Ted Holmberg

Fall 2019: CSCI 6522: Programming Assignment #1 - Part A:

Hill-Climbing Algorithm

Abstract:

In Part A, Hill-climbing algorithm is implemented to find a local maximum solution for a given fitness function that processes on binary strings.

Table of Contents:

I. Problem Statement:

Part 1 [Marks 20]

#1. Write a *Hill-Climbing* algorithm to find the maximum value of a function *f*, where *f* = |13 • one (*v*) -170|. Here, *v* is the input binary variable of 40 bits and the *one* counts the number of '1's in *v*. Set MAX =100, and thus *reset* algorithm 100 times for the global maximum and print the found maximum-value for each *reset* separated by a comma.

To provide context to this problem, assume, the search space is a set of binary strings v of the length 40. The objective function f to be maximized is given as:

$$f(v) = | 13 * one(v) - 170 |$$

where the function one(v) returns the count of 1s that appear in the string v.

For example, the following three strings:

v_1 = 1101101011101011111110110110111100110101
v_2 = 1010001001001101010010101000110000000101
v_3 = 0000100000110010000000100010000000000000

would evaluate to:

Since, one(v_1) = 28, one(v_2) = 15, and one(v_3) = 6.
$f(v_1) = | 13 * 28 - 170| = 194$,
$f(v_2) = | 13 * 15 - 170| = 25$,
$f(v_3) = |13 * 6 - 170| = 92$,

The function *f* is linear and does not provide any challenge as an optimization task. Its used for this report only to illustrate the ideas behind the hill-climbing algorithm.

However, the interesting characteristic of the function *f* is that it has one global maximum for v_g = 1111111111111111111111111111111, such that $f(v_g) = |13* 40 - 170| = 350$,

and one local maximum for
v_l = 0000000000000000000000000000000, such that $f(v_l) = |13* 0 -170| = 170$.

II. Approach

Hill-climbing Algorithm:
A simple iterated hill-climbing algorithm (with MAX iterations) will be used to perform a steepest ascent hill-climbing.

Basic Algorithm:

```
begin
   t ← 0
   repeat
      local ← FALSE
      select a current string v_c at random
      evaluate v_c
      repeat
         select 40 new strings in the neighborhood of v_c by flipping single bits of V_c
         select the string V_n from the set of new strings with the largest value of function f
         if f(v_c) < f(v_n)
            then v_c ← v_n
            else local ← TRUE
      until local
      t ← t + 1
   until t = MAX
end
```

Algorithm Explanation:
Step 1: repeat the following process a number of times equal to MAX, in this case 100.
Step 2: create a local variable to track when local maximum found, defaults to false.
Step 3: randomly generate a 40-bit string
Step 4: evaluate that string with the fitness function
Step 5: Then repeat following process until a local maximum is found:
Step 6: get all 40 possible neighbors to this string, flipping a bit in position 0,1,2...39.
Step 7: compare the current string fitness to each neighbor's fitness value
Step 8: if a neighbor's fitness value is higher, update that as the current string
Step 9: otherwise, a local minimum has been found
Step 10: this iteration ends, go back to step Step 6

Technology & Implementation:
JavaScript selected to implement the project for its portability. As such all testing and executing of this application is made trivially easy for any computing device with a browser. The remainder of this section will detail the approach toward implementation.

Step 1: Initialize Random 40-bit Binary String

A random binary string is initialized with the init function.

```
function init(bits){
    let bit_string = "";
    for (let i=0; i<bits; i++){
        let bit = parseInt(Math.random()*2);
        bit_string += bit;
    }
    return bit_string;
}
```

This function randomly generates a binary sequence that's length is based on the parameter passed in. The bit string is initialized as an empty string. A for-loop iterates a number of times equal to the number of bits needed. A random integer of either 0 or 1 is generated and then concatenated onto the end of the bit string. After the for-loop is complete processing, a random n-bit binary string is then returned.

Step 2: Evaluate Bit-String Fitness

Evaluating the fitness of a binary string occurs across two functions. countOnes and getFitness.

```
function countOnes(bit_string){
    return bit_string.split('').reduce( (a,b) => (+a)+(+b) );
}
```

The countOnes function returns the count of ones in a binary string. Given a string as a parameter, it will split it into an array of singular values and iterate across each value summing them together and returning the result. The lambda function passed to the reduce method will take each text element, cast it into an integer and add them together. The reduce method will then reduce this process into a singular solution.

```
function getFitness(bit_string){
    let ones = countOnes(bit_string);
    return Math.abs(13 * ones - 170);
}
```

The getFitness function returns the fitness of the binary string. Given a string as a parameter, it invokes countOnes to get the number of ones as an integer number. It then performs the given equation which is to multiply that count by 13 then subtract 170 and finally takes its absolute value.

Step 3: Get a Bit-String's Neighbors

Getting all of a given binary string's neighbors occurs across two functions: flipBit and getNeighbors.

```
function flipBit(index, bit_string){
    let bits = bit_string.split("");
    bits[index] = (bits[index] == '0') ? '1' : '0';
    return bits.join("");
}
```

The flipBit function is a helper method to getNeighbors. Given a binary string as a parameter and an index, it will flip the bit at the given index. This is accomplished by splitting the string into an array of values. Then using a conditional operator flip the value at that index. Then join the array back into a string and return it.

```
function getNeighbors(bit_string){
    let neighbors = [];
    for(let i=0; i<bit_string.length; i++){
        let neighbor = flip_bit(i,bit_string);
        neighbors.push(neighbor);
    }
    return neighbors;
}
```

The getNeighbors function returns all the topological neighbors of a given binary string. Where a neighbor may be defined as a binary string that differs from the given string by a single bit value. Since this problem uses 40-bit strings, then there are 40 possible neighbors that must be generated. This is done by first making a local array to hold the neighbors. Then for each index from 0 to the length of the binary string, invoke the helper method flipBit to generate a neighbor and push it into the neighbors array. Once this process is complete, return that array.

Step 4: Hill-Climbing Algorithm

The hill-climbing algorithm is then implemented in one function:

```
function hillClimb(){
    let output = new Plotter();
    let bits = 40
    let max = 100;
    for( let iter=0; iter<max; iter++){
        let local = false;
        let current = init(bits);
        let fitness = getFitness(current);
        let steps = 0;
        output.newIteration(iter, steps, current, fitness);
        while (local == false){
            neighbors = getNeighbors(current);
            let next = current;
            for (let j=0; j<neighbors.length; j++){
                let neighbor_fitness = getFitness(neighbors[j]);
                if (fitness < neighbor_fitness ){
                    fitness = neighbor_fitness;
                    next = neighbors[j];
                }
            }
            if (getFitness(current) < getFitness(next)){
                current = next;
                output.updateIteration(++steps, current, fitness);
            }
            else{
                local = true;
                output.endIteration();
            }
        }
    }
    output.show();
}
```

The hillClimb function executes the main algorithm. First it initializes the number of bits, and the max number of iterations. For graphical output, a separate Plotter class was created which will be further detailed in the next section.

Then a for-loop executes for a number of iterations equal to the max value. Within the for-loop, an inner control variable for finding the local maximum is initialized to false. Then a random binary string is initialized and its fitness is calculated. Then a counter variable is created to store the number of steps it takes to find a local maximum. This initial data is logged into the plotting object that will display the final results.

The inner while-loop executes until a local maximum is found. To accomplish this, all 40 of the neighbors are generated. Then a local variable to store the next binary string is initialized with the current string as its default. For each neighbor in the collection of neighbors, they are compared to the current highest fitness, if a given neighbor's fitness is higher than the current highest, then both the fitness variable and the next variable are both updated.

Finally that next fitness value is compared to the current string's fitness. If it is higher, than a step is performed and the current variable is updated with the value from the next variable. Otherwise, if it is the same or lower, then we have reached a maximum, and we must toggle the loop control variable to true to end this iteration. The end of iteration will be logged into the plotter.

After max iterations occurs, the results will be graphed to the screen.

Step 5: Data Visualization

All visualization for this project is done using Plotly, an open source JS graphics library that is also available on Python and R. All text output is rendered using HTML. To prepare the output data for both graphics and text, a custom class was designed as detailed below.

Plotter is a custom javascript class that contains the following methods for this project.

```
constructor(){
    this.fitness_to_value_plot = [];
    this.steps_to_fitness_plot = [];
    this.steps_to_value_plot = [];
    this.results = "";
 }
```

constructor that holds the data for three different plots and the html text output.

```
getNewTrace(x,y){
    return {x:[x], y:[y], color:'blue'};
 }
```

a helper method to generate a new config object to store data for a future trace object.

```
updateTrace(arr,x,y){
    arr.x.push(x);
    arr.y.push(y);
 }
```

a helper method to updateTrace object with new x, y data values

```
newIteration(iter, steps, current, fitness){
    let value = parseInt(current, 2);
    this.fitness_to_value = this.getNewTrace(fitness, value );
    this.steps_to_fitness = this.getNewTrace(steps, fitness);
     this.steps_to_value = this.getNewTrace(steps, value);

     this.results += `<b>Iteration: ${iter}</b><br>`
     this.results += `Step ${steps}:`
     this.results += `Fitness: ${fitness}, String: ${current}<br>`
    }
```

A method that logs the plot data for a new iteration of the hill-climb

```
updateIteration(steps, current, fitness){
    let value = parseInt(current, 2);
    this.results += `Step ${steps}:`
    this.results += `Fitness: ${fitness}, String: ${current}<br>`

    this.updateTrace(this.fitness_to_value, fitness, value);
    this.updateTrace(this.steps_to_fitness, steps, fitness);
    this.updateTrace(this.steps_to_value, steps, value);
    if ( !current.includes("1") ){
        this.fitness_to_value.color = 'red';
        this.steps_to_fitness.color = 'red';
        this.steps_to_value.color = 'red';
    }
 }
```

A method that updates the plot data for an ongoing iteration of the hill-climb

```
endIteration(){
    this.fitness_to_value_plot.push(this.make_trace(this.fitness_to_value));
    this.steps_to_fitness_plot.push(this.make_trace(this.steps_to_fitness));
    this.steps_to_value_plot.push(this.make_trace(this.steps_to_value));
 }
```

A method that finalizes the data for an iteration of the hill-climb

```
make_trace(data){
        let trace = {
            x: data.x ,
            y: data.y ,
            mode: 'lines',
            marker: {color:data.color, size: 10}
        };
        return trace;
    }
```

A helper method that generates a trace object for Plotly

```
show(){
    let label1 = {title:"Fitness-to-Numerical Value",
                  x:"Fitness Score",
                  y:"Numerical Value of String"};
    this.plot('plot-climbers',this.fitness_to_value_plot, label1);
    let label2 = {title:"Number Steps-to-Highest Fitness",
                  x:"Number of Steps",
                  y:"Fitness Score"};
    this.plot('plot-iterations',this.steps_to_fitness_plot, label2);
    let label3 = {title:"Number Steps-to-Numerical Value",
                  x:"Number of Steps",
                  y:"Numerical Value of String"};
    this.plot('plot3',this.steps_to_value_plot, label3);

    let out = document.getElementById('results-list');
    out.innerHTML += this.results;
}
```

Method to show the logged data as graphics and text.

```
plot(div,trace_list, labels){
        let data = [ ...trace_list ];

        let layout = {
            title: {
                text: labels.title
            },
            xaxis: {
                title: {
                    text: labels.x
                }
            },
            yaxis: {
                title: {
                    text: labels.y
                }
            }
        };
        Plotly.newPlot(div, data, layout);
    }
}
```

A method to generate a single plot using Plotly.

III. Application Instructions

How to execute the Application:
It is very easy to run this hill-climbing application. Go to the following url:

https://scalemailted.github.io/MachineLearning2-HW1A/

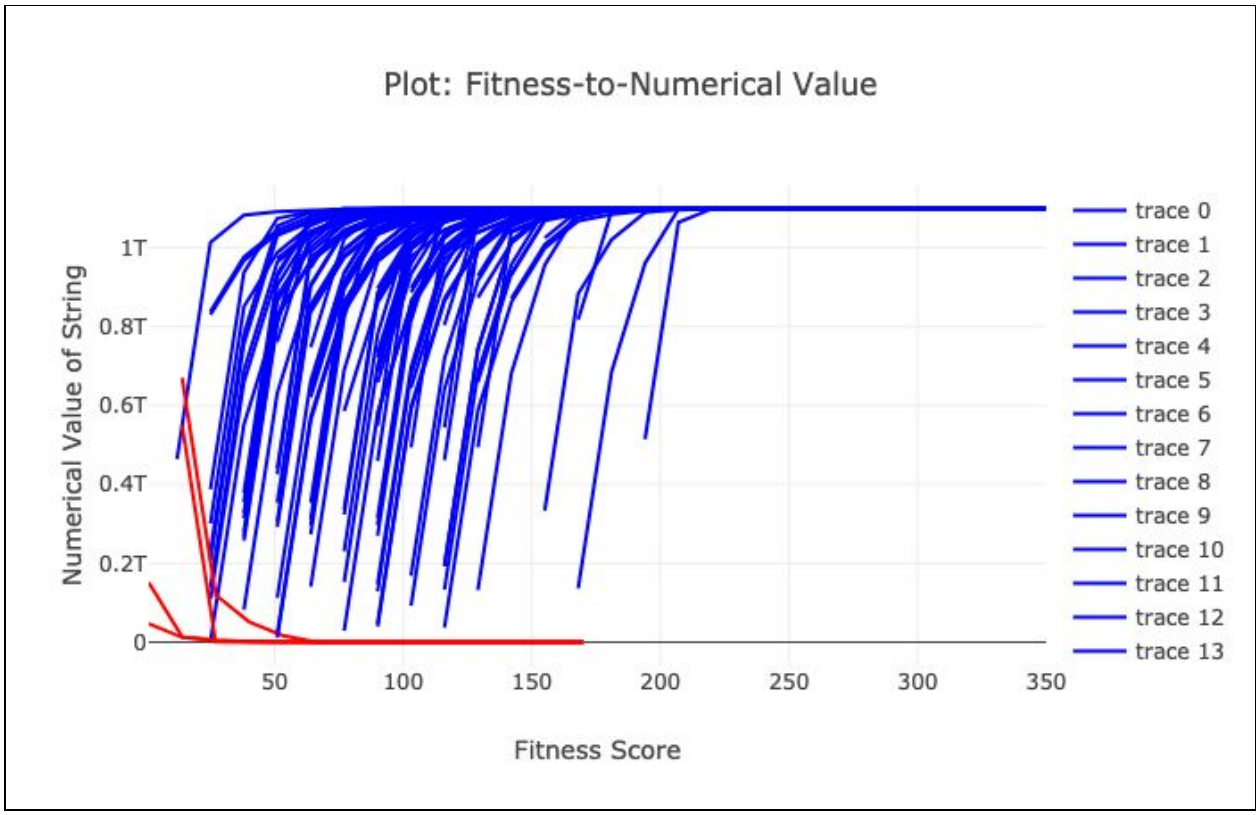
And press the green start button:

Each time you press the button, it will randomly generate new results!

IV. Results

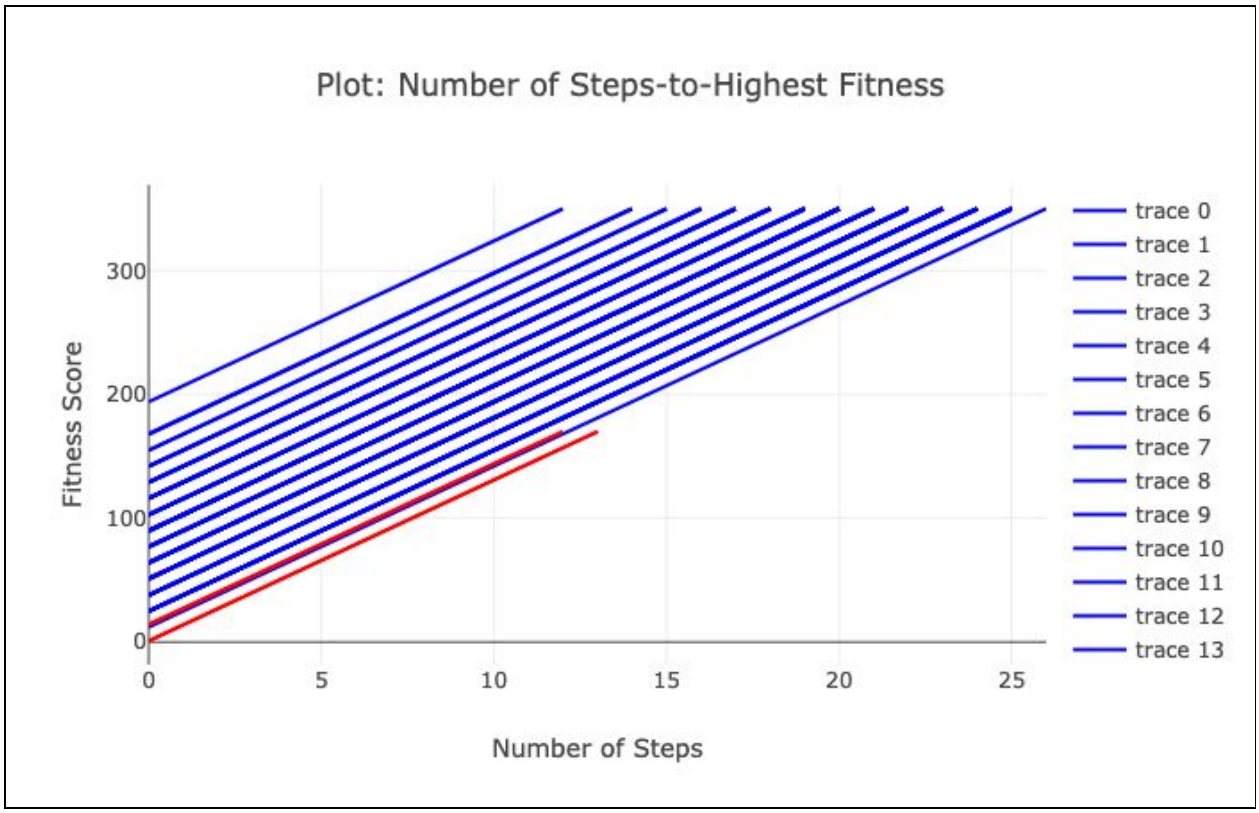
In this section, some examples output is generated to better explain the graphics and text.

Graphic 1: Fitness to Decimal Value



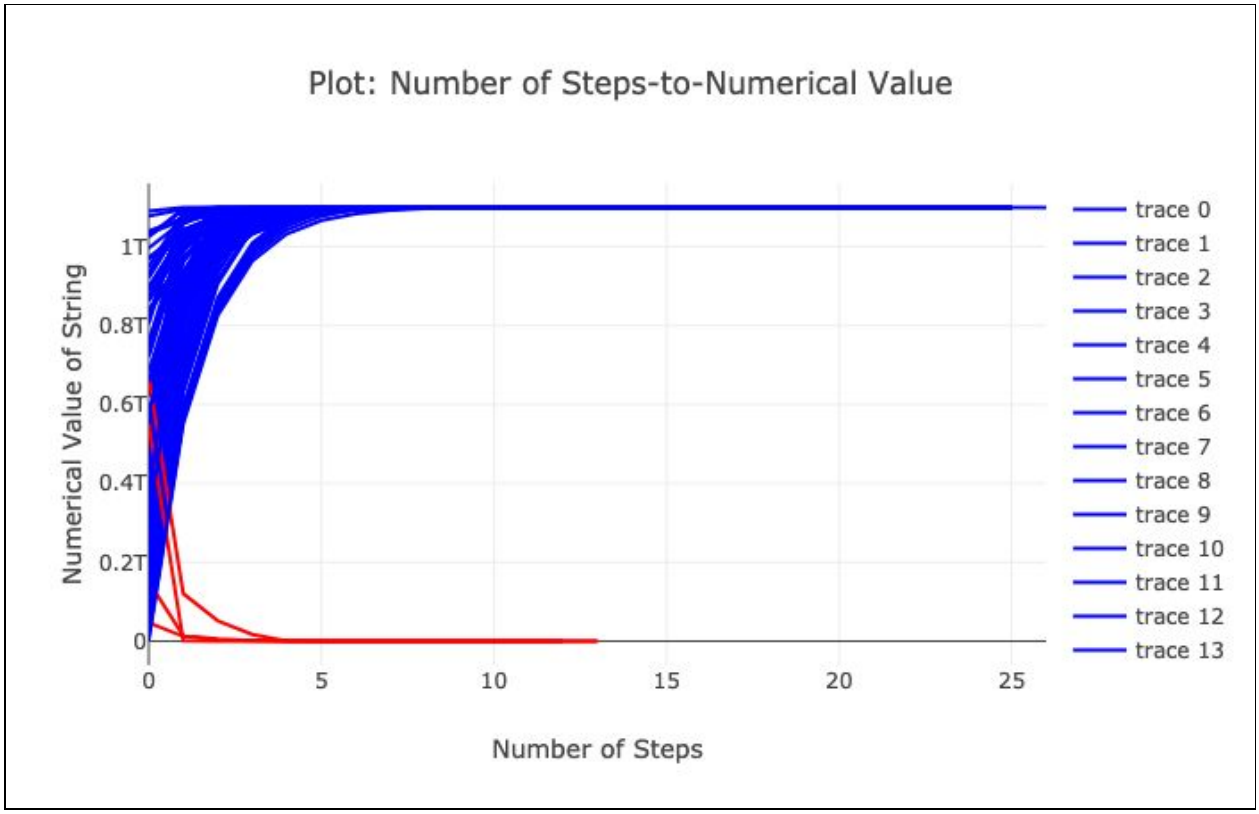
In this plot, you can visualize the relation between the fitness score and the numerical value of the string. Recall that the global maximum is 11111111111111111111111111111111111111111 and that the local maximum is 00000000000000000000000000000000000000000. This plot allows you to see as for each iteration how the fitness score goes up that the string's numerical value trends towards one of the two maximums.

Graphic 1: Number of Steps to Highest Fitness Score



In this plot, you can visualize the relation between the Number of steps to get to a global/local maximum value and the fitness score for each step. This plot allows one to verify that for each step taken, a higher fitness score was always selected. This plot also illustrates how the red plots (local maximum) are lower than the blue plots (global maximum).

Graphic 3: Number of Steps to Numerical Value



In this plot, you can visualize the relation between the number of steps and the numerical value of the string. This plot allows you to see as for each iteration how quickly the hill-climbing algorithm converges towards one of the two maximums.

Text-log 1: A Step-by-Step walk toward Global Maximum

Iteration: 1

Step 0: Fitness: 38, String: 00111110010010000000001110110100100000101
Step 1: Fitness: 51, String: 10111110010010000000001110110100100000101
Step 2: Fitness: 64, String: 11111110010010000000001110110100100000101
Step 3: Fitness: 77, String: 11111111010010000000001110110100100000101
Step 4: Fitness: 90, String: 11111111110010000000001110110100100000101
Step 5: Fitness: 103, String: 11111111111010000000001110110100100000101
Step 6: Fitness: 116, String: 11111111111110000000001110110100100000101
Step 7: Fitness: 129, String: 11111111111111000000001110110100100000101
Step 8: Fitness: 142, String: 11111111111111100000001110110100100000101
Step 9: Fitness: 155, String: 11111111111111110000001110110100100000101
Step 10: Fitness: 168, String: 11111111111111111000001110110100100000101
Step 11: Fitness: 181, String: 11111111111111111100001110110100100000101
Step 12: Fitness: 194, String: 11111111111111111110001110110100100000101
Step 13: Fitness: 207, String: 11111111111111111111001110110100100000101
Step 14: Fitness: 220, String: 11111111111111111111101110110100100000101
Step 15: Fitness: 233, String: 11111111111111111111111110110100100000101
Step 16: Fitness: 246, String: 11111111111111111111111111110100100000101
Step 17: Fitness: 259, String: 11111111111111111111111111110010000101
Step 18: Fitness: 272, String: 11111111111111111111111111111010000101
Step 19: Fitness: 285, String: 11111111111111111111111111111110000101
Step 20: Fitness: 298, String: 11111111111111111111111111111111000101
Step 21: Fitness: 311, String: 11111111111111111111111111111111100101
Step 22: Fitness: 324, String: 11111111111111111111111111111111110101
Step 23: Fitness: 337, String: 11111111111111111111111111111111111101
Step 24: Fitness: 350, String: 11111111111111111111111111111111111111

Text-log 2: A Step-by-Step walk toward Local Maximum

Iteration: 36

Step 0: Fitness: 1, String: 00001010110101001000010011100010100000000
Step 1: Fitness: 14, String: 00000010110101001000010011100010100000000
Step 2: Fitness: 27, String: 00000000110101001000010011100010100000000
Step 3: Fitness: 40, String: 00000000010101001000010011100010100000000
Step 4: Fitness: 53, String: 00000000000101001000010011100010100000000
Step 5: Fitness: 66, String: 00000000000001001000010011100010100000000
Step 6: Fitness: 79, String: 00000000000000001000010011100010100000000
Step 7: Fitness: 92, String: 00000000000000000000100111000101000000000
Step 8: Fitness: 105, String: 00000000000000000000000011100010100000000
Step 9: Fitness: 118, String: 00000000000000000000000011000101000000000
Step 10: Fitness: 131, String: 00000000000000000000000000100010100000000
Step 11: Fitness: 144, String: 00000000000000000000000000000010100000000
Step 12: Fitness: 157, String: 00000000000000000000000000000010000000000
Step 13: Fitness: 170, String: 00000000000000000000000000000000000000000

V. Conclusions

For source code:
https://github.com/scalemailted/MachineLearning2-HW1A