Genetic programming using adaptable stochastic flow of control

ARTUR RATAJ
Institute of Theoretical and Applied Computer Science,
Baltycka 5, Gliwice, Poland
arataj@iitis.gliwice.pl

Received 17 November 2010, Revised 1 December 2010, Accepted 21 December 2010.

Abstract: This paper discusses a replacement of a typical, deterministic flow of control in the evolving program with a probabilistic one, which is then adapted using a supervised learning, until it hopefully converges back to a deterministic flow of control. The probabilistic flow of control allows for a continuous modification of the evolved program, which internally can consist of a set of competing paths of execution.

Keywords: genetic programming, flow of control, supervised learning

1. Introduction

Genetic programming (GP)\cite{Koza92}\cite{Rice95} typically uses a black–box searching known as metaheuristics \cite{Muhlenbein91}. This paper discusses another method of such a searching.

In the proposed method, the evolved program (EP) has a stochastic flow of control (FoC) \cite{Rataj10}, as opposed to the deterministic FoC. In the deterministic, typical FoC, operations are executed in a sequence, except for jumps. In the proposed stochastic FoC, the next operation is randomly picked using probability mass functions (PMF). Each operation has such a single PMF, represented by pairs (probability, address of the next operation). The function, in turn, is iteratively adapted using the MSE of the stochastic processes, that execute the EP. In order to do so, each process collects the data about the actual FoC that was picked while the process was run. After a batch of such processes is run, the collected data of the better fitting processes is considered superficial to that of the less fitting processes. This is a reason, in turn, to update the PMFs of the EP. The update is performed only by a small learning step $\Phi_p$, discussed later. The limited learning step serves the purpose of only partially favouring the best–so–far path of execution. Otherwise, the currently best path of execution could exclude any other path, thus stopping the searching for a possible new path.
A classic individual in a GP population, that can survive by elitist selection [9] or breeding, contains a single deterministic program. In contrast, in the proposed method, because the PMFs defines a number of possible next operations to be executed after the current one, the individual can in a sense be a conglomerate of a very large number of subprograms, sharing the code and picked at random by the processes. Let such a conglomerate is run a number of times $\Phi_K$ in a single fitness evaluation and then the best MSE is chosen.

One can hypothesize about several phenomena that would occur in such a conglomerate:

- the ‘best–so–far’ subprograms can defend the new, better subprograms against extinction, by providing a ‘best–so–far’ MSE;
- the subprograms within a single individual create a family, that shares the same code, allowing for their reuse in various ways and in effect in a progressive evolution in a direction individual for the conglomerate.

The discussed method is similar to simple version of ant colony optimization methods [2], especially these applied to GP like [3][11]. It is also somewhat similar to probability–based program search (PBPS) like Levin Search [6][7] or self-modifying probabilistic learning algorithms [12][13], in the sense, that if some fragment of the path of execution works better, it is favoured. The difference is, though, that the PBPS adapt the probabilistic distributions that decide about the way of searching of the space of all possible programs, while in the proposed method, the adapted probability distributions are a part of the EPs.

The proposed stochastic FoC still needs, as mentioned, a typical GP environment providing at least basic concepts for the evolution of a population of EPs. The premise is, that the concepts be extremely simple, to try the stochastic FoC out in basic, albeit in effect rather harsh conditions, like a flat program with possible type mismatch or argument number mismatch, as opposed to the typical tree of operators approach [5][9], or no explicit, for the EPs, input and output. We will not discuss selection and breeding, as they are not needed to carry out the presented test.

### 2. Processor

Processor ($\Psi$) is an individual within the evolved population, that contains a single EP, and can execute it, that is, run a process.

$\Psi$ contains a common memory for variables, operations and addresses, and can modify the memory. This allows for self–modifications of the program, that in turn allows for a kind of evolution, but in this paper is rather more of a side–effect of the common memory.
Ψ is clearly separated from the further discussed environment, that introduces competition between the processors.

By convention, each coefficient that determines some global, constant trait of the processor begins with (Ψ).

2.1. Memory organisation

The memory of (Ψ_p) has a fixed size (Ψ_M), and contains (W_p^i), where (i) is an address in the range (0, 1, ..., Ψ_M − 1). Each of the cells can contain any of the following:

- operation code,
- arithmetic real value,
- an absolute memory address.

A program is thus a mix of operations, constants and variables within the same space.

2.2. Router

Operations, excluding explicit jumps, are executed in the order specified by the routers. It is thus different from a typical program, where the operations are, except for branches, executed linearly, that is, in the same order in which they are placed in the memory.

The routers’ characteristics, as they decide about the FoC, are the inherent part of the EP, next to the memory contents, and evolve parallelly to the memory contents.

Each cell has its own router. The router (R_p^i) of the cell (W_p^i) contains a table of the size of (Ψ_R). Each entry (e = 0, 1, ..., Ψ_R − 1) in the table contains the address of the next possible operation (A_p^i,e) along with the probability (P_p^i,e), that the respective flow of control, or path, will be realized after the contents of (W_p^i) has been executed. The probabilities for all entries within a single router sum to 1. The router is only used if the operation in (W_p^i) does not realize its own jump, like the unconditional jump or the conditional jump with the condition being true. Note though, that in the tests described in this paper, no jump operations will be allowed. Each time the router is used, an individual address is picked, according to the probabilities. This way, two processes, starting at the same position and with the same contents of the memory, may execute differently, just because of the probabilistic routers.

2.3. Stack

A process has its own stack of a limited size (Ψ_S). The stack is used to pass values between operations. Typically, an operation pops a given number of values from the
stack, processes these values and then pushes the results back to the stack. Overflows and underflows result in finishing of the process.

Any value within the stack can hold exactly the same, what a memory’s cell can. The stack is common for different types, as opposed to some other GP systems like [4].

2.4. Operations

If any operation pops from the stack an incompatible value, for example an operation code when an arithmetic value is expected, a process is finished. If nothing is written about a value in the following list of operations, then the value can be of any type, that is arithmetic, operation code or memory address.

A process, that executed the operations, has its instruction pointer (IP), that defines the operation to execute.

A process is ended, when any of the following occurs:

- **STOP** is encountered, as further discussed;
- an illegal state occurs, as discussed;
- the number of executed operations since the process started reaches the maximum allowed constant ($\Psi_I$).

The list of operations is very limited, as it is only intended for the needs of the further presented test.

- **NONE** does nothing.
- **STOP** stops the process.
- **PUSH** pushes the value read from ($W_i^p$), where ($i$) is equal to the current IP, to the stack; the operation is represented thus not by a distinct operation code, but directly by a plain arithmetic value or a plain address.
- **LOAD** pops an address from the stack, and then pushes the value read from a memory location at that address to the stack.
- **STORE** pops an address from the stack, and then a value, and writes that value to a memory location at the address.
- **MULT** pops two values form the stack, multiplies them and then pushes the result to the stack.
2.5. Input/output

Data transfer to and from the process is performed by defining a set of cells. By convention, these cells are placed at the end of the memory, in the order first input and then output cells. Training data is written to the input cells, the output cells are cleared to have no arithmetic value, then the program is executed, and finally data is read from the output cells, if any.

For the process, the input/output cells cells are just regular memory cells, with the exception that they are externally modified before the process is run. The process is not aware in any explicit way where the input/output cells are located.

3. Environment

Environment \((\Phi)\) introduces a competition between \(\Psi_s\), to drive their evolution. \(\Phi\) contains \((\Phi_N)\) separate \(\Psi_s\). \(\Phi\) evolves the processors iteratively in a loop, that consists of subsequent supervised learning epochs \((\Xi_k)\), where \((k \in 0, 1, \ldots)\) is the epoch number. Within a single epoch, all processors are tested for fitness to given training data.

By convention, each coefficient that determines some global, constant trait of the environment begins with \((\Phi)\).

3.1. Testing

Let \((\Psi_p)\) be tested in order to estimate its fitness. In order to so, \((Q)\) tests \((t_q, q \in 0, 1, Q - 1)\), are be performed.

Let there be \((\Phi_T)\) samples in the training set. For each \((t_q)\), a training sample is randomly picked with replacements. The replacement simulates a typical situation, where there can be repetitions within queries made to a learning machine. Let

\[
Q = \lfloor \Phi_T m T + 1 / 2 \rfloor. \tag{1}
\]

The value of \((\Phi_T m)\) is a trade–off between:

- on the one side, a good coverage of the training set, reduction of noise caused by the stochastic routers and by the random picking of samples;
- on the other side, the computational complexity.

The coefficient \((\Phi_T m)\) is typically larger than \((1)\).

Each test \((t_q)\) runs \((\Phi_K)\) processes \((P_k^q, k = 0, 1, \ldots \Phi_K - 1)\), on a given random sample picked from the training set. Let the local MSE \((E_q^L)\) be computed, achieved by the best \((P_k^q)\) on that sample:

\[
E_q^L = \min_k E_k^q \tag{2}
\]
where \((E^q_k)\) is the MSE of \((P^q_k)\) over the predicted attributes of the random sample. The functionality of \((\Phi_K)\) has already been discussed in Sec. 1.

If the output cell contains no value after the process ends, then the respective \((E^q_k = \Phi_V)\). The constant \((\Phi_V)\) should probably have

- a relatively large value, so that ‘no value’ programs are regarded as the bad ones;
- but not too large, because a good program can possibly also occasionally produce no value, because of the stochastic FoC.

Let the average of all \((E^q_k)\) within the \((Q)\) tests be a global, epoch–level MSE \((E^G_p)\) that defines \((\Psi_p)\)’ performance:

\[
E^G_p = Q^{-1} \sum_q E^L_q \tag{3}
\]

3.2. Learning of routers

The environment encourages certain flows of control by modifying the routers, and in this way, heuristically decreases the evolved program’s \((E^G_p)\).

Let each router \((R^p_i)\) remembers, how many times \((P^p,i,q,k,e)\) each of its entries was chosen during execution of \((P^q_k)\), where \((e)\) is the entry number, \((e = 0, 1, \ldots, \Psi_R - 1)\).

Let,

1. for computational efficiency;
2. to have only one version of the evolved program per processor per epoch, so that it is clear what program is responsible for \((E^G_p)\),

the routers be updated only once per epoch. Thus, let the statistics collected by routers \((F^p,i,q,k,e)\) be accumulated into epoch–level statistic, that sum up router decisions within the whole estimation of \((E^G_p)\). Let the epoch–level statistic of \((F^p,i,q,k,e)\) be \((F^G_{p,i,e})\). As the outcome of the probabilistic routing is in general reflected by the process’ MSE, then let \((F^p,i,q,k,e)\) influence on \((F^G_{p,i,e})\) with \((E^q_k)\) taken into account:

\[
F^G_{p,i,e} = \sum_{q,k} \left((E^q_k + \Phi_\zeta)^{-1} F^p,i,q,k,e\right) \tag{4}
\]

so, the lower the MSE, the more important are statistics of the respective process, and the importance is shaped by \((\Phi_\zeta)\). The lesser the coefficient, the more important are processes with low MSE.

Once we have the global epoch–level router statistics \((F^G_{p,i,e})\), the routers can be updated. Let the probabilities in \((R^p_i)\), while the tests \((t_q)\) were run, be \((P^p_{i,e})\). Let the
new probabilities, after the update of routers, be \((P'_{p,i,e})\). Let

\[
z = \begin{cases} 
\frac{F^G_{p,i,e} - F^G_{p,i,min}}{F^G_{p,i,max} - F^G_{p,i,min}} - \frac{1}{2} & \text{if } F^G_{p,i,min} \neq F^G_{p,i,max} \\
0 & \text{if } F^G_{p,i,min} = F^G_{p,i,max}
\end{cases}
\]

\[
t = P^p_{p,i,e} + \Phi \mu z
\]

\[
P^{sp}_{p,i,e} = \begin{cases} 
0 & \text{if } t < 0 \\
t & \text{if } 0 \leq t \leq 1 \\
1 & \text{if } t > 1
\end{cases}
\]

\[
P'^{p}_{p,i,e} = m P^{sp}_{p,i,e}
\]

where \((\Phi \mu)\) is the learning step, \((F^G_{p,i,min})\) and \((F^G_{p,i,max})\) are respectively the minimum and the maximum values within \((\{F^G_{p,i,0}, F^G_{p,i,1}, \ldots F^G_{p,i,\Psi R-1}\})\), and

\[
m = \left(\sum_e P^{sp}_{p,i,e}\right)^{-1}
\]

makes \(\Sigma_e P'^{p}_{p,i,e} = 1\).

Thus, because of (4), the frequent paths in the processes yielding low MSE are the most encouraged in (5) at the cost of the other paths. Choosing of the value of \((\Phi \mu)\) has been discussed in Sec. 1.

The router learning is only heuristic – the evolving program may better thanks to it, but it may get worse as well. For example, a certain entry can have a high probability \((P^p_{p,i,e})\) because of:

- a great number of meaningless flows of control, that did not decrease the respective MSE in any way;
- a small number of flows of control, that caused the MSE to increase considerably, because the alternate entry \((f \neq e)\) was not chosen instead;
- the respective process still, despite the choice of \((e)\), has a relatively low MSE.

Thus, to make the program better, \((P^p_{p,i,e})\) should be decreased, but the described algorithm may increase it instead.

### 3.3. Mutation of routers

The discussed learning of routers is not enough – no addresses are modified. Routers are mutated in each new child program, and, to reflect the performance of parents on the
number of mutations of the child’s routers, the number of such mutations \(m\) is computed using the parents’ scores \((S^i_k)\) and \((S^j_k)\), where \((i)\) and \((j)\) are parent processors’ indices in the population from the epoch \((\Xi_k)\). Let

\[ m = \Psi^M_p (1 - S^i_k) (1 - S^j_k) \]  

(7)

where \((\Psi^M_p)\) is an environment constant, that specifies the general level of mutations of the routers’ addresses, and scores represent the relative performance within a population. The score of \((\Psi_p)\) is computed as follows. Let \((E^G_{\text{min}})\) and \((E^G_{\text{max}})\) be, respectively, the minimum and maximum MSE among all processors in \((\Xi_k)\). Let

\[ z = \frac{E^G_{\text{max}} - E^G_{\text{min}}}{E^G_{\text{min}} + E^G_{\text{max}}} \]

\[ S^k_p = \begin{cases} 
(1 - z)\Phi_R + z \left(1 - \frac{E^G_p - E^G_{\text{min}}}{E^G_{\text{max}} - E^G_{\text{min}}} \right) & \text{if } E^G_{\text{min}} \neq E^G_{\text{max}} \\
\Phi_R & \text{if } E^G_{\text{min}} = E^G_{\text{max}} 
\end{cases} \]  

(8)

So, the score is in the maximum range of 0 for the worst processors to 1 for the best processors in the extreme case. Yet, the more similar is the MSE the processors earn, the more \((S^k_p)\) converges to \((\Phi_R)\). The reason is, that a similar MSE can mean, that the score becomes too unreliable to be given such extreme values as (0) and (1). This unreliability is caused especially by the random pick of samples for tests, and by the stochastic nature of routers.

The mutation of each router is as follows.

A router \(R^p_i\) is picked, where \(i\) is chosen at random from the address space \(0, 1, \ldots \Psi_M\) using an uniform distribution. Then, an arbitrary entry \(e\) is chosen, that has the minimum value of \(P^p_{i,e}\) within all entries of \(R^p_i\). Finally, the respective address \(A^p_{i,e}\) is set to a random one, using the same probability distribution as for \(i\).

Thus, by choosing the least probability \((P^p_{i,e})\), the likely least usable flow of control is replaced by a random one.

To give the new entry some substantial chance of being tested out, the respective probability \(P^p_{i,e}\) is increased to \(\Psi_M^p_{\text{min}}\) if it is lesser than \(\Psi_M^p_{\text{min}}\), at the cost of the other probabilities, that are multiplied by a certain constant, such that all router’s probabilities sum correctly to 1.

\[ 4. \text{Tests} \]

Let us test, how fast the routers evolute, given a single processor, thus, \((\Phi_N = 1)\). Let its state before each evolution test be as presented in Fig. 1. The state is represented as an array with subsequent cells left–to–right. Each cell contains its address preceded
by ‘#’, its contents, and its router’s state. The router’s state is represented by \((\Psi_R)\) lines, each line contains a pair (respective probability, destination address), separated by an arrow.

As can be seen, \(\Psi_M = 10\) and \(\Psi_R = 3\). Let the fitness function be \(y(x) = 2x\). Let the processor be, in each epoch, subject to the described testing and then router learning and mutation. Let the number of training samples be \(\Phi_T = 5\), and let the samples be \((x, y)\) pairs \((0, 1), (1, 2), \ldots (\Phi_T - 1, 2(\Phi_T - 1))\). Let us test the number of epochs, \(X(\Phi_\mu, \Phi_K)\), and the total number of tests \(t_q\), \((T(\Phi_\mu, \Phi_K))\), needed to evolve an EP that has \(E_p^G = 0\), for different values of the learning step \(\Phi_\mu\) and the number of tests \(\Phi_K\). Only single test is performed per a single point in the domain of the functions \(X(\Phi_\mu, \Phi_K)\) and \(T(\Phi_\mu, \Phi_K)\), what will give noisy results, as the functions are obviously both stochastic. Yet, a very large number of such points \((\Phi_\mu, \Phi_K)\) will be sampled, so that it will be possible to evaluate the character of the noise and the local mean values.

<table>
<thead>
<tr>
<th>#0</th>
<th>#1 LOAD</th>
<th>#2 SAVE</th>
<th>#3 STOP</th>
<th>#4 MULT</th>
<th>#5 NONE</th>
<th>#6</th>
<th>#7 2.000</th>
<th>#8 NONE</th>
<th>#9 NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.004</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Fig. 1. A 10–cell processor with 3–entry routers, in the initial state.

Let the stack size be \(\Psi_S = 10\) and maximum process’ length be \(\Psi_I = 10\), which is enough for the problem, that requires at minimum, the stack of two values, and 6 executed operations. Let \(\Phi_T = 2\), for a reasonable coverage of the training samples.

As can be seen, the initial state contains all operations needed to evolve a \(\Psi_p\) that has \(E_p^G = 0\) for the described task. In particular, only the modification of the router at \([3]\) is needed to obtain a EP that realises \(y(x) = x\). To obtain \(y(x) = 2x\) in turn, two additional operations, scattered around in the memory, need to be additionally incorporated into the program, what required learning of several routers. It can be supposed, that evolving the EP that represents \(y(x) = x\) is easier, as only a single router needs to be modified, and, that evolving the function will be an intermediate step, because it gives much lower \(E_p^G\) than the initial program, which produces no values, and \(\Phi_V = 10000\) gives it a relatively large \(E_p^G\).

Let \(\Phi_\zeta = 0.1\), a much smaller value than \(E_p^G\) of an EP that represents \(y(x) = x\), thus, the system has a strong drive to produce an even better EP. Let \(\Phi_p^{\text{min}} = 0.1\) gives a chance of the new solution to evolve, without considerably disrupting the existing solution.

Let \(\Phi_\mu\) be tested in the range \(0 \ldots 0.02\) at the resolution of \(0.0002\), and let \(\Phi_K\) be tested in the range of \(1 \ldots 100\), at the resolution of \(1\). Let us limit the maximum number
of learning epochs per each argument \((\Phi_\mu, \Phi_K)\) to 1000, as:

1. number of epochs needed will be, in most cases in which \(\Phi_\mu\) and \(\Phi_K\) have near-optimal values, much lower;

2. in real GP, the population is typically subject to selection and mutation, what might destroy an evolving system of operations if the routers need a very large number of epochs to learn the desired function, and in effect to reach MSE low enough to ‘defend’ the system from being destroyed or getting extinct.

Fig. 2. The test’s results: (a) number of epochs \(X(\Phi_\mu, \Phi_K)\), (b) number of tests \(T(\Phi_\mu, \Phi_K)\). Isolines show locally averaged values.

Fig. 2 shows the results of the test. It can be seen, that for very low values of \(\Phi_\mu\), lesser that 0.005, the EPs evolve very slowly, and that further increase of \(\Phi_\mu\) at about 0.15, has a progressively smaller effect on the speed of evolution. We do not test \(\Phi_\mu\) values larger that 0.2, as while in the discussed extremely simple case it might speed up the learning even more, yet in a general case, as discussed, high values of \(\Phi_\mu\) might aggressively select only a single or few ‘best so far’ paths, thus decreasing the search range of alternative paths.

The optimum values of \(\Phi_K\) seems to be around 50 for the test case – lower values cause \(X(\Phi_\mu, \Phi_K)\) to be high, thus making the EPs more likely to be destroyed or extinct, as already discussed in this section. On the other hand, values of \(\Phi_K\) higher than the optimum value make generally \(T(\Phi_\mu, \Phi_K)\) larger, what translates to higher computational cost, thus, slower evolution.

5. Conclusion

The proposed method is a way of evolving programs, that can work beside mutation, selection and breeding, and possibly help these.
Tests have shown, that the learning of routers works in a very simple case without a real evolution of population, and so it is though an open question, if the method would have any substantial advantage a real GP environment.

The tested EPs usually converged to ones with a deterministic FoC, as the routers’ probabilities were converging to either 0 or 1. Yet, the cases are possible where an EP converges to a program with a stochastic FoC. For example, let in some perfect algorithm, a given operation should be executed only once per two loops. what the EP might imperfectly simulate instead with a stochastic FoC. One of the partial solutions to such cases might be computing the statistics of the routers, that do not converge to deterministic ones, in the context of values of variables, and if dependencies between these would be observed, a router might be replaced with, for example, a conditional branch.

References

Artykuł omawia zamienną typowego, deterministycznego przepływu sterowania w programie tworzonego algorytmem genetycznym, na probabilistyczny przepływ sterowania, który z kolei jest adaptowany poprzez uczenie nadzorowane, aż ewentualnie w kolejnych iteracjach uzyska zbieżność do szczególnego przypadku deterministycznego przepływu sterowania.

Stochastyczny przepływ sterowania umożliwia płynną modyfikację tworzonego programu, który wewnętrznie jest złożony ze zbioru konkurujących ze sobą sposobów wykonania.

Metoda jest testowana dla prostego przypadku, w teście szacowane są również optymalne wartości współczynników decydujących o adaptacji przepływu sterowania.