Genetic Program Example - adapted from TinyGP by Moshesipper

Import libraries

In [1]:

```
from random import random, randint, seed
from statistics import mean
from copy import deepcopy
import numpy as np
```

Define parameters

We must now define our parameters, and allow for bloat control.

In [45]:

```
POP_SIZE        = 36   # population size
MIN_DEPTH       = 2    # minimal initial random tree depth
MAX_DEPTH       = 5    # maximal initial random tree depth
GENERATIONS     = 50  # maximal number of generations to run evolution
TOURNAMENT_SIZE = 5    # size of tournament for tournament selection
XO_RATE         = 0.8  # crossover rate
PROB_MUTATION   = 0.2  # per-node mutation probability
```

Defining non-terminals and terminals

Define functions for non terminal set - this can be done natively in functional programming languages.

In [3]:

```
def add(x, y): return x + y
def sub(x, y): return x - y
def sqr(x): return x * x
def cos(x): return np.cos(2 * np.pi * x)
def mul(x, y): return x * y
#def div(x,y): return x/y # Consider what issues might arrise with this function
```

Define terminal and non-terminal sets

In [4]:

```
FUNCTIONS2 = [add, sub, mul]
FUNCTIONS1 = [sqr, cos]
FUNCTIONS = [sqr, cos, add, sub, mul]
TERMINALS = ['x', 10]#, 2, 'd', 'pi']
```

Managing our dataset

In usual settings you will have a dataset which you are working from, much in the same way as in traditional ML tasks; however, for the assignment and for observing, we will consider a target function and a create a dataset from that.

In [5]:

```
def target_func(x): # evolution's target
    return (10 * 1) + (x ** 2 - 10 * np.cos(2 * np.pi * x))
```

In [6]:

```
def generate_dataset(): # generate 101 data points from target_func
    dataset = []
    for x in range(-50,50, 6):
        x = x / 10
        dataset.append([x, target_func(x)])
    return dataset
```

Creating the genetic program class

In [7]:

```
class GPTree:
    def __init__(self, data = None, left = None, right = None):
        self.data  = data
        self.left  = left
        self.right = right

    def node_label(self): # string label
        if (self.data in FUNCTIONS):
            return self.data.__name__
        else:
            return str(self.data)

    def print_tree(self, prefix = ""): # textual printout
        print("%s%s" % (prefix, self.node_label()))
        if self.data not in FUNCTIONS1:
            if self.left:  self.left.print_tree (prefix + "   ")
        if self.right: self.right.print_tree(prefix + "   ")

    def compute_tree(self, x):
        if (self.data in FUNCTIONS2):
            return self.data(self.left.compute_tree(x), self.right.compute_tree(x))
        elif (self.data in FUNCTIONS1):
            return self.data(self.right.compute_tree(x))
        elif self.data == 'x': return x
        elif self.data == 'pi': return np.pi
        elif self.data == 'd': return 1
        else: return self.data

    def random_tree(self, grow, max_depth, depth = 0): # create random tree using either grow or fu
        if depth < MIN_DEPTH or (depth < max_depth and not grow):
            self.data = FUNCTIONS[randint(0, len(FUNCTIONS)-1)]
        elif depth >= max_depth:
            self.data = TERMINALS[randint(0, len(TERMINALS)-1)]
        else: # intermediate depth, grow
            if random () > 0.5:
                self.data = TERMINALS[randint(0, len(TERMINALS)-1)]
            else:
                self.data = FUNCTIONS[randint(0, len(FUNCTIONS)-1)]
        if self.data in FUNCTIONS2:
            self.left = GPTree()
            self.left.random_tree(grow, max_depth, depth = depth + 1)
            self.right = GPTree()
            self.right.random_tree(grow, max_depth, depth = depth + 1)
        elif self.data in FUNCTIONS1:
            self.right = GPTree()
            self.right.random_tree(grow, max_depth, depth = depth + 1)

    def mutation(self):
        if random() < PROB_MUTATION: # mutate at this node
            self.random_tree(grow = True, max_depth = 2)
        elif self.left: self.left.mutation()
        elif self.right: self.right.mutation()

    def size(self): # tree size in nodes
        if self.data in TERMINALS: return 1
        l = self.left.size()  if self.left  else 0
        r = self.right.size() if self.right else 0
        return 1 + l + r
```

```
    def build_subtree(self): # count is list in order to pass "by reference"
        t = GPTree()
        t.data = self.data
        if self.left:  t.left  = self.left.build_subtree()
        if self.right: t.right = self.right.build_subtree()
        return t

    def scan_tree(self, count, second): # note: count is list, so it's passed "by reference"
        count[0] -= 1
        if count[0] <= 1:
            if not second: # return subtree rooted here
                return self.build_subtree()
            else: # glue subtree here
                self.data  = second.data
                self.left  = second.left
                self.right = second.right
        else:
            ret = None
            if self.left  and count[0] > 1: ret = self.left.scan_tree(count, second)
            if self.right and count[0] > 1: ret = self.right.scan_tree(count, second)
            return ret

    def crossover(self, other): # xo 2 trees at random nodes
        if random() < XO_RATE:
            second = other.scan_tree([randint(1, other.size())], None) # 2nd random subtree
            self.scan_tree([randint(1, self.size())], second) # 2nd subtree "glued" inside 1st tree
```

Fitness and selection

In [8]:

```
def fitness(individual, dataset): # inverse mean absolute error over dataset normalized to [0,1]
    return 1 / (1 + mean([abs(individual.compute_tree(ds[0]) - ds[1]) for ds in dataset]))
```

In the example we are using we are using tournament based fitness. What benefits and negatives does tournament selection have?

In [9]:

```
def selection(population, fitnesses): # select one individual using tournament selection
    tournament = [randint(0, len(population)-1) for i in range(TOURNAMENT_SIZE)] # select tournamen
    tournament_fitnesses = [fitnesses[tournament[i]] for i in range(TOURNAMENT_SIZE)]
    return deepcopy(population[tournament[tournament_fitnesses.index(max(tournament_fitnesses))]])
```

Try to implement a roulette wheel selection for this and compare your results.

In [10]:

```
def roulette_selection(population, fitnesses):
    pass
```

In [11]:

```
def init_population(): # ramped half-and-half
    pop = []
    for md in range(3, MAX_DEPTH + 1):
        for i in range(int(POP_SIZE/6)):
            t = GPTree()
            t.random_tree(grow = True, max_depth = md) # grow
            pop.append(t)
        for i in range(int(POP_SIZE/6)):
            t = GPTree()
            t.random_tree(grow = False, max_depth = md) # full
            pop.append(t)
    return pop
```

Main Loop

In [47]:

```
dataset = generate_dataset()
population = init_population()
best_of_run = None
best_of_run_f = 0
best_of_run_gen = 0
fitnesses = [fitness(population[i], dataset) for i in range(POP_SIZE)]

# go evolution!
for gen in range(GENERATIONS):
    nextgen_population=[]
    for i in range(POP_SIZE):
        parent1 = selection(population, fitnesses)
        parent2 = selection(population, fitnesses)
        parent1.crossover(parent2)
        parent1.mutation()
        nextgen_population.append(parent1)
    population=nextgen_population
    fitnesses = [fitness(population[i], dataset) for i in range(POP_SIZE)]
    if max(fitnesses) > best_of_run_f:
        best_of_run_f = max(fitnesses)
        best_of_run_gen = gen
        best_of_run = deepcopy(population[fitnesses.index(max(fitnesses))])
#           print("________________________")
#           print("gen:", gen, ", best_of_run_f:", round(max(fitnesses),3), ", best_of_run:")
#           best_of_run.print_tree()
    if best_of_run_f == 1: break

print("\n\n_________________________________________________\nEND OF RUN\nbest_of_run attained a
      " and has f=" + str(round(best_of_run_f,3)))
best_of_run.print_tree()
```

```
_________________________________________________
END OF RUN
best_of_run attained at gen 33 and has f=1.0
add
   mul
      sub
         mul
            sqr
               10
            sub
               10
               10
         10
      cos
         x
   add
      10
      mul
         x
         x
```

In [ ]: