Arian Jalalian - 99243026

We implemented 2 optimization algorithms to solve a CNF : genetic and simulated annealing algorithm

Genetic Algorithm :

In the genetic one we defined a crossover reproduce with the probability of 70 %
In this function a random index within the boundaries specifies where the 2 parents should break and change parts to form an offspring. We selected the bigger portion of the more fit parent to produce a better child. (the more fit parent is always passed as first argument)

```
def reproduce(x, y):
    """
    do the cross over with a probability of 70 %
    """
    if random.random() < 0.3:
        return x

    m = len(x)
    cross_point = random.randint(0, m - 1)

    if cross_point < (m - 1) / 2 :
        cross_point = m - 1 - cross_point

    child = np.array(x[0:cross_point])
    child = np.append(child, y[cross_point:])
    return child
```

Selection gets the CNF and a population of individuals and selects the best 2 individuals to be parents

```
def mutate(individual):
    """
    flip a single variable value with probability of 30 %
```

```
    """
    if random.random() > 0.3:
        return

    index = random.randint(0, len(individual) - 1)
    individual[index] *= -1
```

At the end the genetic algorithm is implemented . It first generates a random population with 10 individuals then in a for loop parent selection happens among the population and then 2 parents reproduce . Then the produced child is evaluated on the CNFto see if it is the solution or not. We keep track of the best child that is produced to be the answer at the end. Then we add the produced child to the population and check if the population size exceeded 200 or not, if that is the case, we terminate the function and return the best child that is produced.

```
def genetic_algorithm(cnf, m):
    max = 0
    population = random_population(10, m)
    while(True):

        x, y = selection(cnf, population)
        child = reproduce(x, y)
        mutate(child)

        evaluation = cnf_eval(cnf, child)

        if evaluation[0]:
            return child


        if evaluation[1] > max:
            max = evaluation[1]
            max_individual = child

        print(evaluation[1])

        if (len(population) > 200):
            print("max TRUE clauses count is : ", max)
            print("solution : ", max_individual)
            return max_individual
```

```
        child.reshape(1, m)
        population = np.vstack((population, child))
```

Simulated Annealing Algorithm :

This algorithm is similar to hill climbing but it accepts the child which doesn't have better energy than its parent with a probability that decreases over time.
In the other words, it first explores for a global minimum then it tries to converge to it.

First schedule is implemented which decreases the value of temperature over time in a linear manner.

Children of each parent are generated with flipping value of a single variable so, m children (100 in this case) for each parent.

Then the algorithm is implemented with an initial value of 400 degrees for temperature . the algorithm generates the children for current parent then select a random child and calculate energy difference of the parent and the child; if it is better then it is accepted if it's not it is better, it is accepted with a probability of : $e^{\frac{-|\Delta E|}{t}}$
If the probability condition doesn't satisfy, the parent is selected again. This routine goes on until the temperature reaches 0; then the last selected child is returned.

```
def simulated_anealing(root, cnf):
    t = 400
    current = root
    alpha = 2

    while t > alpha:

        current_evaluation = cnf_eval(cnf, current)
        if current_evaluation[0]:
            return current

        t = schedule(t, alpha)
```

```
        child = child_generator(root)
        child_evaluation = cnf_eval(cnf, child)
        delta_e = child_evaluation[1] - current_evaluation[1]

        if delta_e > 0 or random.random() < math.exp((-1 * abs(delta_e)) /
t):
            print(child_evaluation[1])
            current = child

    return current
```

If we run algorithms, we get these results (which may vary from time to time) :

For genetic :

```
max TRUE clauses count is :  380
solution :  [ 1 -1  1 -1  1  1 -1  1 -1 -1 -1 -1  1  1 -1  1 -1 -1  1 -1  1
-1 -1 -1
 -1 -1  1  1  1  1  1 -1  1  1  1 -1 -1 -1  1  1  1  1  1 -1 -1 -1 -1 -1
 -1  1 -1 -1 -1  1  1 -1 -1  1  1  1 -1  1  1 -1  1 -1 -1  1  1  1  1  1
 -1 -1  1  1  1  1 -1  1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1  1  1 -1 -1 -1
 -1  1  1  1]
```

For simulated annealing :

```
final solution :  [-1. -1.  1.  1.  1. -1.  1. -1.  1.  1.  1. -1. -1. -1.
1.  1. -1. -1.
  1.  1. -1.  1.  1. -1.  1.  1. -1.  1.  1.  1.  1. -1.  1. -1.  1. -1.
 -1.  1.  1. -1.  1. -1.  1. -1. -1. -1.  1.  1.  1. -1. -1.  1. -1.  1.
 -1. -1. -1. -1.  1. -1.  1.  1.  1.  1. -1. -1.  1. -1. -1. -1.  1. -1.
 -1.  1.  1. -1. -1. -1.  1.  1.  1. -1.  1.  1.  1.  1.  1.  1. -1.  1.
  1. -1. -1. -1. -1.  1.  1.  1.  1.  1.]
count :  381
```

We can see that simulated annealing is running faster than genetic but genetic converge faster and often find a slightly better answer.