Transgenic Evolution for Classification Tasks with HERCL

Alan D. Blair

School of Computer Science and Engineering
University of New South Wales
Sydney, 2052 Australia

blair@cse.unsw.edu.au

Abstract. We explore the evolution of programs for classification tasks, using the recently introduced Hierarchical Evolutionary Re-Combination Language (HERCL) which has been designed as an austere and general-purpose language, with a view toward modular evolutionary computation, combining elements from Linear GP with stack-based operations from FORTH. We show that evolved HERCL programs can successfully learn to perform a variety of benchmark classification tasks, and that performance is enhanced by the sharing of genetic material between tasks.

Keywords: transgenic evolutionary computation, evolutionary automatic programming, stack-based genetic programming

1 Introduction

The Hierarchical Evolutionary Re-Combination paradigm and associated HERCL programming language were recently introduced [3] in an effort to provide a novel framework for evolutionary automatic programming, designed to be suitable for transfer learning between tasks [18] as well as for the future development of modular evolving systems [4].

HERCL has been designed as an austere and general-purpose language, combining elements from Linear GP [16] with stack-based operations from FORTH [5]. As such, it draws on the tradition of induced subroutines [1] and Automatically Defined Functions [11, 17, 9, 23] as well as stack-based GP [19, 6] and related approaches [20, 22].

In previous work, we have shown how HERCL programs can be evolved to perform dynamically unstable control tasks [4] as well as coding tasks such as the Caesar and Vigenere cipher [3]. Although challenging, these coding tasks dealt exclusively with synthetic data derived from very precise rules, so that as soon as the training error reached zero, the test set error generally also went to zero. In the present work, we take this research in a new direction and explore the ability of HERCL programs to capture underlying patterns in real-world datasets and generalize to unseen data, by testing them on six benchmark classification tasks. We are particularly interested in the question of whether the evolution of one task can be improved (in terms of speed, parsimony or accuracy) by the sharing of transgenic material from agents evolved on other, related tasks.

```
  out: [-0.98]
  mem: .............................
  reg: gtattctcaacaagattaaccgacagattcaatctcgtggatggacgttcaacattg
stack: [0.93]t[1.39][1.77]..............................
                     ^
 0[15<14<49g=:x|33=x:35=:1:|15g}:38=:21<38g:q|cn%y3.906#p3.#thwo]
                                                               ^
```

Fig. 1. HERCL simulator, showing an evolved agent executing the PROMOTERS task. Note the output buffer, memory, registers, stack and code. All items are floating point numbers, but the simulator prints them as a dot (zero), an ASCII character, or bracketed in decimal format, depending on their value.

2 The HERCL programming language

HERCL agents have a stack, registers and memory. The number of registers, size of memory and (maximum) size of the stack are part of the specification of the agent, along with the code – which is divided hierarchically into *cells*, *bars* and *instructions*. Each *cell* contains a sequence of executable instructions, and might alternatively be thought of as a "procedure" or "subroutine". The pipe symbol (|) is used as a kind of *bar line* to divide each cell into smaller chunks or *bars*, like the bars in a musical score. Every instruction consists of a (single-character) *command*, optionally preceded by a sequence of dot/digits which form the "argument" for that command. The various commands are listed in Table 1.

The language has been designed with the specific aim of allowing new programs to be created by combining portions or *patches* of code from other (heterogenous) programs, at multiple scales (bar, cell or multi-cell) in such a way that the functionality of the transplanted code would be substantially preserved. Whenever such a patch is applied, it is followed up by a series of smaller-scale patches or mutations in the vicinity of the original patch, in an effort to make the new code integrate felicitously with the surrounding code. These smaller patches are, in turn, followed up by yet smaller mutations, recursively, to produce a global random search strategy known as hierarchical evolutionary re-combination.

3 Hierarchical Evolutionary Re-Combination

HERCL does not use a population as such, but instead maintains a stack or *ladder* of candidate solutions (agents), and a *codebank* of potential mates (Figure 2). At each step of the algorithm, we select the agent at the top rung of the ladder and apply either mutation or crossover with a randomly chosen agent from the codebank, or from an external *library* (explained below). We distinguish different *levels* of mutation/crossover (TUNE, POINT, BAR, BRANCH, CELL, JUMP or BLOCK) which vary according to the portion of code from the primary (ladder) parent that is either mutated or crossed over with a commensurate portion of

Table 1. HERCL Commands

	Input and Output	
`i`	fetch INPUT to input buffer	
`s`	SCAN item from input buffer to stack	
`w`	WRITE item from stack to output buffer	
`o`	flush OUTPUT buffer	
	Stack Manipulation and Arithmetic	
`#`	PUSH new item to stack	$\ldots\ldots \mapsto \ldots\ldots x$
`!`	POP top item from stack	$\ldots\ldots x \mapsto \ldots\ldots$
`c`	COPY top item on stack	$\ldots\ldots x \mapsto \ldots\ldots x, x$
`x`	SWAP top two items	$\ldots y, x \mapsto \ldots x, y$
`y`	ROTATE top three items	$z, y, x \mapsto x, z, y$
`-`	NEGATE top item	$\ldots\ldots x \mapsto \ldots\ldots (-x)$
`+`	ADD top two items	$\ldots y, x \mapsto \ldots (y + x)$
`*`	MULTIPLY top two items	$\ldots y, x \mapsto \ldots (y * x)$
	Mathematical Functions	
`r`	RECIPROCAL	$.. x \to .. 1/x$
`q`	SQUARE ROOT	$.. x \to .. \sqrt{x}$
`e`	EXPONENTIAL	$.. x \mapsto .. e^x$
`n`	(natural) LOGARITHM	$.. x \mapsto .. \log_e(x)$
`a`	ARCSINE	$.. x \mapsto .. \sin^{-1}(x)$
`h`	TANH	$.. x \mapsto .. \tanh(x)$
`z`	ROUND to nearest integer	
`?`	push RANDOM value to stack	
	Double-Item Functions	
`%`	DIVIDE/MODULO	$.. y, x \mapsto .. (y/x), (y \bmod x)$
`t`	TRIG functions	$.. \theta, r \mapsto .. r \sin\theta, r \cos\theta$
`p`	POLAR coords	$.. y, x \mapsto .. \text{atan2}(y, x), \sqrt{x^2+y^2}$
	Registers and Memory	
`<`	GET value from register	
`>`	PUT value into register	
`^`	INCREMENT register	
`v`	DECREMENT register	
`{`	LOAD from memory location	
`}`	STORE to memory location	
	Jump, Testing, Branching and Logic	
`j`	JUMP to specified cell (subroutine)	
`|`	BAR line (RETURN on `.|` HALT on `8|`)	
`=`	check register is EQUAL to top of stack	
`g`	check register is GREATER than top of stack	
`:`	if TRUE, branch FORWARD	
`;`	if TRUE, branch BACK	
`&`	logical AND	
`/`	logical OR	
`~`	logical NOT	

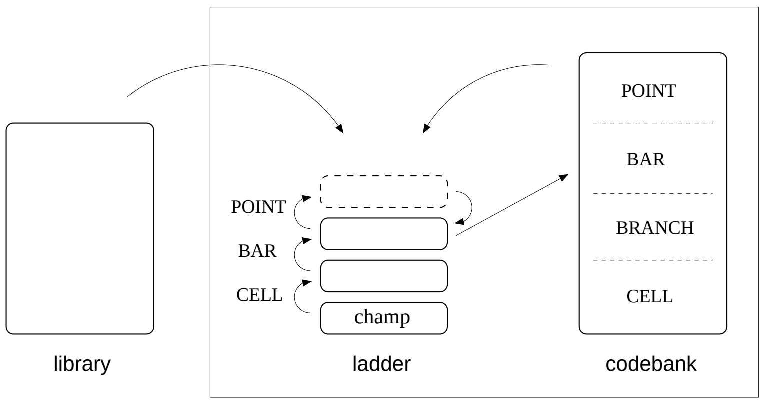

Fig. 2. Hierarchical Evolutionary Re-Combination. If the top agent on the ladder becomes fitter than the one below it, the top agent will move down to replace the lower agent (which is transferred to the codebank). If the top agent exceeds its maximum number of allowable offspring without ever becoming fitter than the one below it, the top agent is removed from the ladder (and transferred to the codebank).

code from the secondary (codebank or library) parent. The level of each mutation/crossover is chosen randomly, with lower levels weighted more heavily than higher ones, and with the constraint that the mutation levels must strictly decrease as we move up the ladder. The agents in the codebank are grouped according to mutation/crossover level, with a limited number of agents in each level. Further details can be found in [3].

During the evolution, comparison between agents is based on five criteria: *length*, *time*, *cost*, *penalty* and *reject*. The *length* is the total number of commands, dots and digits in the program. The *time* is the average number of instructions executed for each training input. We draw a distinction between the *cost* — which is a measure of the difference between actual and desired output — and the *penalty* — which is a count of more serious violations of the "rules" (for example, producing the wrong number of outputs, or failing to produce any output at all). If an agent exceeds a certain maximum number of execution steps (usually due to an infinite loop) it is classified as *reject* and culled immediately. If two agents differ in terms of *penalty*, the one with lower penalty is always considered fitter, regardless of the *cost*. When comparing two penalty-free agents, the fitness is calculated as the *cost* plus tiny multiples of the *length* and *time* — thus favoring shorter and faster agents, and serving as an effective means of *bloat control* [12, 13].

When two agents with the same non-zero *penalty* are compared, the winner is chosen probabilistically using a Boltzmann distribution based on the difference in *length* and *time*. This gives rise to a Metropolis search [15] in the early stages of evolution, until a penalty-free agent is achieved. After that, the fitness comparison strictly favors shorter and faster agents, and relies purely on the hierarchical nature of the search in order to escape from local optima (see Fig-

Table 2. Mutation Levels (Low to High)

TUNE:	Modify one or more PUSH values
POINT:	Choose one or more points at which to insert, remove or replace an instruction, or modify the dot/digits of an instruction
BAR:	Replace the front, back, middle, fringe or whole of a bar in P_0 with the front, back, middle, fringe or whole of a bar in P_1
BRANCH:	Insert a conditional branch, to skip some existing instructions and/or execute newly added instructions
CELL:	Replace front, back, middle or fringe of a cell in P_0 with the front, back, middle or fringe of a cell in P_1
JUMP:	Introduce an instruction to jump to a cell in P_0 and (optionally) replace that cell with a cell from P_1
BLOCK:	Replace a block of cells in P_0 with a block from P_1

ure 2). Depending on its level, each (penalty-free) agent is guaranteed to survive long enough to produce a certain number of offspring, thus promoting diversity in a manner comparable to the age-layered population structure of [10].

Once an agent is found which achieves a *cost* less than some pre-determined threshold, the algorithm moves into a final *trimming* phase in which instructions can be deleted and replaced but not inserted, thereby removing extraneous code and reducing the agent to a minimal size.

4 Training Paradigm

We consider classification tasks where the input consists of a fixed number of (binary, discrete or continuous) features and the target output is either 1 or −1.

The stack and memory of the agent are initially re-set, and the input features are loaded into its registers (one register for each feature). The code of the agent is then executed, and it is required to eventually output one single-item message and halt; otherwise it incurs a penalty. The cost function between the target T and output Z is defined as:

$$\text{cost} = \begin{cases} 0\ , & \text{if } \mathrm{T} = \ \ 1 \text{ and } \mathrm{Z} \geq \ \ 1, \\ 0\ , & \text{if } \mathrm{T} = -1 \text{ and } \mathrm{Z} \leq -1, \\ (\mathrm{Z}-\mathrm{T})^2, & \text{otherwise.} \end{cases}$$

The progress of the evolution is measured in *applications*, *evaluations* and *epochs*. Each *application* refers to the code of a new agent being executed once, to classify one training item. Each *evaluation* refers to a new agent having its cost (fitness) evaluated and compared to that of its (primary) parent. For this, the new agent is applied to successive training items until either (a) all items in the training set have been exhausted, or (b) the cost accumulated by the new agent is already so large that it would remain inferior to the parent even if it were to classify all subsequent items with zero cost. For convenience the evolution is divided into *epochs*, with the number of evaluations in each epoch gradually increasing as the evolution progresses (equal to $2^{\frac{n}{2}}$ for epoch n, up to a maximum of $10,000$ comparisons). If more than one agent remain on the ladder when this limit is reached, the epoch continues for a few additional evaluations until the ladder is reduced to a single agent on the lowest rung. For the experiments described in this paper we will only consider single-cell HERCL programs, so CELL level crossovers are the highest ones available.

In order to put a limit on the computation time, and avoid overfitting, we stop the evolution when either (a) the total number of applications has exceeded 900 million, or (b) the average cost per training item has reached a pre-defined threshold. We then commence the final "trimming" phase, for an additional 10,0000 evaluations. The limit of 900 million epochs has been chosen so that, allowing for the completion of the current epoch, plus the trimming phase, the total number of applications in the entire process will not exceed one billion. For these experiments a threshold value of 0.2 was chosen. (Preliminary tests indicated that a threshold of 0.1 leads to similar results, while 0.3 leads to slightly degraded performance.) During the trimming phase agents with lower cost are preferred, so the average cost per training item may ultimately reach a value lower than the pre-defined threshold.

Once the evolution and trimming are complete, the agent is tested on the (unseen) test data. We will be investigating the number of evaluations required to achieve the threshold cost per item, and the length of the resulting code, as well as the final accuracy.

It is also possible to ensemble a number of evolved agents to produce a collective prediction. The ensembling is done by voting, with the sum of the output values used as a tie-breaker when there are equal numbers of positive and negative votes.

This kind of ensembling method — now a standard technique in machine learning — can perhaps be traced back to Solomonoff [21] who proposed that the optimal predictive agent should be one which maintains a collection of Turing machines compatible with the data so-far observed, and weighs their predictions (inversely) exponentially by the size of the machine. Programs in a language like HERCL (or any kind of linear, tree- or stack-based GP) arguably comprise a better set of "base learners" than Turing machines for this purpose, because (a) the evolutionary algorithm provides an effective search mechanism, giving preference to shorter (and faster) agents, and (b) these programs operate natively on floating point numbers rather than discrete symbols, reflecting the fact that modern computers can perform floating-point operations in a single clock cycle.

5 Tasks and Experimental Method

Six benchmark classification tasks were selected from the UCI repository [2]: IONOSPHERE, PROMOTERS, HEPATITIS, AUSTRALIAN, SONAR and PIMA. These tasks were chosen on the criteria that the number of training items should not exceed 1000, the number of features should not exceed 100, and the number of classes should be two. Each dataset was split randomly into 10 parts (stratified) for 10-fold cross-validation. In each case, we consider the target output to be $+1$ for positive items and -1 for negative items.

The input features for these six tasks are quite diverse. The IONOSPHERE task takes 34 continuous inputs in the range -1 to 1. The E. Coli Promoter Gene Sequences (PROMOTERS) task uses 57 nucleotides, which we represent as ASCII characters `a`, `c`, `g` or `t`. The inputs for the HEPATITIS task are the age and sex of the patient, together with 17 medical indicators (12 binary, 5 continuous). The Australian Credit Card Approval (AUSTRALIAN) dataset has 14 inputs of which 6 are continuous, 4 are binary (0,1) and 5 are categorical (which we treat as integers 1, 2, etc.). The SONAR task (Mines vs. Rocks) uses 60 continuous values between 0 and 1, representing the energy in different frequency bands. The Pima Indians Diabetes (PIMA) Dataset has 6 medical indicators together with the age of the patient and the number of times they have been pregnant.

In order to investigate the effect of sharing genetic material between different tasks, we will compare two different training regimes – referred to as Single and Transgenic. In the Single regime, evolution for the six tasks are run completely independently and do not share any genetic material. In the Transgenic regime, the six evolutions are run concurrently, and a common *library* is maintained, consisting of the current best agent (champ) for each of the six tasks. The code in the library is updated at the conclusion of each epoch. At each step of the algorithm, the secondary parent can be chosen either from the codebank or from the library. In other words, the current best agent from each task is made available as a potential mate for the evolution of the other five tasks. Each regime was run 25 times from different random seeds.

6 Results

In order to compare training times between the two regimes, we count the total number of evaluations required for the threshold cost per item to be achieved. The median number of evaluations (rounded to the nearest thousand) are shown in Column 2 of Table 3, while Column 3 shows the Z-score and p-value obtained from a (two-tailed) Mann-Whitney U-test [14] (the AUSTRALIAN and PIMA datasets are excluded, because the majority of runs failed to attain the threshold cost per item before exceeding the maximum number of allowed applications). For the IONOSPHERE dataset, the number of evaluations for the Transgenic regime is significantly smaller than for the Single regime. For the other datasets, the difference is not statistically significant.

The median Code Length (total number of commands, dots and digits) in the final evolved agent is shown in Table 3, Column 4. For the PROMOTERS, AUSTRALIAN, SONAR and PIMA datasets, the Transgenic regime produces significantly shorter code than the Single regime, as measured by a Mann-Whitney U-test (Column 5).

Table 3. Evaluations and Code Length

	Evaluations		Mann-Whitney		Code Length		Mann-Whitney	
	Single	Trans	z-score	p-value	Single	Trans	z-score	p-value
IONOSPHERE	67,000	55,000	2.01	0.044	62	63	1.27	0.20
PROMOTERS	484,000	439,000	1.28	0.20	95	87	3.25	0.0012
HEPATITIS	689,000	575,000	1.35	0.18	118	114	1.71	0.087
AUSTRALIAN	–	–			249	216	5.32	0.0001
SONAR	1436,000	1612,000	-0.97	0.33	215	206	2.34	0.019
PIMA	–	–			266	244	2.58	0.010

Table 4. Accuracy

	Items	Features	Single	Trans	t-score	p-value	Ensemble
IONOSPHERE	351	34	89.8 ± 0.3	89.8 ± 0.2	0.12	0.91	93.2
PROMOTERS	106	57	65.2 ± 0.6	68.4 ± 1.0	2.54	0.014	83.0
HEPATITIS	155	19	73.6 ± 0.6	73.3 ± 0.5	-0.42	0.67	80.6
AUSTRALIAN	690	14	79.1 ± 0.3	79.0 ± 0.3	-0.30	0.76	85.2
SONAR	208	60	64.4 ± 0.7	66.3 ± 0.7	1.76	0.084	82.2
PIMA	768	8	62.6 ± 0.5	62.9 ± 0.6	0.47	0.51	72.3

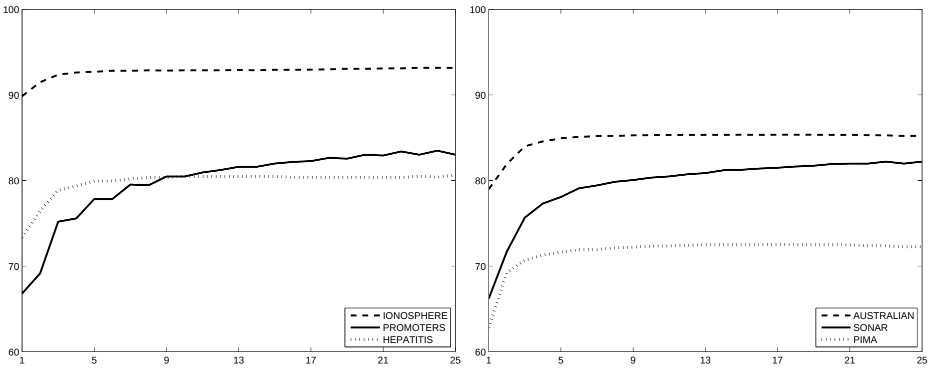

Fig. 3. Accuracy achieved by different sized ensembles of evolved HERCL programs, with classification by voting, using summed output to break ties when votes are equal.

Columns 4 and 5 of Table 4 show the mean accuracy (on the test set) for the Single and Transgenic regimes, averaged over the 25 runs, together with the standard error of the mean. Column 6 shows the result of a (two-tailed) Welch's t-test. We see that, for the PROMOTERS dataset, the Transgenic regime provides a statistically significant improvement in mean accuracy, compared to the Single regime. For the other datasets, the difference is not statistically significant.

The accuracy achieved by ensembling different numbers of agents is graphed in Figure 3, and the accuracy obtained from an ensemble of all 25 evolved agents is listed in the final column of Table 4.

By ensembling a number of moderately accurate agents, we ultimately achieve a level of accuracy which is broadly competitive with what has previously been reported for other evolutionary approaches such as GP with clustering, or GP refined using a gain criterion [7]. For the SONAR dataset, our methods seem to achieve an accuracy comparable to that of a neural network with two or three hidden units [8] (although a more thorough analysis based on multiple splits of the data would be needed in order to make a comprehensive comparison).

The evolved HERCL agents which happen to classify the largest number of items correctly for each dataset are shown in Table 5. We see that the agents evolved for the IONOSPHERE and PROMOTERS task make extensive use of testing and branching instructions, while those evolved for the SONAR and PIMA tasks rely more heavily on stack manipulation and arithmetic.

Table 5. Evolved Agents

IONOSPHERE:

```
[13{<q14g:.06#11g|.v~:<17={:28=v:15<p6=.{:4=.v:4<|<wo]
```

HEPATITIS:

```
[10<13g:<1.#12g~y:ztq|13ge2}:>7}|5.#t%1vh<8<y14g15{:7=|+px
 !xgx:y|y}>14g{p6^{:3{|p13g:8{15g~:2g:e|px>=~:.647059#-|-wo]
```

PROMOTERS:

```
[15<14<49g=:x|33=x:35=:1:|15g}:38=:21<38g:q|cn%y3.906#p3.#thwo]
```

AUSTRALIAN:

```
[e2g=:1.83242496#10<aeh.3#6gy:{g{:*ycc5>p+|txy.>e<12va7<21#9g9:3.#
 5=~!:{|!n8<c11g:e|*4=~:15#-|3<n12vrn13<cp12ge~*:+}4^c2<htxxn4#-
 c1}>g2:|t.}><+8{+5{+c9=!{++c13<11=~!:n3vy<h++12{5>7}{2{11<!g:5:
 |a10<x6g.=aaa/~aqt:!!zcg:a*xaaaaaaa2#e>aaaaaaah<|5<t7vnzt}xg2:+
 11{+caaa1.#->aag~:0>|13<3=~*z:8}|0g:2>4<|9#=!:+11#g}:+a8{2=y:h|wo]
```

SONAR:

```
[10<21<14g~!:aaaaaaaaa|h15g:.1#p<|+<xqcpya>h+g46<x.99#-:*a
 tt**>44<qh|tx>+zcc>+g+46<<1:19<+27<6g+:51<*4<+{}13gq}p:<|c
 r+9#-t50<q.13#>g~:!42<{+.8#-+a|z35{++aaaxr-q<-+hz+a14{q+wo]
```

PIMA:

```
[7.#27.8#5gzq6<10.88#0<p>ct>pz><q:%pp<8#-.>ppch7}e5ge|q>g<:5v**y+|
 1gzh<ppppp!:%t6vxy%|n<p5}at.1565#->g{4:x^xzg:9:|9g+>aa+r.1521#-g:>g
 1:x|}a.58#4gac:%*9<x|x6gx:+*tp>py|%!<ac0g~!:a|1.#>g:pa%nzc6>pc1v}pp
 <p>|<4g!!p!.55393#:ae+1:|7.#.>0g*:n-|.g~:x=x1:zg:++|htx6g~ax:tt|wo]
```

7 Conclusion and Further Work

We have shown that HERCL programs can successfully be evolved to perform six benchmark classification tasks.

Although the tasks are quite disparate, the evolution for each task can make productive use of transgenic crossovers which adapt portions of code from the evolution of other, related tasks. For all except the HEPATITIS task, the availability of transgenic crossovers provides a statistically significant improvement in either the training time, code length or final accuracy. Overall accuracy can be improved by ensembling a number of evolved agents.

The genetic diversity of the codebank and the hierarchical nature of the evolutionary search allow it to escape from local optima even though the number of agents competing at any point in time is very small. The selective pressure for shorter and faster code enables the evolved programs to capture the underlying patterns in the data and avoid overfitting.

In future work, we plan to investigate sequence prediction and further refine the HERCL framework with a view to exploring multi-agent systems and modular evolution.

References

1. Angeline, P.J. and J.B. Pollack, 1992. "The evolutionary induction of subroutines", *Proc. 14th Annual Conference of the Cognitive Science Society*, 236–241.
2. Bache, K. and M. Lichman, 2013. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
3. Blair, A., 2013. Learning the Caesar and Vigenere Cipher by Hierarchical Evolutionary Re-Combination, *Proc. 2013 Congress on Evolutionary Computation*, 605–612.
4. Blair, A., 2014. "Incremental evolution of HERCL programs for robust control", *Proc. 2014 Conf. on Genetic and Evolutionary Computation Companion*, 27–28.
5. Brodie, L., 1987. *Starting Forth*, 2nd ed. (Prentice-Hall, NJ).
6. Bruce, W.S., 1997. "The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions", *Proc. 2nd Annual Conf. Genetic Programming*, 52-57.
7. Eggermont, J., J.N. Kok and W.A. Kosters, 2004. "Genetic programming for data classification: Partitioning the search space", *Proc. 2004 ACM Symposium on Applied Computing*, 1001-1005.
8. Gorman, R.P. and T.J. Sejnowski, 1988. "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets", *Neural Networks*, Vol. 1, pp. 75-89.
9. Harper, R. and A. Blair, 2006. "Dynamically Defined Functions in Grammatical Evolution", *Proc. 2006 Congress on Evolutionary Computation*, 1420–1427.
10. Hornby, G.S., 2006. "ALPS: the age-layered population structure for reducing the problem of premature convergence", *Proc. 2006 Conf. on Genetic and Evolutionary Computation*, 815–822.
11. Koza, J.R., 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press).

12. Langdon, W.B. and W. Banzhaf, 2000. “Genetic Programming Bloat without Semantics”, *Parallel Problem Solving from Nature VI*, 201–210.
13. Luke, S. and L. Panait, 2006. “A Comparison of Bloat Control Methods for Genetic Programming”, *Evolutionary Computation* 14(3), 309–344.
14. Mann, H.B. and D.R. Whitney, 1947. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”, *Annals of Math. Statistics* 18(1):50-60.
15. Metropolis, N., A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, 1953. “Equation of State Calculations by Fast Computing Machines”, *J. Chem. Phys.* 21, 1087–1092.
16. Nordin, P., 1994. “A compiling genetic programming system that directly manipulates the machine code”, *Advances in genetic programming* 1: 311-331.
17. O’Neill, M. and C. Ryan, 2000. “Grammar based function definition in Grammatical Evolution”, *Proc. GECCO 2000*, 485–490.
18. Pan, S.J. and Q. Yang, 2010. “A Survey on Transfer Learning”, *IEEE Trans. Knowledge and Data Engineering* 22(10), 1345–1359.
19. Perkis, T., 1994. “Stack-based genetic programming”, *Proc. IEEE World Congress on Computational Intelligence*, 148-153.
20. Salustowicz, R. and J. Schmidhuber, 1998. “Evolving Structured Programs with Hierarchical Instructions and Skip Nodes”, *Proc. 15th Int’l Conf. Machine Learning (ICML’98)*, 488–496.
21. Solomonoff, R.J., 1964. “A formal theory of inductive inference: Parts 1 and 2”, *Information and Control* 7:1–22 and 224–254.
22. Spector, L. and A. Robinson, 2002. “Genetic Programming and Autoconstructive Evolution with the Push Programming Language”, *Genetic Programming and Evolvable Machines* 3(1), 7–40.
23. Walker, J.A. and J.F. Miller, 2008. “The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming”, *IEEE Trans. Evolutionary Computation* 12(4), 397–417.