Eric Mouledoux

I.0: Requirements Documentation

I.1: Description of the Problem

**Name:** Cloth Physics

**Problem Statement:** Implement spring physic to simulate realistic cloth simulation.

**Problem Specification:** The application should simulate cloth using springs and physical forces such as gravity and wind.

I.2: Input Information

Description: Runtime interaction with the cloth

Type: N/A

Range of Acceptable Values: Mouse position and mouse buttons

I.3: Output Information

The program will simulate cloth based on input given before, and during runtime of the appilication.

I.4: User Interface Information

N/A

II.0: Design Documentation

II.1: System Architecture Description

The program is split into 4 different classes for orgranization. The main entry point into the program is started in Driver.py. This will read , from a file (named Expression.txt,) strings in Conjunctive Normal Form and create CNF objects from them. CNF.py is responsible for turning the strings that it receives into an object that is useable by the algorithm. Once a CNF object is created Driver.py will send that object to Genetic.py where the algorithm will begin. Genetic.py will execute the following algorithm.

procedure genetic algorithm;

begin

set time t:= 0;

initialize the population P(t);

while the termination condition is not met do

begin

1)evaluate fitness of each member of the population P(t);

2)select members from population P(t) based on fitness;

3)produce the offspring of these pairs using genetic operators;

4)replace, based on fitness, candidates of P(t), with these offspring;

5)set time t := t + 1

end

end.

The initialization of the population will utilize Chromosome.py to create Chromosomes that can be evaluated in the algorithm.

===Please Read====

Notable design decisions include the following.

1)The initial population is the number of variables divided by 2. If the number of variables is less than 2 then the effective population will be set to 4. This is to ensure that the new population will have a large enough pool to choose random parents that will produce offspring

2)Fitness calculations are done inside the chromosome class.

3)When creating offspring, the random selection has already been chosen so the loop will pull out sets of two and perform crossover on those.

4)When crossover is performed, the mutate will occur if the two children are the same every third generation. This is to accommodate for plateau problems when producing new generations.

5) Individual testing of the functions found in Genetic.py can be found at the bottom of the file.

II.2: Information about the functions

**File: Chromosome.py**

Class: Chromosome

Data Members

Name: id

Description: identifier for the chromosome

Name: generation

Description: which iteration of the loop is this chromosome part of

Name: cnf

Description: the expression this Chromosome will evaluate to

Name: genes

Description: the variable representation Ex: 10001

Name: geneCount

Description: the length of the genes

Name: fitness

Description: number of correct disjuncts that this chromosome matches

Name: literals

Description: dictionary of variables to the genes.

Name: geneClauses

Description: clauses associated with this chromosome. Ex:(1\*1)

Name: Elite

Description: Is this Chromosome an elite. This was used in previous implementation whenever we cared about the state of this object

Name: fitness\_ratio

Description: value between 1-100 that gives a weight of correctness vs all other genes

Name: minRange

Description: the lower bound of the fitness ratio

Name:max Range

Description: the upper bound of the fitness ratio

Function: display

Parameters: self

Description: display relevant information for this object

Return: n/a

Function: setGenes

Parameters: string genes

Description: takes in strings for manual assignment or performs random assignment based on the variable length.

Return: n/a

Function: mapGenes

Parameters: self

Description: Maps the genes to the variables. Appending to the literals dictionary. Sorts the resulting list by the variable. This is only for organized output information

Return: n/a

Function: mapVars

Parameters: Clause Ex: (a \* b)

Description: Takes a clause and replaces the appropriate variable with it.

Return: list with variables replaced Ex: (1 \* 0)

Function: evalFitness

Parameters: self

Description: evaluates the clauses of this gene using bitwise operations

Return: n/a

Function: set\_fitness\_ratio

Parameters: tf

Description: tf is the total fitness of the population this chromosome is associated with. divide the fitness value by the total fitness to get the weight of this chromosome

Return: n/a

Function: set\_fitness\_range

Parameters: chrom

Description: chrom is a reference to the previous chromosome to calculate the minimum and maximum bounds for the current chromosome. It is initially set to 0 then set to the current member as it evaluates. This is to ensure the correct ranges are calculated

Return: n/a

Function: displayGenes

Parameters: self

Description: displays only gene information

Return: n/a

**File: CNF.py**

Class: CNF

Data Members:

Name: expression

Description: The literal expression this CNF is associated with. Ex: "(a \* b) + (c \* d)"

Name: clauses

Description: list of the clauses Ex: [(a\*b), (c\*d)]

Name: valid

Description: valid variables this CNF can use

Name: varcount

Description: the number of total unique variables

Name: variables

Description: unique variables for this specific CNF

Function: setClauses

Parameters: self

Description: checks for valid cnf clause by incrementing the current charactesr until it finds a "+". Once it matches to a "+" matches the next char for ")" . After passing this check it appends the disjunct to the clause list. It does not care to append the "+" b/c these are irrelevant due to the nature of cnf. Updates the number of clauses. Then converts them to the bitwise string

Return: n/a

Function: convert

Parameters: c

Description: c is a clause of the form (a\*b). Strips spaces and the "(" ")".It replaces the following: ! to ~ + to & and \* to |.

Return: the string with the values replaced

Function: setVariables

Parameters: self

Description: set the variables and number of variables in the cnf

Return: n/a

Function: display

Parameters: n/a

Description: prints relevant information for this cnf

Return: n/a

**File: Driver.py**

Function: main

Parameters: n/a

Description: takes in from file expressions in cnf. strips white space and newline characters. It then performs the genetic algorithm on all expressions.

Return: n/a

Function: LoadPresets

Parameters: file

Description: file is the name of the file to be loaded. If no file is found it will load the default cnf "(a \* b) + (c \* d)

Return: the string representation of the cnf

***File: Genetic.py***

Function: run

Parameters: cnf, debug

Description: cnf is the string representation of the cnf to be solved, debug is a boolean that will enable all print statements if true. Performs the algorithm as described in the system architecture.

Return: the number of generations it took to solve the expression

Function: generate\_population

Parameters: id gen cnf debug

Description: id is the unique identifier for this chromosome, gen is the generation this chromosome was created, cnf is the expression to be solved, debug is the boolean for showing print statements. Generates a population based on the number of variables /2.

Return: list of chromosomes

Function: calc\_fitnesses

Parameters: pop is the list of chromosomes to be evaluated for fitness

Description: evaluates population based on number of correct disjuncts with respect to the expression.

Return: sorted chromosome population by fitness values

Function: get\_selection

Parameters: fit cnf

Description: fit is the list of chromosomes after their fitness values have been calculated, cnf is the current expression. Randomly selects members of the population based on their contribution to the overall fitness. This is to ensure more randomness and giving chromosomes with a low initial fitness ratio to be selected

Return: list of selected chromosomes. the size of this list is the same size of the initial population

Function: create\_offspring

Parameters: selection, t, debug

Description: selection is the randomly selected list, t is the generation count, debug is for printing.

Return: list of offspring from resulting crossover and mutation

Function: crossover

Parameters: t, dad, mom, randNum, debug

Description: t is the generation, dad is the chromosome that will crossover with mom, randNum is for the random crossover point, debug is for printing. Generate a random number then use that number as a crossover point to perform crossover. for example: random = 2 dad = 0011 mom = 1100 the resulting children will be 0000 and 1111.

Return: list of two chromosomes from resulting crossover operations

Function: replace\_population

Parameters: offspring

Description: sorts the offspring by fitness value. This can be taken out but was left in for program integrity

Return: sorted offspring by fitness values

Function: mutate

Parameters: child debug

Description: selects a random point in a chromosomes gene and flips it.

Return: gene with bit flipped

Function: test\_crossover

Parameters: n/a

Description: solo function to test the crossover functionality

Return: n/a

Function: test\_offspring

Parameters: n/a

Description: solo function to test the offspring functionality

Return: n/a

Function: test\_cnf

Parameters: n/a

Description: solo function to test the cnf functionality

Return: n/a

Function: test\_chromosome

Parameters: n/a

Description: solo function to test the chromosome functionality

Return: n/a

Function: test\_replace

Parameters: n/a

Description: solo function to test the replace functionality

Return: n/a

III.0: Implementation Documentation

III.1 Program Code

**#File: Driver.py**

import os

import Chromosome

import Genetic

import CNF

from Genetic import \*

from Chromosome import \*

from CNF import \*

#Function: main

#Parameters: n/a

#Description: takes in from file expressions in cnf. strips white space and newline characters. It then performs the genetic algorithm on all expressions.

#Return: n/a

def main():

tmp = []

file = 'Expressions.txt'

presets = LoadPresets(file)

for p in presets:

p = p.replace(" ", "")

p = p.replace("\n", "")

tmp.append(p)

presets = tmp

all = True

#testCnf = CNF(presets[4])

#gens = Genetic.run(testCnf)

if(all):

for i in presets:

#print(i)

cnf = CNF(i)

gens = Genetic.run(cnf, False)

else:

#print("testing bit operaations 1 & 0 =", 1 & 0)

#print("testing bit operations 1 | 1 = ", 1 | 0)

gens = Genetic.run(testCnf)

while True:

if(gens > 1):

return

else:

gens = Genetic.run(testCnf)

#Function: LoadPresets

#Parameters: file

#Description: file is the name of the file to be loaded. If no file is found it will load the default cnf "(a \* b) + (c \* d)

#Return: the string representation of the cnf

def LoadPresets(file):

inFile = open(file, 'r')

conjunct = []

if(inFile):

for i in inFile:

conjunct.append(i)

else:

preset = "(a \* b) + (c \* d)"

v = ""

print("Loading presets")

for p in preset:

disjunct += p

v += p

if(p == ')' or p == '\*'):

conjunct.append(disjunct)

disjunct = ''

return conjunct

'''====================================='''

main()

#main(input("1 for preset, 2 for user specified: "))

**#File: CNF.py**

class CNF:

def \_\_init\_\_(self, exp):

#Name: expression

#Description: The literal expression this CNF is associated with. Ex: "(a \* b) + (c \* d)"

self.expression = exp

#Name: clauses

#Description: list of the clauses Ex: [(a\*b), (c\*d)]

self.clauses = []

#Name: valid

#Description: valid variables this CNF can use

self.valid = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

#Name: varcount

#Description: the number of total unique variables

self.varcount = 0

#Name: variables

#Description: unique variables for this specific CNF

self.variables = ""

self.numClauses = len(self.clauses)

self.setClauses()

#set the variables and number of variables in the cnf

self.setVariables()

#Function: setClauses

#Parameters: self

#Description: checks for valid cnf clause by incrementing the current charactesr until it finds a "+". Once #it matches to a "+" matches the next char for ")" . After passing this check it appends the disjunct to the #clause list. It does not care to append the "+" b/c these are irrelevant due to the nature of cnf. Updates #the number of clauses. Then converts them to the bitwise string

#set the clauses for this gene

#the clauses will come from the cnf and then

#converted into a format that is compatible with bit operations

#ex: chromosome.clause = a & b

#Clauses: ['~a&b', 'c&d', 'e&~f', 'g&h']

#Return: n/a

def setClauses(self):

tmp = ""

clauses = []

for c in self.expression:

if(c != '+'):

tmp += c

if(c == ')'):

clauses.append(tmp)

tmp = ""

self.clauses = self.convert(clauses)

self.numClauses = len(self.clauses)

#Function: convert

#Parameters: c

#Description: c is a clause of the form (a\*b). Strips spaces and the "(" ")".It replaces the following: ! to ~ #+ to & and \* to |.

#Return: the string with the values replaced

#convert a cnf string to it's literal form

#ex: "(!a + b) \* (c + d) = (~a & b) | (c & d)

def convert(self, c):

for i in range(len(c)):

c[i] = c[i].replace(" ", "")

c[i] = c[i].replace("(", "")

c[i] = c[i].replace(")", "")

c[i] = c[i].replace("!", "~")

c[i] = c[i].replace("+", "&")

c[i] = c[i].replace("\*", "|")

return c

#Function: setVariables

#Parameters: self

#Description: set the variables and number of variables in the cnf

#Return: n/a

def setVariables(self):

for i in self.expression:

if(i in self.valid and i not in self.variables):

self.variables += i

self.varcount = len(self.variables)

#Function: display

#Parameters: n/a

#Description: prints relevant information for this cnf

#Return: n/a

def display(self):

print("===== CNF =====")

print("CNF:", self.expression)

print("Clauses:", self.clauses)

print("Variables:", self.variables)

print("=========================\n")

**#File: Chromosome.py**

import random

import operator

import math

class Chromosome:

def \_\_init\_\_(self, id, generation, cnf, elite, genes):

#Name: id

#Description: identifier for the chromosome

self.id = id

#Name: generation

#Description: which iteration of the loop is this chromosome part of

self.generation = generation

#Name: cnf

#Description: the expression this Chromosome will evaluate to

self.cnf = cnf

#Name: genes

#Description: the variable representation Ex: 10001

self.genes = genes

#Name: geneCount

#Description: the length of the genes

self.geneCount = len(self.cnf.variables)

#Name: fitness

#Description: number of correct disjuncts that this chromosome matches

self.fitness = 0

#Name: literals

#Description: dictionary of variables to the genes. #literals dictionary a:1 b:0 c:1

self.literals = {}

#Name: geneClauses

#Description: clauses associated with this chromosome. Ex:(1\*1)

self.geneClauses = []

#Name: Elite

#Description: Is this Chromosome an elite. This was used in previous implementation whenever we cared about the state of this object

self.Elite = elite

#Name: fitness\_ratio

#Description: value between 1-100 that gives a weight of correctness vs all other genes

self.fitness\_ratio = 0

#Name: minRange

#Description: the lower bound of the fitness ratio

self.minRange = 0

#Name:max Range

#Description: the upper bound of the fitness ratio

self.maxRange = 0

#begin the setup for this objects genes

self.setGenes(genes)

#Function: display

#Parameters: self

#Description: display relevant information for this object

#Return: n/a

def display(self):

print("===== Chromosome", self.id, "=====")

print("Generation:", self.generation)

print("CNF:", self.cnf.expression)

print("Clauses:", self.cnf.clauses)

print("Variables:", self.cnf.variables)

print("Genes:", self.genes)

print("Gene Count:", self.geneCount)

#print("Gene Literals:", self.literals)

print("Gene Clauses:", self.geneClauses)

print("Fitness:",self.fitness)

print("Fitness Ratio:",self.fitness\_ratio)

#print("Elite: " , self.Elite)

print("=========================\n")

#Function: setGenes

#Parameters: string genes

#Description: takes in strings for manual assignment or performs random assignment based on the variable length.

#Return: n/a

def setGenes(self, genes):

if(genes != ""):

if(len(genes) != len(self.cnf.variables)):

print("Bit length to variables problem for", self.id)

return

else:

self.genes = genes

self.mapGenes()

self.evalFitness()

else:

#random binary assignment

for i in range(0, self.geneCount):

tmp = str(random.randint(0,1))

self.genes += tmp

self.mapGenes()

#Function: mapGenes

#Parameters: self

#Description: Maps the genes to the variables. Appending to the literals dictionary. Sorts the resulting list by the variable. This is only for organized output information

#Return: n/a

def mapGenes(self):

#create a dictionary of the variables with the key of the gene

for i,k in zip(self.cnf.variables, self.genes):

self.literals.update([(i,k)])

#map the variables of the clause to the gene

for clause in self.cnf.clauses:

self.geneClauses.append(self.mapVars(clause))

#sort the literals dictionary by the variable

self.literals = sorted(self.literals.items(), key=operator.itemgetter(0))

#Function: mapVars

#Parameters: Clause Ex: (a \* b)

#Description: Takes a clause and replaces the appropriate variable with it.

#Return: list with variables replaced Ex: (1 \* 0)

def mapVars(self, clause):

tmpClause = clause

for c in clause: #for each character in the clause

if(c in self.cnf.variables):#check if its in variables

tmpClause = tmpClause.replace(c, self.literals.get(c))

return tmpClause

#Function: evalFitness

#Parameters: self

#Description: evaluates the clauses of this gene using bitwise operations

#Return: n/a

#~ Not ^ XOR | Or & And

#evaluate the fitness and clear elite flags

def evalFitness(self):

self.Elite = False

self.fitness = 0

for i in self.geneClauses:

if(eval(i)):

self.fitness +=1

#Function: set\_fitness\_ratio

#Parameters: tf

#Description: tf is the total fitness of the population this chromosome is associated with. divide the fitness value by the total fitness to get the weight of this chromosome

#Return: n/a

def set\_fitness\_ratio(self, tf):

self.fitness\_ratio = round(self.fitness / tf, 3)

#Function: set\_fitness\_range

#Parameters: chrom

#Description: chrom is a reference to the previous chromosome to calculate the minimum and maximum bounds for the current chromosome. It is initially set to 0 then set to the current member as it evaluates. This is to ensure the correct ranges are calculated

#Return: n/a

def set\_fitness\_range(self, chrom):

self.minRange = round(chrom, 2)

self.maxRange = round(self.fitness\_ratio + chrom, 3)

if(self.maxRange >= .99):

self.maxRange = math.ceil(self.maxRange)

#Function: displayGenes

#Parameters: self

#Description: displays only gene information

#Return: n/a

def displayGenes(self):

print("======ID:", self.id,"=====Genes:", self.genes,"Fitness:", self.fitness, "nFitness Range:",self.minRange,"->",self.maxRange)

**#File: Genetic.py**

import Chromosome

import operator

import CNF

from Chromosome import \*

from CNF import \*

'''

=======================================================================

Encoding table Class Hair Color Agenda Fears

===================================================================

00 Rogue Black Gold Prison

01 Warrior Brown Fame Spiders

10 Mage Grey Knowledge Idiocy

11 Wildcard Blonde Evil Noobs

===================================================================

'''

'''

Get Input:

Ex: Enter an expression to solve.

Ex: (!a ^ b) v (c ^ d) v (e ^ !f) v (g ^ h)

Solution: 01 11 10 00 (Warrior, Blonde, Knowledge, Noobs)

a = 0 c = 1 e = 1 g = 1

b = 1 d = 1 f = 0 h = 1

Get the clauses.

Ex: cnf = (!a + b) (c + d) (e + !f) (g + h)

Get the variable count.

Ex: 8

Initialize chromosomes based on the variable count.

Chromosome length based on number of variables.

Number of chromosomes is arbitrary.

Ex: C1.gene = 10 11 11 00;

C2.gene = 11 00 00 00;

C3.gene = 11 11 00 01;

C4.gene = 00 01 10 10;

Evaluate fitness by summing the number of true clauses in each chromosome

Ex: C1.fitness = f t f f >>1

C2.fitness = f f f f >>0

C3.fitness = f t f f >>1

C4.fitness = f f t f >>1

Select parents based on fitness.

Ex: select(C1) and select(C3)

Perform Crossover on C1 and C3.

Ex: Breed(C1, C3)

===Genetic Algorithm===

time t = 0

initialize population P(t)

repeat

evaluate each member of P(t)

select member of P(t) based on fitness

produce offspring using genetic operators

replace, based on fitness, members of P(t) with these

t+=1

until solved

=== ===

'''

#Function: run

#Parameters: cnf, debug

#Description: cnf is the string representation of the cnf to be solved, debug is a boolean that will enable all print statements if true. Performs the algorithm as described in the system architecture.

#Return: the number of generations it took to solve the expression

def run(cnf, debug):

t = 0

f = 0

population = generate\_population("", t, cnf, debug)

population = sorted(population,key = lambda p: p.fitness, reverse = True)

if(debug):

print("Solving for", cnf.expression)

totalFitness = 0

while True:

fitnesses = calc\_fitnesses(population)

selection = get\_selection(fitnesses, cnf)

#for i in population:

#i.displayGenes()

offspring = create\_offspring(selection, t, debug)

population = replace\_population(offspring)

fitnessGoal = population[0].fitness

if(debug):

print("Fitness Goal:", cnf.numClauses, "@Generation", t)

if(fitnessGoal == cnf.numClauses):

winner = population[0]

print("====Solved=====\n",cnf.expression,"\n@ GENERATION", t,"with",winner.genes)

break;

t += 1

if(t >= 1000):

break

#restart loop

return t

#Function: generate\_population

#Parameters: id gen cnf debug

#Description: id is the unique identifier for this chromosome, gen is the generation this chromosome was created, cnf is the expression to be solved, debug is the boolean for showing print statements. Generates a population based on the number of variables /2.

#Return: list of chromosomes

#generate a population of Chromosomes

#returns a list of Chromosomes

def generate\_population(id, gen, cnf, debug):

pop = []

genCount = int(len(cnf.variables) / 2 )

if (genCount <= 2):

if(debug):

print("Initial population too low... doubling it")

genCount = 4

for i in range(0, genCount):

c = Chromosome(str(id) + str(i), gen, cnf, False, "")

pop.append(c)

return pop

#Function: calc\_fitnesses

#Parameters: pop is the list of chromosomes to be evaluated for fitness

#Description: evaluates population based on number of correct disjuncts with respect to the expression.

#Return: sorted chromosome population by fitness values

#calculates the strongest of the population based on fitness

#and returns the most fit

def calc\_fitnesses(pop):

tf = 0

count = 0

for i in pop:

i.id = str(count)

i.evalFitness()

tf += i.fitness

count += 1

prev = 0

pop = sorted(pop, key = lambda p: p.fitness, reverse = True)

for i in pop:

i.set\_fitness\_ratio(tf)

i.set\_fitness\_range(prev)

prev = i.maxRange

return pop

#Function: get\_selection

#Parameters: fit cnf

#Description: fit is the list of chromosomes after their fitness values have been calculated, cnf is the current expression. Randomly selects members of the population based on their contribution to the overall fitness. This is to ensure more randomness and giving chromosomes with a low initial fitness ratio to be selected

#Return: list of selected chromosomes. the size of this list is the same size of the initial population

#returns a list of the parents

def get\_selection(fit, cnf):

sel = []

new\_gen\_size = cnf.numClauses

#clamp size to 4

if(new\_gen\_size <= 2):

new\_gen\_size = 4

for loop in range(0, new\_gen\_size):

rNum = random.randint(0, 100)

rNum /= 100

#print("Random Selection",loop,"is", rNum)

for i in fit:

if (rNum >= i.minRange and rNum <= i.maxRange):

sel.append(i)

#print("selecting", i.id)

break

return sel

#Function: create\_offspring

#Parameters: selection, t, debug

#Description: selection is the randomly selected list, t is the generation count, debug is for printing.

#Return: list of offspring from resulting crossover and mutation

#takes the random selected population and generates offspring from them

def create\_offspring(selection, t, debug):

children = []

subchildren = []

upperBound = len(selection)

if(debug):

print("===Selection===")

for i in selection:

if(debug):

i.displayGenes()

if(len(selection) % 2 == 1):

upperBound = len(selection) - 1

else:

upperBound = len(selection)

for i in range(0, upperBound, 2):

first = i

second = i + 1

p1 = selection[first]

p2 = selection[second]

parents = [(p1), (p2)]

if(debug):

print("====Parents====")

for i in parents:

if(debug):

i.displayGenes()

geneLength = len(p1.genes)

crossoverPoint = random.randint(0, geneLength)

if(debug):

print("crossover @", crossoverPoint)

subchildren = crossover(t, p1, p2, crossoverPoint, debug)

for i in subchildren:

if(debug):

i.displayGenes()

children.append(i)

subchildren = []

if(debug):

print("====SELECTION====")

for i in selection:

i.displayGenes()

print("====CHILDREN====")

for i in children:

i.displayGenes()

return children

#Function: crossover

#Parameters: t, dad, mom, randNum, debug

#Description: t is the generation, dad is the chromosome that will crossover with mom, randNum is for the random crossover point, debug is for printing. Generate a random number then use that number as a crossover point to perform crossover. for example: random = 2 dad = 0011 mom = 1100 the resulting children will be 0000 and 1111.

#Return: list of two chromosomes from resulting crossover operations

#does a crossover on two Chromosomes

#return a list of two Chromosomes containing the result of crossover

def crossover(t, dad, mom, randNum, debug):

dadPt1, dadPt2, momPt1, momPt2 = ("",) \*4

dadGenes = dad.genes

momGenes = mom.genes

for i in range(0, randNum):

dadPt1 += dadGenes[i]

momPt1 += momGenes[i]

for i in range(randNum, len(dadGenes)):

dadPt2 += dadGenes[i]

momPt2 += momGenes[i]

child1 = Chromosome("child1", t, dad.cnf, False, dadPt1 + momPt2)

child2 = Chromosome("child2", t, dad.cnf, False, momPt1 + dadPt2)

if(child1.genes == child2.genes and (t % 3) == 0):

child1.genes = mutate(child1, debug)

children = [(child1), (child2)]

for i in children:

i.evalFitness()

return children

#Function: replace\_population

#Parameters: offspring

#Description: sorts the offspring by fitness value. This can be taken out but was left in for program integrity

#Return: sorted offspring by fitness values

def replace\_population(offspring):

#return sorted list based on fitness

#the first element of this list will be checked by fitness

#if it is the num clauses we have an answer

return sorted(offspring, key = lambda p: p.fitness, reverse = True)

#Function: mutate

#Parameters: child debug

#Description: selects a random point in a chromosomes gene and flips it.

#Return: gene with bit flipped

def mutate(child, debug):

genes = list(child.genes)

mutatePoint = random.randint(0, len(child.genes) - 1)

if(debug):

print("mutate point = ", mutatePoint, " length of genes =", len(child.genes))

if(genes[mutatePoint] == "1"):

genes[mutatePoint] = "0"

else:

genes[mutatePoint] = "1"

return "".join(genes)

#Function: test\_crossover

#Parameters: n/a

#Description: solo function to test the crossover functionality

#Return: n/a

def test\_crossover():

cnf = CNF("(a \* b) + (c \* d)")

dad = Chromosome(1, 0, cnf, True, "")

mom = Chromosome(2, 0, cnf, True, "")

parents = [(dad), (mom)]

children = crossover(0,dad, mom, 2, debug)

for i in parents:

print(i.genes)

for i in children:

print(i.genes)

#Function: test\_offspring

#Parameters: n/a

#Description: solo function to test the offspring functionality

#Return: n/a

def test\_offspring():

cnf1 = CNF("(a \* b) + (c \* d) + (e \* f)")

dad = Chromosome(1, 0, "(a \* b) + (c \* d) + (e \* f)", True, "")

mom = Chromosome(2, 0, "(a \* b) + (c \* d) + (e \* f)", True, "")

parents = [(dad), (mom)]

offspring = create\_offspring(parents)

#Function: test\_cnf

#Parameters: n/a

#Description: solo function to test the cnf functionality

#Return: n/a

def test\_cnf():

cnf = CNF("(a \* b) + (c \* d) + (e \* f)")

cnf.display()

#Function: test\_chromosome

#Parameters: n/a

#Description: solo function to test the chromosome functionality

#Return: n/a

def test\_chromosome():

cnf = CNF("(a \* b) + (c \* d) + (e \* f)")

chro = Chromosome(1, 0, cnf, 0, "")

chro.display()

#Function: test\_replace

#Parameters: n/a

#Description: solo function to test the replace functionality

#Return: n/a

def test\_replace():

cnf = CNF("(a \* b) + (c \* d) + (e \* f)")

#cnf = CNF("(!a \* b) + (c \* d) + (e \* !f) + (g \* h) + (i \* j) + (k \* l) + (m \* n) + (o \* p) + (q \* r) + (s \* t) + (u \* v) + (w \* x) + (y \* z)")

pop = generate\_population("initial ", 0, cnf)

#for i in pop:

#i.display()

#manual parent set

pop[0].setElite()

pop[1].setElite()

#manual add

child1 = Chromosome("child " + str(1), 0,cnf, False, '101111')

child2 = Chromosome("child " + str(2), 0,cnf, False, '000001')

children = [(child1), (child2)]

#for i in children:

#i.display()

replace\_population(pop, children)

#test\_crossover()

#test\_offspring()

#end test crossover

#test\_cnf()

#test\_chromosome()

#test\_replace()

IV.0: Verification and Validation Documentation

IV.1: Test Plan

Test plan includes the following cnf expressions included in the Expressions.txt file

**Tests for functionality**

**Test case:** Simple functionality

**Results:** All pass

(!a \* c) + (!a\*c\*!e)+(!b\*c\*d\*!e)+(a\*!b\*c)+(!e\*f) result:

(a \* b) + (c \* d) + (e \* f) + (g \* h)

(!a \* b) + (c \* d) + (e \* !f) + (g \* h)

(r \* b) + (c \* d) + (c \* a \* !b \* b \* c \* z)

(a \* b \* c) + (c \* d \* e)

(f \* g \* k) + (n \* m)

(q \* z \* l \* v) + (z \* k \* r) + (m \* n \* f) + (b \* r \* t)

(!a \* b) + (c \* d) + (e \* !f) + (g \* h) + (i \* j) + (k \* l) + (m \* n) + (o \* p) + (q \* r) + (s \* t) + (u \* v) + (w \* x) + (y \* z)

(!a \* b \* !c) + (c \* d \* !e \* !g \* !m) + (e \* f) + (g \* h) + (a \* !g \* !f) + (e \* f \* g \* k \* r) + (!e \* a \* b) (a \* b) + (c \* d)

**Tests with Expected results**

**Test case: All values must evaluate to true**

(a) +(b) +(c) +(d) +(e) +(f) +(g) +(h) +(i) +(j) +(k) +(l) +(m)+ (n)+ (o)+(p) +(q) +(r) +(s)+(t)+(u)+(v)+(w)+(x)+(y)+(z)

**Expected: 11111111111111111111111111**

**Result: 11111111111111111111111111**

**Test case first half of cnf must evaluate to true**

(a) +(b) +(c) +(d) +(e) +(f) +(g) +(h) +(i) +(j) +(!k) +(!l) +(!m)+ (!n)+ (!o)+(!p) +(!q) +(!r) +(!s)+(!t)+(!u)+(!v)+(!w)+(!x)+(!y)+(!z**)**

**Expected: 11111111111101100010101011**

**Result: 11111111111101100010101011**

**Results:**

====Solved=====

(!a\*c)+(!a\*c\*!e)+(!b\*c\*d\*!e)+(a\*!b\*c)+(!e\*f)

@ GENERATION 0 with 100110

====Solved=====

(a\*b)+(c\*d)+(e\*f)+(g\*h)

@ GENERATION 0 with 01111010

====Solved=====

(!a\*b)+(c\*d)+(e\*!f)+(g\*h)

@ GENERATION 0 with 10110001

====Solved=====

(r\*b)+(c\*d)+(c\*a\*!b\*b\*c\*z)

@ GENERATION 4 with 011011

====Solved=====

(a\*b\*c)+(c\*d\*e)

@ GENERATION 0 with 10111

====Solved=====

(f\*g\*k)+(n\*m)

@ GENERATION 0 with 10001

====Solved=====

(q\*z\*l\*v)+(z\*k\*r)+(m\*n\*f)+(b\*r\*t)

@ GENERATION 0 with 01100111101

====Solved=====

(!a\*b)+(c\*d)+(e\*!f)+(g\*h)+(i\*j)+(k\*l)+(m\*n)+(o\*p)+(q\*r)+(s\*t)+(u\*v)+(w\*x)+(y\*z)

@ GENERATION 9 with 10111110101011101001011111

====Solved=====

(!a\*b\*!c)+(c\*d\*!e\*!g\*!m)+(e\*f)+(g\*h)+(a\*!g\*!f)+(e\*f\*g\*k\*r)+(!e\*a\*b)

@ GENERATION 0 with 00110111000

====Solved=====

(a\*b)+(c\*d)

@ GENERATION 0 with 1110

@ GENERATION 148 with 11111111111101100010101011

====Solved=====

(a)+(b)+(c)+(d)+(e)+(f)+(g)+(h)+(i)+(j)+(k)+(l)+(m

+(u)+(v)+(w)+(x)+(y)+(z)

@ GENERATION 849 with 11111111111111111111111111

Observations: Most cnfs were solved within the first generation. This is due to the nature of the test cases.

IV.2: Operating Directions

To run this program navigate to the directory that is extracted to and type python driver.py