Codifier
An Artificial Intelligence C Programmer

Saulo Queiroz da Fonseca

Universidade Aberta de Portugal

fonseca@astrotown.de

Abstract

A compiled computer program is no more than the interpretation of a sequence of numbers by an electronic processor. What if we generated a sequence of random numbers and let them be interpreted? What if we used a genetic algorithm to modify the sequence so that, once interpreted, the program could solve a task? The following represents my efforts to chart one of the few remaining frontiers of artificial intelligence: the creation of a program capable of creating another program. The Codifier algorithm receives two vectors of numbers, in the form of a source and a target, after which it presents a program in C language that transforms the source vector into the target.

Keywords: Genetic Algorithm, Automatic Programing, Artificial Intelligence

1. Introduction

For a long time, the search for a program capable of creating another program was the Grail quest of Automatic Programing (Mahoney, 2002). Successful progress toward this goal, however, was impeded by limited computer power and an absence of sufficiently complex code relationships.

My aim is to successfully automate the creation of small pieces of code that constitute a bigger project. An A.I. agent could perform this task, thereby liberating developers to pursue higher-level abstractions.

In the following discussion, I present my contribution to this field, in the form of a program that reads source and target vectors, to create a piece of code capable of transforming this source into its target equivalent. To do this, an agent must interpret a sequence of random numbers as a program. It must subsequently apply a genetic algorithm, to prompt this sequence to evolve into a code capable of accomplishing a given task. This process will produce a program, in ANSI C language, that can be sent to and read by any standard compiler.

This application reflects the next intuitive in the logical development of software engineering. Slowly, dynamic applications are overtaking the inflexible applications that characterize the current era. We should thus have all possible tools at our disposal. This approach may yield more adaptable agents resulting in the future creation of applications with even greater flexibility. Such amplified versatility may ultimately bear agents of unprecedented capabilities that include "thinking."

2. Interpreting Random Numbers

A computer program is no more than the interpretation of a sequence of numbers by an electronic processor. This sequence results from a compilation process that consists of the conversion of the instructions of a program into numbers that correspond to the processors' instructions.

In our task, the first step consists of an inversion of this process. We should be capable of making sense of a given sequence of numbers, enabling us to "decompile" it to read the instructions in a higher-level language capable of generating some sequence of commands that ultimately evolves to accomplish the given task.

Imagine that our high-level language consists of five distinctive instructions. Let us define these instructions as follows: a value of 0 corresponds to the first instruction; a value of 1 corresponds to the second; and so forth. In this way, the first 5 cases are defined, and the other values ignored:

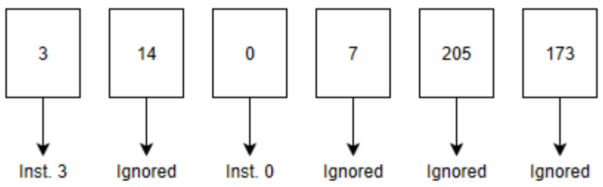


Figure 1

Figure 1 depicts the interpretation of a small sequence of random numbers or bytes. This is a moderately silly method that compels us to waste all numbers above our scope of instructions. A preferable solution would employ the rest of a division, also known as a module, thereby rendering each number useful. Let us use the symbol "%" to represent the calculation of the module—for example, 11%3=2.

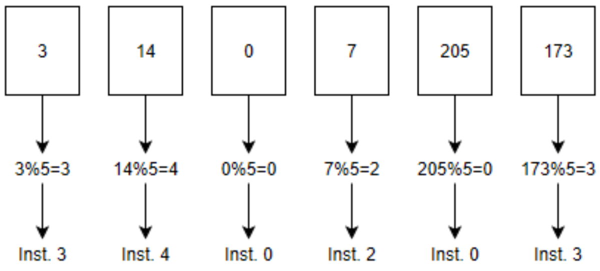


Figure 2

We could also expand our instructions to include additional parameters. For example, our instruction 0 could use the next number as a parameter, which could then be interpreted as A, B, or C. In such a case, where there exists an instruction 0, we would move on to the next number and use the module of 3 (i.e., number of parameters) to define the specific parameter at that point. In that case, zero would be A, 1 would be B, and 2 would be C.

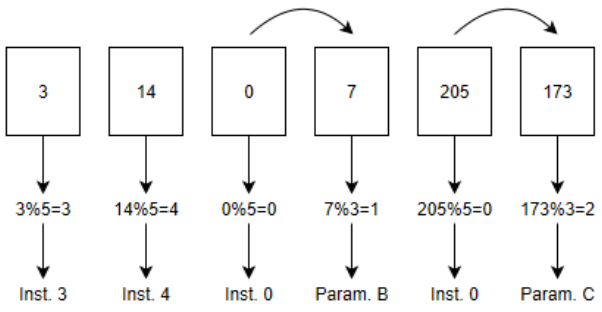


Figure 3

Defining a scope of instructions for our language facilitates a clearer understanding of this process.

3. Scope of Instructions

Let us define a small set of instructions to elucidate the concept:

- **Instruction 0** – Set a variable to a value.
- **Instruction 1** – Increment the value of a variable by 1.
- **Instruction 2** – To impose some flow control, an *if* statement that contains 1-3 instructions inside its block.

In the first 2 instructions, we must define the available variables:

- **Variable 0** – A temporary variable called *tmp*.
- **Variable 1** – A number defined by the next byte.
- **Variable 2** – The size of the source vector.

- **Variable 3** – An element of the source vector indexed by the next byte.

So, we have defined 3 instructions and 4 variables. Given the scope, let us define the source as src={2,1,3} and a random sequence of numbers as follows:

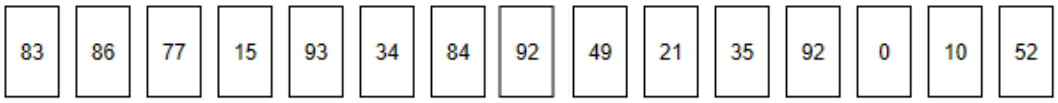


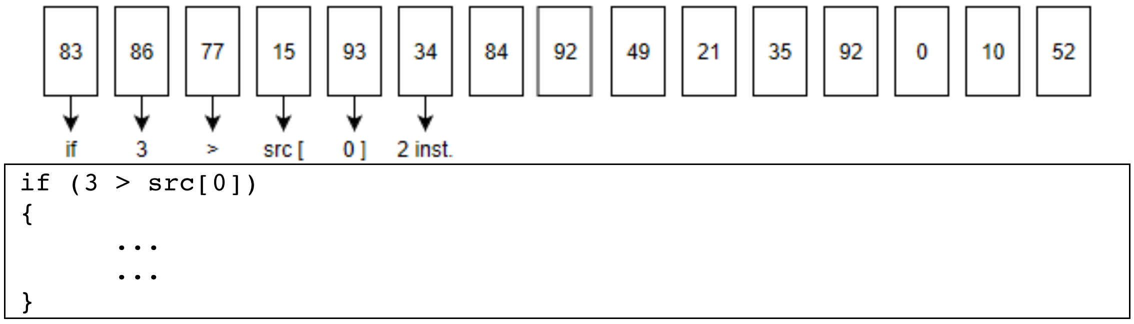
Figure 4

Let us check the first byte to determine which instruction it represents: 83%3=2. In our scope, this represents the *if* statement. Such an instruction requires additional parameters, defined below:

- The first variable used in the bool comparison.
- The kind of comparison (let us use only *less than* and *greater than*).
- The second variable used in the bool comparison.
- The quantity of instructions inside the block.

The next byte defines the first variable: 86%4=2. This instruction corresponds to the size of the source vector, which contains 3 entries. The following byte defines 1 of 2 comparison types. Zero is *less than* (<) and 1 is *greater than* (>). 77%2=1. The next byte defines the second variable: 15%4=3. This is an element of the source vector indexed by the next byte. The vector contains 3 entries: 93%3=0. Therefore, the second variable is *src[0]*.

The last parameter applicable to the *if* block is the quantity of instructions inside it; we want it to have 1-3 instructions. If we take the module of 3, we get a number that ranges from 0 to 2. To obtain at least 1 instruction, we must we add 1 to the calculated module: 34%3+1=2. This leaves us with an *if* block that contains 2 instructions. So, the first 6 bytes create the instruction rendered in Figure 5.



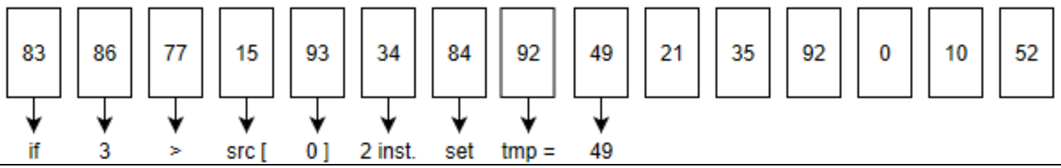
```
if (3 > src[0])
{
       ...
       ...
}
```

Figure 5

Let us continue on, to observe the operation of the next instruction. 84%3=0. This assigns a variable to a value. This instruction also requires additional parameters, defined below:

- The variable that is being set.
- The value to which the variable will be assigned.

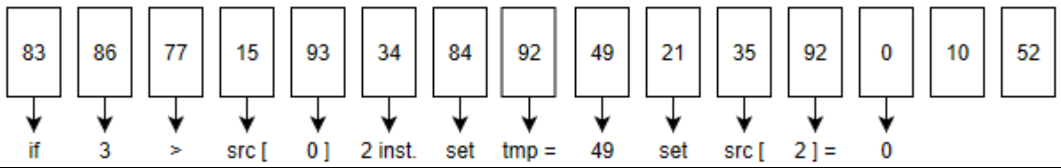
The next byte defines the variable as: 92%4=0. This is the temporary variable, and it will be assigned to the value of the next byte, which is 49.



```
if (3 > src[0])
{
      tmp = 49;
      ...
}
```

Figure 6

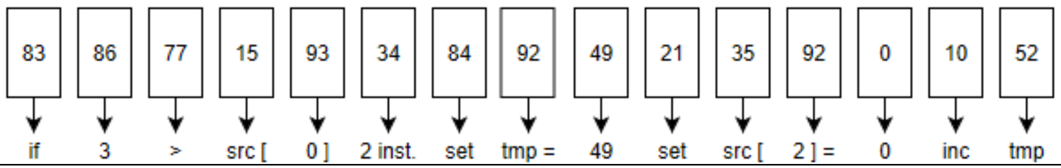
The next instruction is 21%3=0, which is also a *set* instruction. 35%4=3: the variable is an element of the source vector indexed by the next byte, which is 92%3=2, making it *src[2]*. This will be assigned to the value of the next byte, which is 0.



```
if (3 > src[0])
{
      tmp = 49;
      src[2] = 0;
}
```

Figure 7

We still have some bytes left to use. At the moment, the *if* block is complete, containing 2 instructions, so the next instruction must remain outside the *if* block: 10%3=1. This is the instruction to increment the value of a variable by 1, and it requires an additional parameter to define which variable to increment: 52%4=0. This is the temporary variable, which leaves us with *tmp++*.



```
if (3 > src[0])
{
      tmp = 49;
      src[2] = 0;
}
tmp++;
```

Figure 8

If the remaining available bytes are insufficient to complete an instruction, they should be ignored.

As we can observe, it would be possible to convert the 15 bytes presented here into a small program. If the source is {2,1,3}, it is transformed by these instructions into {2,1,0}.

4. Proof of Concept

I created a program called *Codifier* to generate a proof of concept. Written in C++11 (ISO/IEC 14882, 2011), the program is composed of some classes, one for each kind of available instruction. The main class is also called *Codifier* and has some static variables that should be set as initial parameters:

- **size** - The size of the sequence of numbers, which determines the length of a vector called *instructions* that contains those numbers.
- **source** - A vector of vectors of integers that contains the numbers that the program changes. Ex: source[][] ={{2,1,3},{7,0,4},{1,9,2}}. Each run of the algorithm selects 1 of those for insertion into the vector *src*. Ex: src[]={2,1,3}.
- **target** - A vector of vectors of integers that contains the numbers that act as a reference and instruct the algorithm to run. If our program sorts the numbers, we could identify the following: target[][]={{1,2,3},{0,4,7},{1,2,9}}.

This class also contains some private variables:

- **instructions** - A vector of integers that stores the sequence of numbers. It is initially filled by a sequence of random numbers that ranges from 0 to 99.
- **src** - Contains a temporary copy of one of the source vectors. It is altered by the generated code.
- **var** - A vector of integers that contains temporary variables. Its size is defined by the first byte in the instructions vector, and is defined to range between 1 and 10 temporary variables that can be used by the generated code.
- **index** - An integer that informs the next byte of the instructions vector for interpretation.
- **i** - An iteration variable used in *for* loops that is changed by the generated code.

The instructions selected for generation by the program are a subset of the ANSI C programming language (ISO/IEC 9899, 2011). We should have enough instructions to cover a basic Turing Machine (Turing, 1937).

- **nop** - No operation. This instruction is useful, to ensure that we have a variable number of commands for a given quantity of random numbers. It is represented as an isolated semicolon *";"* in C.
- **set** - Defines the value of a variable.
  Ex: var[3]=i.
- **opr** - The same as *set*, but uses 2 variables to generate an operation.
  Ex: var[3]=src[1]/var[0].
- **inc** - Increments or decrements a variable.
  Ex: var[3]++.
- **for** - Creates a loop, using the predefined variable *i* for iteration. At this point, multiple *for* loops inside each other are not permissible.
- **if** - Makes a flow decision, based on a comparison between 2 variables.
- **while** - Runs a block of code for as long as a given comparison between 2 variables holds true.

Currently, the program does not generate new variables at runtime. The existing variables should be divided into 2 categories: left and right. A left variable can stay on the left side of an attribution and be changed by the code. Ex: var[1]=3. A right variable is used only on the right side of an attribution, such as the size of the source vector or a fixed number. The term *variable* makes no sense in such cases, but it is important to keep those values available for the code at points where a left variable might also reasonably be.

Here are the left variables, generated by the respective class:

- **src[i]** - An element of the source vector indexed by the iteration variable.
- **src[n]** - An element of the source vector indexed by the next byte.
- **var[n]** - An element of the temporary variable vector indexed by the next byte.
- **src[var[n]]** - An element of the source vector indexed by an element of the temporary variable vector, which is itself indexed by the next byte.
- **var[src[n]]** - An element of the temporary variable vector indexed by an element of the source vector, which is itself indexed by the next byte.

The right "variables" are as follows:

- **i** - The iteration variable, which can only be changed by the *for* loop.
- **number** - A number defined by the next byte.
- **src.size()** - The size of the *src* vector, present in the code as its absolute value, because *src.size()* is a C++ instruction, not an ANSI C one.

The quality of the numbers generated also depends on the random algorithm that is used. Fortunately, I am running the program on a 2017 iMac model that employs Yarrow (Kelsey, Schneier, & Ferguson, 1999) as the cryptographic pseudorandom number generator in its compiler's library, thereby increasing our odds of attaining a better instance. After generating this moderately expanded set of instructions, let us consider an example of the potential results yielded by a random 50-number sequence:

```
Instructions: 25, 64, 70, 02, 65, 44, 78, 47, 97, 94, 56, 63, 62, 84,
              31, 08, 01, 04, 46, 69, 13, 07, 72, 69, 94, 85, 10, 77,
              66, 05, 51, 43, 22, 21, 98, 87, 18, 28, 35, 15, 22, 91,
              30, 37, 28, 62, 97, 81, 66, 43
```

```
int var[] = {0,0,0,0,0,0};
int i = 0;
src[i] -= var[2];
src[1] /= src[i];
while (var[src[1]] < var[4])
{
      while (i != src[i])
      {
            src[i]++;
            src[i]--;
            src[0] -= 3;
            for (i=1; i<3; i++)
            {
                  var[0] += var[src[2]];
            }
      }
}
```

Figure 9

Figure 9 illustrates the random sequence of numbers generated by the program, along with the ANSI C instructions that they form. The first byte defines how many temporary variables are present in the *var* vector. The variable *i* is always set to 0 as its initial state. The other instructions result from the conversion process, using the defined scope.

Evidently, we can use this set of instructions to generate some complex code. This does not, however, mean that this instance performs a useful function, but rather that this system has the potential to do so. The next step involves inserting the generated code inside a *main(),* with the source vector defined, to see what changes it performs.

5. Running the Code

Let us consider a single example of our Codifier program's output:

```
// Description: Dynamic generated code
// Author: Codifier - The Artificial Intelligence C Programmer
#include <stdio.h>

// Function created by Codifier
int codifier(int *src, int *var)
{
      int i = 0;
      for (i=2; i>0; i--)
      {
            if (var[2] <= var[0])
            {
                  src[0] = src[var[1]] % src[2];
                  src[i]++;
            }
      }
      return i;
}
int main()
{
        // Define source vector
        int source[][3] = {{2,1,3},{7,0,4},{1,9,2}};

        // Run codifier function for each source vector
        int m = 0, n = 0;
        for (m=0; m<3; m++)
        {
            // Initialize variables and run
            int var[] = {0,0,0};
            int i = 0;
            i = codifier(source[m],var);

            // Print status
            printf("// i:%d, src:",i);
            for (n=0; n<3; n++)
                  printf("%d,",source[m][n]);
            printf(" var:");
            for (n=0; n<3; n++)
                  printf("%d,",var[n]);
            printf("\n");
        }
        return 0;
}
```

Figure 10

It is worth noting here that Figure 10 is not a representation of the Codifier program itself, but rather an example of the output that Codifier generates. Codifier, as a program that generates a program, is a C programmer operating via artificial intelligence.

The dynamic part of the code only consists of the block of instructions within the function *codifier()*. The other instructions merely exist to define the initial values and show the results. Running this output as a program would transform the source vector as follows:

```
source = {{2,2,4},{3,1,5},{1,10,3}};
```

Figure 11

To prompt the code to evolve, we must run many different instances, to observe how they work to modify the source vector. My initial idea involved sending the instances to a compiler, such as *gcc* (Free Software Foundation), to analyze the output of the program subsequent to running it. However, I soon identified a few problems associated with such an approach:

- It is too slow; given the many thousands of instances that would have to be generated per second, compiling and running each instance would consume too much time.
- Certain instances have specific problems, such as buffer overflow, which should be recognized and discarded.

To avoid these problems, it was necessary to create an interpreter, which was, by far, the most difficult aspect of creating the whole program. Although the numbers are interpreted as instructions, they are also executed against the variables, which represent a complicated set of demands. The program needs to ignore instructions inside an *if* that is untrue, run multiple times when inside a *for*, and even do both simultaneously within *while* instructions.

The interpreter recognizes the following problems, and discards any instance where any such problem occurs:

- **Infinite loop** - A *while* loop could never ends, because the Boolean expression is tied to variables that never change. A loop limit is set as the size of the instruction vector multiplied by the size of the source vector.
- **Buffer overflow** - An effort to read or write a value outside vector limits can result in a serious error that may crash a computer. The program stops the interpretation before this happens.
- **Division by zero** - This error is also detectable by a compiler, but still included in the list of errors that Codifier detects.
- **Integer overflow** - An integer has a limit for the number it represents. In a 32-bit system, a signed integer ranges from -2^{31} to $2^{31}-1$. If a calculation extends beyond this limit, it overflows. This is not necessarily an error, but can lead to unexpected behaviors. The program uses a set of overflow-detection instructions to identify those cases (Warren, 2013).

To make the interpreter work, I employed a trick that permits us to use a pointer to indirectly change the value of a variable. In this way, I can, for example, define an integer *left* that points to any of the left variables defined in our scope. Afterwards, if I set a value to *left*, the corresponding variable also changes.

Here is a small example of the concept in C++:

```
int main()
{
   int val = 4;                 // Variable to be controlled
   int *ptr;
   ptr = &val;
   int &left = *ptr;            // “left” controls “val”
   left = 3;
   cout << val << endl;         // Outputs "3" as a result
}
```

Figure 12

6. Genetic Algorithm

Alan Turing (1950) was the first to propose the idea of an algorithm that mimics evolution. Stuart Russel and Peter Norvig (2009) published this specific variant of genetic algorithm.

This is the motor of the program. The genetic algorithm creates four instances of *Codifier* class and then evolves until the source vector corresponds to the target.

The flowchart used by the algorithm is as follows:

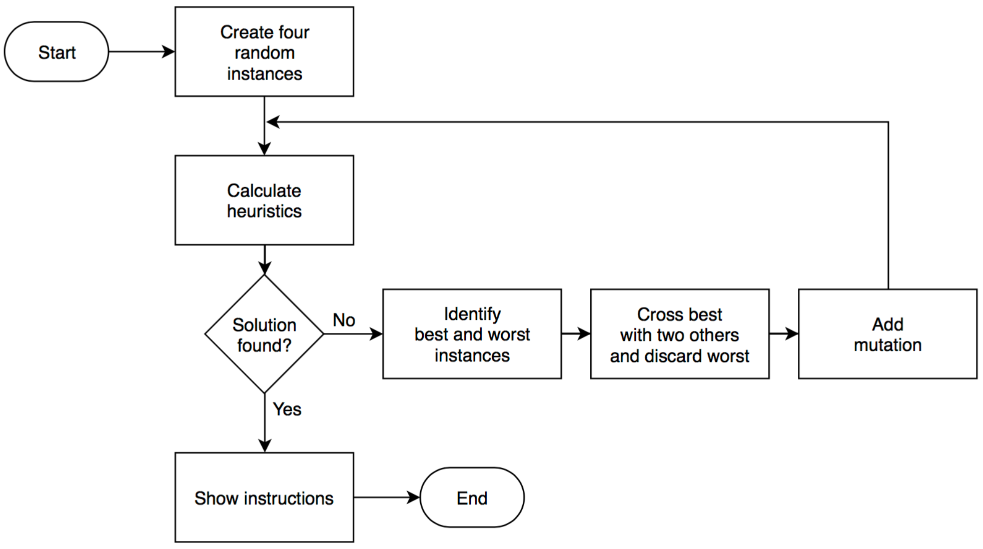

Start
Create four random instances
Calculate heuristics
Solution found?
No
Identify best and worst instances
Cross best with two others and discard worst
Add mutation
Yes
Show instructions
End

Figure 13

Let us examine each block more closely:

6.1. Create Four Instances

The algorithm creates four instances of *Codifier* class as a starting point. In each instance, the *instructions* vector is filled by random numbers that range from 0 to 99. The static variable *size* defines the quantity of random numbers.

The program interprets each instance before accepting it, to avoid errors and ensure that only viable instances are moved forward in the algorithm.

6.2. Calculate Heuristics

Heuristics are a way of evaluating an instance, determining how far it is from achieving the goal (which, in this case, is the transformation of the source into the target). Each instance should be interpreted, largely to evaluate which changes it performs on a copy of the source vector. Let us create an example:

```
source[][] = {{2,1,3},{7,0,4},{1,9,2}};
target[][] = {{1,2,3},{0,4,7},{1,2,9}};
```

Figure 14

Now let us imagine that our instance transforms the source into the following values:

```
source[][] = {{0,1,3},{0,0,4},{0,9,2}}
```

Figure 15

One way to calculate a heuristic is to determine the distance between the transformed source and the desired target. Figuring this out simply requires us to calculate the absolute difference between each corresponding number, as follows:

```
source[][] = {{0,1,3},{0,0,4},{0,9,2}};
target[][] = {{1,2,3},{0,4,7},{1,2,9}};
difference =   1+1+0 + 0+4+3 + 1+7+7 = 24
```

Figure 16

So, for this instance, the heuristic would be 24. The smaller the heuristic is, the better the instance.

6.3. Solution Found

If, in a particular instance, we perform the calculation and determine that the heuristic is 0, this indicates that the source vector has been transformed into the target. In turn, this means that the genetic algorithm identified a set of instructions capable of achieving the goal. In this case, the instruction set in the relevant instance will be noted, to demonstrate the program's ability to transform the source vector into the target.

6.4. Identify Best and Worst

In each cycle of the genetic algorithm, we must rate the instances according to their comparative heuristic values. The best instance is the one that has the smallest heuristic value, and the worst instance has the largest. If more than a single instance has the same—and best—heuristic value, the first instance identified is selected. The same principle applies to multiple instances that share the same—and worst—heuristic value.

6.5. Cross the Best with 2 Others and Discard the Worst

Given that we have 4 instances, we know that there one has the best heuristic value, another has the worst, and the other 2 have intermediate heuristic values. Let us refer to each half of this pair as *other1* and *other2,* respectively.

Let us consider an example. Imagine that each of the 4 instances contains 10 bytes in the instructions vector and are rated as Figure 17 indicates.

```
Best:   25, 64, 70, 02, 65, 44, 78, 47, 97, 94
Other1: 56, 63, 62, 84, 31, 08, 01, 04, 46, 69
Other2: 13, 07, 72, 69, 94, 85, 10, 77, 66, 05
Worst:  51, 43, 22, 21, 98, 87, 18, 28, 35, 15
```

Figure 17

The best instance will execute a sort of crossover with the 2 other instances by exchanging a random number of bytes in the *instructions* vector, for which a random point of exchange will be generated. Let us suppose that, in this example, this random point is the 3rd number for *other1* and the 7th number for *other2*:

```
Best:   25, 64, 70, 02, 65, 44, 78, 47, 97, 94
Other1: 56, 63, 62, 84, 31, 08, 01, 04, 46, 69

Child1: 25, 64, 70, 84, 31, 08, 01, 04, 46, 69
Child2: 56, 63, 62, 02, 65, 44, 78, 47, 97, 94

Best:   25, 64, 70, 02, 65, 44, 78, 47, 97, 94
Other2: 13, 07, 72, 69, 94, 85, 10, 77, 66, 05

Child3: 25, 64, 70, 02, 65, 44, 78, 77, 66, 05
Child4: 13, 07, 72, 69, 94, 85, 10, 47, 97, 94
```

Figure 18

These children will replace the 4 original instances in the genetic algorithm; as you can observe, the program did not use the worst instance, but rather discharged and thereby excluded it.

The program interprets each instance before accepting it, to avoid errors. In such erroneous cases, it would select an alternative point of exchange at which to execute the cross.

6.6. Add a Mutation

As occurs in nature, we give the instances the opportunity to have one of its numbers changed randomly. We should assign it a very low numerical probability for this to occur, and as such, in fact, setting the probability level at about 0.01 yields the optimal results (Grefenstette, 1986).

```
Child1: 25, 64, 70, 84, 31, 08, 01, 04, 46, 69 // Before mutation
Child1: 25, 64, 70, 84, 05, 08, 01, 04, 46, 69 // After mutation
```

Figure 19

Here, the program also interprets each instance before accepting it, to avoid errors.

7. Removing Useless Instructions

After we find a solution, the next step is to remove useless instructions. These are instructions that fail to contribute to the work of transforming the source into target, and are present because the program generates many random instructions. The following algorithm eliminates them:

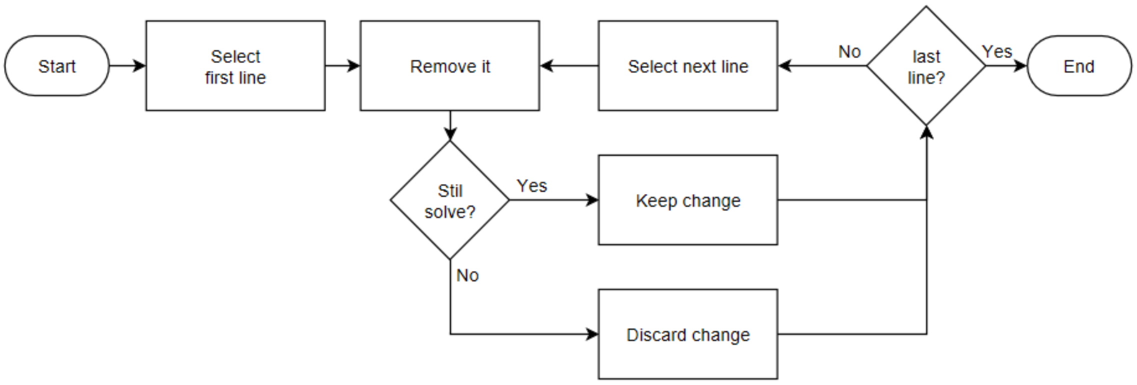


Figure 20

This operates by replacing the byte used on the beginning of the instruction with -1 and all of its parameters with -2. The following serves as an example:

```
Inst: 26, 06, 90, 94, 01, 77, 93, 66, 37, 08, 61, 45, 91, 31, 37, ...
src[i] = 1 * i;
src[2] = i + src[var[3]];
src[var[5]] -= src[var[3]];
```

```
Inst: 26, 06, 90, 94, 01, 77, 93, -1, -2, -2, -2, -2, -2, -2, 37, ...
src[i] = 1 * i;
; // Killed
src[var[5]] -= src[var[3]];
```

Figure 21

The program subsequently ignores those numbers and reassumes the interpretation following the final -2.

But remember that flow instructions such as *if*, *for* and *where* defines how many instructions are inside their blocks. Therefore, such a scenario could happen:

```
for (i=1; i>-1; i--) // 2 instr.
{
    if (3 < 74) // 2 instr.
    {
        var[src[1]]--;
        src[i] = src[i];
    }
    src[var[4]]--;
}
```

```
for (i=1; i>-1; i--) // 2 instr.
{
    ; // Killed
    var[src[1]]--;
}
src[i] = src[i];
src[var[4]]--;
```

Figure 22

As you can observe in Figure 22, following the removal of *if*, its instructions are shifted over to the *for* block. This modification, however, only works for a maximum of 2 instructions, forcing the remaining instructions outside. I defined a flow instruction, along with the instructions inside its block, to count once when inside another block. Killed instructions, as above, also count as regular ones. We must guarantee that all instructions remain in a static location after a removal, which we can ensure by saving the quantity of commands for each flow instruction in a vector called *heads*. For this example, such a vector appears as follows:

```
for (i=1; i>-1; i--) // 2 instr.
{
    if (3 < 74) // 2 instr.
    {
        var[src[1]]--;
        src[i] = src[i];
    }
    src[var[4]]--;
}
```

```
Inst: 53, 97, 36, 91, 89, 79, 26, 30, 74, 71, 59, 29, 58, 33,  8, 90,
      66, 24, 38, 53,  4, 97
Head:  2,  0,  0,  0,  2,  0,  0,  0,  0,  0,  1,  0,  0,  0,  1,  0,
       0,  0,  1,  0,  0,  0
```

Figure 23

Each number in the *heads* vector refers to the number in the same position in the *instructions* vector.

In Figure 23, the *heads* vector starts with a number 2, followed by three 0s; these correspond to the *for* instruction and its arguments. The number 2 signifies that are 2 instructions inside its block which, in this case, corresponds to the *if* block and *src[var[4]]--*.

Then we can observe another number 2 followed by five 0s; these correspond to the *if* instruction and its arguments, which is the one we want to kill. Subsequently, we observe 3 groups of 1s followed by 0s; these correspond to the other 3 instructions and their arguments.

To avoid the risk that the killing process might modify the position of the instructions, we must add to the parent block the same number of instructions that were inside the killed block. Doing this produces in the following result:

```
for (i=1; i>-1; i--) // Now contains 4 instr.
{
    ; // Killed
    var[src[1]]--;
    src[i] = src[i];
    src[var[4]]--;
}
```

```
Inst: 53, 97, 36, 91, -1, -2, -2, -2, -2, -2, 59, 29, 58, 33,  8, 90,
      66, 24, 38, 53,  4, 97
Head:  2,  0,  0,  0,  2,  0,  0,  0,  0,  0,  1,  0,  0,  0,  1,  0,
       0,  0,  1,  0,  0,  0
```

Figure 24

8. Results

Let us apply the term *sample* to each vector (i.e., each group of numbers) inside the source and target vectors. If we provide more than 1 sample, the algorithm identifies a solution that fits them all. As such, we are using a learning algorithm that processes samples and returns a solution applicable even to samples that were not provided in the learning phase. This approach is similar to others that rely on a small neural network, but the advantage of this method is that it is followed by an adjustable piece of code instead of a black box. Here, I present some results obtained by running the Codifier algorithm. Let us consider the results, to fully grasp the implications and capabilities of this algorithm.

8.1. Inverting

Let us start with something simple. We can search for a code that inverts the order of the elements in the source vector. As each of the following samples has only 3 numbers, the code needs only to swap the first and last elements. After a few attempts, I found this instance:

```
size = 50
srand(64983)
source = {{2,1,3},{7,0,4},{1,9,2}}
target = {{3,1,2},{4,0,7},{2,9,1}}
```

```
int codifier(int *src, int *var)
{
      int i = 0;
      src[var[2]] = src[2] - src[i];
      src[2] -= src[var[2]];
      src[i] += src[2];
      return i;
}
// Generations: 56272
// Instances:   929428
// Time: 19 sec
```

Figure 25

The code presented in Figure 25 shows only the dynamic part of the output. It required 19 seconds to generate this solution and managed to create 929428 instances. As each generation uses 4 instances, we actually used 225088 (i.e., 56272*4) instances. This suggests that errors led the code to discard roughly 76% of the instances, which is predicable, given the randomness of the initial instances.

It is capable of inverting any vector with 3 numbers. The following edited version of the code executes the same operations in a clearer way:

```
int codifier(int *src, int *var)
{
      int i = 0;                 // if {a,b,c}
      src[0] = src[2] - src[0]; // src[0] = c-a
      src[2] -= src[0];         // src[2] = c-(c-a) = a
      src[0] += src[2];         // src[0] = c-a+a = c
      return i;                 // then {c,b,a}
}
```

Figure 26

8.2. Summation

Now, let us move on to something more difficult. We can search for a piece of code that determines the sum of all elements in the sample and puts the result on the target. Given that, in this example, we must maintain the number of elements for placement on the target, the result will replace all elements of the sample.

After some attempts, I found this instance:

```
size = 75
srand(78079)
source = {{2,1,3},{ 7, 0, 4},{ 1, 9, 2}}
target = {{6,6,6},{11,11,11},{12,12,12}}
```

```
int codifier(int *src, int *var)
{
        int i = 0;
        for (i=0; i<3; i++)
        {
                src[var[4]] += src[1];
                src[1] = src[var[0]];
                var[4]++;
        }
        for (i=2; i>0; i--)
        {
                src[var[2]] = src[2] % 48;
                src[1] = src[i] + var[1];
        }
        return i;
}
// Generations: 17836
// Instances:   296312
// Time: 11 sec
```

Figure 27

Codifier required 11 seconds to generate the solution rendered in Figure 27 and did so by creating 296312 instances, about 76% of which were discarded as errors.

It is extremely compelling to observe how the program obtains the result by playing around with the numbers. Nobody would ever create such a code, but it does its intended job perfectly, provided that the sum is less than 48.

As the solution is the result of a genetic algorithm, it contains some indirect references and useless parts. This "noise" that can be removed from the code without altering its functionality is the main characteristic of a code generated via evolution. Searching for it can enable us to determine whether the code was designed or evolved. I believe that this kind of analysis is applicable to other fields, such as genomics.

8.3. Sorting

Now, let us attempt something much harder. For example, let us suppose that we wanted to sort the numbers in ascending order. At first, I found the following instance:

```
size = 75
srand(60426)
source = {{2,1,3},{7,0,4},{1,9,2}}
target = {{1,2,3},{0,4,7},{1,2,9}}
```

```
int codifier(int *src, int *var)
{
        int i = 0;
        src[1] = src[2] - var[3];
        for (i=0; i<2; i++)
        {
                var[src[2]]++;
                if (var[src[2]] < src[var[3]])
                {
                        src[i] = var[3];
                }
        }
        src[i] = 3 * var[src[1]];
        src[var[4]]++;
        src[i]++;
        src[i] = src[var[5]] + src[i];
        src[var[4]]--;
        return i;
}
// Generations: 56840
// Instances:   945603
// Time: 30 sec
```

Figure 28

Codifier required 30 seconds to generate the solution rendered in Figure 28 and did so by creating 945603 instances, of which, again, roughly 76% were discarded as errors.

It is capable of sorting the provided source vector, but does not work properly with other numbers. To render the code more flexible, we must furnish it with more samples and perhaps enlarge the instructions vector. After quite a few attempts, I found the instance in Figure 29.

In this new instance I provide 8 samples, which yields much better results, sorting not only the source samples, but many others as well. It remains imperfect, but is a clear improvement over the code employed in Figure 28. It required 174 seconds to be generated, and discarded about 76% of the instances as errors.

We can observe the extent to which this tool is much more suitable to the task of creating small pieces of code, as opposed to a single complex one. This is logical and to be expected, as it is sensible to approach a problem by breaking it into small pieces that can then be addressed and resolved individually. Such a process is the core of abstraction.

Please recall that this is merely a test of a concept, operating on a domestic computer. It would certainly be easier to achieve more complex results within a shorter time period if we created a multithread version and ran it on a powerful machine with lots of processors.

```
size = 100
srand(1153489657)
source = {{2,1,3},{7,0,4},{1,9,2},{2,4,8},{9,2,1},{3,7,5},{2,0,2},{1,1,1}}
target = {{1,2,3},{0,4,7},{1,2,9},{2,4,8},{1,2,9},{3,5,7},{0,2,2},{1,1,1}}
```

```
int codifier(int *src, int *var)
{
      int i = 0;
      for (i=2; i>0; i--)
      {
            var[0] = src[i] * 1;
            if (src[var[9]] <= 3)
            {
                  var[9] = 3 % i;
            }
            while (src[i] < src[var[9]])
            {
                  src[i] = src[var[9]];
                  src[var[9]] = var[0] + var[6];
            }
      }
      return i;
}
// Generations: 83782
// Instances:   1398167
// Time: 174 sec
```

Figure 29

9. Conclusion

The foregoing discussion described a process wherein we determined how to transform a random sequence of numbers in a group of instructions and a way to evolve it to achieve a goal. I provided the steps indicating how to construct it and attached a working program, written in C++, to serve as a test of concept described in this paper.

This approach represents a contribution to the foundational efforts to design and create an agent capable of thinking and, perhaps at some future time, evolving by means of adding blocks of code to itself.

We should regard artificial intelligence as a partner in our search for a better future, instead of a potential enemy or threat. By contrast to the yet blameless agents of artificial intelligence, human beings have demonstrated our own inability to live peacefully with other beings; it may even be possible to employ artificial intelligence to enable us to avoid our own self-destruction.

References

Free Software Foundation. (2016). *Using the GNU Compiler Collection*. Retrieved from GNU Operating System: http://gcc.gnu.org/onlinedocs/gcc/

Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics SMC-16(1)*, 122-128.

ISO/IEC 14882. (2011). Information technology -- Programming languages -- C++. *International Organization for Standardization*.

ISO/IEC 9899. (2011). Information technology -- Programming languages -- C. *International Organization for Standardization*.

Kelsey, J., Schneier, B., & Ferguson, N. (1999, Aug). Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. *Sixth Annual Workshop on Selected Areas in Cryptography*. Retrieved from https://www.schneier.com/academic/archives/2000/01/yarrow-160.html

Mahoney, M. S. (2002). Software: The Self-Programing Machine. In A. Akera, & F. Nebeker, *From 0 to 1: An Authoritative History of Modern Computing.* New York: Oxford U.P.

Russel, S., & Norvig, P. (2009). Genetic algorithms. In *Artificial Intelligence - A Modern Approach* (pp. 126-129). New Jersey: Pearson Education, Inc.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society v.S2-42*, 230–265.

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind 49*, 433-460.

Warren, H. S. (2013). Overflow Detection. In J. Henry S. Warren, *Hacker's Delight* (pp. 31-34). Westford, Massachusetts: Pearson Education, Inc.

Saulo Fonseca, Graduated in Electronics from Federal Technical College in Brazil and in Software Engineering from Universidade Aberta de Portugal. He began his professional life as a networking technician and now works in a major automotive parts manufacturer in Germany.