An Algorithmic Chemistry for Genetic Programming

Christian W.G. Lasarczyk[1] and Wolfgang Banzhaf[2,⋆]

[1] Department of Computer Science, University of Dortmund,
D-44221 Dortmund, Germany
christian.lasarczyk@uni-dortmund.de
[2] Department of Computer Science, Memorial University of Newfoundland,
St. John's, NL, A1B 3X5, Canada
banzhaf@cs.mun.ca

Abstract. Genetic Programming has been slow at realizing other programming paradigms than conventional, deterministic, sequential von-Neumann type algorithms. In this contribution we discuss a new method of execution of programs introduced recently: Algorithmic Chemistries. Therein, register machine instructions are executed in a non–deterministic order, following a probability distribution. Program behavior is thus highly dependent on frequency of instructions and connectivity between registers. Here we demonstrate the performance of GP on evolving solutions to a parity problem in a system of this type.

1 Introduction

Representations in genetic programming encode functionality both explicitly by choosing from a set of operations and implicitly by choosing a position within the genome. While it is "easy" to inherit the explicitly encoded portion of functionality, variable genome length leads to difficulties in inheritance of implicitly encoded functionality.

In this contribution we present a different way of looking at transformations from input to output that does not require a prescribed sequence of computational steps and therefore no implicitly coded functionality. Instead, the elements of the transformation, which in our case are single instructions from a multiset $I = \{I_1, I_2, I_3, I_2, I_3, I_1, \ldots\}$ are drawn in a random order to produce a transformation result. In this way we dissolve the explicit sequential order usually associated with an algorithm for our programs.

A program in this sense is thus not a sequence of instructions but rather an assemblage of instructions that can be executed in arbitrary order. By randomly

⋆ The authors gratefully acknowledge support from a grant of the Deutsche Forschungsgemeinschaft DFG (German Research Foundation) to W.B. under Ba 1042/7–3.

M. Keijzer et al. (Eds.): EuroGP 2005, LNCS 3447, pp. 1–12, 2005.

choosing one instruction at a time, the program proceeds through its transformations until a predetermined number of instructions has been executed. It is therfore more akin to a chemical system with data as educts and products, and operations as reactions than to an "orderly" execution of code.

Programs of this type can be seen as artificial chemistries, where instructions interact with each other (by taking the transformation results from one instruction and feeding them into another). Different multisets can be considered different programs, whereas different passes through a multiset can be considered different behavioral variants of a single program.

Because instructions are drawn randomly in the execution of the program, it is really the concentration of instructions that matters most. It is thus expected that "programming" of such a system requires the proper concentration of instructions, while an explicit sequencing is not required.

At first, this kind of repeated execution of instructions seems to be a waste of computational power. While it is always possible to transform a snapshot[1] of an individual into a linear program, hardware centered improvements of execution speed are imaginable, too. E.g., a huge number of processors could execute the same multiset of instructions in parallel. In the extreme case of the number of processors equal to the number of instructions running time is reduced to a minimum predetermined by depth of data flow. Specialized multiprocessor systems, such as the wavescalar–architecture[1, 2], hold potential to achieve this speed up using less processors.

Due to the stochastic nature of results, it might be advisable to execute a program multiple times before a conclusion is drawn about its "real" output. In this way, it is again the concentration of output results that matters. Therefore, a number of n passes through the program should be taken before any reliable conclusion about its result can be drawn. Reliability in this sense would be in the eye of the beholder. Should results turn out to be not reliable enough, simply increasing n would help to narrow down the uncertainty. Thus the method is perfectly scalable, with more computational power thrown at the problem achieving more accurate results.

We believe that, despite the admitted inefficiency of our approach in the small, it might well beat sequential or synchronized computing at large, if we imagine tens of thousands or millions of processors at work.

Algorithmic Chemistries were considered earlier in the work of Fontana [3]. In our contribution we use the term as an umbrella term for those kinds of artificial chemistries [4] that aim at algorithms. As opposed to terms like randomized or probabilistic algorithms, in which a certain degree of stochasticity is introduced explicitely, our algorithms have an implicit type of stochasticity. Executing the sequence of instructions every time in a different order has the potential of producing highly unpredictable results.

It will turn out, however, that even though the resulting computation is unpredictable in principle, evolution will favor those multisets of instructions that

[1] Ambiguousness starts, if different instructions share the same target.

turn out to produce approximately correct results after execution. This feature of approximating the wished-for results is a consequence of the evolutionary forces of mutation, recombination and selection, and will have nothing to do with the actual order in which instructions are being executed. Irrespective of how many processors would work on the multiset, the results of the computation would tend to fall into the same band of approximation. We submit, therefore, that methods like this can be very useful in parallel and distributed environments.

Following previous work on Artificial Chemistries (see, for example [5, 6, 7, 8]), [9] introduces a very general analogy between chemical reaction and algorithmic computation, arguing that concentrations of results would be important. [10] was the first step in this new direction. Here we want to deepen our understanding of the resulting system by studying the GP task of even-parity.

2 Algorithmic Chemistry

On executing a sequence of instructions using linear GP[11], each point in execution time is assigned to exactly one instruction, which is executed at that very moment. This principle is even the same, if instructions are stored in a tree like data structure (e.g. Tree–GP[12]).

Applying Tree–GP, functional dependence of instructions is related to their distance within the tree. Subtrees possess sub–functionality, an edge carries an implicit specification for the subtree it connects to the tree. This specification has to be satisfied during recombination. Using linear GP, functional dependence is determined by both, distance within the genome and source and target registers used by instructions. Therefore successful recombination has to consider both.

2.1 GP to AC — A Gradual Transition

Here we shall use 3–address machine instructions. The genotype of an individual is a list of those instructions. Each instruction consists of an operation, a destination register, and two source registers[2]. Initially, individuals are produced by randomly choosing instructions. As is usual, we employ a set of fitness cases in order to evaluate (and subsequently select) individuals.

A time–dependent probability distribution determines the sequence of instructions. Linear GP uses a discrete distribution:

$$P_t(X = x_i) = \begin{cases} 1, \text{if } i = t \\ 0, \text{else} \end{cases}, \quad t, i \in 1, 2, \ldots, n. \tag{1}$$

Position in memory is denoted by x_i, and the individual consists of n instructions. Starting at $t = 1$ exactly one instruction gets executed at each moment in time, followed by the next instruction in memory until at $t = n$ all instructions got executed in exactly the same order as they appear in memory. This is shown on left side of Fig. 1. Thus, the location in memory space determines

[2] Operations, which require only one source register, simply ignore the second register.

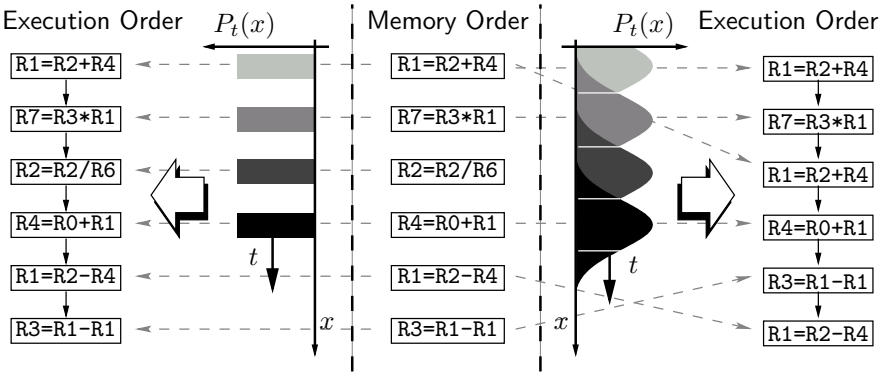

Fig. 1. Execution of an individual. Transition from memory to execution order is determined by a time dependent distribution function. Left side shows transition using distribution function of Eq. 1, emulating linear GP. On right side transition occurs by using a normal distribution. Different gray tones of distribution functions represent different points in time

the particular sequence of instructions. Classically, this is realized by the program counter. Each instruction is executed, with resulting data stored in its destination register.

If we use a distribution to access instructions as described above, we come to a new class of algorithms by changing this distribution. On the right side of Fig. 1 shows a different execution order result from using a Gaussian distribution. If the standard deviation σ is increased the influence of time on instruction selection decreases. In the extreme case $\sigma \to \infty$ a uniform distribution results and all instructions have the same probability to be drawn at any moment. This we call an Algorithmic Chemistry.

Using a uniform distribution, behavior of a program during execution will differ from instance to instance. There is no guarantee that an instruction is executed, nor is it guaranteed that this happens in a definite order or frequency. If, however, an instruction is more frequent in the multi-set, then its execution will be more probable. Similarly, if it should be advantageous to keep independence between data paths, the corresponding registers should be different in such a way that the instructions are not connecting to each other. Both features would be expected to be subject to evolutionary forces.

As shown in Fig. 2, also 1–Point–*Crossover* could be described, using time depended distributions. While the first part of an offspring is formed by instructions drawn from first parent during time interval $1 \leq t \leq c_1 (\leq n_1)$, the second part is drawn from second parent during time interval $(1 \leq) c_2 \leq t \leq n_2$.

Though the instructions inherited from each of the parents are located in contiguous memory locations, the actual sequence of the execution is not determined by that order once we use a distribution to access instructions. The probability that a particular instruction is copied into an offspring again depends on the frequency of that instruction in the parent. Inheritance therefore

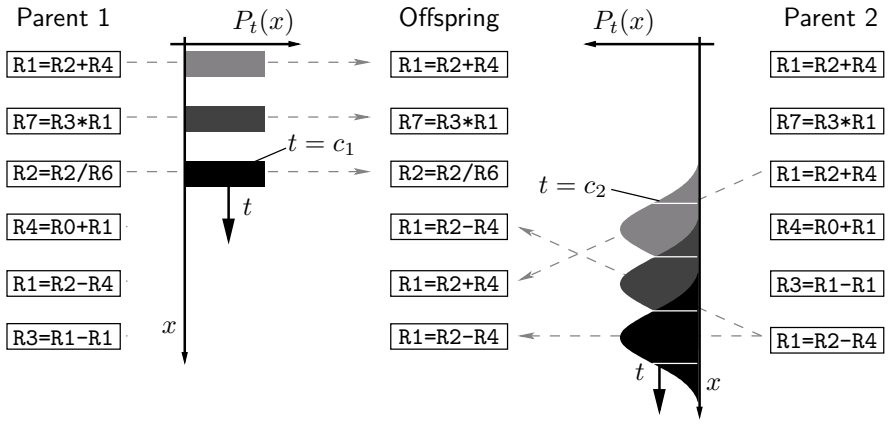

Fig. 2. Recombination accesses parents instruction via the same time dependent distribution function $P_t(x)$ used at evaluation. While we take instruction from the first parent during time span $1 \leq t \leq c_1$, we draw instruction from the second parent during time span $c_2 \leq t \leq n_2$. In contrast to the depict situation $P_t(x)$ is the same for accessing both parents instructions

is inheritance of frequencies of instructions, rather than of particular sequences of instructions.

Estimation of Distribution Algorithms (EDAs). Estimation of Distribution Algorithms[13] are a relatively new class of approaches to evolutionary computation. These population based algorithms generate offsprings by two steps, omitting crossover and mutation. At first, they estimate the probability distribution of a selected subset of the current population, and subsequently they sample a new population from this distribution. We can think of an Algorithmic Chemistries as an implicit description of an instruction distribution by storing a set of samples from this distribution. Recombination is similar to creating a new common distribution based on two of the selected individuals and sampling an offspring from it. While this kind of sampling is not able to create something new, mutation is still needed.

2.2 Algorithmic Chemistry in Detail

Having derived execution, utilizing a 3–address–machine, and crossover of individuals on Algorithmic Chemistry for Genetic Programming(ACGP) from linear GP, we explain further details in this section, including additional information on crossover and evaluation.

Registers. We distinguish between three different kinds of registers:

- connection registers
- input registers
- registers containing constant values

While instructions could read from all of them, and thus can use them as a source register, they can just write to *connection registers*. Therefore these are the only valid targets to store instruction results. Information flows among them in the course of computation. The number of connection registers could be set as an evolution parameter. Values in connection registers are set to zero at the beginning of an evaluation. Each *input register* contains data of a single fitness case at the beginning of execution, where the number of these registers is determined by the problem tackled. The third type of registers contain *constant values* evolved during evolution. The choice of a *result register* out of connection registers is done by evolution.

More About Crossover and Evaluation. Parents are chosen randomly for each offspring. Crossover rate assigns the proportion of offspring created by recombination, the rest of offsprings is created by reproductive cloning. In both cases mutation is applied afterwards. During crossover constant register values will be copied with equal probability from each parent, as is done for the choice of the result register if necessary.

The number of executed instructions on linear GP and Tree–GP is limited by the number of instructions contained in an individual's genome. As described in Sec. 1 an instructions in an Algorithmic Chemistry can be executed successfully – in fitness improving sense – if all required sources contain correct inputs. Therefore, it could be reasonable to increase the upper limit on execution and cycle ($P_{n+t} = P_t$) through individuals more than once. In the case of a constant uniform distribution, as used by the Algorithmic Chemistry presented here, this means that we could execute available instructions multiple times by drawing them randomly. The number of *cycles*, is an additional evolution parameter.

Because evaluating an individual is a stochastic process, it could be useful to evaluate individuals more than once and combine the results to get a single fitness value. We will discuss this in detail later.

Initialization and Mutation. Initialization and mutation of an individual are the same for both the ACGP and usual linear GP. Mutation changes single instructions by changing operation, target register and the source registers according to a prescribed probability. Register values are mutated by using a Gaussian with mean at present value and standard deviation 1.

Selection. We use a (μ, λ)–strategy. In doing so a set of μ parents produce λ offspring first. The λ best individuals of these offspring form the set of next generation's parents.

3 Results and Outlook

Since in [10] we already discussed an approximation and a real-world classification problem, we now evolve a Boolean function using Genetic Programming of Algorithmic Chemistries.

3.1 Even–Parity Problem

Boolean problems are used as popular benchmark problems in GP. The even-parity problem, widely discussed in [12], tries to generate the value of a bit, so that with an input of three external bits, even parity is provided. The individuals can use four logical operations {AND, OR, NAND, NOR}. The cost for random search has been discussed in [14].

ACGP uses real–valued registers. For boolean operations, > 0 values will be mapped to true, ≤ 0 values to false. The fitness function corresponds to the fraction of fitness cases an individual could not generate even parity for. The solution hoped for is to have a fitness of zero.

3.2 ACGP Settings

We do non claim, that we use optimized settings. Nevertheless we think it is important to describe the amount of optimization done so far and describe our settings.

To chose an appropriate setting we create a space filling latin hypercube design with 50 runs on a reduced subspace of our parameter space. Roughly speaking this means, that we divide each parameter into 50 evenly spaced levels[3] and then choose 50 points in parameter space maximizing minimum distance between points considering theses levels.

Table 1. ACGP settings and ranges of our space filling design

parameter	setting	design range min	design range max
offsprings	450	200	500
crossover rate	0.45	0.0	0.6
mutation rate	0.01	0.0	0.05
initial length	50	10	50
maximal length	150	300	500
cycles	3.5	1	5
connection register	40	30	60
parents	100		
evaluations per ind. (m)	1,2,4,8,16	8	
evolved constants	2		

For each design point we start four runs, executing 10^{10} instructions each. Influenced by those runs showing good average test performance we choose our setting. Table 1 shows ranges of considered parameter subspace and finally selected settings.

[3] We even do so for integers and round afterwards.

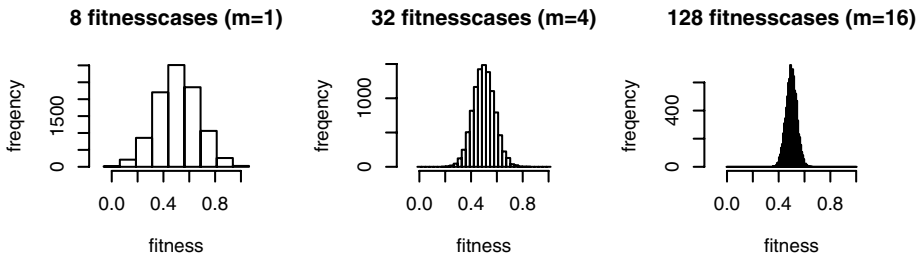

Fig. 3. Fitness distribution of a single initial individual using different training set sizes. These training sets are generated by multiplication of original set containing 8 fitness cases m times. As this is an individual of initial population mean value is expected to be 0.5, standard deviations of noise are $\sigma_{m=1} = 0.163, \sigma_{m=4} = 0.083$ and $\sigma_{m=16} = 0.041$

3.3 Stochastic Noise

During the observation of the even–parity problem a difficulty has occurred, that is unknown for other GP variants: Originating from the small training set of only 8 fitness cases the stochastic noise gains in influence. Through the non–deterministic order of execution of instructions, a multiple execution of a single individual can lead to different results. Especially at the beginning of evolution, while instructions of a chemistry are purely random, and different instructions share the same target register.

To reduce this noise we initially use the concept of repeated evaluation. To do this, we not only execute the algorithmic chemistry of the individual once on the training set of 8 fitness cases, but m times. Accordingly, with $m = 2$ the fitness corresponds to the fraction of 16 fitness cases that did not generate even parity, while there are still just 8 different fitness cases. As a side effect, the resolution of the fitness calculation increases. Instead of 8 different values for $m = 1$ resulting in a 0.125 margin distance, an $m = 4$ approach results in 32 values with a 0.03125 margin distance.

Figure 3 displays the histograms of the fitness values from 10000 analyses of an individual of the initial population using differently sized sets as basis of valuation. In addition to the higher resolution, reduced standard deviation can be observed.

Each increase on size of training set reduces the number of generations for an unchanged number of instructions allowed to be executed. In our system the number of executed instructions serves as a measure of time, each run is allowed to execute 10^{10} instructions.

Figure 4 reveals different fitness development for $m \in \{1, 2, 4, 8, 16\}$, averaged over 100 runs. For evaluation during the training phase, a corresponding number of fitness cases was used. A validation was performed in regular intervals, choosing the best individual based on 128 fitness cases. For testing purposes, fitness is compared to this larger number of fitness cases afterwards.

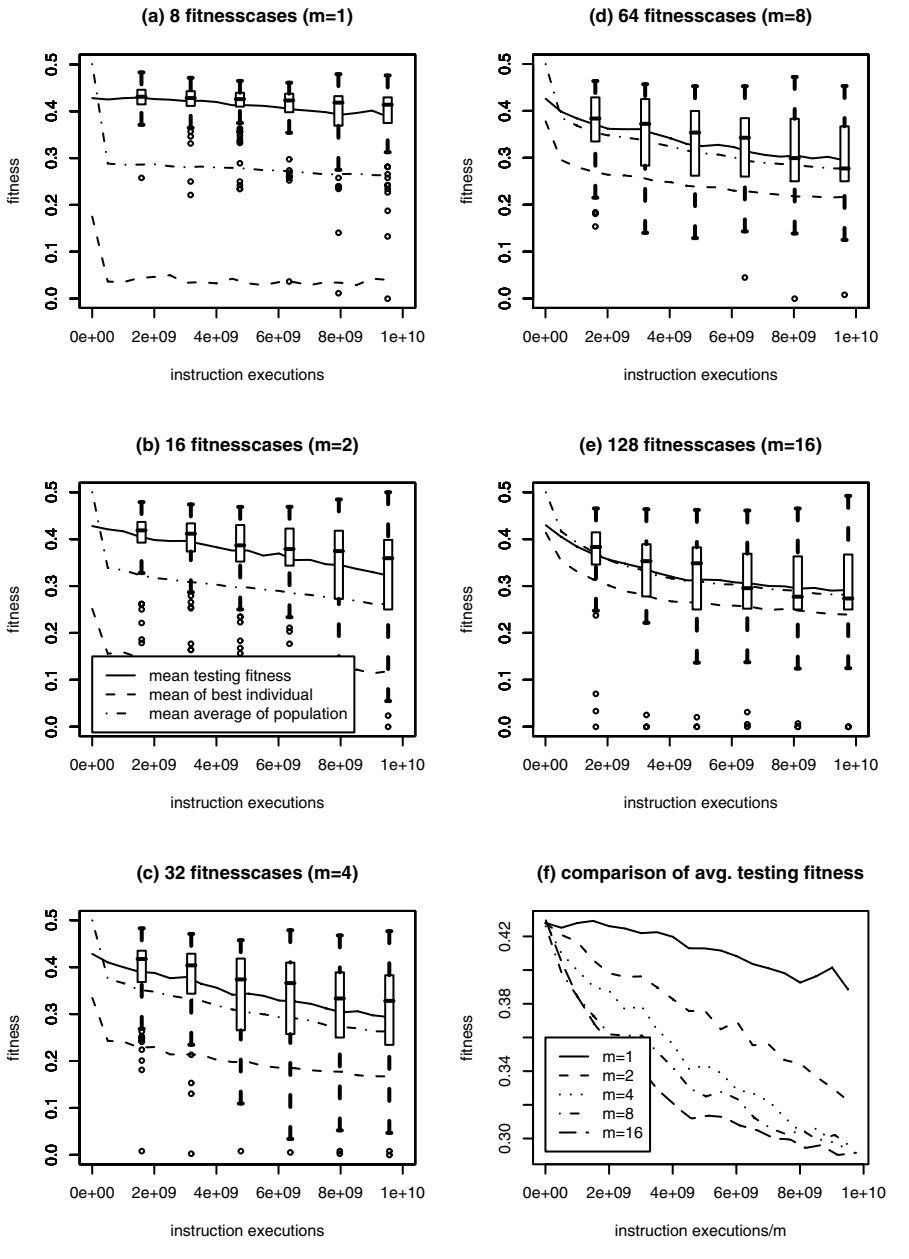

Fig. 4. Using different training set sizes we show fitness of best individual, mean population fitness and fitness on testing set of population's best individual on validation set averaged over 100 runs. The chart in bottom right corner is for direct comparison of achieved testing set fitness

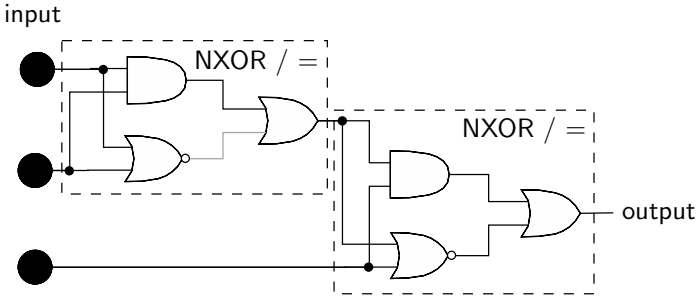

Fig. 5. Circuit of a solution found using Algorithmic Chemistries

The smaller the size of training set, the better the best individual of a population looks on average. Assuming, that the noise for all initial individuals equals that of figure 3, it easily becomes obvious, that among 100 observed individuals one might be chosen whose noise leads to a good fitness. This could prove problematic for evolution, because this selection is not founded and does not hold out against further evaluation. Also, in the next generation, offspring of individuals selected this way might underachieve. This leads to early stagnation among runs with a small training set. For instance, if $m = 1$ (original training set size) most runs do not improve much after initialization and good runs are a very rare event (cf. Fig.4(a) and solid line in Fig.4(f)). Even worse, these runs can pass most generation tests, because using the same individual $m = 1$ requires half as much instruction executions – which are limited – as $m = 2$.

Things get better for $m = 2$, Fig.4(b). While evolutionary improvement take place slowly, these Algorithmic Chemistries continue to evolve. Fitness of the best individual, however, is on average inferior to the case $m = 1$, though more realistic. This trend continues in Fig.4(c-e) with $m \in 4, 8, 16$. Here we can also observe that testing fitness converges to the mean population fitness, which is an indication that effective evolutionary progress is more strongly coupled with population dynamics.

While Algorithmic Chemistries suffer from noise they introduced by non–deterministic instruction execution, this problem can be handled by increasing training set size: By duplicating fitness cases, as done here, noise can be reduced. This reduction in noise, however, comes with an increase in computational power demand. We expect noise reduction in Algorithmic Chemistries to be an important topic for further investigations.

3.4 A Glance at a Solution

Because the data flow of this individual is "nearly unique", it is easy to extract the corresponding circuit, as shown in Fig.5. Here "nearly unique" means, that there are for one connection register two different instructions using it as their target. Because data flow in Algorithmic Chemistries via connection registers, this flow can be symbolized on circuits though conductors. "Nearly unique" con-

ductors are drawn using gray color. Missing uniqueness is caused by a functional intron like this:

```
r27 = r27 AND true
```

The Boolean value `true` resides in a constant register, and it is obvious that register `r27` is not changed through execution. The evolved solution presented here consists of two NXOR–gates. These gates test for equivalence and return 'true' if their inputs are equal.

3.5 Outlook

While using additional cases to evaluate an individual's fitness is able to reduce noise, enlarging a training set is computationally expensive. In fact, applying a training set m times only reduces noise by $\sqrt{m}$. In the future, therefore, sequential sampling techniques [15] shall be used to compare the fitness of individuals. This technique does not fix the number of fitness cases in advance, but ensures a desired level of confidence by treating fitness cases one at a time.

In Sec.3.2 we used a very simple approach to choose settings for ACGP from parameter space. In further work we plan to use methodologies described in [16, 17] useful to analyze and optimize evolutionary algorithms and other search heuristics. Beside an improved system performance we hope for further insights in behavior of algorithmic chemistries.

In this study on our Algorithmic Chemistries for Genetic Programming we considered only uniform distributions of instruction choice. However, other distributions are possible as well. A uniform distribution is, however, the most extreme case, since it ignores order completely when drawing instructions from an equal distribution. By using a normal distribution, we plan to investigate the algorithmic space between linear GP and ACGP.

As mentioned before, there are already some similarities of EDAs and the present approach. By mixing all selected Algorithmic Chemistries (creating a common multi–set of instructions) and drawing new offspring from this "common distribution" we can go one further step in the direction of EDAs.

References

1. Swanson, S., Oskin, M.: Towards a universal building block of molecular and silicon computation. In *1st Workshop on Non-Silicon Computing (NCS-1)* (2002)
2. Swanson, S., Michelson, K., Oskin, M.: Wavescalar. Technical report, University of Washington, Dept. of Computer Science and Engineering (2003)
3. Fontana, W.: Algorithmic chemistry. In Langton, C.G., Taylor, C., Farmer, J.D., Rasmussen, S., eds.: Artificial Life II, Redwood City, CA, Addison-Wesley (1992) 159–210
4. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial Chemistries - A Review. Artificial Life **7** (2001) 225–275
5. Banzhaf, W.: Self-replicating sequences of binary numbers. Comput. Math. Appl. **26** (1993) 1–8

6. di Fenizio, P.S., Dittrich, P., Banzhaf, W., Ziegler, J.: Towards a Theory of Organizations. In Hauhs, M., Lange, H., eds.: Proceedings of the German 5th Workshop on Artificial Life, Bayreuth, Germany, Bayreuth University Press (2000)
7. Dittrich, P., Banzhaf, W.: Self-Evolution in a Constructive Binary String System. Artificial Life **4** (1998) 203–220
8. Ziegler, J., Banzhaf, W.: Evolving Control Metabolisms for a Robot. Artificial Life **7** (2001) 171–190
9. Banzhaf, W.: Self-organizing Algorithms Derived from RNA Interactions. In Banzhaf, W., Eeckman, F., eds.: Evolution and Biocomputing. Volume 899 of LNCS. Springer, Berlin (1995) 69–103
10. Banzhaf, W., Lasarczyk, C.W.G.: Genetic programming of an algorithmic chemistry. In O'Reilly, U.M., Yu, T., Riolo, R., Worzel, B., eds.: Genetic Programming Theory and Practice II. Volume 8 of Genetic Programming. Kluwer/Springer, Boston MA (2004) 175–190
11. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction. Morgan Kaufmann, San Francisco, CA (1998)
12. Koza, J.: Genetic Programming. MIT Press, Cambridge, MA (1992)
13. Mühlenbein, H., Paaß, G.: From recombination of genes to the estimation of distributions: I. Binary parameters. In Voigt, H.M., Ebeling, W., Rechenberg, I., Schwefel, H.P., eds.: Parallel Problem Solving from Nature – PPSN IV, Berlin, Springer (1996) 178–187
14. Langdon, W.B., Poli, R.: Boolean functions fitness spaces. In Poli, R., Nordin, P., Langdon, W.B., Fogarty, T.C., eds.: Genetic Programming: Second European Workshop EuroGP'99, Berlin, Springer (1999) 1–14
15. Branke, J., Schmidt, C.: Sequential sampling in noisy environments. In Yao, X., Burke, E., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J., Kabán, P.T.A., Schwefel, H.P., eds.: Parallel Problem Solving from Nature - PPSN VIII. Volume 3242 of LNCS., Birmingham, UK, Springer-Verlag (2004) 202–211
16. Bartz-Beielstein, T., Markon, S.: Tuning search algorithms for real-world applications: A regression tree based approach. In Greenwood, G.W., ed.: Proc. 2004 Congress on Evolutionary Computation (CEC'04), Portland. Volume 1., Piscataway NJ, IEEE Press (2004) 1111–1118
17. Bartz-Beielstein, T., Parsopoulos, K.E., Vrahatis, M.N.: Analysis of particle swarm optimization using computational statistics. In Simos, T.E., Tsitouras, C., eds.: Proc. Int'l Conf. Numerical Analysis and Applied Mathematics (ICNAAM), Weinheim, Wiley-VCH (2004) 34–37