ПРОСТЕЙШИЕ ШИФРЫ И ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

Борисенко О.Ф.[1], доцент кафедры высшей математики, кандидат физико-математических наук, доцент, e-mail: borisenkoof@bsuir.by

Протько М.А.[1], студент, e-mail: mari.protko@mail.ru,

2022

1. Белорусский государственный университет информатики и радиоэлектроники

Ключевые слова: Генетический алгоритм, шифр перестановки, шифр подстановки, ген, целевая функция, селекция, мутация, турнирный отбор

Аннотация:В данной работе исследуется применение генетических алгоритмов в криптографии. Цель работы – создание системы генерации и анализа шифротекстов.

Введение

В нынешних реалиях вычислительная техника совершенствуется по законам Мерфи, постоянно увеличивая свою производительность, из чего следует, что ни один из современных шифров не будет в конце концов способен обеспечить достаточный уровень безопасности - цена получения информации методом грубой силы будет приемлема для злоумышленника. Современное решение данной проблемы – использование шифров с подходящим уровнем безопасности [1].

Стратегия, которую используют при выборе шифра, на данный момент, такова: цена дешифровки должна быть больше, чем та выгода, которую можно получить от приобретаемой информации. Возможно ли автоматизировать этот процесс (подробнее о самом процессе в [2] [3] и [4])? Что если позволить системе самой выбирать шифр в зависимости от желаний пользователя? Что если позволить системе не только выбирать шифр, но и создавать его, добиваясь необходимого уровня безопасности?

Основная часть

1.1 Постановка цели

Создание системы, имеющей на входе ключ и текст, а на выходе шифр, причем в системе изначально не должно быть алгоритмов шифрования.

То есть, целью данного проекта является создание некоего алгоритма, выдающего в зависимости от внешних характеристик один из шифров из данного списка: шифр Цезаря, Виженера, Бофора, Плейфера, простой замены, одноразовый блокнот.

1.2 Выбор технологии:

Поскольку условие динамического и самостоятельного развития соответствует области машинного обучения, выбор технологии пал на генетический алгоритм.

Язык реализации Python3, поскольку в дальнейшем будет работа с большими массивами данных, комфорт работы с которыми достижим средствами данного языка.

1.3 Разработка.

Идея использования генетического алгоритма в шифровании нигде до этого не использовалась (по крайней мере, какие-либо свидетельства обратного не были найдены), поэтому все этапы необходимо было проектировать с нуля.

Генетический алгоритм состоит из следующих этапов [5]:

- Создание начальной популяции – генерируется случайно с заданным размером
- Вычисление приспособленности – в данной программе – fitness_value. Принципы ее вычисления напрямую зависят от структуры гена
- Отбор
- Скрещивание и/или мутации
- Выбор индивидуума с максимальной приспособленностью

Подробности о каждом этапе описаны в соответствующих подпунктах.

Входные данные – текст и ключ, представляют собой текстовые строки кода ASKII (текст на английском языке).

Особь, или же ген представляет собой генерируемую последовательность действий, в итоге приводящую к алгоритму шифрования (функции биекции). Особь в коде (см. приложение А) представлена классом cipher, имеющем такие поля как fitness_value, gen_code, decipher_text и cipher_text, заполняющиеся в соответствии с этапом.

Эпоха (поколение) в данном случае представлена несколькими этапами, некоторые из которых повторяются. Каждый этап применяется к особи (элементу списка population), при условии, если особь соответствует заданному критерию. Результат эпохи – создание новой популяции посредством скрещивания и мутации, или же выход из программы (нахождение оптимального решения или же доказательство отсутствия такового – когда поколения не отличаются по свойствам друг от друга).

В программе реализованы следующие этапы (см. приложение Б):

Начальный – создание популяции определенного количества, а также ввод текста и ключа пользователем. Создаются все изначальные элементы графического интерфейса, а также потоки.

После начального этапа программа входит в цикл, состоящий из следующих этапов:

Подсчет fitness_value – или же, характеристики приспособленности. Представляет собой обыкновенное десятичное число. Присваивается в зависимости от наличия выполнимых операций в гене особи, а также от того, как эти операции следуют друг за другом. Если присутствует минимум операций для создания шифра, для данной особи начинает выполняться ген, в зависимости от результатов выполнения которого к уже имеемой fitness_value прибавляется некое дополнительное значение.

Выполнение гена – вызывается из предыдущего этапа в зависимости от fitness_value. Последовательно выполняется каждый элемент гена с проверкой результата. На выполнение дается определенное количество циклов (элемент CICLES – средство синхронизации потоков выполнения гена для каждой особи популяции), по окончанию которых происходит дешифровка (с тем же количеством циклов и при наличии результата соответствующего с текстом размера). В зависимости от того, на каком этапе выполнения гена закончиться процесс, к значению fitness_value прибавляется соответствующая величина. Если все этапы выполнения гена привели к нахождению решения, цикл генетического алгоритма завершается.

Эволюция – или же, кроссинговер и/или мутация. Из какого-то количества особей с наибольшим показателем fitness_value выбираются пары для кроссинговера, а для оставшихся особей с некой вероятностью происходит мутация гена (параметры мутации тоже задаются случайно функцией random языка python, таким образом, получая либо совершенно новую особь – либо особь с геном, изменённым в некоторых случайных местах). Во время кроссинговера ген каждого элемента пары делиться на две части (размер частей также задается функцией random), после чего создаются два потомка, у каждого из которых будет часть из гена каждой особи из пары (части разные!).

Естественный отбор – этот этап возникает, когда количество особей превышает заданное значение (сделано для того, чтобы потомки проявили себя, прежде чем избавляться от родителей). По значению fitness_value выбираются особи с минимальным значением данного критерия, после чего они удаляются из списка population, тем самым прекращая участие в генофонде.

После выполнения основных этапов при отсутствии решения цикл повторяется, запуская все этапы в той же последовательности для имеющихся особей.

1.4 Описание графического интерфейса приложения.

Самая первая версия приложения так и не была закончена по причине отсутствия графического интерфейса [6]. Реализация непосредственно алгоритма – это важно, но не менее важно понимать происходящее в собственной программе. Поэтому, остановимся на подробном описании реализации графической интерпретации происходящего.

Графический интерфейс разделен на следующие области:

Область ввода – здесь пользователь вводит значение ключа и текста, а также будущие константы, такие как – количество циклов (CICLE), количество пар для кроссинговера, количество особей в начальной популяции (POPULATION_SIZE), максимальное количество особей (POPULATION_MAX_SIZE), желаемый уровень безопасности, а также mutation – частота мутации, размер гена. В случае, если таковые значения не были введены, программа начинает работу с заранее определенными значениями в ней.

Область гена – в данной области показывается генетический код для советующей особи популяции (каждому элементу списка population присваивается номер, который выводиться в этой области). Особи-лидеры помечаются графическим интерфейсом. Во время этапа выполнения гена советующий элемент генетического кода выделяется при условии нажатия на строку элемента в окне области гена. В верхней части окна помечается номер эпохи.

Область прогресса – в ней присутствует номер особи в популяции (аналогично предыдущей области), значение fittness_value, а также зашифрованное и расшифрованное сообщение для каждого элемента популяции. В этой области также помечаются особи-победители.

Область контроля - включает в себя следующие кнопки: кнопку остановки программы (необходима для остановки потоков), кнопку остановки всех циклов, кнопку ускорения работы программы, кнопку отрисовки графиков .

Область отрисовки графиков (см приложение В) – отдельное окно, появляющееся при нажатии кнопки. Состоит из 10 графиков, с отложенным по осям *x* и *y* частям алфавита. На графике изображается целевая функция каждой особи популяции соответственно. Достигается это следующим образом: на графике изображается желаемый результат (график одного из шифров) и результат данного поколения. Таким образом, можно наглядно проследить эволюцию (значения fittness_value будут стремится к значениям целевой функции).

Внешний вид графика представлен на рисунке 4.1.

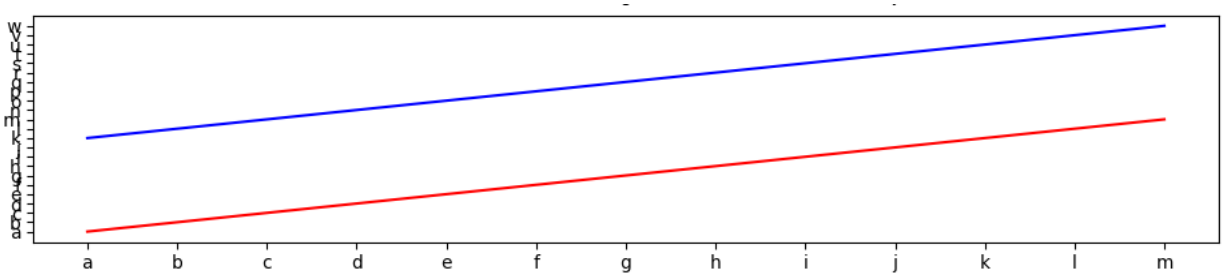

Рисунок 4.1 – пример графика из области отрисовки графиков

На графике из области отрисовки ось абсцисс – изначальный текст (в примере на рисунке 1– первые 13 букв алфавита), ось ординат – шифрованный текст.

На данных графиках не обозначается ключ, с помощью которого производилось шифрование, поэтому при каждом новом запуске программы графики одного и того же шифра могут отличаться.

Область отрисовки и внешний вид программы показаны на рисунках 2 и 3 соответственно. На рисунках показана первая эпоха после инициализации программы.

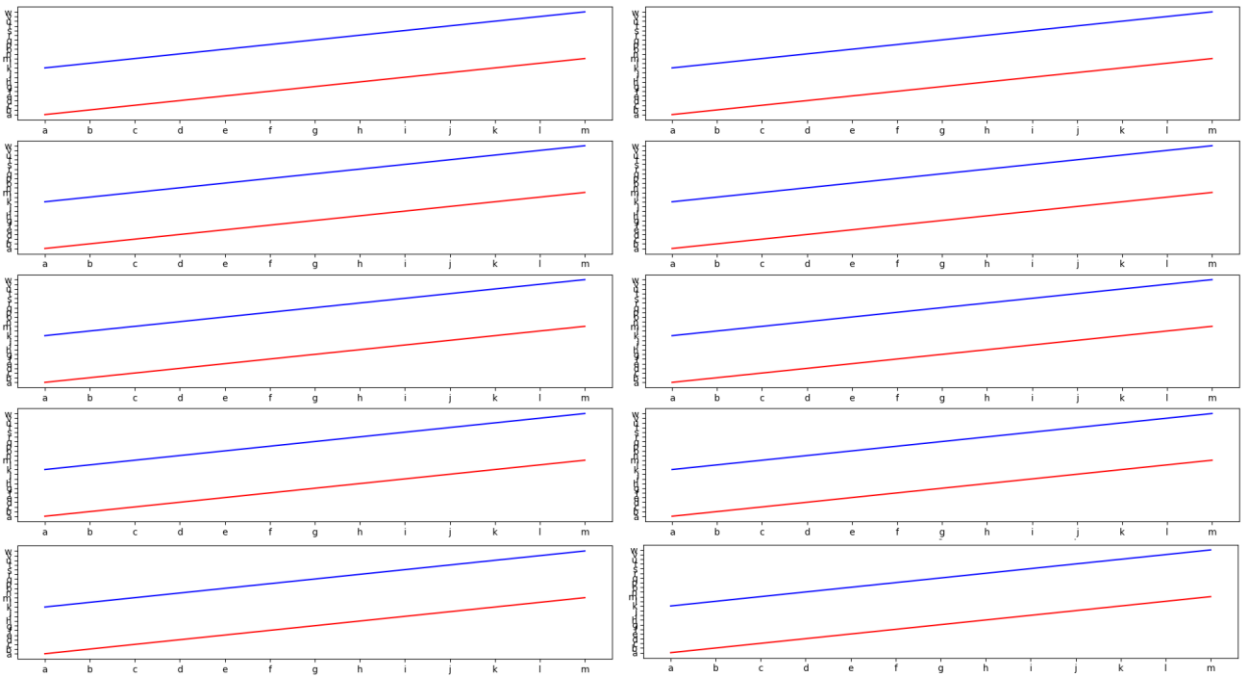

Рисунок 4.2 – область отрисовки графиков

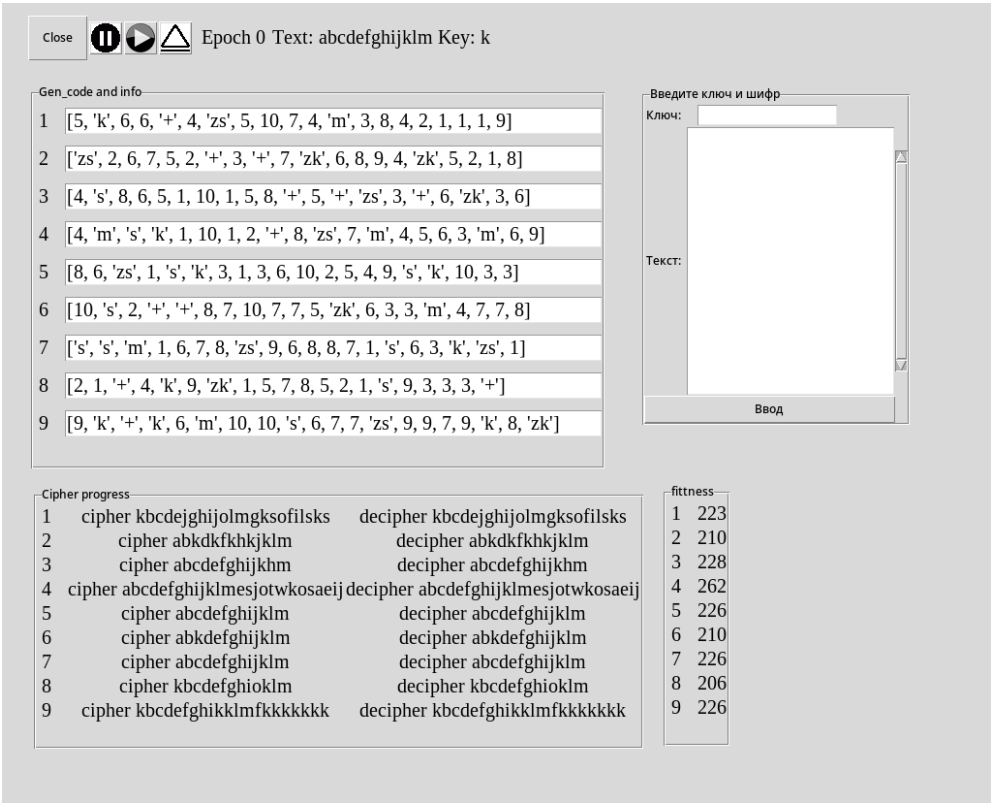

Рисунок 4.3 – графический интерфейс

1.5 Описание гена алгоритма.

Ген, по fitness_value которого будет производиться отбор, состоит из элементов двух категорий – операции и ее аргумента. Операции представлены в гене буквами английского алфавита, в то время как аргументы – цифрами, имеющими любое положительное значение.

Для возможности реализации шифров из определенной выборки (описана в цели) необходимы следующие операции:

- ‘k’ – взять элемент ключа
- ‘s’ – взять элемент текста
- ‘p’ – проверка на конец шифрованного текста
- ‘m’ – перенос указателя читаемого места в гене
- ‘zs’ – взять элемент текста на позицию x больше
- ‘zk’ – взять элемент ключа на позицию x больше
- ‘mo’ – операция взятия модуля
- ‘+’ – операция сложения по mod 26
- ‘-‘ – операция вычитания по mod 26
- ‘*’ – операция умножения по mod 26
- ‘/‘ – операция деления по mod 26

После некоторых операций следует аргумент:

- ‘k' 15 –означает – взять (15 по mod мощность ключа) элемент ключа
- ‘s' 15 - взять (15 по mod мощность текста) элемент текста
- ‘m' 15 – изменить индекс читаемой в гене позиции на (15 mod мощность гена)

После математических операций ‘+’, ‘-‘, ’*’, ’/’, ’md’ следует два аргумента:
‘+’ 14 22 – прибавить к 22 значение 14

Приоритеты операций: ‘p’,’m’,‘k’ и ‘s’, ‘*’ и ‘/’,‘+’ и ‘-‘.

Запись всех операций в гене осуществляется в префиксной форме (польская запись), то есть, для элемента, аналогичного первой итерации шифра Цезаря получим: ‘+’ ’s’ 5 ’k’ 8

1.6 Ускорение работы.

Программа разбита на две секции – графическую, для взаимодействий с пользователем и основную - для создания гена. Для повышения скорости работы программа разбивается на потоки – поток графики mainloop() (средства библиотеки tkinter), который возможно временно приостанавливать в случае необходимости сбора большого объема данных (фоновый режим работы). В основной части также присутствует разбиение на потоки: поток первоначального расчёта fitness_value – их количество пропорционально размеру популяции (один поток обрабатывает до 5 элементов), в случае, если значение fitness_value больше 12 (позиции операций и их аргументов с большей вероятностью образуют исполняемый алгоритм способный произвести шифрование), запускается новый поток, выполняющий ген данного элемента популяции. При этом, все потоки, выполняющие ген, синхронизируются по значению CICLE и запускаются только в случае, если fitness_value всех элементов популяции уже были посчитаны, если программа не работает в фоновом режиме, иначе – потки выполняют свои операции независимо, а основной цикл программы ожидает выполнения всех потоков прежде чем перейти к следующему этапу.

1.7 Стратегии отбора.

Для ускорения работы генетического алгоритма, до выполнения шифрования с помощью гена необходимо убедиться в следующем:

- наличие минимума операций для создания шифра
- наличие необходимых операндов за операциями (проверка смысловой наполненности гена без его выполнения)
- отсутствие последовательности, состоящей только из операндов (“+’ ‘-‘ ‘*’ – пример). При наличии данной последовательности ее необходимо пропускать.

- проверка гена (эту проверку возможно отобразить в отдельном окне – то есть, из всей мощности гена будут выбраны только активные элементы, которые будут выполняться в течение определенного промежутка времени, заданного переменной CICKLE)

Поскольку по условию ген алгоритма должен описывать функцию биекции, следует выделить следующую проверку гена при расшифровке/дешифровке одной буквы:

- из множества букв на входе можно получить единственное множество зашифрованных букв (новый алфавит), причем буквы в полученном алфавите не повторяются. (по свойству инъективности)
- для полученного гена возможно составить ген дешифровки, позволяющий получить обратный результат (изначальный текст).

1.8 Расчёт приспособленности.

Для расчёта fitness_value, или же приспособленности, необходимо знать целевую функцию. На рисунках 8.4-8.7 показаны функции шифрования. На графике по оси абсцисс – изначальный текст. На оси ординат – шифротекст. Первые графики рисунков 8.5-8.7 – шифры с использованием ключа длиной в один символ, вторые графики – использование шифра длиной в 3 символа, причем подобранного таким образом, чтобы получить наибольшую вариативность.

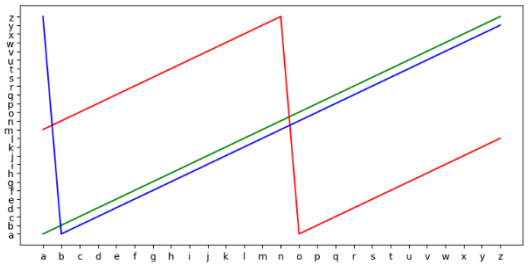

Рисунок 8.4 –шифр Цезаря при разных ключах

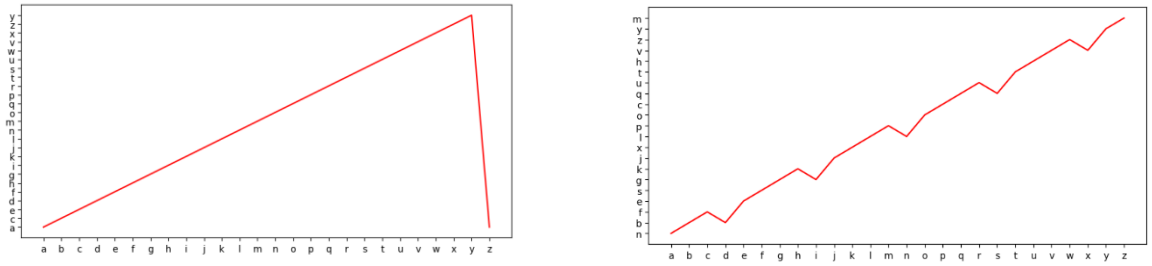

Рисунок 8.5 – Шифр Виженера

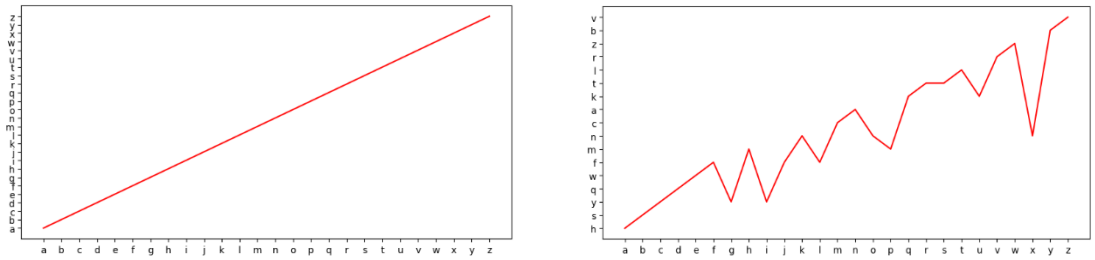

Рисунок 8.6 – Шифр Вернама

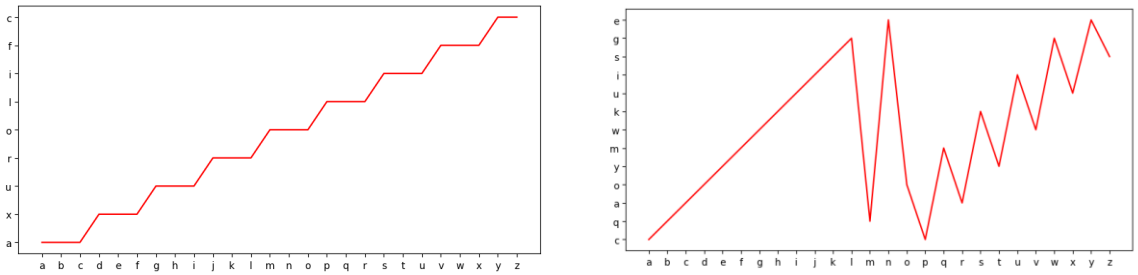

Рисунок 8.7 – Шифр Бофора

Из графиков 8.4-8.7 можно убедиться в том, что создание системы образования гена, способной образовывать на своем выходе один из шифров подстановки или/и перестановки возможно, поскольку функции шифрования имеют в некоторых случаях похожую форму.

Для расчёта fitness_value, или же приспособленности, можно воспользоваться двумя стратегиями. Первая – расчет fitness_value по значениям гена. В данном случае составляется «идеальный» ген, дающий желаемый шифр и имеющий минимальное количество значащих генов для его осуществления. После определяются ключевые характеристики этого гена – соотношение количества операций к количеству аргументов, порядок следования операций друг за другом и т.д. На основании выделенных характеристик составляется алгоритм, проверяющий соответствие каждому выделенному пункту элемента популяции. Чем больше соответствий, тем больше значение fitness_value.

К примеру, для гена шифра Цезаря (на основании таблицы 9.1), получается следующий набор соответствий:

- Количество различных операций – 5
- Порядок следования: две операции – аргумент, одна операция – аргумент.
- Наличие переноса
- Наличие операции, требующей двух аргументов

Также, стоит упомянуть изменение fitness_value при прохождении особью популяции этапа выполнения гена – в данном случае, каждый элемент получаемого на выходе выполнения гена шифротекста сравнивается с изначальным текстом. Чем

больше несоответствий между шифротекстом и изначальным текстом и чем их, соответственно, меньше между расшифрованным текстом и изначальным, тем больше fitness_value.

Второй способ расчета fitness_value стремится наоборот, не к ее росту, а к уменьшению. В данном случае, искомое решение будет получено, если fitness_value особи будет равна 0. При выполнении этапа расчета приспособленности будет рассчитываться разница между функцией шифрования и функцией гена особи (графики 8.4-8.7). Чем ближе значения гена особи к функции шифрования, тем выше ее приспособленность. Разница рассчитывается путем вычитания значения элемента шифротекста из значения соответствующего элемента шифротекста гена. Знак при этом отбрасывается.

1.9 Результаты и сравнительный анализ.

После выбора системы операций и их аргументов, была проведена проверка выполнимости поставленных целей данной системой, а именно, возможность шифрования и расшифрования соответственно при этапе выполнения гена.

Для проверки была написана вспомогательная программа (см. приложение Г), позволяющая выполнять записанный вручную ген. Было проверено выполнение генов в соответствии с шифрами, на основании которых они базировались. Проверка считалась пройденной, если ген позволял расшифровать зашифрованное им же сообщение. Ген относился к шифру при соответствии шифротекстов гена и шифра при использовании теста и ключа, общего как для гена, так и для функции шифрования.

Таблица 9.1 – Шифры и соответствующие гены

Шифр Цезаря:	‘+’ ‘s’ 0 ‘k’ 0 ‘+’ ‘zs’ 1 ‘k’ 0 ‘m’ 4
Шифр Виженера:	‘+’ ‘s’ 0 ‘k’ 0 ‘+’ ‘zs’ 1 ‘zk’ 1 ‘m’ 4
Шифр Бофора:	‘-’ ‘s’ 0 ‘k’ 0 ‘-’ ‘zs’ 1 ‘zk’ 1 ‘m’ 4
Шифр Вернама (одноразовый блокнот)	‘mo’ ‘s’ 0 ‘k’ 0 ‘mo’ ‘zs’ 1 ‘zk’ 1 ‘m’ 4
Аффинный шифр:	‘+’ ‘*’ 5 ‘s’ 0 ‘+’ ‘*’ 5 ‘zs’ 1 ‘m’ 4
Шифр простой перестановки:	‘s’ 1 ‘zs’ 2 ‘m’ 4

Также была проверена работоспособность алгоритма выполнения гена в случае последовательностей гена, не дающих шифротекс или образующих его неполное

множество. Было проверено более 30 случайных последовательностей, после чего были исправлены возникшие ошибки.

После этого этапа был найден алгоритм выполнения гена, позволяющий не только производить шифрование и расшифрование, но и не заканчивающий работу всего приложения в случае возникновения последовательностей в гене, не дающих никакого вывода или входящих в бесконечный цикл.

Были проверены следующие сочетания стратегий эволюции с использованием турнирного отбора и расчета fitness_function на основании содержания гена:

Мутация: В данном случае преждевременная сходимость не наблюдается, но приблизительно на 500 эпохе ген входит в цикл, где значения целевых функций повторяются (в какой-то момент достигают максимума в районе 12, при этом процесс шифрования не полный или не наблюдается, после чего постепенно популяция приходит к значениям целевой функции 3-1).

Кроссинговер: Преждевременная сходимость, приближаясь к 50 эпохе, значение fitnes_value стабильно уменьшается, пока не стабилизируется на значении 1 для всей популяции. Можно наблюдать, как ген каждой популяции последовательно «собирает» все элементы операций, пока не появится доминирующая, которая после и останется во всей популяции.

Кроссинговер и мутация: Преждевременная сходимость гена в районе 300 эпохи к последовательности элементов, состоящей из какой-либо операции, причем числовые значения локализуются в определенном месте гена и посредством следующих этапов естественного отбора эти значения перемещаются в иную позицию. При возникновении мутаций разнообразие не достигается. Значения fitnes_value для популяции ведут себя также, как и при кроссинговере, только преждевременная сходимость происходит значительно позже.

Полученный на данном этапе результат позволил убедиться в невозможности выполнения поставленной цели при использовании мутации и кроссинговера по отдельности. Получение результата возможно с наибольшей вероятностью, когда они используются вместе.

Заключение

В итоге проделанной работы была получена система значений гена для образования функций шифрования простейших шифров (подстановки и перестановки). Были найдены стратегии формирования популяций следующих эпох на этапе эволюции, при этом был проведен их сравнительный анализ по таким параметрам, как количество эпох перед достижением преждевременной сходимости или нахождения решения. Также было проведено сравнение простейших шифров и их характеристик.

В итоге была создана система, позволяющая из множеств текстов и ключа получить множество функций шифрования, аналогичных простейшим шифрам подстановки и перестановки.

Список использованных источников

1. *Статья из электронного журнала* **Jorgen Veisdal, Shannon Ciphers and Perfect Security** /Cantor’s Paradise/2020
2. **Протько М.А., Борисенко О.Ф. Принципы построения алгоритмов шифрования**: Материалы V международной научно-практической конференции студентов, аспиранов, преподавателей. АМТИ, г. Армавир, Россия, 30-31 октября 2021 г. – С. 279-282.
3. **Протько М.А. Пример построения алгоритма шифрования**: Материалы XII Республиканской научной конференции молодых ученых и студентов, г. Брест, 18-19 ноября 2021 года.– С. 117-120.
4. **Протько М.А. Теория чисел в ассиметричном шифровании**: 57-я научная конференция аспирантов, магистрантов и студентов // БГУИР 2022.
5. **Панченко Т.В., Генетические алгоритмы**: учеб. пособие / под ред. Ю. Ю. Тарасевича.; Изд-во Астраханского ун-та, 2007. — 87 с.
6. **Протько М.А. Шифр Цезаря и генетический алгоритм**: 58-я научная конференция аспирантов, магистрантов и студентов // БГУИР 2022. – С. 150-159.

Приложение А

Реализация индивидуума

```
import random
import string

alphabet=list(string.ascii_lowercase)

#список с номерами операций (используется для разброса значений при изначальной #генерации популяций
operations_decoder = {
0:'k', # взять элемент ключа по номеру
1:'m', # сместить выполняемую позицию гена по номеру
2:'s', # взять элемент текста по номеру
3:'+', # сложить по модулю 26
4:'zs', # взять следующий элемент текста по индексу + смещение
5:'zk' # взять следующий элемент ключа по индексу + смещение
7:'mo', # взять модуль
4:'-'  # отнять по модулю 26
}

dec_operations={ #в случае расшифровки операнды гена меняются на обратные им
'+':'-',
'-':'+'
}

class cipher:
	def __init__(self, text, key):
		self.key=key # ключ
		self.text=text # изначальны текст
		self.cipher_text=text #Шифр
		self.decipher_text=text # Расшифровка
		self.operations = { #соответствие операндов гена с их функциями
			'k': lambda x: self.key[x%len(self.key)],
			'm': 0,
			's': lambda x: self.cipher_text[x%len(self.cipher_text)],
			'+': lambda x, y: alphabet[((alphabet.index(x) + alphabet.index(y))%len(alphabet))],
			'e': 0, # для быстрого выхода из рекусии в rec_funk
			'zs': lambda x:self.cipher_text[x%len(self.cipher_text)],
			'zk': lambda x: self.key[x%len(self.key)],
			'-': lambda y, x:alphabet[((alphabet.index(x) - alphabet.index(y))%len(alphabet))]
			'mo':lambda x,y: alphabet[((alphabet.index(x)%alphabet.index(y))%len(alphabet))]
			'*': lambda x, y: chr((ord(x) * ord(y))%26),
			'/': lambda x, y: chr((ord(x) / ord(y))%26)
			}
		self.GEN_SIZE=20 # размер гена
		self.CICLES=50*len(self.cipher_text) # количество итераций
		#следующие параметры нужны для контроля равномерного распределения
	операндов и их значений в первоначальной популяции
```

```
        self.MAX_CODE_NUMBER=10
        self.ARG_BEG=6
        self.ARG_END=10

        self.gen_code=[]
        self.fittness_value=0 # значение целевой функции

        self.ind_gen_code=0
        self.ind_text=0
        self.ind_key=0
        self.count_for_cicles=0
        self.decipher=False

        self.generate_primal_code() # функция создания первоначального гена

def __lt__(self, obj):
        return ((self.fittness_value) < (obj.fittness_value))
def __gt__(self, obj):
        return ((self.fittness_value) > (obj.fittness_value))
def __le__(self, obj):
        return ((self.fittness_value) <= (obj.fittness_value))
def __ge__(self, obj):
        return ((self.fittness_value) >= (obj.fittness_value))
def __eq__(self, obj):
        return (self.fittness_value == obj.fittness_value)
def __repr__(self):
        return str((self.gen_code, self.fittness_value))

def generate_primal_code(self):
        #в начале создаются только значения, позже в случайные места добавляются
операнды (второй while)
        tmp=0
        while tmp < self.GEN_SIZE:
                self.gen_code.append(random.randint(1, self.MAX_CODE_NUMBER))
                tmp+=1
        tmp=0
        while tmp<random.randint(self.ARG_BEG,self.ARG_END):
        self.gen_code[random.randint(0,self.GEN_SIZE-
        1)]=operations_decoder[random.randint(0,5)]
        tmp+=1

def mutation(self):
        for element in range(0,random.randint(1,10)):
                if random.randint(0,self.MAX_CODE_NUMBER+5)<5:
                        self.gen_code[random.randint(0,self.GEN_SIZE-
                1)]=operations_decoder[random.randint(0,5)]
                else:
```

```
                self.gen_code[random.randint(0,self.GEN_SIZE-1)]=random.randint(1,
            self.MAX_CODE_NUMBER)

def ciper_do(self,text,key):
    self.text=text
    self.key=key

    self.beg_funk()
    chipher=self.cipher_text
    self.decipher=True
    self.beg_funk()

def rec_funk(self,zn):
    self.ind_gen_code=(self.ind_gen_code+1)%self.GEN_SIZE
    self.count_for_cicles+=1
    if self.count_for_cicles>=self.CICLES: zn='e'
        try:
            self.operations[zn]
            if zn=='k' or zn=='s' or zn=='zs' or zn=='zk' or zn=='m':
            try:
                x=self.rec_funk(self.gen_code[self.ind_gen_code])
                if type(x)==type('a'): x=alphabet.index(x)
                if type(x)==type(None):
                x=0
                if zn=='k':
                    self.ind_key=int(x)
                elif zn=='s':
                    self.ind_text=int(x)
                elif zn=='zs':
                    self.ind_text+=int(x)
                    x=self.ind_text
                elif zn=='zk':
                    self.ind_key+=int(x)
                    x=self.ind_key
                elif zn=='m':
                    self.ind_gen_code=int(x)%len(self.gen_code)
                    print(self.ind_gen_code)
                    return
            ret=self.operations[zn](x)
            return ret
        except IndexError:
            print("Out of range - The end")
            return
    elif zn=='e':
        print("THE END!")
        return
    else:
        print("we have twice "+ str(zn))
        if self.decipher:
```

```
                zn=dec_operations[zn]
                print("Deciper! "+ str(zn))
                try:
                        y=self.rec_funk(self.gen_code[self.ind_gen_code])
                        x=self.rec_funk(self.gen_code[self.ind_gen_code])
                        print("twice counted: "+ str(y)+" "+ str(x))
                        if type(x)==type(1): x=alphabet[x%len(alphabet)]
                        if type(y)==type(1): y=alphabet[y%len(alphabet)]
                        if type(x)==type(None):
                                x='a'
                        if type(y)==type(None):
                                y='a'
                        ret=self.operations[zn](x,y)
                        return ret
                except IndexError:
                        print("Out of range - The end")
                        return
                except KeyError:
                        return zn

def beg_funk(self):
        test_text=""
        number_of_returned_letters=0
        while number_of_returned_letters<len(self.text)and self.count_for_cicles<self.CICLES:
        if self.gen_code[self.ind_gen_code] == 'e':
                print("END!")
                return
        elif self.gen_code[self.ind_gen_code] in self.operations:
                fin= self.rec_funk(self.gen_code[self.ind_gen_code])
                if fin==None:
                        print("NONE")
                        continue
                number_of_returned_letters+=1
                self.cipher_text=self.cipher_text[:self.ind_text]+str(fin)+self.cipher_text[self.ind_te
        xt+1:]
                test_text+=str(fin)
                continue
                self.ind_gen_code=(self.ind_gen_code+1)%self.GEN_SIZE
        self.ind_gen_code=0
        self.ind_text=0
        self.ind_key=0

def count_fittness_new(self,chipher):
        self.fittness_value=0
        for index in range(0,len(chipher)-1):
                tmp= alphabet.index(chipher[index])-
                alphabet.index(self.cipher_text[index%len(self.cipher_text)])
                if tmp<0: self.fittness_value-=tmp
                else: self.fittness_value+=tmp
```

Приложение Б

Основная программа

```
import threading
import time
import sys
import tkinter as tk
from tkinter import ttk
from tkinter import *
from tkinter import scrolledtext
from PIL import ImageTk, Image
from Cipher import *

class conditions:
	def __init__(self,frame,pos_col,pos_row):
		self.row=pos_row
		self.column=pos_col
		self.crown_frame = Frame(frame, width=10, height=10)
		self.crown_frame.grid(column=pos_col,row=pos_row)
		img= (Image.open("/home/mash0/project/gen_alg/pictures/crown"))
		resized_image= img.resize((30,30), Image.ANTIALIAS)
		self.new_image= ImageTk.PhotoImage(resized_image)
		tmp=0
		while tmp<4:
			self.crown_labels.append(Label(self.crown_frame,image=self.new_image))
			tmp+=1
	def mark_winner(self,ciphers):
		array=sorted(ciphers)[:4]
		#выбираем ряд, где нужно нарисовать корону
		i=0
		for element in array:
			self.crown_labels[i].grid(column=self.column,row=self.row+i)
			i+=1

class Mains:
	def __init__(self):
		self.TEXT=""
		self.KEY=""

		self.letter_list="".join(alphabet) #алфавит шифра (английские буквы нижнего регистра
		с пробелом
		self.POPULATION_SIZE=10 # размер популяции
		self.TURNIR_MAX_NUMBER=10 # максимальное количество особей, участвующих
		в турнире
		self.MUTATION_MAX_NUMBER=10 # максимальное количество мутаций
		self.population=[] # изначальный массив популяции
		self.CICLES=6 #количество итераций по шифру
		self.epoha=0 # номер эпохи
```

```
        self.gui_done=False # флаг окончания отрисовки при запуске приложения
        self.running =True # флаг для остановки потоков

        self.gui_thread = threading.Thread(target = self.gui_run) # поток контроля графики
        self.main_thread = threading.Thread(target = self.main_cicle) # поток выполнения
        этапов для каждого элемента популяции
        self.genetate_population() # создание первоначальных генов каждого элемента
        популяции
        #запуск потоков
        self.gui_thread.start()
        self.main_thread.start()

def gui_run(self):
        self.win = tk.Tk()
        self.win.geometry('920x920')

        #создание верхнего меню
        self.frame_menu = ttk.Frame(self.win)
        self.frame_menu.grid(row = 0, column = 0, pady = 10)

        close_button = tk.Button(self.frame_menu,text="Close",command=self.stop, width = 5,
        height = 2)
        close_button.grid(column=0, row=0,columnspan = 2)

        self.epoch_label=ttk.Label(self.frame_menu,
        text = "Epoch "+str(self.epoha),
        font = ("Times New Roman", 15),
        foreground = "black")
        self.epoch_label.grid(column = 14,row=0)

        self.text_label=ttk.Label(self.frame_menu,
        text = " Text: "+str(self.TEXT)+ " Key: "+ str(self.KEY),
        font = ("Times New Roman", 15),
        foreground = "black")
        self.text_label.grid(column = 15,row=0)

        self.gen_code_labels=[]
        self.frame_gen_code_and_info = ttk.LabelFrame(self.win, text="Gen_code and info")
        self.frame_gen_code_and_info.grid(row = 1, column = 0, columnspan = 8, pady =
        15,padx=30)

        #создание начальных номеров популяции
        tmp=0
        while tmp<self.POPULATION_SIZE:
                ttk.Label(self.frame_gen_code_and_info,
                text = str(tmp+1)+" " ,
                font = ("Times New Roman", 15),
                foreground = "black").grid(
```

```
            column = 1,row=1+tmp)
            tmp+=1

        #создание начальных слов
        tmp=0
        while tmp<self.POPULATION_SIZE:
            self.gen_code_labels.append(Text(self.frame_gen_code_and_info,height=1,width=
            55,
            font = ("Times New Roman", 15),
            foreground = "black"))
            self.gen_code_labels[tmp].insert('end',str(self.population[tmp].gen_code))
            self.gen_code_labels[tmp].configure(state = 'disable')
            self.gen_code_labels[tmp].grid(column = 2,row=1+tmp,pady=5)
            tmp+=1

        #информационное поле
        self.cipher_process_labels=[]
        self.fittnes_and_cipher_labels=[]
        self.text_of_a_chipher=[]
        self.text_of_a_dechipher=[]
        self.frame_cipher_process = ttk.LabelFrame(self.win, text="Cipher progress")
        self.frame_cipher_process.grid(row = 1, column = 9, columnspan = 10)

        #создание полей номеров популяции
        tmp=0
        while tmp<self.POPULATION_SIZE:
            ttk.Label(self.frame_cipher_process,
            text = str(tmp+1)+" " ,
            font = ("Times New Roman", 15),
            foreground = "black").grid(
            column = 1,row=1+tmp)
            tmp+=1

        #создание полей текстов шифра:
        tmp=0
        while tmp<self.POPULATION_SIZE:
            self.text_of_a_chipher.append(ttk.Label(self.frame_cipher_process,
            text = "cipher "+str(self.population[tmp].cipher_text),
            font = ("Times New Roman", 15),
            foreground = "black"))
            self.text_of_a_chipher[tmp].grid(column = 2,row=1+tmp,columnspan=2)
            tmp+=1

        #создание полей текстов дешифра:
        tmp=0
        while tmp<self.POPULATION_SIZE:
            self.text_of_a_dechipher.append(ttk.Label(self.frame_cipher_process,
            text = "decipher "+str(self.population[tmp].decipher_text),
```

```
                font = ("Times New Roman", 15),
                foreground = "black"))
                self.text_of_a_dechipher[tmp].grid(column = 4,row=1+tmp,columnspan=2)
                tmp+=1

        self.gui_done=True
        self.win.mainloop()

def genetate_population(self): # создание первоначальных генов популяции
        tmp=0
        while tmp<self.POPULATION_SIZE:
                self.population.append(cipher(self.letter_list,'k'))
                tmp+=1

def main_cicle(self):
        while self.gui_done==False: # ожидание окончания отрисовки
                time.sleep(1)
        while self.running:
                # выполнение всех этапов эпохи
                self.cipher_cicle() # выполнение гена (шифрование и расшифрование)
                self.whether_it_works() # проверка на нахождение решения
                self.competition()
                self.change_gen_code( ) # кроссинговер и мутация
                self.epoha+=1
                self.epoch_label.config(text="Epoha "+str(self.epoha))

def cipher_cicle(self):
        for creater in self.population:
                creater.ciper_do(self.TEXT,self.KEY)

def whether_it_works(self):
        number_of_different_letters=0
        for element in self.population:
                for index in range(0,len(element.cipher_text)-1):
                        if element.cipher_text[index]!=self.TEXT[index]:
                                number_of_different_letters+=1

        print("Number:")
        print(number_of_different_letters)

        for element in self.population:
                for index in range(0,len(element.decipher_text)-1):
                        if len(element.decipher_text)!=len(self.TEXT) or
                        element.decipher_text[index]!=self.TEXT[index]:
                                print("Not perfect")
                                return
                        print("PERFECT!!!!")
                        self.add_highlighter(self.population.index(element))
```

```
def crossover(self,gen_code1,gen_code2): # кроссинговер
        divider=random.randint(0,cipher(self.letter_list,'k').GEN_SIZE-1)
        new_gen_code1=gen_code1[divider:]+gen_code2[:divider]
        new_gen_code2=gen_code1[:divider]+gen_code2[divider:]
        final1=cipher(self.letter_list,'k')
        final2=cipher(self.letter_list,'k')
        final1.gen_code=new_gen_code1
        final2.gen_code=new_gen_code2
        final1.count_pre_cipher_fittnes()
        final2.count_pre_cipher_fittnes()
        if final1.fittness_value>=final2.fittness_value: return final1
        return final2

def competition(self): # турнирное соревнование, после соревнования – кроссинговер, затем
мутации

        #турнирный отбор
        new_population=[]
        middle_array=[]
        for a in range(self.TURNIR_MAX_NUMBER):
                selection_list=[]
                for b in range(random.randint(2, self.TURNIR_MAX_NUMBER)):
                        selection_list.append(self.population[random.randint(0,
                        self.POPULATION_SIZE-1)])
        selection_list.sort()
        middle_array.append(selection_list[0])

        for a in range(self.TURNIR_MAX_NUMBER): #кроссинговер
                kkk=self.crossover(middle_array[random.randint(0, len(middle_array)-
        1)].gen_code,middle_array[random.randint(0, len(middle_array)-1)].gen_code)
                new_population.append(kkk)

        self.population=new_population

        #мутация
        for element in range(random.randint(1,self.MUTATION_MAX_NUMBER)):
                self.population[random.randint(0, len(self.population)-1)].mutation()

def stop(self): # выход из программы
        self.running=False
        self.win.destroy()
        time.sleep(1)
        sys.exit()

def change_gen_code(self): # перерисовка полей гена в графической части программы
        tmp=0
        for element in self.gen_code_labels:
```

```
                element.configure(state='normal')
                element.delete("1.0","end")
                element.insert('1.00',str(self.population[tmp].gen_code))
                element.configure(state='disable')
                tmp+=1

    def print_chipher(self): # перерисовка полей шифра гена в графической части программы
        tmp=0
        for element in self.text_of_a_chipher:
            element.configure(state='normal')
            element.configure(text="cipher "+str(self.population[tmp].cipher_text))
            element.configure(state='disable')
            tmp+=1
        tmp=0
        for element in self.text_of_a_dechipher:
            element.configure(state='normal')
            element.configure(text="decipher "+str(self.population[tmp].cipher_text))
            element.configure(state='disable')
            tmp+=1

mai=Mains()
```

Приложение В

Отрисовка графиков

```
from Cipher import *
import numpy as np
import tkinter
import threading
import time

from matplotlib.backends.backend_tkagg import (
FigureCanvasTkAgg, NavigationToolbar2Tk)
from matplotlib.backend_bases import key_press_handler
from matplotlib.figure import Figure
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

kkk="".join(alphabet) # список алфавита (английский алфавит, нижний регистр)

из класса Main приложения Б:
def gui_run(self):
        root = tkinter.Tk()
        fig=Figure(constrained_layout=True)
        self.ax=[]
        spec2 = gridspec.GridSpec(ncols=2, nrows=5, figure=fig)
        #создание областей графиков для каждой особи популяции
        self.ax.append(fig.add_subplot(spec2[0, 0]))
        self.ax.append(fig.add_subplot(spec2[0, 1]))
        self.ax.append(fig.add_subplot(spec2[1, 0]))
        self.ax.append(fig.add_subplot(spec2[1, 1]))
        self.ax.append(fig.add_subplot(spec2[2, 0]))
        self.ax.append(fig.add_subplot(spec2[2, 1]))
        self.ax.append(fig.add_subplot(spec2[3,0]))
        self.ax.append(fig.add_subplot(spec2[3,1]))
        self.ax.append(fig.add_subplot(spec2[4,0]))
        self.ax.append(fig.add_subplot(spec2[4,1]))
        for index in range(0,9):
                tmp=self.values[index]
                self.ax[index].plot(self.names, list(tmp),color='r', label='main')
                self.ax[index].plot(self.names,list(self.ciph)[:13], color='b', label='perf')
        plt.suptitle('Categorical Plotting')
        self.canvas = FigureCanvasTkAgg(fig, master=root)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(side=tkinter.TOP, fill=tkinter.BOTH, expand=1)

        self.toolbar = NavigationToolbar2Tk(self.canvas, root)
        self.toolbar.update()
        self.canvas.get_tk_widget().pack(side=tkinter.TOP, fill=tkinter.BOTH, expand=1)
```

```
        self.canvas.mpl_connect("key_press_event", self.on_key_press)
        tkinter.mainloop()

def for_graphic(self): # обновление графиков при изменении fitness_value
        while 1:
                if self.running:
                        for index in range(0,9):
                                self.population[index].beg_funk()
                                self.count_fittness()
                                self.competition()
                        for index in range(0,9):
                                self.ax[index].clear()
                                self.ax[index].plot(list(self.values[index])[:13],self.names,color='r',
                                label='main')
                                self.ax[index].plot(self.names,list(self.ciph)[:13], color='b', label='perf')
                                self.canvas.draw()
                                self.running=False
                else: time.sleep(1)

def count_fittness(self): # второй вариант расчета fitness_value с использованием графиков шифров
        for element in self.population:
                element.count_fittness_new(self.ciph)
```

Приложение Г

Проверка гена на выполнимость

```
#Тест на выполняемость гена при любых сочетаниях
#Цикл не должен прерываться
while True:
	popul=cipher('apricot','k')  #создание особи с текстом и ключом соответственно
	print(popul.gen_code)
	popul.beg_funk()

#Тест на реализуемость некой части из выборки простейших шифров
#результат после операций шифрования-расшифрования должен совпадать с изначальным вводом
popul=cipher('apricot','k')

gen_codes=[
('+','s',0,'k',0,'+','zs',1,'k',0,'m',4), #шифр Цезаря
('+','s',0,'k',0,'+','zs',1,'zk',1,'m',4), #шифр Виженера
('s',0,'zs',2,'m',1), #простейший шифр перестановки
('mo','s',0,'k',0,'mo','zs',1,'zk',1,'m',4), #шифр Вернама
('-' 's' 0 'k' 0 '-' 'zs' 1 'zk' 1 'm' 4), #шифр Бофора
('+' '*' 5 's' 0 '+' '*' 5 'zs' 1 'm'  4) # Аффинный шифр
]

for chipher in gen_codes:
	popul.gen_code=chipher
	popul.beg_funk()
	popul.decipher=True
	popul.beg_funk()
	print(popul.cipher_text)
	print(popul.decipher_text)
	print(popul.text)
	if not functools.reduce(lambda x, y : x and y, map(lambda p, q: p ==
q,popul.text,popul.decipher_text), True):
		print ("The outputs are not the same")
		break
```