Computer Assignment #2 – Text Decryption

Ghazal Kalhor

810196675

kalhorghazal1378@gmail.com

Abstract **— In this computer assignment, we want to find a suitable key for the "Text Decryption" problem; this is done by using the Genetic Algorithm that we learnt in Artificial Intelligence.**

Keywords **— key, Genetic Algorithm, Artificial Intelligence**

I. Introduction

The goal of this computer assignment is to decrypt the given encrypted text by the substitution method. In this method, each letter is being mapped to another letter and replaced in the encrypted text in order to reach the goal text. This 26-literal mapping is called a key.

CODE 1
Translating the Text

```
def translate_text(text, dictionary):
    translated_text = ""
    for i in range(len(text)):
        if text[i].isalpha():
            if text[i].isupper():
                translated_text += dictionary[text[i].lower()].upper()
            else:
                translated_text += dictionary[text[i]]
        else:
            translated_text += text[i]
    return translated_text
```

II. Cleaning the Text

In this section, we made a dictionary from the given training text. Firstly, we replaced all the non-alphabetic characters by the space. Secondly, we converted the text into a list of words by using the "split" function. Thirdly, we converted each word into the lower case. Fourthly, we removed the English stop word from the list. Fifthly, we removed words with the length below two. Finally we converted the list into a set, in order to remove duplicate words.

CODE 2
Cleaning the Text

```
def process_text(text):
    processed_text = "".join([c if (c.isalpha()) else ' ' for c in text])
    processed_text = processed_text.split()
    processed_text = [word.lower() for word in processed_text]
    processed_text = [word for word in processed_text if word not in STOP]
    processed_text = [word for word in processed_text if len(word) > 1]
    processed_text = set(processed_text)
    return processed_text
```

III. Defining Chromosome

In this modeling, each chromosome is represented as a key that we mentioned in the last sections.

IV. Generating Initial Population

In this section, we generated each chromosome and appended it to the population list; this is done by using the "sample" function from Random library in python. Furthermore, we defined the "Decoder" class that contains necessary datasets and methods for decoding.

CODE 3
Generating Initial Population

```
def initiate_population(self):
    for i in range(POPULATION_SIZE):
        key = random.sample(ALPHABET, len(ALPHABET))
        while key in self.population:
            random.shuffle(key)
        self.population.append(key[:])
```

CODE 4
Decoder Class

```
class Decoder:
    def __init__(self, encoded_text):
        self.encoded_text = encoded_text
        self.decoded_text = ""
        self.population = []
        self.mating_pool = []
        self.fitness = [0] * POPULATION_SIZE
        training_text = open(TRAINING_DIR).read()
        self.dictionary = process_text(training_text)
        self.finished = False
```

V. Evaluating Fitness

In this section, we evaluated fitness for each chromosome in the population. Firstly, we translated the encrypted text by using the key and the "translate_text" function. Secondly, we cleaned the translated text by using the "process_text"

function. Thirdly, we initialized fitness by zero. Finally, we iterated over obtained words and if a word was in the dictionary, we added its length to the fitness. Furthermore, if all of the words were in the dictionary, we set the "finished" parameter to true in order to terminate the reproduction cycle.

CODE 5
EVALUATING FITNESS

```
def evaluate_fitness(self):
    for i in range(POPULATION_SIZE):
        table = dict(zip(self.population[i], ALPHABET))
        translated_text = translate_text(self.encoded_text, table)
        processed_text = process_text(translated_text)

        state = True
        self.fitness[i] = 0
        for word in processed_text:
            if word in self.dictionary:
                self.fitness[i] += len(word)
            else:
                state = False

        if state:
            self.decoded_text = translated_text
            self.finished = True
```

VI. SELECTION

In this section, we selected parents for the next generation by using the tournament method and added them to the mating pool. In this method, at each step we randomly selected a specific number of chromosomes from the population called tournament-size, then we compared their fitness and returned the best one.

CODE 6
SELECTION

```
def tournament_select(self):
    best_index = NONE_INT
    for i in range(TOURNAMENT_SIZE):
        rand_index = random.randint(0, POPULATION_SIZE-1)
        if best_index == NONE_INT or self.fitness[best_index] < self.fitness[rand_index]:
            best_index = rand_index
    return best_index
```

CODE 7
FILL MATING POOL

```
def fill_mating_pool(self):
    self.mating_pool = []
    while  len(self.mating_pool) != POPULATION_SIZE:
        index = self.tournament_select()
        self.mating_pool.append(self.population[index][:])
```

VII. CROSSOVER

In this section, at each step we randomly selected two parents from the mating-pool, then we applied a one-point crossover method on them with crossover-probability, otherwise we copied the parents. In this method, we randomly selected the crossover index, then we copied the genes before the index from the first parent, then we iterated over genes of the second parent and if they were not in the child, we added to it. Finally we applied the crossover on the other child by exchanging the parents.

CODE 8
APPLY-CROSSOVER FUNCTION

```
def apply_crossover(self, chromosome1, chromosome2, point):
    child = [None] * CHROMOSOME_SIZE
    for i in range(point):
        child[i] = chromosome1[i]

    pos = point
    for i in range(CHROMOSOME_SIZE):
        if pos > (CHROMOSOME_SIZE-1):
            break

        if chromosome2[i] not in child:
            child[pos] = chromosome2[i]
            pos += 1
    return child
```

CODE 9
CROSSOVER FUNCTION

```
def crossover(self):
    mating_pool = []
    index = set()
    while len(mating_pool) != POPULATION_SIZE:
        point = random.randint(1, CHROMOSOME_SIZE-2)

        index1 = random.randint(0, POPULATION_SIZE-1)
        while index1 in index:
            index1 = random.randint(0, POPULATION_SIZE-1)

        index.add(index1)

        parent1 = self.mating_pool[index1]

        index2 = random.randint(0, POPULATION_SIZE-1)
        while index2 == index1:
            index2 = random.randint(0, POPULATION_SIZE-1)

        while index2 in index:
            index2 = random.randint(0, POPULATION_SIZE-1)

        index.add(index2)

        parent2 = self.mating_pool[index2]

        if random.random() < CROSSOVER_PROBABILITY:
            child1 = self.apply_crossover(parent1, parent2, point)
            child2 = self.apply_crossover(parent2, parent1, point)
            mating_pool.append(child1)
            mating_pool.append(child2)
        else:
            mating_pool.append(parent1)
            mating_pool.append(parent2)


    self.mating_pool = mating_pool
```

VIII. MUTATION

In this section, for each offspring, we iterated over its genes and applied a swapping mutation method on them with mutation-probability. In this method, we randomly selected two indexes and swapped the genes in these indexes.

CODE 10
APPLY-MUTATION FUNCTION

```
def apply_mutation(self, chromosome, point1, point2):
    key = chromosome[:]
    key[point1] = chromosome[point2]
    key[point2] = chromosome[point1]
    return key
```

CODE 11
MUTATION FUNCTION

```
def mutation(self):
    for i in range(POPULATION_SIZE):
        for j in range(CHROMOSOME_SIZE):
            if random.random() < MUTATION_PROBABILITY:
                point1 = random.randint(0, CHROMOSOME_SIZE-1)
                point2 = random.randint(0, CHROMOSOME_SIZE-1)
                while point1 == point2:
                    point2 = random.randint(0, CHROMOSOME_SIZE-1)
                self.mating_pool[i] = self.apply_mutation(self.mating_pool[i], point1, point2)
```

IX. REPRODUCTION CYCLE

In this section, we repeated the steps that mentioned above, until we reached the wanted key. After each iteration, we copied the obtained mating-pool to the population.

CODE 12
DECODE FUNCTION

```
def decode(self):
    self.initiate_population()
    while self.finished != True:

        self.evaluate_fitness()
        self.fill_mating_pool()
        self.crossover()
        self.mutation()
        self.population = self.mating_pool[:]

    return self.decoded_text
```

X. VARIABLE POPULATION-SIZE

If we increase the population size at each iteration, both speed of convergence and accuracy will increase. The variable population-size genetic algorithm is more efficient than the standard genetic algorithm.

XI. EFFECT OF THE MUTATION

The effect of the mutation is to increase the diversity in the population and avoid converging to local minimum (maximum). If we do not use this method, the individuals in the population will be too similar, the speed of the algorithm will be reduced and in some cases, the algorithm will not reach the goal.

XII. COMPARISON BETWEEN CROSSOVER AND MUTATION

Crossover is more effective than mutation. Mutation is more effective in small population-size. Crossover causes more accuracy.

XIII. INCREASING DIVERSITY

To reach this goal, we used random selection in different steps of the algorithm. For example, we randomly selected crossover and mutation points by using the “randint”. Also we initialized the population without duplication. Finally, based on the class discussion, we find out a way to optimize the algorithm. The closer to the solution the algorithm gets, the lower mutation probability we need. Therefore we decided to reduce this parameter after each iteration with considering a lower bound.

CODE 13
UPDATED DECODE FUNCTION

```
def decode(self):
    self.initiate_population()
    while self.finished != True:

        self.evaluate_fitness()
        self.fill_mating_pool()
        self.crossover()
        self.mutation()
        self.population = self.mating_pool[:]
        global MUTATION_PROBABILITY
        if MUTATION_PROBABILITY > 0.02:
            MUTATION_PROBABILITY -= 0.0002

    return self.decoded_text
```

XIV. RESULTS

The result of execution of the algorithm with different values for parameters search are shown in Table 1. The result of the variable mutation probability is shown in the pink color.

TABLE 1
RESULTS

Population Size	Tournament Size	Crossover Probability	Mutation Probability	Execution Time
40	16	0.6	0.06	7.812
60	20	0.65	0.06	13.807
30	12	0.55	0.055	9.305
60	18	0.68	0.1	17.488

XV. CONCLUSIONS

In this computer assignment we learned that genetic algorithms are good methods to solve optimization problems, when the search space is very large. Also we were introduced to one of the applications of Artificial Intelligence in cipher substitution.

References

[1] COMPUTER ASSIGNMENT MANUAL, Computer Assignment 2, *Genetic*

[2] Shi, X.H., Wan, L.M., Lee, H.P., Yang, X.W., Wang, L.M. and Liang, Y.C., 2003, November. An improved genetic algorithm with variable population-size and a PSO-GA based hybrid evolutionary algorithm. In *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 03EX693)* (Vol. 3, pp. 1735-1740). IEEE.

[3] Luke, S. and Spector, L., 1997. A comparison of crossover and mutation in genetic programming. *Genetic Programming*, *97*, pp.240-248.