Parallel Hierarchical Evolution of String Library Functions

Jacob Soderlund, Darwin Vickers and Alan Blair

School of Computer Science and Engineering
University of New South Wales
Sydney, 2052 Australia
blair@cse.unsw.edu.au

Abstract. We introduce a parallel version of hierarchical evolutionary re-combination (HERC) and use it to evolve programs for ten standard string processing tasks and a postfix calculator emulation task. Each processor maintains a separate evolutionary niche, with its own ladder of competing agents and codebank of potential mates. Further enhancements include evolution of multi-cell programs and incremental learning with reshuffling of data. We find the success rate is improved by transgenic evolution, where solutions to earlier tasks are recombined to solve later tasks. Sharing of genetic material between niches seems to improve performance for the postfix task, but for some of the string processing tasks it can increase the risk of premature convergence.

1 Introduction

Evolutionary Computation typically involves a population of *agents* (individuals) undergoing repeated cycles of selection, crossover and mutation. It has long been recognized that a large-scale crossover would normally result in an initially inferior agent and that subsequent, smaller crossovers or mutations would be needed before the new agent becomes competitive in the general population. Methods have therefore been proposed to protect these young individuals for a period of time in an age-layered population structure or similar scheme [6].

Hierarchical evolutionary re-combination and the associated HERCL programming language were introduced as an alternative approach to this problem [1]. HERCL agents have a stack, registers and memory (Fig. 1), thus combining elements from linear GP [8] and stack-based GP [9, 10]. Programs are divided hierarchically into *cells*, *bars* and *instructions*. Each *cell* is effectively a procedure or subroutine, containing a sequence of executable instructions. Cells are divided into smaller chunks called *bars*, delimited by the pipe symbol (|) – much like the bars in a musical score. Each 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 (see [1] for further details).

Hierarchical Evolution does not use a population in the usual sense, but instead maintains a stack or *ladder* of candidate solutions (agents), and a *codebank*

```
    INPUT: ickey
   OUTPUT:
   MEMORY: Minnie.............................
           ..................................
REGISTERS: .....[6]..[1].  [7]
    STACK: MM

     CODE: 0[is|.<syˆ5>};i|8{ˆs-˜:+7=;wo8|-wo]
                                 ˆ
```

Fig. 1. HERCL simulator, showing an evolved agent executing the `strcmp` task, to compare the strings "Minnie" and "Mickey". 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.

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

Registers and Memory

`<`	GET value from register
`>`	PUT value into register
`ˆ`	INCREMENT register
`v`	DECREMENT register
`{`	LOAD from memory location
`}`	STORE to memory location

Jump, Test, Branch and Logic

`j`	JUMP to specified cell (subroutine)
`|`	BAR line (RETURN on `.|` HALT on `8|`)
`=`	register is EQUAL to top of stack
`g`	register is GREATER than top of stack
`:`	if TRUE, branch FORWARD
`;`	if TRUE, branch BACK
`&`	logical AND
`/`	logical OR
`˜`	logical NOT

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	$\ldots x \to \ldots 1/x$
`q`	SQUARE ROOT	$\ldots x \to \ldots\sqrt{x}$
`e`	EXPONENTIAL	$\ldots x \mapsto \ldots e^x$
`n`	(natural) LOGARITHM	$\ldots x \mapsto \ldots \log_e(x)$
`a`	ARCSINE	$\ldots x \mapsto \ldots \sin^{-1}(x)$
`h`	TANH	$\ldots x \mapsto \ldots \tanh(x)$
`z`	ROUND to nearest integer	
`?`	push RANDOM value to stack	

Double-Item Functions

`%`	DIVIDE/MODULO	$\ldots y, x \mapsto \ldots (y/x), (y \bmod x)$
`t`	TRIG functions	$\ldots \theta, r \mapsto \ldots r\sin\theta, r\cos\theta$
`p`	POLAR coords	$\ldots y, x \mapsto \ldots \mathrm{atan2}(y,x), \sqrt{x^2+y^2}$

of potential mates. At each step of the algorithm, the agent at the top rung of the ladder is selected and either mutated or crossed over with a randomly chosen agent from the codebank, or from an external library.

Crossovers and mutations are classified into different *levels* (TUNE, POINT, BAR, BRANCH, CELL or BLOCK) according to what portion of code from the pri-

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). When the algorithm is parallelized, each niche has its own ladder and codebank, but all of them may share a common library, containing the current champ from each niche.

mary (ladder) parent is either mutated or replaced with code from the secondary (codebank or library) parent. A large crossover at the lowest rung of the ladder is followed up by a series of progressively smaller crossovers and mutations at higher rungs, concentrated in the vicinity of the large crossover.

In previous work, single-cell HERCL programs have successfully been evolved for coding tasks [1], dynamically unstable control problems [2] and classification tasks [3]. However, a number of drawbacks emerged:

(a) the algorithm sometimes experienced long periods of stagnation,
(b) for some of the more complex tasks, the single-cell programs became very long and difficult to evolve,
(c) the number of competing agents in a single ladder is rather low, potentially missing out on the benefits if parallelism inherent in other EC paradigms.

In the present work, we address these issues by introducing:

(a) incremental training, with reshuffling,
(b) multi-cell evolution,
(c) parallelized hirerachical evolution, on a multi-core architecture.

Aided by these enhancements, we test whether programs can be evolved to emulate a postfix calculator, and to perform ten string processing tasks modeled on functions from the standard C library.

2 HERCL Enhancements

(a) Incremental training, with reshuffling: Incremental training has previously been used to evolve HERCL programs for control problems such as the double pole balancing task [2]. We now extend this approach to supervised learning, with some additional modifications. The training items are shuffled into a random order and, initially, the fitness evaluation is based only on the first two items. Once a certain target cost has been achieved on the first k items, additional items are added, until the per-item target cost is no longer achieved. If the system runs for an entire *epoch* (100,000 offspring) without adding any new items, the last item on the list is swapped out and replaced with the next (in order) item – thus giving the system a chance to "move on", rather than getting stuck on a particularly difficult item. If two difficult items occur in succession, the algorithm will swap back and forth between them – up to a maximum of six attempts (three for each item). After the sixth failed attempt, a *reshuffle* event occurs: the training items are reshuffled into a new order, and training begins again with the first two items (according to the new ordering). Note that this is not the same as a random re-start, because the codebank and the champ are retained, and only the subset of the training data changes.

This reshuffling, combined with hierarchical search, gives rise to a process of *creative destruction*, where the components of agents evolved under previous orderings are re-combined, to optimize the fitness under the new ordering. Over time, code fragments that are advantageous for multiple sets of training items are more likely to survive to be incorporated into a global solution.

(b) Multi-cell evolution: In order to evolve multi-cell programs we introduce new levels of crossover beyond the CELL level, labeled as BLOCK-1, BLOCK-2, BLOCK-4, etc. For a BLOCK-k mutation, a block of k cells from the secondary parent is transplanted into the primary parent (a JUMP instruction to the modified cell(s) may optionally be inserted elsewhere in the code). Subsequent (lower-level) mutations are concentrated in the vicinity of the transplanted block.

(c) Parallel Hierarchical Evolution: We parallelize the algorithm to run on a multi-core machine. Each core maintains a separate *niche* with its own ladder and codebank, but all of them may share a common library, comprised of the current best agent (champ) from each of the niches (Fig. 2). As soon as a global solution is found in one niche, a terminating signal is sent to the niches running on the other cores. Since competition between agents occurs only within a niche, data can be rechuffled independently in each niche, thus creating additional diversity in the system without compromising the "fairness" of the competition.

For comparison, we include some experiments where each niche is running completely independently, with no sharing of code between them. This allows us to examine to what extent improved performance is due to sharing of code, and to what extent it is due simply to a greater amount of searching.

Table 2. String processing tasks and programming constructs.

TASK	DESCRIPTION	L	R	V	S	M	I
`strcpy`	input: a string output: the same string	✓					
`strcat`	input: a string followed by another string output: first string concatenated with second string	✓					
`strlen`	input: a string output: the length of that string	✓	✓				
`idxstr`	input: an index followed by a (non-empty) string output: the character at that index in the string	✓	✓				✓
`chrstr`	input: a character followed by a string output: index of first occurrence of that character (or an empty message, if it does not occur)	✓	✓	✓			
`stridx`	input: a (non-empty) string followed by a list of indices output: the list of characters at the specified indices	✓	✓	✓		✓	✓
`catstr`	input: a string followed by another string output: second string concatenated with first string	✓	✓	✓		✓	
`strchr`	input: a string followed by a character output: index of first occurrence of that character (or an empty message, if it does not occur)	✓	✓	✓		?	
`strrchr`	input: a string followed by a character output: index of last occurrence of that character (or an empty message, if it does not occur)	✓	✓	✓		?	
`strcmp`	input: a string followed by another string output: difference between characters at the first place where the two strings differ (or zero, if they are identical)	✓	✓	✓	✓	✓	

KEY: L = LOOP R = REGISTER V = compare VALUE
S = SUBTRACT M = MEMORY I = compare INDEX

3 String Processing Experiments

In our first set of experiments, we attempt to evolve solutions for a set of ten string processing tasks, adapted from functions in the standard C String Library `string.h` (listed in Table 2). The main motivation for choosing these tasks is to see whether certain general-purpose programming constructs could be evolved, and form a kind of "standard library" for HERCL, to facilitate the learning of more complex tasks such as those proposed in [5]. The right column of Table 2 gives a general indication of the kind of programming constructs that are required for each task. Using this information, along with preliminary experiments, we have tried to arrange the tasks roughly in order from easiest to hardest.

For each task, 1000 training and 1000 test cases are randomly generated. String contents are chosen uniformly from the set of all printable ASCII characters. For tasks involving a distinguished character or index, the number of characters before and after it are geometrically distributed with a mean of 3

characters. This is equivalent to choosing the string lengths from a negative binomial distibution NB(2, $\frac{3}{4}$) and then choosing the distinguished character uniformly within the string. Each run is conducted on a 16-core machine with 15 separate niches, plus one core dedicated to communication between the other cores (in a star network arrangement). For the cost function we use the generalized Levenshtein edit distance [1], which is suitable for comparing outputs that may vary in length. The target cost is zero.

Table 3 shows the number of minutes to completion for the various evolutionary runs. In the first five runs (labeled Sa to Se) each task was evolved on its own, up to a maximum of 8 hours. In the remaining runs, labeled as *transgenic*, the system attempts to evolve solutions for each task in turn, using a library consisting of the solutions evolved for all of the previous (successful) tasks on the list. Up to three attempts are made for each task, with each attempt running for a maximum of 8 hours. As soon as one attempt is successful, the system adds the solution to its library and moves on to the next task. If all three attempts fail, the system moves to the next task without adding anything to the library.

For the runs labeled as *sharing*, code was shared between the 15 niches, with space reserved in the common library for the current best agent from each niche (updated asynchronously, at the end of each epoch). For the last three runs (labeled as non-sharing) the cores were run completely independently, with no genetic material transmitted between cores (and the library consisting only of the solutions to previous tasks).

Table 4 shows the evolved code from three selected runs (TSb, TSc and Ta).

We see that almost all the runs succeeded in evolving solutions for `strcpy`, `strcat`, `strlen`, `idxstr`, `chrstr` and `stridx`. The first two tasks – `strcpy` and `strcat` – are easily solved within a few minutes and the resulting code is practically identical across all runs, as follows:

`strcpy` `0[i|sw;o]` `strcat` `0[i|sw;i|sw;o]`

The solutions for `idxstr` all involve incrementing or decrementing a register, until the required index is reached. Those for `strlen` and `chrstr` involve counting items – either by incrementing an index, explicitly adding 1, or computing tanh of each character (which saturates to 1). The solutions for `stridx` all work by storing the string into successive memory locations and then accessing the value at each index in the list.

The last four tasks were solved considerably more often by the transgenic runs, and we can see several instances where code from previous tasks has been re-combined to solve later tasks. The solution for `catstr` in run Ta uses code from `stridx` to store the first string into memory, then transfers the second string to the output buffer, before retrieving the first string from memory. Indeed, `stridx` seems to be a kind of *bottleneck* task in the sense that failure on `stridx` in Run Tb has led to failure on all the subsequent tasks. Run TSc has found solutions for `strchr` and `strrchr` which invert the roles of the character and index, storing each index (plus 0.2) into the memory location specified by the character. If the same character occurs multiple times, the first or last occurrence

Table 3. Evolution time for string tasks.

Run	Sa	Sb	Sc	Sd	Se	TSa	TSb	TSc	TS2	Ta	Tb	T2
transgenic	← No →					← Yes →						
time limit	480 mins					$3 \times 480 = 1440$ mins						
sharing	← Yes →									← No →		
cells	← 1 →								2	1		2
`strcpy`	2	3	1	4	2	2	4	1	4	1	3	3
`strcat`	15	6	19	15	15	15	15	16	17	16	8	11
`strlen`	2	1	190	2	12	109*	5	2	3	1	2	1
`idxstr`	8	2	275*	3	2	4	5	2	4	3	81	5
`chrstr`	122	44*	46	×	466	6	16	1	109	22	7	21
`stridx`	319	124	117	214	251	420	127	163	795	749	×	191
`catstr`	×	×	×	×	×	×	×	×	20	342	×	130
`strchr`	×	×	×	×	×	×	56	32	×	×	×	×
`strrchr`	97	×	37	×	×	272	5*	3	×	918	×	14
`strcmp`	×	×	×	×	×	×	×	174	×	×	×	×

KEY: × = failed to achieve zero cost on the training set
∗ = achieved zero cost on training set but not on test set

(as appropriate) will overwrite the others and its index (rounded to the nearest integer) will be the one that prevails in the relevant memory location. We can recognize a fragment of code from `strrchr` in the solution for `strcmp`, which stores the first string into memory and then scans the second string, comparing it with the one in memory, one character at a time, until a non-zero difference is found.

Runs Sa, Sc, TSa, Ta and T2, which failed to evolve a solution for `strchr`, end up solving `strrchr` by storing each index and character alternately on the stack, then searching backwards. When the relevant character is found, the next number on the stack will be the correct index.

Run TSb provides a good example of how a suboptimal "gene" from one task can find its way into later tasks. For a string of length n, the solution for `strlen` is achieved not by incrementing a register but instead by computing the nearest integer to $\log \sqrt{(7.5)^n} \simeq 1.0075\,n$. This gives the correct answer for all strings up to length 67. The pattern then finds its way (with a slight twist) into the solution for `strchr` – which computes a formula based on the sum of the logs of the characters in the string, with an expected value of $1.04 + 0.985(k-1)$. In other words, the agent is exploiting the independent distribution of the characters, to find a solution which is good enough to give the correct answer for all 1000 training and test cases, but would not work for all possible inputs. The solution for `strrchr` similarly computes a function whose expected value is $0.79 + 1.02(k-1)$. It satisfies all the training data but makes an off-by-one error on 1 of the 1000 test items.

In three cases (Sb `chrstr`, TSa `strlen` and `idxstr`) a very long suboptimal solution has been found which explicitly scans items one at a time, then prints

Table 4. Evolved code for selected runs of the string tasks.

	Run TSb (sharing)		
strlen	0[1#i|y*7.5#s;}qnzwo]		
idxstr	0[isi|=^:sx;|swo]		
chrstr	0[i{s>i|=h:+s;|!wo]		
stridx	0[i|s}^;is:o.|>{ws;o]		
catstr	–		
strchr	[i|ss;>is|=:x>;o.|!=h;eex=*;{e|6#**x.=x1;7><xgq;qnzwo]		
strrchr	[i|ss;>is|=:x>;o8|!he|7#**><xgq;qnzwo]		
strcmp	–		
	Run TSc (sharing)	Run Ta (no sharing)	
strlen	0[i|<s^;wo]	0[is:{wo1:|}1.#+s;wo]	
idxstr	0[is>2gi|<.=:s5^;|swo]	0[isi|%^g:s;|swo]	
chrstr	0[i<s>is|=h:+s;|!wo]	0[i<s>is|=h:+s;|>wo]	
stridx	0[i|s}^;is:o8|>{ws;o]	0[is|}^s;is|~:>{|ws1;o]	
catstr	–	0[i|}^s;is|ws;1^{g:;|o]	
strchr	0[.2#i>|<s^;}|>};is>{9=z~:o8|wo]	–	
strrchr	0[.2#i>|.<s^5>};is>{3=z:w|o]	0[i|<^s;is>}|=:};o8|%wo]	
strcmp	0[is|.<sy^5>};i|8{^s-~:+7=;wo8|-wo]	-	

out a hard-coded answer. In general, the non-sharing runs tend to produce code which is shorter and more robust than those the sharing runs. The reason may be that a suboptimal solution can develop in one niche and then spread like a virus to the other niches. In that case, a system with all niches "quarantined" from each other may produce a global solution faster. For example, Run Ta (without sharing) has found a global solution for `catstr` using memory, whereas the three sharing runs all got stuck in suboptimal solutions which manage to store and retrieve up to 5 or 6 characters of the string using specific registers and stack manipualations, but fail for longer strings. The two-cell runs (TS2 and T2) were able to solve `catstr`, but their solutions avoid using memory and instead use recursion, with one recursive call for each item.

It is hard to say whether sharing makes a significant difference in the evolution times – although the sharing runs may finish slightly faster (or more often) than the non-sharing runs on tasks like `stridx` and `strrchr` where suboptimal solutions are unlikely to occur. It may be that some of these tasks – when provided with evolved solutions to the preceding tasks – effectively require only one new "trick" in order to succeed. Once this trick is discovered, the evolution rapidly proceeds to completion. Under this hypothesis, 15 niches running concurrently would be expected to find the vital "trick" in equal time, whether they are sharing code or evolving independently.

4 Postfix Calculator

In order to further explore the issue of sharing vs. non-sharing evolution, we tried evolving on a different kind of task, whose solution is more likely to require

Table 5. Probabilistic Grammar for generating postfix data.

S	→	Val	(0.1)	Op	→	+ (0.25)
S	→	Tree Tree Op	(0.9)	Op	→	− (0.25)
Tree	→	Val	(0.6)	Op	→	∗ (0.25)
Tree	→	Tree Tree Op	(0.4)	Op	→	/ (0.25)
Val	→	space, followed by numeric value from a Cauchy distribution, rounded to two decimal places				

Table 6. Evolution times and evolved code for the postfix task.

MINS	CODE
314	`0[]` `1[is|>39#g!:ss;|<ct>cnt!gn1:>g:r|*c{|hz-*+s3;wo]`
395	`0[43#>g~:!*s:wo8]` `1[isss:wo.|!ss|41#>g1;0j43#>g;47#>g:yr*<xy|.7#-t!z*+s1;wo]`
721	`0[isss:wo8|!s46#>s|g:!r*s;2;|vvg:}-<|vg:!+s:wo8|vg4;46#>g:3;|vvg:}-<|vg:!+s1:wo.|!*s:wo.|4;]`

a combination of separately evolved modalities. The task we chose is a `postfix` calculator. For this task, the input is a sequence of numbers and operators, forming an arithmetic expression in *Postfix* or *Reverse Polish* notation. The output is the numerical value to which the given expression evaluates.

The allowed operators are $+, -, *$ and $/$. Again, 1000 training and test cases are produced by a generative process, according to the probabilistic grammar shown in Table 5. Expressions involving divide by zero are excluded. For this task we again use the Generalized Levenshtein Edit Distance. The target cost is set at 10^{-6} in order to avoid failure due to roundoff errors.

For the `postfix` task, two-cell programs rather than single-cell programs were evolved in the first instance. Ten runs were performed with sharing of genetic material, and ten runs without sharing. All of the non-sharing runs failed to find a solution within the 8 hour limit. Only two of the ten sharing runs managed to find a solution.

Ten additional runs were performed to see whether single-cell evolution, with sharing of code between niches, could produce solutions for the `postfix` task. For these runs, the time limit was extended to 24 hours per run. Only one of these ten runs produced a solution. (In order to save computing time, single-cell experiments without sharing of code were not performed.) The exact evolution times and evolved solutions are shown in Table 6.

Note that the first of these solutions ultimately uses only one of the two cells. The code is quite short, and uses a serendipitous combination of trig functions and logs to distinguish the characters $+, -, *$ and $/$. The other two solutions perform an explicit comparison against (the ASCII values of) these characters.

Having two cells available during the evolution seems to free up the evolutionary process, and provide greater flexibility for targeted crossovers and mutations. But, later in the process the code from one cell may get randomly transplanted into the other cell, allowing it to perform the whole task on its own.

5 Conclusion and Further Work

We have parallelized the hierarchical evolutionary re-combination algorithm, and shown that HERCL programs can successfully be evolved to emulate a postfix calculator and perform ten different string processing tasks. In the process, fundamental programming constructs emerge such as incrementing and decrementing of registers, storing items into successive memory locations, looping until certain conditions are met and distinguishing between various arithmetic symbols to select between different computation paths.

This process can be compared to the primordial stages of biological evolution, where random combinations in geographically separated regions over long periods of time eventually lead to beneficial fragments of genetic code, which can then be recombined into successively more complex organisms.

In future work we plan to package up the solutions for these string tasks into a "standard library" and test to what extent the inclusion of this library (and others like it) would speed up the learning of new tasks.

References

1. Blair, A., Learning the Caesar and Vigenere Cipher by Hierarchical Evolutionary Re-Combination, Congress on Evolutionary Computation, 605–612 (2013)
2. Blair, A., Incremental evolution of HERCL programs for robust control, Conference on Genetic and Evolutionary Computation Companion, 27–28 (2014)
3. Blair, A., Transgenic Evolution for Classification Tasks with HERCL, Artificial Life and Computational Intelligence, Springer, 185-195 (2015)
4. Bruce, W.S., The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions, Conf. Genetic Programming, 52-57 (1997)
5. Helmuth, T. & Spector, L., General Program Synthesis Benchmark Suite, Genetic and Evolutionary Computation Conference, 1039-1046 (2015)
6. Hornby, G.S., ALPS: the age-layered population structure for reducing the problem of premature convergence, Genetic and Evolutionary Computation Conference, 815–822 (2006)
7. Lehman, J. and K.O. Stanley, 2011. "Abandoning Objectives: Evolution through the Search for Novelty Alone", *Evolutionary Computation* journal **(19):2**, 198–223.
8. Nordin, P., A compiling genetic programming system that directly manipulates the machine code, Advances in genetic programming 1, 311-331 (1994)
9. Perkis, T., 1994. Stack-based genetic programming, IEEE World Congress on Computational Intelligence, 148-153 (1994)
10. Spector, L. and A. Robinson, Genetic Programming and Autoconstructive Evolution with the Push Programming Language, Genetic Programming and Evolvable Machines 3(1), 7–40 (2002)