Learning the Caesar and Vigenere Cipher by Hierarchical Evolutionary Re-Combination

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

***Abstract*—We describe a new programming language called HERCL, designed for evolutionary computation with the specific aim of allowing new programs to be created by combining patches of code from different parts of other programs, at multiple scales. Large-scale patches are followed up by smaller-scale patches or mutations, recursively, to produce a global random search strategy known as hierarchical evolutionary re-combination. We demonstrate the proposed system on the task of learning to encode with the Caesar or Vigenere Cipher, and show how the evolution of one task may fruitfully be cross-pollinated with evolved solutions from other related tasks.**

I. Introduction

There have been a number of efforts over the years to introduce function calls and other modular structures to evolutionary computation, with the aim of *scaling up* to larger and more complex task domains.

Induced subroutines and Automatically Defined Functions have been incorporated into Genetic Programming [1], [6], Cartesian Genetic Programming [18] and Grammatical Evolution [12], [5]. Meanwhile, new evolutionary paradigms have been developed such as H-PIPE [14], Linear GP [2], and stack-based GP [4] as well as the PUSH programming language [16], borrowing features from stack-based languages like FORTH [3].

In the present work, we describe a novel approach to the problem of modular evolutionary computation, using a new language called HERCL which has been specifically designed for the purpose. HERCL combines elements of Linear GP with stack-based operations from FORTH. It is intended to be an austere and general-purpose language with a very small instruction set. Although it has been designed primarily for the purpose of evolutionary computation, we also allow for the possibility of programs being hand-coded by humans (using rudimentary assembler and debugging tools). Algorithms such as QuickSort, HeapSort and multi-layer neural network backpropagation have all been successfully (and succinctly) implemented in HERCL.

The code of a HERCL program 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. 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 is 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.

II. 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 into cells, bars and instructions. 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 I, and described in detail in the following subsections.

```
i: astimegoesby
o: eta
m: piano.............................
r: .....[4][5]...   [1]
s: [2][2][2][1][2][1]yn

 0[is|sx^g6}:;|5{^.g+;84#%26#%97#+ws;o]
                  ^
```

Fig 1. HERCL simulator, showing an evolved agent executing the VIGENERE task. Note the input and output buffer, memory, (global and local) 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.

A. Input and Output

The input and output of an agent are thought of as a stream of *messages*, with each message consisting of a sequence of *items*. These items, which are stored as floating point numbers, may comprise a set of features, a time series, or a sequence of ASCII characters, etc, depending on the task. As well as a stack, the agent has an input buffer and an output buffer (see Figure 1). The INPUT command (`i`) fetches the next message to the input buffer. The SCAN command (`s`) scans the next item from the input buffer and pushes it to the stack. The WRITE command (`w`) pops the top item from the stack and transfers it to the output buffer. The OUTPUT command (`o`) flushes the output buffer, sending its contents as a message to the output stream.

TABLE I. 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	$\ldots x \to \ldots 1/x$
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	$\ldots y, x \mapsto \ldots \mathrm{atan2}(y, x), \sqrt{x^2+y^2}$
	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	

B. Stack Manipulation and Arithmetic

The PUSH command (#) pushes a new item to the stack. The value of the new item is specified in decimal notation. If no value is specified, we assume a default value of zero. All items on the stack are floating point numbers. The stack is implemented as a circular array of a predetermined size, initialized with all zeros. If the agent fills up the stack by pushing more and more items, the values near the top will be preserved while those near the bottom are erased. The POP command (!) pops the top item from the stack. It has no effect if the stack is already empty. The COPY command (c) copies (or duplicates) the top item on the stack. The SWAP command (x) swaps the top two items on the stack while the ROTATE command (y) rotates the top three items. The NEGATE command (-) replaces the top item on the stack with its negative. The ADD and MULTIPLY commands (+ and *) pop the top two items from the stack, compute their sum or product, and push the result back to the stack.

C. Mathematical Functions

There are several commands which pop the top item x from the stack, apply a mathematical function to it, and push the result back to the stack:

r computes the RECIPROCAL of x.
q computes the SQUARE ROOT of x (or 0, if $x < 0$).
e computes the EXPONENTIAL function of x.
n computes the natural LOGARITHM of x (or $-\infty$, if $x \leq 0$).
a computes ARCSINE of x (or $\frac{\pi}{2}$ sign(x), if $|x| > 1$).
h computes the HYPERBOLIC TANGENT function.
z rounds to the nearest integer.

The RANDOM command (?) pushes a random number to the stack (uniformly distributed between 0 and 1).

D. Double-Item Functions

There are three commands which pop two items and then push two new items onto the stack. The DIVIDE command (%) pops two items x and y from the stack and replaces them with the two numbers y/x (whole number division) and (y modulo x). The TRIG command (t) pops two items r and θ from the stack and replaces them with $r\sin(\theta)$ and $r\cos(\theta)$. The POLAR command (p) performs the inverse operation, popping x and y from the stack and replacing them with $\mathrm{atan2(y, x)}$ and $\sqrt{x^2 + y^2}$.

Some computations can be done entirely on the stack without using any registers or memory. For example, this program computes the month and date of Easter Sunday for a given year, using a formula from Butcher's 1873 Almanac:

```
[isc19#cy%x!cy*yy100#%ycc4#%y-+yyc8#+25#%!-+1#
 +3#%!-+15#+xy+30#%x!yx4#%y+2#*x-+xcycy-+32#+
 7#%x!cyc++y+y11#*+451#%!7#*-+114#+31#%1#+xwwo]
```

E. Registers and Memory

The agent has a specified number of global registers, and there is also a local register associated with each (invocation of a) cell. Register commands may be preceded by a dot (indicating the local register of the current cell) or a sequence

of digits (specifying a global register 0, 1, 2, etc.). The GET command (<) gets the value from the specified register and pushes it to the stack. The PUT command (>) pops the top item from the stack and puts it into the specified register. The INCREMENT command (^) increases the value of the specified register by 1, the DECREMENT command (v) decreases it by 1. Memory locations are always addressed indirectly, through one of the registers. If necessary, the value from the register is rounded to the nearest integer and computed modulo M (the total memory size) in order to ensure an index between 0 and $M-1$. The LOAD command ({) loads the value from the memory location indexed by the specified register and pushes it to the stack. The STORE command (}) pops the top item from the stack and stores it into the memory location indexed by the register. For all register and memory commands, if no register is specified, by default the command will apply to the most recently specified register (or the local register of the current cell, if no other register has yet been specified).

F. Jump, Testing, Branching and Logic

The agent can test certain conditions, and then branch forward or back by a specified number of bars if those conditions are satisfied. The EQUAL command (=) tests whether the value of a register is equal to the top item on the stack. The GREATER THAN command (g) tests whether the value of a register is greater than the top item on the stack. As a "side effect", the INPUT command (i) also tests whether there is actually any input to fetch, and the SCAN command (s) tests whether there are still item(s) available to be scanned from the input buffer. The FORWARD command (:) and BACK command (;) can optionally be preceded by digits specifying the number of bars to branch forward or back. If no digits are specified, we assume a default value of zero, meaning a branch to either the beginning or end of the current bar. The (Boolean) results of any i, s, = or g commands are stored temporarily (until the next branch instruction) so they can be combined, using the three logical commands: AND (&), OR (/) and NOT (~).

For the experiments described in this paper we will only consider single-cell programs. But in general, a HERCL program may have multiple *cells*, which behave like functions or subroutines. Program execution begins at the highest numbered cell; control can be transferred from one cell to another using a JUMP instruction (consisting of the letter j preceded by the number of the cell being called). Upon reaching the end of its code, the current cell will terminate and *return* control to the calling cell. We also need the capability to RETURN from the middle of a cell or HALT execution of the entire program. We treat these not as independent commands but instead as modifications of the BAR line, using the combination .| for RETURN and 8| for HALT. This keeps the instruction set small, and helps to preclude "unreachable" instructions – by restricting RETURN and HALT to occur only at the end of a bar. For example, here is a HERCL implementation of the HeapSort algorithm, using 5 registers, 4 cells and 15 bars:

```
0[4<1>3<|1<c+0<-+2>^=1:g~:!.|2{^{
  v.>g~!:2^|2{1{.>g:1}.<2}<1>2;|!!]
1[0<^<v-3<+2#%!+4>|g:=:.|0jv1;]
2[3>c0>1j|0{3{0}3}v=:c4>0j;|!]
3[1#c.>c1>vis|^}s;<^<y2j|.=:{^w;|o]
```

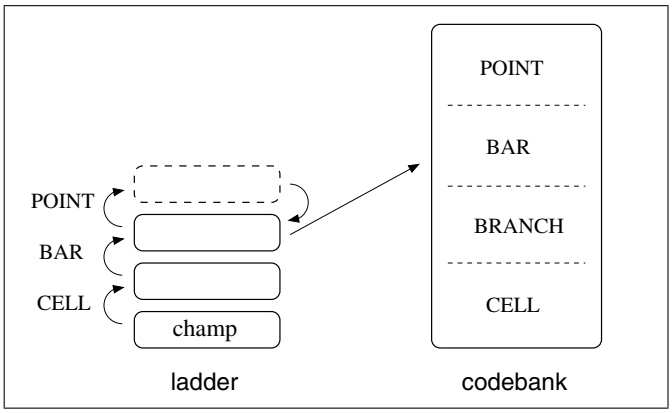


Fig 2. Hierarchical Evolutionary Re-Combination. If the top agent on the ladder becomes fitter than the one below it, the top agent will move down to replace the lower agent (which is transferred to the codebank). If the top agent exceeds its maximum number of allowable offspring without ever becoming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

III. HIERARCHICAL EVOLUTIONARY RE-COMBINATION

For this research we introduce a new global random search strategy known as *hierarchical evolutionary re-combination*. We do not use a population as such, but instead maintain a *stack* of candidate solutions (agents), and a *codebank* of potential mates. For convenience, we will refer to the stack as a *ladder*, on which the agents occupy different *rungs* (see Figure 2). At any point in time, the agent with the best currently known fitness (the *champ*) is always residing at the bottom (lowest rung) of the ladder. The remaining rungs contain increasingly distant relatives of the champ – each agent having been generated, by a series of mutations, from the one below it. We limit the size of the ladder by insisting that the mutation *level* must strictly decrease as we move from one rung to the next, as explained below.

During the evolution, comparison between agents will be 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).

The mating process normally involves two parents, of which we distinguish one as the *primary* parent (P_0) and the other as the *secondary* parent (P_1). The offspring is thought of as being mainly a copy of the primary parent, but with some (smaller) amount of code being either randomly generated, or copied from the secondary parent. For simplicity we will use the generic term "mutation" to refer to both mutation and

TABLE II. 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

crossover. We distinguish seven different *levels* of mutation, as shown in Table II.

The evolution begins with a single (trivial) agent at the lowest rung of the ladder. At each step of the algorithm, we select the agent at the top rung (R) of the ladder and *mutate* it (by mating with a randomly chosen agent from the codebank) to produce an offspring. The *level* of the mutation is chosen from a random distribution, with lower levels weighted more heavily than higher ones, and with the constraint that the mutation level at rung R must be strictly lower than the one previously used at rung R-1. (Note: in the very beginning, the codebank is empty, so the agent must mate "with itself".)

The intuition behind the algorithm is as follows: in the case of a low-level (TUNE or POINT) mutation, because the change is so small, it seems logical to cull the offspring immediately if it is less fit than the parent. In the case of a higher-level mutation, however, it is unrealistic to expect that the offspring would immediately be superior to its parent (see Figure 3). Therefore, provided the offspring is penalty-free, instead of culling it straight away, we install it at the next rung (R+1) of the ladder and let it continue to evolve until either (a) it becomes superior to its (original) parent or (b) it exceeds some maximum number of allowable offspring (depending on the mutation level). In the former case, the new (superior) agent at rung R+1 will replace the old one at rung R, and the old one will be removed from the ladder and inserted into the *codebank* (where it remains available, for a time, as a potential mate for future agents). In the latter case, the new (inferior) agent at rung R+1 will be removed (and transferred to the codebank).

Higher level mutations (BAR, BRANCH, CELL, JUMP or BLOCK) all require code to be transcribed from P_1 to (a copy of) P_0. In the process, the digits preceding any branch or jump instruction will be modified so as to preserve the destination bar or cell (if it lies within the transcribed code). We also keep track of a *focus bar* and *focus cell(s)*, so that subsequent mutations can be focused on (a) the bar or cell containing the transplanted code, (b) the bar containing the inserted jump instruction, (c) the transplanted cell(s), or (d) other cells, weighted by proximity to the transplanted code.

The agents in the codebank are grouped according to mutation *level*, with a limited number of agents per level.

```
[ic}~c^/>g/s:&::ish<=~&|csww;o] (rung 1)
CELL    [i::ish<=~&|csww;o] (add rung 2)
  POINT [i::ish<=~&|csw/;o]
  POINT [i::sh<=~&|csw/;o]
  POINT [i::sh<=~&|sw/;o]
  BAR   [i:}^;v|sw/;o]
  BAR   [i:}^i|sw/;o]
  POINT [i:}i|sw/;o]
  POINT [i:i|sw/;o]
  BAR   [i|sw/;o]
  BAR         [i|scws;o]      (add rung 3)
    POINT     [i|scwn;7go]
        [i|scwn;7go]     (replace rung 2)
  POINT [i|scwn;go]
  BAR         [isw/|scwn;go] (add rung 3)
    POINT     [isw/|scwn;&o]
    POINT     [isw/|scw/;&o]
        [isw/|scw/;&o]  (replace rung 2)
[isw/|scw/;&o]          (replace rung 1)
```

Fig 3. Chain of hierarchical mutations observed during the evolution of the DUPLICATE task. The current champ is shown on the top line; a CELL mutation is applied – replacing the front part of the cell with that of a randomly chosen cell from the codebank. The new agent is initially inferior to its parent but, after a sequence of lower-level mutations (including combinations of BAR and POINT mutations), it improves its fitness and eventually achieves superiority, thus moving up the ladder to become the new champ (shown on the last line).

Once the limit is reached, the insertion of any new agent will cause the oldest agent at that level to be discarded. This means that agents arising from higher-level mutations (which tend to be more diverse) will remain longer in the codebank than those from lower-level mutations (which tend to be very similar to each other). In addition to this (local) codebank, we can also *cross-pollinate* the evolution by making available a (global) *library* comprised of previously evolved (or hand-coded) solutions to other related tasks. In this paper we evolve agents on a series of tasks of increasing complexity, and investigate the extent to which the evolution of later tasks might fruitfully be cross-pollinated with evolved solutions for earlier tasks.

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 [11] 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. As soon as a solution with zero *cost* has been achieved, 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. Typically, as the evolution progresses, from time to time a new agent will emerge which is significantly fitter than the previous agents, but initially somewhat "bloated". Subsequent mutations, together with the *trimming* phase at the end, serve as an effective means of *bloat control* [7], [10].

IV. Generalized Levenshtein Edit Distance

For supervised learning tasks, we need a *cost* function to measure the difference between the target (T) and actual output (O). In cases where all targets have the same length (number of items) we use the L_1-Norm for the cost function:

$$L_1(O,T) = \sum_i |O_i - T_i|$$

If the length of the output is different from that of the target, the agent incurs a *penalty* equal to the difference in the two lengths. If the agent does not produce any output, the penalty will be one plus the length of the target.

For cases where the targets have different lengths, we use a new cost function which is a generalization of the Levenshtein Edit Distance [9]. The traditional LED is defined as the smallest number of *edits* required to transform one sequence into the other, where each *edit* consists of inserting or deleting an item, or exchanging one item for another. Our Generalized LED is defined in the same way, except that we impose a variable cost on the exchange of one item (σ) for another (τ), depending on the distance between the two items, as follows:

$$\text{edit}(\sigma,\tau) = 1 - \frac{1}{1+\log_4(1+|\sigma-\tau|)}$$

This Generalized LED can be computed efficiently using a straightforward variation of the algorithm for the traditional LED. Here are some examples:

cost (" bat "," cat ")	$= \frac{1}{3}$	cost (" cat "," rat ")	$= \frac{2}{3}$
cost (" fat "," cat ")	$= \frac{1}{2}$	cost (" cat "," at ")	$= 1$
cost (" fat "," mat ")	$= \frac{3}{5}$	cost (" cat "," crate ")	$= 2$

V. Tasks and Experimental Method

The algorithm and cost function we have described are of a general nature, and could potentially be applied to many tasks. For this pilot study, we focus on the task of learning to encode a given text, using either the Caesar or Vigenere Cipher.

CAESAR task:
Input: a single character (*code letter*) followed by a sequence of characters (*text*).
Output: Caesar-encoding of the text, using the code letter.

VIGENERE task:
Input: a sequence of characters (*key*), followed by zero, followed by another sequence of characters (*text*)
Output: Vigenere-encoding of the text, using the key.

There has been some previous work using Genetic Algorithms with a specialized fitness function to search the key space for a Vigenere Cipher [17]. We are interested in a quite different task, where the agent must learn to perform the encoding of a text, based purely on input and output pairs. For example, consider this input for the CAESAR task:

```
vcryhavocandletslipthedogsofwar
```

Using the first character ('v') as the code letter, the (text) characters that follow ("cry..." etc.) need to be rotated backwards by 5 places in the alphabet to become "xmt.." etc. so the agent should respond with this output:

```
xmtcvqjxviygzongdkoczyjbnjarvm
```

Turning to the Vigenere task, consider this input:

```
piano playastimegoesby
```

(where the space indicates the number zero). In order to encode the text, the codeword "piano" must be aligned with the text multiple times as follows:

```
playastimegoesby
pianopianopianop
```

So the correct output is:

```
etalohbizsvwefpn
```

While the CAESAR task has a reasonable chance of being learned directly, the VIGENERE task is more complex, since it requires the key to be stored in successive memory locations and then retrieved, one character at a time, with multiple passes, as the text is processed. We therefore devise a series of simpler tasks which are intended as stepping stones toward the VIGENERE task:

ITEMNUMBER task:
Input: a sequence of numbers, followed by an integer k.
Output: the k^{th} number in the sequence (starting from 0).

For example,

Input: -2.20, -2.00, 0.77, -1.22, -0.21, -0.38, 0.24, 4
Output: -0.21

DUPLICATE task:
Input: a sequence of characters.
Output: a sequence composed of two copies of the original sequence.

For example,

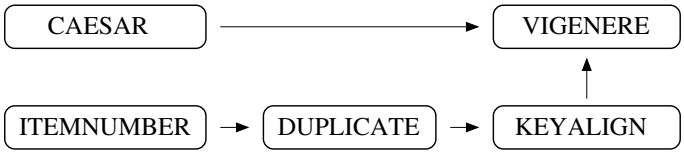
Input: `piano`
Output: `pianopiano`

KEYALIGN task:
Input: a sequence of characters (*key*) followed by zero, followed by another sequence of characters (*text*)
Output: a sequence of characters the same length as the text, consisting of several copies of the key laid end-to-end (the last of which may be incomplete)

For example,

Input: `piano playastimegoesby`
Output: `pianopianopianop`

Our experiments can be summarized in the following diagram, in which an arrow from task A to task B indicates that the evolution for task B is to be *pollinated* with an evolved solution for task A:



For each task, 100 training examples and 100 test examples were randomly generated. For the ITEMNUMBER task, the input items were real numbers drawn from a standard Normal distribution; for the other tasks, the input items were lower case ASCII letters (uniformly distributed). For the CAESAR, ITEMNUMBER and DUPLICATE tasks, the length of each sequence was chosen from a geometric distribution with $r = \frac{1}{4}$. For the KEYALIGN and VIGENERE tasks, the length of the key and text were chosen from geometric distributions with $r = \frac{1}{3}$ and $r = \frac{1}{6}$, respectively, with the constraint that the text must be at least as long as the key and, in the case of the VIGENERE task, the key must be at least two characters in length. All agents had one cell, 10 registers, 64 memory locations and a maximum stack size of 64. The codebank was limited to 64 agents per mutation level. The evolution was divided into *epochs*, where each epoch corresponds to 10000 fitness evaluations, plus whatever additional steps are needed to bring the ladder back down to a single rung. Each experiment was repeated 10 times with different random seeds, and evolved to a maximum of 10000 epochs.

VI. RESULTS

A. CAESAR *task*

The cost function on the training and test set for 10 runs of the CAESAR task are plotted in Figure 4. Of the 10 runs, 8 of them reached zero cost for the training set within the 10000 epochs. The following table shows the number of epochs to convergence, and the resulting code:

	Epochs	Code
C1.	825	`[issx40.#+|c}+26.#%97.#+ws{;o]`
C2.	1193	`[is>s<+|27.4#%+88.#+zws<+;o]`
C3.	2843	`[is>s|<+84#%26#%97#+ws;o]`
C4.	4244	`[iscs|px7.#*2#t*#yc*hh+xpx2.#{}` `py+-p.34#*pxqthp%!110.#+wcs;o]`
C5.	4541	`[is>s|<+cpqg:>3.09#>*!+<p|cczt{9>g1;` `6.#>g~:*pq+.14#-+|108.#.{!++zws2;o]`
C6.	5421	`[is>s|.<2.#-++5><#.77#yyt16.#` `3#e.764#>y+tv<tp%!110#+ws;o]`
C7.	7256	`[219.#>is}s{|+g=/~:` `149.#+|123.#%ws{1;o]`
C8.	8318	`[i230#1#p.82#t106#+c+c+sp}sp|y*{+` `.3#ty+{p.24#tpa%>108#{a++zws;o]`

The first 7 of these agents were *successful* in the sense of also achieving zero error on the test set. The last one (C8) handles the training inputs correctly, but makes slight mistakes on 4 of the 100 test inputs (substituting 'l' for 'k', 'a' for 'z', 'z' for 'y' and 'k' for 'j'). Among the successful agents, C3 is perhaps the most elegant. It fetches the input (`i`), scans the code letter (`s`), puts it into a register (`>`) and scans the first character from the text. It then repeatedly retrieves the code letter from the register (`<`), adds it to the text character, computes the remainder modulo 84 and then 26, and finally adds 97 (the ASCII code for 'a'). The result is written (`w`) to the output buffer and the next character is scanned (`s`) before branching back (`;`) to the beginning of the bar (`|`). After the entire text has been processed, the scan command fails, the branch command (`;`) is skipped, and the encoded message

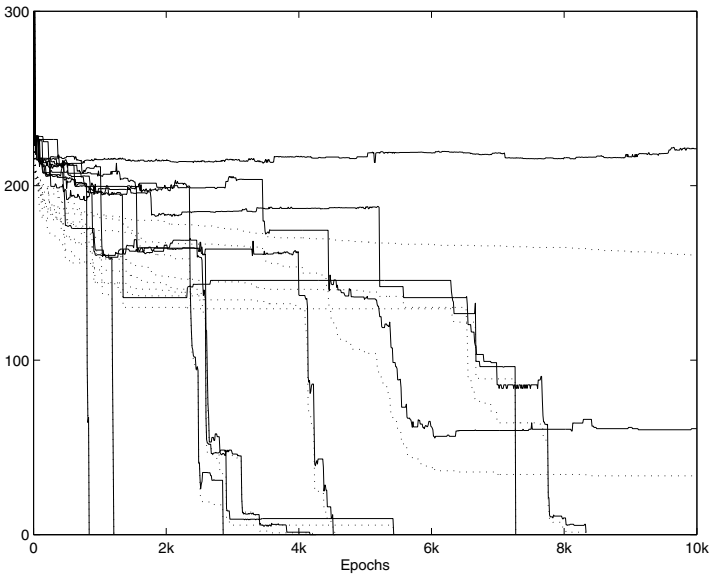


Fig 4. Cost function on the training set (dotted) and test set (solid) for the CAESAR task (10 separate runs). Note that in a typical (successful) run, the cost on the test set fluctuates and remains somewhat above that of the training set, until the moment where both of them become zero as the solution is achieved.

is sent as output (`o`). Solutions C1 and C2 follow a similar pattern, with C1 using a memory location rather than a register to store the code letter (`}`) and load it back again (`{`). Solution C7 uses the combination `g=/~` to check that the value in the local register (219) is not (`~`) greater than (`g`) or (`/`) equal to (`=`) the value on top of the stack, in which case it branches forward (`:`) to the next bar (`|`) thus skipping the part where 149 is pushed to the stack and added to the sum. In either case, the result is computed modulo 123, and written to the output buffer. Solutions C4, C5 and C6 achieve the same result by rather more elaborate computations – including square roots (`q`), hyperbolic tangents (`h`) and conversions between polar and rectangular coordinates (`p`,`t`) followed by division without remainder (`%!`) or rounding to the nearest integer (`z`).

B. ITEMNUMBER *task*

For the ITEMNUMBER task, 7 of the 10 runs were successful (reaching zero cost on both the training and test set) while the other 3 failed to converge. The successful runs were as follows:

	Epochs	Code
I1.	124	`[is}|s^};v{>{wo]`
I2.	169	`[is}s|^sx};>{wo]`
I3.	356	`[iss|x}^s;>{wo]`
I4.	948	`[is}|^sc};>{wo]`
I5.	2035	`[i|sc}^;>{wo]`
I6.	3196	`[i|sc}^;>{wo]`
I7.	3997	`[is}|^s}{;>{wo]`

These evolved solutions are all quite similar to each other. They first input (`i`) the message, then scan the items (`s`) one at a time – putting them (`}`) into the memory location indexed by a register, and incrementing that register (`^`). They repeatedly branch back (`;`) until the SCAN command fails. The final item

(index) is then stored into the same register (>) and the value from that memory location is loaded ({), written (w) and output (o).

C. DUPLICATE *task*

For the DUPLICATE task, the evolution was *pollinated* with an evolved solution for the ITEMNUMBER task (meaning that this code was always available as a potential "mate"). Of the 10 runs, 9 were successful, as follows:

	Epochs	Code
D1.	122	`[ig|scw.}&c^;*1g*~{^;o]`
D2.	327	`[isccw|xv|scwc}1;!!5=3v~{;o]`
D3.	447	`[isc|cw}^sc;1>*=~9{;o]`
D4.	696	`[is|cw^8}s;>4{g~;o]`
D5.	1772	`[i|sc}+w0{<4^;.^{g~;o]`
D6.	1938	`[iscw|sc}^w;.vg9{;o]`
D7.	2350	`[iscw|^sccw1}y;y>*g8{;o]`
D8.	2591	`[isc|vcw1}sc;+h>=~8{;o]`
D9.	4766	`[iscw|s}c{^w;+g*~2{;o]`

All the evolved solutions follow a similar strategy of scanning and writing the items one at a time, meanwhile storing a copy (c) of each item into successive memory locations (}) using a register which is either incremented (^) or decremented (v) at each cycle. The items are then retrieved from memory in the same order, until the number zero is encountered. Taking D5 as an example – after storing the sequence in memory it proceeds to increment (^) the local register (.) and load the item from the corresponding memory location ({). It then checks that the value in the register itself (i.e. the index) is not (~) greater than (g) the value on top of the stack (which is either a lower case ASCII character or, once the end of the sequence has been reached, zero). If the condition is satisfied, the program branches back (;) otherwise it proceeds to the output (o) instruction. (The program would fail if the input sequence were to exceed 97 characters, but the Vigenere Cipher does not normally employ such a long key). For comparison, another set of 10 runs were performed, on the same (DUPLICATE) training data, but without pollinating the evolution with the solution from the ITEMNUMBER task. Of these 10, only 3 were successful, as follows:

Epochs	Code
1951	`[iscw|svc8}w;<1g{;o]`
4485	`[isc|ws}{^;7{g:x;|o]`
6491	`[iscw|8.#-sc}pzwp^;1{g:;|o]`

D. KEYALIGN *task*

For the KEYALIGN task, the evolution was pollinated with one of the evolved solutions from the DUPLICATE task (D5). Of the 10 runs, 5 of them were successful, as follows:

	Epochs	Code
K1.	583	`[i|sc}9^;<{%>+{w!0=:6v;|o]`
K2.	1047	`[sc}^;c>i;|>4{g;^2}{9.#>wg~;o]`
K3.	1271	`[sc9}^;iv|*.{g~^1;v2{.}^}2{^wg~;o]`
K4.	3501	`[i|sx^9g}:;|.{^9<yg;ws;o]`
K5.	5422	`[i|^sc1};2g{~:^|9{x0}^x!2=1;9g}^g5{wh~1;o]`

The second of these solutions (K2) nicely illustrates the potential benefits of the final *trimming* phase of the algorithm. Prior to trimming, the agent is as follows:

```
[sc}^;c}c}^c}{c^}^c}}^}^^.2#3>c=~:;2.#
 c!}c=cw*>.#&:w!w|1}i1;c{x>i|!3{^7.#.>
 .1#-2g!^:.79#c^}^c}^}^}^^3}i;c{x>c+7{
 ^.#1>{><!1.#>g|g3;4}{34.#><!cg}w4g~1;o]
```

This (un-trimmed) agent correctly handles all of the training inputs and 97 out of the 100 test inputs, but fails on the other three test inputs (by getting into an infinite loop). The trimming process leads to a more parsimonious agent (K2 above) which in this case is also more *robust* – since it is able to handle all of the training and test inputs correctly.

E. VIGENERE *task*

For the VIGENERE task, the evolution was pollinated with evolved solutions for both the KEYALIGN task (K4) and the CAESAR task (C3) (i.e. both of these programs were available as potential mates, throughout the run). In this case, all 10 runs were successful in fewer than 1000 generations, producing these solutions:

```
[is|sx^1g}:;|6{c^1}+84#%26#^%97#+ws;o]
[i|sx^2g}1:;|.#{2>g:1.#.>|p.{+84#%26#%97#+^ws1;o]
[i.4#s|sx^4g}:+;|6{+97#%{e%3=~:6g>;|z26#%97#+6^ws1;o]
[i|sx^6g}:;|.{+97#%26#%97#+w{^6<!}s^;o]
[is|s^g}:;|csy0}|+97#%26#%97#+w^{g~:>{|s1;o]
[is|sx^9g}:;|4.#-+6{g+~:{>+|^27#%0{y+89#+ws1;o]
[is|sx^3g}:;|7{g+~:#>;|97#%26#^%97#+ws1;o]
[is|sx^g6}:;|5{^.g+;84#%26#%97#+ws;o]
[isc|sx^g2}:;|+97#%.{g~:>x|x26#%97#+w{^s1;o]
[isv|sx^g}:;|7{+272#-+26#%{.}^7^97#+ws;o]
```

We can see that the algorithm has successfully "patched" together solutions from the KEYALIGN and CAESAR task, in order to solve the VIGENERE task.

VII. CONCLUSION AND FURTHER WORK

In this paper we have introduced a new programming language called HERCL and a novel global random search strategy called hierarchical evolutionary re-combination. We have shown that the proposed system can learn to encode text using the Caesar and Vigenere Cipher, and examined how the evolution of one task may fruitfully be cross-pollinated with evolved solutions from other related tasks.

This research opens up many possible avenues for future investigation. We could continue to build up an "atlas" of programming tasks, identifying which of them can be learned "from scratch", which can be learned when pollinated from which other tasks, and so on. Alternatively, a large group of tasks might be learned one-by-one, pollinating the evolution for each new task with the solutions to all previously learned tasks, in the spirit of OOPS [15]. Another approach might be to let the system invent its own tasks, in an open-ended exploration akin to "stepping-stone collecting" [8]. Finally, since the HERCL language has been designed in a very general way, with agents sending and receiving messages of arbitrary

length, it could provide a very interesting platform for research into message passing and multi-agent learning.

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] Brameier, M., 2004. *On Linear Genetic Programming*, Ph.D. Thesis, University of Dortmund.

[3] Brodie, L., 1987. *Starting Forth*, 2nd ed. (Prentice-Hall, NJ).

[4] Bruce, W.S., 1997. "The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions", *Proc. 2nd Annual Conf. Genetic Programming*, 52–57.

[5] Harper, R. and A. Blair, 2006. "Dynamically Defined Functions in Grammatical Evolution", *Proc. 2006 Congress on Evolutionary Computation*, 1420–1427.

[6] Koza, J.R., 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press).

[7] Langdon, W.B. and W. Banzhaf, 2000. "Genetic Programming Bloat without Semantics", *Parallel Problem Solving from Nature VI*, 201–210.

[8] Lehman, J. and K.O. Stanley, 2011. "Abandoning Objectives: Evolution through the Search for Novelty Alone", *Evolutionary Computation* journal **(19):2**, 198–223.

[9] Levenshtein, V.I., 1966. "Binary codes capable of correcting deletions, insertions, and reversals", *Soviet Physics Doklady* **10**, 707-710.

[10] Luke, S. and L. Panait, 2006. "A Comparison of Bloat Control Methods for Genetic Programming", *Evolutionary Computation* **14(3)**, 309–344.

[11] 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.

[12] O'Neill, M. and C. Ryan, 2000. "Grammar based function definition in Grammatical Evolution", *Proc. GECCO 2000*, 485–490.

[13] Pan, S.J. and Q. Yang, 2010. "A Survey on Transfer Learning", *IEEE Trans. Knowledge and Data Engineering* **22(10)**, 1345–1359.

[14] 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.

[15] Schmidhuber, J., 2004. "Optimal Ordered Problem Solver", *M*achine Learning **54**, 211–254.

[16] 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.

[17] Toemeh, R. and S. Arumugam, 2008. "Applying Genetic Algorithms for Searching Key-Space of Polyalphabetic Substitution Ciphers", *Int'l Arab Journal of Information Technology* **5(1)**, 87–91.

[18] 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.