A Tale of Fitness Hacking in Evolutionary Program Synthesis

Nick Moran
Analog Devices
nemtiax@gmail.com

September 30, 2021

This report describes a curious and (hopefully) entertaining result of a simple experiment in evolutionary program synthesis, in which the evolved programs displayed a surprising adaptation to specific parameters of the simulation. The specific details of the experiment itself are not particularly interesting, and many others have conducted similar experiments. What is interesting is the solution discovered by evolution and its interpretation.

The goal of the experiment was to evolve simple programs in a lightly modified version of the P" esoteric programming language. A program in this modified language is a string of symbols which control the movement of a tape pointer on a tape of cells, similar to that of a Turing machine. Each cell on this tape holds a byte, which is interpreted as an integer. The cell currently containing the program pointer can be incremented or decremented, with values wrapping around in the case of overflow or underflow. The symbols in this programming language are:

- `<` : Move the tape pointer one cell to the left. If already at the left-most cell, do nothing.
- `>` : Move the tape pointer one cell to the right. If already at the right-most cell, do nothing.
- `+` : Increment the byte in the current cell. If the value is already 255, wrap around to 0.
- `-` : Decrement the byte in the current cell. If the value is already 0, wrap around to 255.
- `[` : If the value of the current cell is 0, jump forward to the symbol after the matching `]`.
- `]` : If the value of the current cell is not 0, jump backwards to the symbol after the matching `[`.

Once the end of the program string is reached, the program terminates. Further, programs are automatically terminated after their execution step count reaches 500, to avoid infinite loops. If a `[` or `]` lacks a matching pair, it is treated as a no-op, to ensure that all strings are valid programs.

In this experiment, the first two cells were each populated with randomized values, and the goal of the evolved programs was to produce the sum of these two values in the first cell. A population of programs was evolved using fitness-based selection, mutation, and crossover. Fitness was calculated by comparing the value in the first cell at the end of the program to the target, and each individual was evaluated on a randomized set of inputs in the range [0,127] (to avoid overflow in the answer).

It's simple to hand-design a short program which performs this task:

```
>[-<+>]
```

This program moves to the second cell, decrements it, moves back to the first cell, increments it, and repeats until the second cell is 0. This leaves the sum of the two initial values in the first cell, completing the task.

By contrast, the experiment produced the following program:

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<->
```

Of course, it's difficult to judge such esoteric programs by eye, but this one stands out as particularly odd. Among other surprises, the first thing it does is move to the second cell and increment it many times! Despite this, the program acheives maximum fitness, and solves the task for every possible pair of inputs. The remainder of this report is a step-by-step analysis of how this odd program works.

We'll display the state of the program using an underline to indicate the position of the program pointer, and a pair of values to represent the state of the tape[1], again with an underline to represent the position of the tape pointer. Finally, we'll keep track of the number of steps of execution. Thus, at the start of execution, the environment looks like this:

$$[\underline{x}, y], step = 0$$

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<->
```

The first thing the program does is to move left, and then increment the second cell 15 times. This takes 20 steps, due to extra movement. We thus arrive at this state:

[1] Conveniently, the program only uses the first two cells of the tape

$$[x, \underline{y+15}], step = 20$$

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<-><
```

We now check whether the current cell is 0 to decide whether to jump for the `[` instruction. Because y is in the range [0,127], $y + 15$ cannot be 0. We therefore do not jump, and proceed to increment the second cell three more times, thus consuming four more steps.

$$[x, \underline{y+18}], step = 24$$

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<-><
```

Now the program enters a small loop, `[+]`. We will spend a single step on the `[`, again deciding not to jump forward, and begin looping. The effect of this loop will be to increment the second cell until its value wraps around to zero. Each pass through the loop will increment by one, and will take two steps (one for the `+` and one for the `]`). Recall that a cell wraps around to zero when incremented past 255. Thus, this loop will be executed $(256 - (y + 18))$ times, giving us the following worrisome state:

$$[x, \underline{0}], step = 25 + 2 * (255 - (y + 18))$$

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<-><
```

At this point, we've seemingly destroyed our data. y is no longer present on the tape, but our program is supposed to add it to x! But, the fitness evaluation function says this program is correct, so we continue on. We now move back to the left, and increment twice, spending three more steps.

$$[\underline{x+2}, 0], step = 28 + 2 * (256 - (y + 18))$$

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<-><
```

Again we ender a small loop. We spend one step entering the loop, evaluating the first `[`, and then begin incrementing x until it overflows to zero. It's at this point that we encounter the cleverness of the evolved solution. Recall that programs are allowed to run unti reaching 500 steps, and we've already burned quite a few steps zeroing out the second cell. Thus, we have $501 - (2 * (256 - (y+18)) + 29)$ steps remaining[2], after entering the loop. We get to evaluate the loop half that many times before our execution times out, and each evaluation increments x by one. Thus, we will time out in the following state:

[2]Note that execution is terminated *after* the 500th step is executed, thus 501

$$[\underline{x + 2 + (501 - (2 * (256 - (y + 18)) + 29))/2}, 0], step = 501$$

```
>+><+++++++><+++++++[+++[+]<++[+]+<-[<-+<++]]-<->++>++<->
```

A bit of arithmetic simplifies this as follows:

$$x + 2 + (501 - (2 * (256 - (y + 18)) + 29))/2$$

$$x + 2 + (501 - (476 - 2 * y) + 29))/2$$

$$x + 2 + (2y - 4)/2$$

$$x + y$$

And thus, the program has, against all odds, succeeded at the task. But how? The key, as the reader may have recognized by this point, relies on two key facts about the execution environment. First, the size of the bytes stored in each cell provides the value of 256, and the length of the execution provides the value of 501. If either of these is changed, the program fails. But evolution is a master of adaptation, and using the tools at hand, even those not intended to be used.

With a simple change of randomizing the number of steps of execution in the range [500,510], the simulation was able to discover a simple program equivalent to our hand designed one. But that result is much less entertaining than the above.