{cross}(v_1) := \gamma \cdot p_{cross}(v_1)$, where $0 < \gamma < 1$. Let π_m be the total discounted probability mass, i.e. $\pi_m \stackrel{\text{def}}{=} (1 - \gamma) \sum_{v_1 \in V_1} p_{cross}(v_1)$.
- Distribute the probability mass π_m to the nodes in V_2, i.e. let $p'_{cross}(v_2) := p_{cross}(v_2) + \frac{\pi_m}{|V_2|}$ for each $v_2 \in V_2$.
- Normalize the modified probabilities, i.e. let $p'_{cross}(v) := \frac{1}{\sum_{v' \in U(C_m)} p'_{cross}(v')} p'_{cross}(v)$ for each $v \in U(C_m)$.

For the paternal tree C_p:

- Let W_1, W_2 and W_3 be a set of nodes to be neutral, of nodes to be protected, and of nodes to be unprotected, respectively. That is, $W_1 \stackrel{\text{def}}{=} V_{anc}(C_p) \setminus V_{root}(C_p)$, $W_2 \stackrel{\text{def}}{=} U(C_p) \setminus (V_{root}(C_p) \cup V_{anc}(C_p))$, and $W_3 \stackrel{\text{def}}{=} V_{root}(C_p)$.
- For a node $v_1 \in W_1$, we do not modify its crossover probability, i.e. let $p'_{cross}(v_1) := p_{cross}(v_1)$.
- For a node $v_2 \in W_2$, discount the crossover probability using $p'_{cross}(v_2) := \gamma \cdot p_{cross}(v_2)$, where $0 < \gamma < 1$. Let π_p be the total discounted probability mass, i.e. $\pi_p \stackrel{\text{def}}{=} (1 - \gamma) \sum_{v_2 \in W_2} p_{cross}(v_2)$.
- Distribute the probability mass π_p to the nodes in W_3, i.e. let $p'_{cross}(v_3) := p_{cross}(v_3) + \frac{\pi_p}{|W_3|}$ for each $v_3 \in W_3$.
- Normalize the modified probabilities.

The constant γ above is hereafter called the *discount rate*.

Now, let us see Fig. 5 (b) again, and consider the patterns comprised of shaded nodes as the frequent patterns. By the modifications of crossover probabilities above, we can find that the frequent patterns in the maternal tree will be less likely to be destroyed, and the frequent patterns in the paternal tree will be more likely to be inherited by its child.

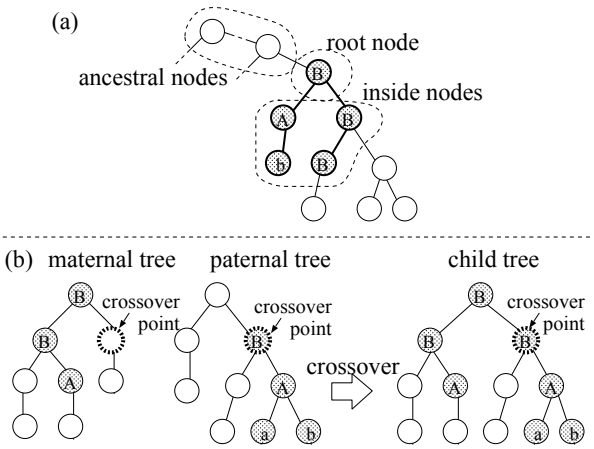

Figure 5: (a) The root node, the ancestral nodes, and the inside nodes for a frequent pattern. (b) Crossover operation with frequent patterns.

2.4 Depth-dependent crossover

Although the modification of crossover probabilities seems to accelerate the evolution, the risk of being trapped unwanted local optima should increase, since unpromising code fragments also tend to be fixed in an early stage of evolution. To get out of this dilemma, the current GPTM also adopts the *depth-dependent crossover* [9] as an option.

In a usual crossover operation in GP, we set the crossover probabilities as uniform. We refer to such a crossover operation as the *uniform crossover*. In the depth-dependent crossover, on the other hand, the crossover probabilities are exponentially decreased with respect to the depths of the corresponding nodes. Roughly speaking, with the depth-dependent crossover, shallow nodes are more often chosen, and hence we have more chance to escape from the local optima. To be specific, for a node v, we first set $p_{cross}(v)$ proportional to $\frac{1}{2^d} \cdot \frac{1}{|L_d|}$, where d is the depth of v and L_d is a set of nodes at the depth d. We then modify $p_{cross}(v)$ to $p'_{cross}(v)$ following the procedure in the previous section,

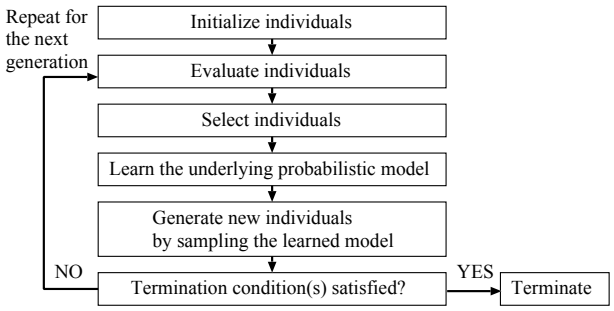

Figure 6: Outline of PMBGP methods.

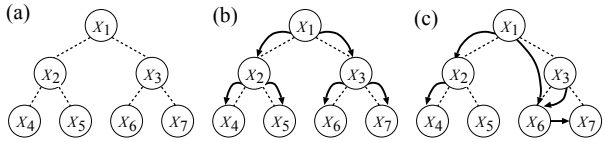

Figure 7: Bayesian network on a probabilistic prototype tree.

except that the discounted probability mass (π_m or π_p) is also distributed decreasingly according to the depths.

3. BENCHMARK EVALUATIONS

In this section, we report the results of comparative evaluation with three benchmark problems — symbolic regression, artificial ants for the Santa-Fe trail, and the royal tree problem. The building blocks in the last problem has an apparent regularity in their structures, while the building blocks in the former two seem not. Before showing the results, we first describe briefly a probabilistic model building GP method, POLE, used in the evaluation.

3.1 Probabilistic model building GP

Fig. 6 shows a typical workflow in PMBGP methods. PMBGP methods no longer have genetic operations, and instead each has a (parametric) probabilistic model by which new populations are generated. In each evolutionary loop, the model is learned from the superior individuals, and generates a new population by sampling. By this strategy, we can expect that the probabilistic model turns to approximate well the distribution of superior individuals, and that the biases in this approximated distribution leads to a formation of building blocks.

There have been two classes of PMBGP methods in which the difference is the underlying probabilistic model. The former class uses a Bayesian network (BN) on the data structure called a *probabilistic prototype tree* (PPT) [14], and POLE belongs to this class. The latter (e.g. [15]) uses probabilistic grammars. Fig. 7 gives examples of a BN on a PPT, where dashed lines indicate the edges of a PPT, and arcs with solid lines indicate the edges of a Bayesian network on the PPT. A PPT is an n-ary complete tree in which n is the maximum arity of the functions in the problem domain. In sampling, along with the edge directions in the BN, function/terminal symbols in the chromosome are determined one by one. The parameters in the BN are the local probabilities among a child (e.g. X_6 in Fig. 7 (c)) and its parents (X_1 and X_3), and they are learned according to the number of occurrences of function/terminal symbols in the chromosomes of superior individuals.

Fig. 7 (a) is a probabilistic model used in PIPE (Probabilistic Incremental Program Evolution) [14], where the BN has no edges, i.e. all function/terminal symbols are determined independently of each other. EDP (Estimation-of-Distribution Programming) [17] considers BNs in a *fixed* form as depicted in Fig. 7 (b). POLE can be seen as the most advanced BN-based PMBGP methods, since it allows an arbitrary form of the BN structure such as Fig. 7 (c), except that the edges should be directed from ancestors to descendants, or from left siblings to right siblings.

The most characteristic feature of POLE is to learn the BN structure, as well as the parameters, using the K2 algorithm.[4] For computational efficiency, on the other hand, we put a restriction that the number of *undirected* edges in the base PPT between a parent and its child in the BN, which are connected by a *directed* edge, cannot be longer than β (> 0) [7]. That is, β works as a control parameter for POLE which limits the range of possible parents in the BN. Also, to get statistically reliable parameters from a limited number of superior individuals, POLE adopts maximum a posteriori (MAP) estimation, where we use δ as the *default count*.

Table 1: Settings common to all problems.

Method	Control parameter	Value
Common to all methods	Initialization method	Grow
	Probability of selecting function symbols in Grow	0.9
	# of elites	1
	# of runs	50
GP/GPTM	Selection method	tournament
	Mutation probability	0.05
GPTM	# of superior individuals	50

3.2 Settings and implementations

For benchmark evaluations, control parameters were configured as shown in Table 1 and Table 2. For symbolic regression and artificial ants, the depth-dependent crossover (Section 2.4) did not work well, so we used the uniform crossover instead. The default count δ in POLE is chosen based on the best fitness value of the last generation, averaged on 50 runs. POLE and GPTM were implemented by extending GPsys-2b (`ftp://cs.ucl.ac.uk/genetic/gp-code/`), a Java-based GP system by A. Qureshi. We also extended GPsys-2b so that the standard GP can deal with both the uniform crossover and the depth-dependent crossover.[5]

3.3 Results

3.3.1 Symbolic regression

In symbolic regression, we aim to obtain an approximate function $f'(x)$ for a given function $f(x)$. We are even unaware of the existence of building blocks, but we expect this problem to exhibit the general performance of GPTM. Fol-

[4]POLE adopts the Bayesian Information Criterion (BIC) to choose the most plausible BN structure.

[5]In the old implementation of GPTM, the depth-dependent crossover was only incorporated in part. We also reconfigured the control parameters for GPTM. As a result, the evaluation results reported in this paper are rather different from the ones in our previous reports.

Table 2: Settings for three benchmark problems.

Method	Control parameter	Ant	SymReg: f_a	SymReg: f_b	SymReg: f_c	Royal tree
Common	Population size	500	200			5000
	# of generations	50	1000			100
	Max. tree depth	7	6			6
GP/GPTM	Selection method	tournament	tournament			tournament
	Tournament size	7	7			7
GP	Crossover operation	uniform	uniform			depth-dep.
GPTM	Size of frequent patterns (ξ_{max}/ξ_{min})	2/5	5/5			3/3
	# of freq. patterns at max. size (ζ_{max}/ζ_{min})	10/50	5/20			5/20
	Discount rate γ	0.9	0.5	0.9	0.9	0.3
	Crossover operation	uniform	uniform			uniform/depth-dep.
POLE	Selection method	truncate	truncate			truncate
	Truncate probability	0.2	0.2			0.2
	Default count δ	0.001	1.0	1.0	0.1	0.5
	Range β of possible parents	4	4	4	4	2

lowing [17], we use the following three target functions:

$$\begin{cases} f_a(x) = (2 - 0.3x)\sin(2x)\cos(3x) + 0.01x^2 \\ f_b(x) = x\cos(x)\sin(x)(\sin^2(x)\cos(x) - 1) \\ f_c(x) = x^3\cos(x)\sin(x)e^{-x}(\sin^2(x)\cos(x) - 1) \end{cases}$$

The function and terminal symbols are given as {`Add`, `Sub`, `Mul`, `Div`, `Sin`, `Cos`} and {`x`, `0.05`, `0.10`, `0.15`, ..., `1.00`}, respectively. For each individual, we compute a (raw) fitness value by $1000 - 50\sum_{j=1}^{30} |f(x_j) - f'(x_j)|$, where $x_j = 0.2(j - 1)$. The fitness value takes 1000 at maximum though it can be negative. The settings of control parameters basically follow [17], and in POLE, the default count δ was chosen from 0.05, 0.1, 0.5, 1.0 and 5. The results are shown in Fig. 8. GPTM works best for functions f_b and f_c, while POLE works best for f_a. Except that the standard GP succeeded to find the exact definition of f_b in a run, GPTM works as an improvement of the standard GP.

3.3.2 Artificial ants for the Santa-Fe trail

In the artificial ant problem, an ant walks around on a 32×32 toroidal grid to eat 89 pieces of foods. These foods are placed on an irregular trail, called the Santa-Fe trail, which has gaps. The ant is only allowed to move in 400 time steps. The (raw) fitness is the number of pieces of foods eaten by the ant. For the ants with the same fitness, simpler one is preferred. The population size and the number of generations were set following [10]. The default count δ for POLE was chosen from 0.0005, 0.001, 0.005, ..., 5.

The results is shown in Fig. 9, where the meanings of x-axis and y-axis of the graphs are the same as those in Fig. 8. We can read from the graphs that the methods except POLE succeeded to find an optimal ant, i.e. an ant which ate all pieces of foods on the trail. The difference between GP and GPTM on the average performance is small, but the probability of success, the relative frequency of runs that achieved to find an optimal ant until the last (50th) generation, in GPTM is as twice as that in GP. The simplest optimal ant obtained through 50 runs of GPTM is illustrated in Fig. 10. This ant turns around clockwise, and moves forward if there is a food ahead. Otherwise, i.e. if there is no food around, it moves forward in the original direction. Surely this ant is not versatile (since it only checks the squares adjacent to the current square), but is rather reasonable. In the run which obtained the ant in Fig. 10, GPTM extracted a pattern `(If_Food_Ahead Move *)`, continuously from the first generation to the last generation. This partial IF-THEN-ELSE pattern is very simple but seems useful in most cases (i.e. at any position in the chromosome) for the artificial ant problem. Also, this pattern appears three times in the optimal ant shown in Fig. 10, and can be seen as a simple instance of a building block discussed in Section 1.

3.3.3 Royal trees

In the royal tree problem [12], the optimal chromosome tree is obtained by building up the code fragments in bottom-up, and from this nature, this problem can be considered as suitable to measure the efficiency in collecting building blocks. A program for the royal tree problem consists of the functions symbols `A`, `B`, ... and the terminal symbols `x`, `y`, ... The arities of functions increase one by one in the alphabetical order (i.e. the arities of `A`, `B`, ... are 1, 2, ..., respectively) of the function labels. In royal trees, we use a notion of 'perfect' trees. A perfect tree is a complete tree such that every function node has a label which is the immediate successor of the child node's label in the alphabetical order, except that every node labeled with `A` has a leaf node `x`. For instance, the trees (i), (ii) and (iii) in Fig. 11 are perfect trees, while the tree (iv) is not.

The (raw) fitness value is a score computed by the procedure as follows. The score of a tree is the score of its root node. At each function node n, we take a weighted sum s of the scores of its child trees. In this summation, if a child tree t_c of n is a perfect tree, we add the t_c's score multiplied by the full bonus (= 2) to s. Even when t_c is not a perfect tree, if the root label of t_c is the immediate predecessor of n's label, the score of t_c multiplied by the partial bonus (= 1) is added to s. If both conditions fail to be satisfied, we add the t_c's score multiplied by the penalty (= 1/3) to s. The score of n is basically given as the weighted sum s, but if the tree rooted at n is a perfect tree, the score of n is then multiplied by the complete bonus (= 2). Also, to each leaf node, we give a score 1 if the label is `x`, and give 0 otherwise. Fig. 11 also shows the score of each tree. We see that the fitness grows exponentially (only) if the subtrees are correctly built in bottom-up. In the evaluation, we used five function symbols {`A`, `B`, `C`, `D`, `E`} and three terminal symbols {`x`, `y`, `z`}. The optimal fitness is 122,880 ($\approx 10^{5.09}$), the score of the perfect tree whose root label is `E`. The settings of control parameters are shown in Table 2. In GPTM, we used the depth-dependent crossover. In POLE, the default count δ was chosen from 0.1, 0.5 and 1, and the range β of possible parents was restricted to 2 due to the memory space.

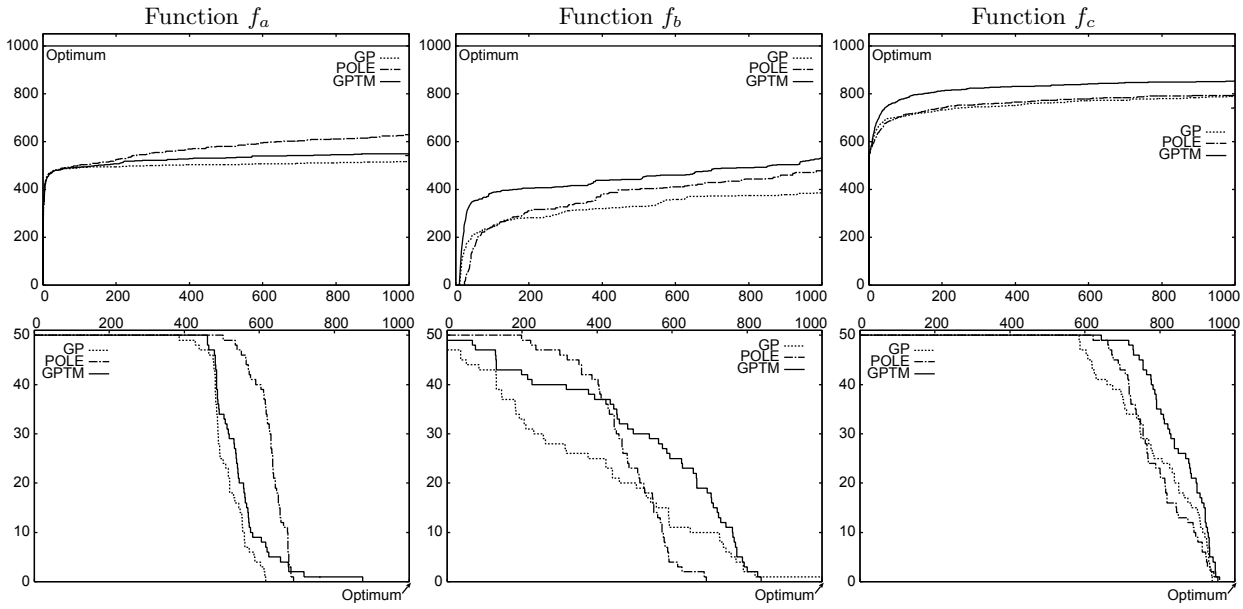

Figure 8: Results for symbolic regression problems. Above: Transition of the best fitness value, averaged on 50 runs. — x-axis: generation, y-axis: fitness value. Below: Cumulative number of successful runs given a threshold, at the 1000th generation — x-axis: threshold fitness value (say, x), y-axis: number of runs where the best individual achieved a fitness greater than or equal to the threshold x.

Fig. 12 shows the results, where the meanings of x-axis and y-axis of the graphs are the same as those in Fig. 8. In the graphs, 'uniform' (resp. 'depth-dep.') indicates the use of the uniform (resp. the depth-dependent) crossover. The results tell us that GPTM found the optimal individual earlier than the standard GP. POLE also found the optimal individual on 19 runs, but from Fig. 12 (a), it should require more time to find the optimal individual on all runs (this observation also applies to the result for the artificial ant problem). We can also see that, in the royal tree problem, the depth-dependent crossover is quite effective for GPTM. Indeed, GPTM with the uniform crossover often trapped in a locally optimal individual, i.e. the perfect tree of depth 5, whose root label is 'D' and whose score is 6,144 ($\approx 10^{3.79}$).

4. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new GP system, called GPTM, which utilizes a subtree mining algorithm for finding building blocks in the superior individuals. GPTM is expected to identify position-independent building blocks in a flexible form such as IF-THEN-ELSE structures. The experimental results show that GPTM is superior to the standard GP in most cases, and is comparable to or better than POLE, one of the most advanced probabilistic model building GP methods. Also GPTM tends to find the optimal individual earlier than the other methods.

To our knowledge, GPTM is a first attempt to combine GP and an efficient data mining technique, so there is much room for improvement. For example, the discount rate is currently set to be constant, but it seems not so adequate since the quality of frequent patterns tends to be poor in an early stage of evolution. So the scheduled adjusting of the discount rate, like [5] proposed for mutation probabilities in genetic algorithms (GAs), seems to be important. To incorporate well-developed heuristics [8, 13] or to give weight to the maximal frequent patterns (found by the techniques such as [4]) is another possible improvement. In a broader context, it seems that GAs can benefit from the basic idea presented in this paper. That is, a sophisticated sequential data mining (motif mining) algorithm such as [2] would be likely to improve the performance of GAs.

5. REFERENCES

[1] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proc. of the 14th Conf. of the Cognitive Science Society*, pages 236–241, 1992.

[2] H. Arimura and T. Uno. A polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. In *Proc. of the 16th Intl. Symp. on Algorithms and Computation (ISAAC-2005)*, pages 724–737, 2005.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficent substructure discovery from large semi-structured data. In *Proc. of the 2nd SIAM Intl. Conf. on Data Mining*, 2002.

[4] Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. CMTree Miner: mining both closed and maximal frequent subtrees. In *Proc. of the 8th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD-2004)*, pages 63–73, 2004.

[5] T. C. Fogarty. Varying the probability of mutation in the genetic algorithm. In *Proc. of the 3rd Intl. Conf. on Genetic Algorithms (ICGA-89)*, pages 104–109, 1989.

[6] Y. Hasegawa and H. Iba. Optimizing programs with estimation of Bayesian network. In *Proc. of the 2006 Congress on Evolutionary Computation (CEC-2006)*, pages 1378–1385, 2006.

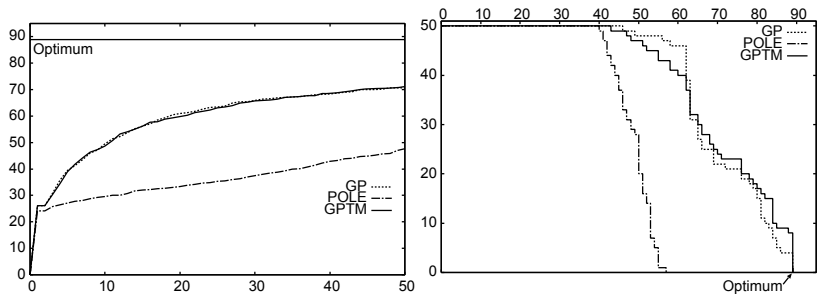

Figure 9: Results for artificial ants. Left: Transition of the best fitness value, averaged on 50 runs. Right: Cumulative number of successful runs given a threshold, at the 1000th generation.

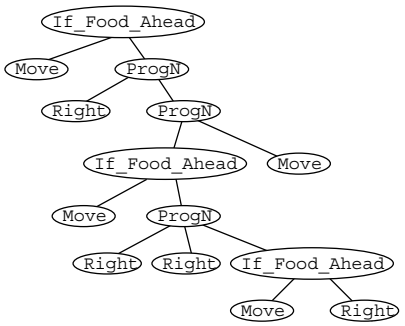

Figure 10: Chromosome tree of the best ant obtained by GPTM.

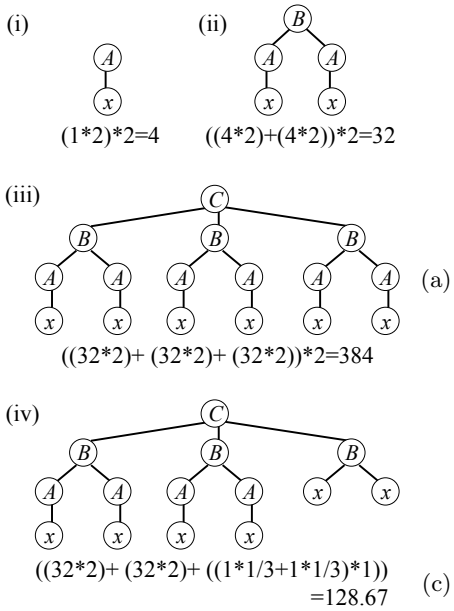

Figure 11: Examples of a royal tree.

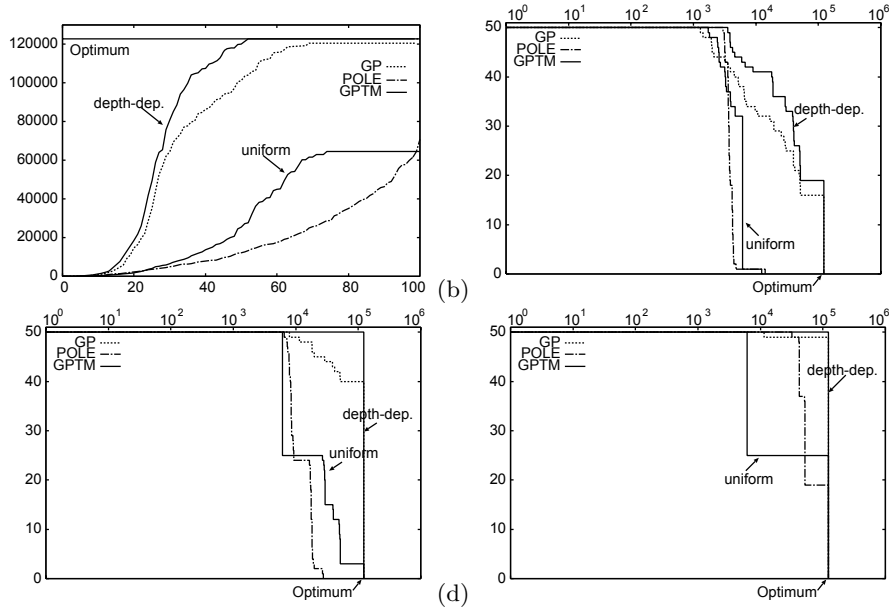

Figure 12: Results for royal trees. (a) Transition of the best fitness value, averaged on 50 runs. (b)∼(d) cumulative number of successful runs given a threshold, at (b) the 25th, (c) the 50th and (d) the 100th gen.

[7] Y. Hasegawa and H. Iba. Probabilistic model building genetic programming based on estimation of Bayesian network. *Trans. of the Japanese Society for Artificial Intelligence*, 22(1):37–47, 2007. In Japanese.

[8] R. D. Howard and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In *Proc. of the 4th European Conf. on Genetic Programming (EuroGP 2001)*, pages 160–175, 2001.

[9] T. Ito, H. Iba, and S. Sato. Depth-dependent crossover for genetic programming. In *Proc. of the 1998 IEEE Intl. Conf. on Evolutionary Computation (ICEC-98)*, pages 775–780, 1998.

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

[11] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Subprograms*. MIT Press, 1994.

[12] B. Punch, D. Zongker, and E. Goodman. The royal tree problem — a benchmark for single and multi-population genetic prgramming. In P. J. Angeline and K. E. Kinnear (Eds), *Advances in Genetic Programming II*. MIT Press, 1996.

[13] J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. In P. J. Angeline and K. E. Kinnear (Eds), *Advances in Genetic Programming II*. MIT Press, 1996.

[14] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[15] Y. Shan, R. I. McKay, R. Baxter, H. Abbass, D. Essam, and H. X. Nguyen. Grammar model-based program evolution. In *Proc. of the 2004 Congress on Evolutionary Computation (CEC-2004)*, pages 478–485, 2004.

[16] W. A. Tackett. Mining the genetic program. *IEEE Expert*, 10(3):28–38, 1995.

[17] K. Yanai and H. Iba. Program evolution by integrating EDP and GP. In *Proc. of the Genetic and Evolutionary Computation (GECCO-2004)*, pages 774–785, 2004.