MECS E4510_001_2019_3 - EVOLUTIONARY COMPUTATION&DESIGN AUTOMATION
HW2 Symbolic Regression

Chiqu Li
UNI: cl3895

Instructor: Hod Lipson

Grace Hours Used: 0
Grace Hours Remaining: 124h

October 20, 2019

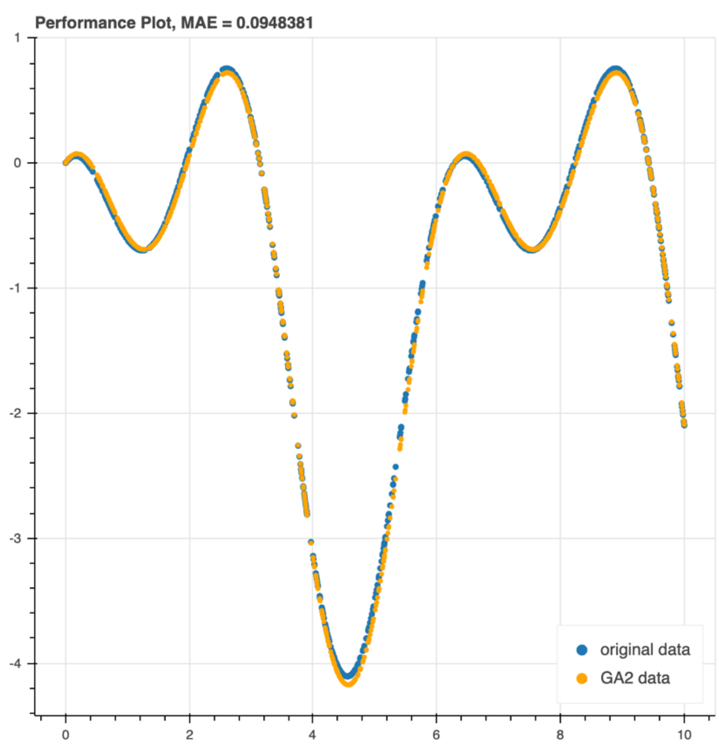
1. Result

Result Table			
Method	Evaluation	Error	Overall Efficiency
Random Search	50000	0.8837209	12.85%
Hill Climber	50000	0.6482721	38.86%
GA1	50000	0.5038362	54.81%
GA2	50000	0.0948381	99.00%
GA3	50000	0.4138948	64.75%

The function can be expressed as

$$f(x)=(4.509666579846594*\sin(0.506732941976681)+0.3250184118293763)*(0.7084635833175384+\sin(x-2.7508491966805))*\sin(x)$$

Performance Plot



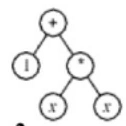
2. Methods

In homework2, we were asked to use genetic programming to get a symbolic regression, which means we are going to find a math expression that best fit 1000 points provided. Assuming that this symbolic regression only uses operators in (+, -, *, /, sin, cos), and real constant(±10), and x.

2.1 Representation

I used a heap data structure to represent the equation. Root element at position 1 and children of element I in position 2i and 2i+1.

Index	Value
1	+
2	1
3	*
4	
5	
6	x
7	x



$X^2 + 1$

2.2 Random Search

Since we have figured out a way to represent the function, for random search algorithm, what we need to do is to define a rational way to generate a new equation randomly, which must obey math regulation. The depth of the tree is under 8, so 256 positions in a heap are required to fully represent the equation.

2.3 Hill Climber

The hill climber algorithm is based on random search. The main difference is a mutation process. At first, we randomly generate a function. Then we apply the mutation process into that function. If the mutation's fitness is better than the original one, then we record it and use the new one to mutate.

2.4 Genetic Programming

I implemented 3 kinds of operators in genetic programming. The selection method, mutation and crossover method are crucial parts of my GA programming. For Each GA methods, in order to improve the diversity, I randomly add new people into population.

Selection Methods: I applied two kinds to selection methods. First one is to select parents from population randomly, and generate children 2/3 of population size. Then I pop the worst 2/3 from the larger population. The second selection method is to select better parents by Roulette Algorithm. When the fitness is too low (lower than a fixed value 25), then I dropped this gene and select another one until the fitness is acceptable. The second selection method was implemented in my GA2.

Crossover: The crossover was applied to all of my GA methods, the difference of these three methods are the crossover rate. The crossover method will randomly choose a crossover node and switch the subtrees between parents.

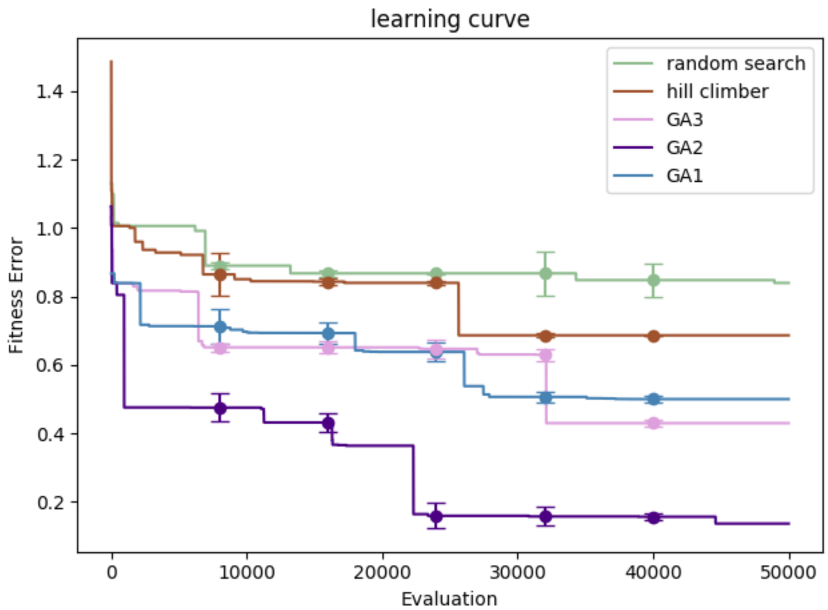
Mutation: The mutation method for GA is exactly the same as the mutation method for Hill Climber. It will select a mutation node randomly and implement mutation.

2.5 Analysis of Performance

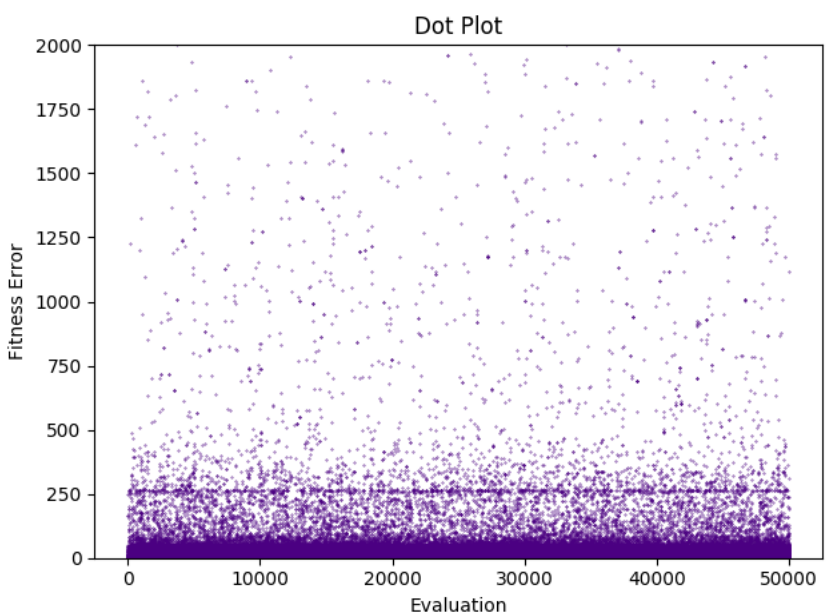
The selection and variation of the GA impact the consequence significantly. As we can see from the path, after 50000 evaluations, GA2 had an excellent performance, which shows the powerful ability of Genetic Programming. This mainly because I implemented a good selection in GA2. Overall, we can learn from the performance plot that GA is better than hill climber and random search. Even though GA runs slower, but it gets more efficiency with time goes by.

3. Performance

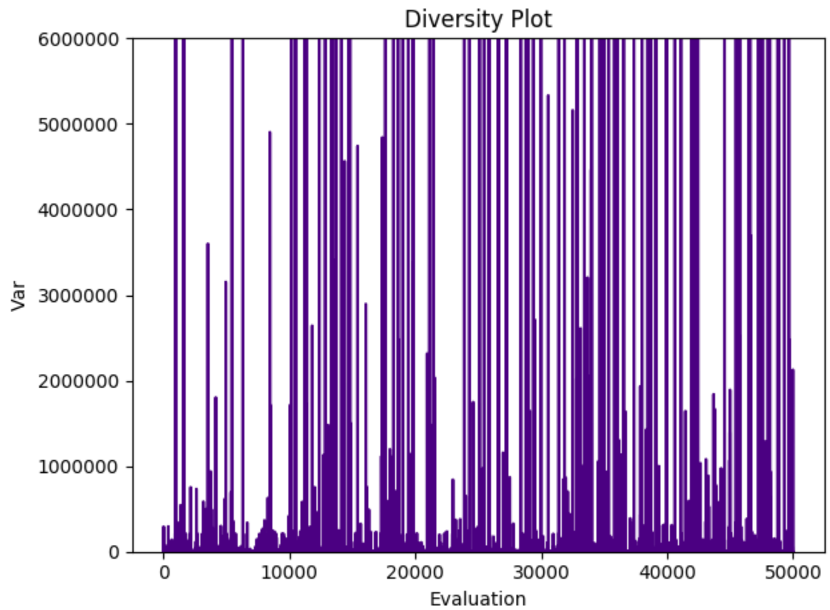
3.1 Performance Plots



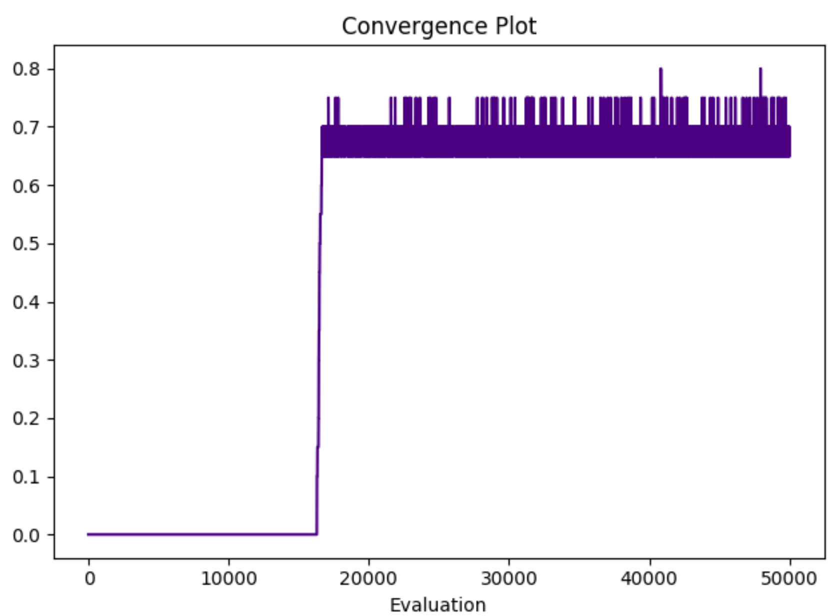
3.2 GA2 Dot Plot



3.3 GA2 Diversity Plot

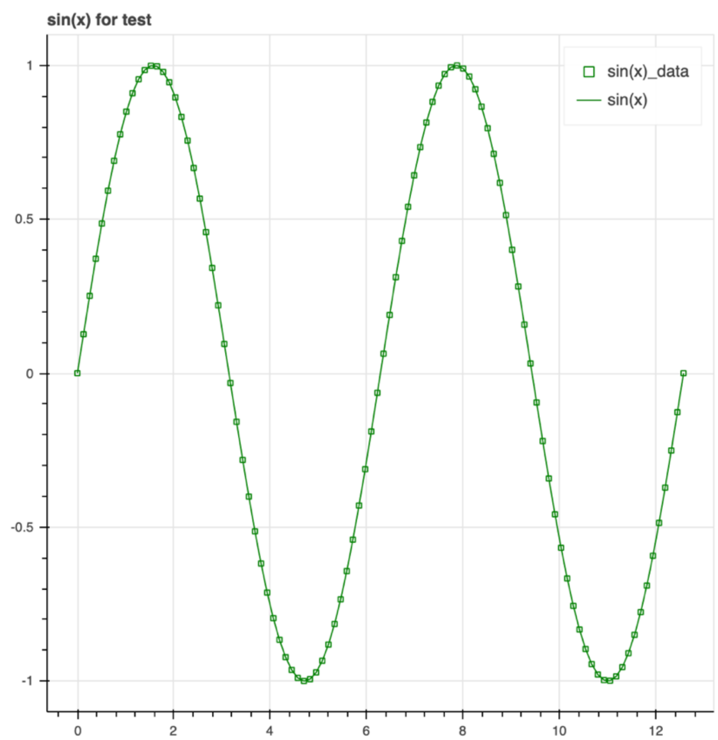


3.4 GA2 Convergence Plot



3.5 Simpler Problem Test

For a more comfortable experience of debugging, I implemented my code on a simpler problem sin(x). It is very easy for the GP to find the correct function.



4. Index

```
import numpy as np
import numexpr as ne
from random import random
from numba import jit
import math
import random
import time
import matplotlib.pyplot as plt

def cos(q):
    return math.cos(q)
def sin(q):
    return math.sin((q))
operator = ['+', '-', '*', '/', 'cos', 'sin']
generationtimes = 100
population_size = 20
group_size = 20
crossrate = 0.5
mutationrate = 0.1
selectrange = population_size/2
selectsize = int(2*population_size/3)
heapsize = 256
a = np.loadtxt('data.txt')
x0 = np.array([])
y0 = np.array([])

@jit
def generateHeap():
    heap = ['']*heapsize
    single = False
    Double = False
    for i in range(1, len(heap)):
        constant = str(constant1())
        c = ['', operator[np.random.randint(0, len(operator))], constant, 'x']
        parent = heap[i//2]
        if i == 1:
            heap[1] = c[1]
            continue
        if parent.isdigit() or parent == 'x' or ('(-' in parent):
            continue
        if parent != '':
            r1 = c[np.random.randint(1, len(c))]
            r2 = c[np.random.randint(2, len(c))]
            if Double:
                if i < 128:
                    heap[i] = r1
                else: heap[i] = r2
                Double = False
                continue
            if single:
                single = False
                continue
            if parent in ['sin', 'cos']:
                if i < 128:
                    heap[i] = r1
                else:
                    heap[i] = r2
                single = True
            if parent in ['+', '-', '*', '/']:
                if i < 128:
                    heap[i] = r1
                else:
                    heap[i] = r2
                Double = True
    return heap

def makevaluate(heap):
    heap_n = []
    for i in heap:
```

```
        heap_n.append(i)
    #print(heap_n)
    for i in range(len(heap)-1, 0, -1):
        if heap_n[i] != '':
            #print(type(heap_n[i-1]))
            if heap_n[i//2] in ['sin', 'cos']:
                heap_n[i//2] = heap_n[i//2] + '(' + heap_n[i] + ')'

            else:
                heap_n[i//2] = heap_n[i-1] + heap_n[i//2] + heap_n[i]
                heap_n[i-1] = ''
    return heap_n[1]

def constant1():
    b = random.random()
    a = random.random()
    while a == 0:
        a = random.random()
    if b < 0.5:
        return str(10 * a)
    else:
        return '(' + '{}'.format(-10 * a) + ')'


def fitness(x0, func):
    sums = 0
    for i in range(len(x0)):
        func1 = func.replace('x', str(x0[i]))
        try:
            fit_single = abs(y0[i] - eval(func1))
        except:
            fit_single = 1.5
        sums += fit_single
    return sums/len(x0)

def combinefit(heap):
    # heap_2 = mutation(heap)
    func = makevaluate(heap)
    fitn = fitness(x0, func)
    return fitn

@jit
def operator_generator(heap, i):
    op = operator[np.random.randint(0, len(operator)-2)]
    sc = operator[np.random.randint(len(operator)-2, len(operator))]
    changerate = np.random.rand()
    if heap[i] in '+-*/':
        if changerate < 0.5:
            heap[i] = op
        else:
            heap[i] = sc
            heap[i*2+1] = ''
        return heap
    if heap[i] == 'sin':
        if changerate < 0.5:
            heap[i] = 'cos'
        else:
            heap[i] = op
            heap[i*2+1] = num_genertator()
        return heap
    elif heap[i] == 'cos':
        if changerate < 0.3:
            heap[i] = 'sin'
        else:
            heap[i] = op
            heap[i*2+1] = num_genertator()
        return heap
    else:
        if i < 128:
            child1, child2 = i * 2, i * 2 + 1
            if changerate < 0.5:
                heap[i] = op
                heap[child1], heap[child2] = num_genertator(), num_genertator()
```

```
            else:
                heap[i] = sc
                heap[child1] = num_genertator()
        return heap


def num_genertator():
    rate = np.random.rand()
    if rate < 0.5:
        return constant1()
    else:
        return 'x'

@jit
def find_index(heap):
    indexmutation = []
    for i in range(1, len(heap)):
        if heap[i] != '':
            indexmutation.append(i)
    return int(indexmutation[random.randrange(len(indexmutation))])

@jit
def mutationforGA(heap1):
    heap = []
    for i in range(len(heap1)):
        heap.append(heap1[i])
    index = find_index(heap)
    change = np.random.rand()
    if '.' in heap[index]:
        b = ''
        for j in heap[index]:
            b = b + j
        if change < 0.25:
            if '(' in heap[index]:
                heap[index] = '(' + str(float(b[1:-1]) * 1.1) + ')'
            else: heap[index] = str(float(heap[index])* 1.1)
        elif change < 0.5:
            if '(' in heap[index]:
                heap[index] = str(float(b[1:-1]) / 1.1)
            else: heap[index] = str(float(heap[index])/ 1.1)
        elif change < 0.75:
            heap[index] = 'x'
        else:
            if index < 128:
                heap = operator_generator(heap, index)
    elif heap[index] == 'x':
        if change < 0.2:
            heap[index] = str(constant1())
        elif change < 0.5:
            heap = operator_generator(heap, index)
    elif heap[index] in operator:
        if change < 0.5:
            if index < 128:
                heap = operator_generator(heap, index)
        if change < 0.75:
            heap[index] = constant1()
        else:
            heap[index] = 'x'
    for i in range(2, len(heap)):
        if heap[i//2] == '' or heap[i//2] not in operator:
            if i != 1:
                heap[i] = ''
    # print('mutation')
    return heap
def modify_none(heap):
    for i in range(2, len(heap)):
        if heap[i//2] == '' or heap[i//2] not in operator:
            heap[i] = ''
    return heap

def find_spring_index(heap, index):
    index_list = [index]
    for i in range(len(heap)):
```

```
        if heap[i] != '':
            if i // 2 in index_list:
                index_list.append(i)
    return index_list
@jit
def switch(p1_switch, p_clist):
    for s in range(1, len(p_clist)):
        if p_clist[s] / p_clist[s-1] == 2:
            p1_switch.append(p1_switch[-1]*2)
        elif p_clist[s]-1 == p_clist[s-1]:
            p1_switch.append(p1_switch[-1]+1)
    return p1_switch

def init_pop(population_size):
    poplist = []
    for i in range(population_size):
        poplist.append(generateHeap())
    return poplist

@jit
def crossover(p1, p2):
    n = heapsize

    p1_switch_1 = [0]
    p2_switch_2 = [0]
    p2_clist = [0]
    while p1_switch_1[-1]+n > heapsize or p2_switch_2[-1]-n > heapsize:
        index1 = find_index(p1)
        index2 = find_index(p2)
        p1_clist = find_spring_index(p1, index1)
        p2_clist = find_spring_index(p2, index2)
        n = index2 - index1
        mid = []
        for i in p1:
            mid.append(i)
        p1_switch = [p1_clist[0]]
        p1_switch_1 = switch(p1_switch, p2_clist)
    for j, char in enumerate(p1_switch_1):
        p1[char] = p2[p2_clist[j]]
    p1 = modify_none(p1)
    return p1


def select1(population):
    flag = np.random.rand()
    individual = population[int(flag * population_size)]
    return individual
@jit
def select2(population, sumfitness):
    flag=np.random.rand()
    flag = flag * sumfitness
    for individual in population:
        flag = flag - combinefit(individual)
        if flag <= 25:
            return individual




def fitnessall(population):
    best_individual=population[0]
    best_fitness=combinefit(best_individual)
    fitness_list=[0]*population_size
    sumfitness=0
    best_indlist = [0]*(selectsize)
    for i, individual in enumerate(population):
        fitness_list[i] = combinefit(individual)
        if fitness_list[i]<best_fitness:
            best_individual=individual
            best_fitness=fitness_list[i]
        sumfitness+=fitness_list[i]
    best_indlist1 = sorted(population, key=combinefit)
    for i in range(selectsize):
```

```python
        best_indlist[i] = best_indlist1[i]
    return sumfitness,best_individual,fitness_list,best_fitness, best_indlist


def generation(population, best_inlist, sumfitness):
    new_population=[0]*len(best_inlist)
    new_population[0] = best_inlist[0]
    for i in range(0, selectsize):
        new_population[i] = best_inlist[i]
    while len(new_population) < population_size:
        new_population.append(born(population, sumfitness))
    return new_population


def born(population, sumfitness):
    dad=select2(population, sumfitness)
    mom=select2(population, sumfitness)
    rate = random.uniform(0, 1)
    if rate < crossrate:
        child = crossover(dad, mom)
    if rate < mutationrate:
        child = mutationforGA(dad)
    else:
        child = generateHeap()
    return child

def run_GA(i):
    population = init_pop(population_size)
    shortest_fitness = []
    over_fitness = []
    while i>0:
        sumfitness, best_individual, fitness_list,best_fitness, b = fitnessall(population)
        print("dis:%f,  generation:%d"%(best_fitness,generationtimes-i))
        shortest_fitness.append(best_fitness)
        over_fitness.append(fitness_list)
        population=generation(population, b, sumfitness)
        print('y=', makevaluate(best_individual))
        i-=1
    np.savetxt('GA2_best_fitness',shortest_fitness)
    np.savetxt('GA2_overall_fitness', over_fitness)
    np.savetxt('GA2_bestfunction', makevaluate(best_individual), fmt='%s')

def run_hillclimber(times, a):
    heap = [''] * 256
    heap_1 = generateHeap(heap)
    goodfit = 100000
    funcgood = ''
    shortest_fitness = []
    for i in range(times):
        heap_2 = mutation(heap_1)
        func = makevaluate(heap_2)
        fitn = fitness(x0, func)
        if fitn < goodfit:
            goodfit = fitn
            funcgood= func
            heap_1 = heap_2
        shortest_fitness.append(goodfit)
        print('good:', goodfit)
        print('funcgood=', funcgood)
        print('times', i)
    np.savetxt('hc_best_fitness',shortest_fitness)
    #np.savetxt('hc_overall_fitness', over_fitness)
    np.savetxt('hc_bestfunction', funcgood, fmt='%s')
    return funcgood, fitn

def run_Random(times, x0):
    funcgood = ''
    goodfit = 100000
    shortest_fitness = []
    for i in range(times):
        heap = [''] * 256
        heap_new = generateHeap(heap)
        func = makevaluate(heap_new)
```

```
        #print(func)
        fitn = fitness(x0, func)
        if fitn < goodfit:
            goodfit = fitn
            print(goodfit)
            funcgood= func
        shortest_fitness.append(goodfit)
        print('fg', funcgood)
        print('fitnnnnnnnnn', goodfit)
        print('times=', i)
    np.savetxt('rs_goodfitness.txt', shortest_fitness)
    return funcgood, fitn

run_Random(times, x0)
run_hillcilmber(times, a)
run_GA(generationtimes)
```