МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского (ННГУ)»

Институт информационных технологий, математики и механики

Кафедра алгебры, геометрии и дискретной математики

Направление подготовки: «Прикладная математика и информатика»
Профиль подготовки: «Прикладная математика и информатика (общий профиль)»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

на тему:

«Алгоритм Ал.А. Маркова распознавания взаимной однозначности алфавитного кодирования»

Выполнил:
студент группы 3821Б1ПМоп1
Рымарь А.А.
Научный руководитель:
доцент кафедры АГДМ, к.ф.-м.н.
Смирнова Т.Г.

Нижний Новгород
2025

Содержание

1. Введение

Задача распознавания взаимной однозначности алфавитного кодирования заключается в определении возможности уникального восстановления исходного сообщения из закодированного. Эта задача была особенно актуальна в XX веке, во время активного развития теории кодирования. Взаимная однозначность широко применяется в задачах сжатия, передачи и хранения информации, где необходимо исключить неоднозначность декодирования.

Поиском алгоритмов распознавания взаимной однозначности алфавитного кодирования занимались многие ученые, включая Ал.А. Маркова - нижегородского математика, работающего в Горьковском институте физико-технических исследований. Свое решение он предложил в 1963 году. Несмотря на то, что в настоящее время на практике чаще пользуются другими способами распознавания взаимной однозначности, алгоритм Маркова до сих пор имеет большую теоретическую значимость.

Например, алгоритм Ал.А. Маркова рассматривается в курсах «Кодирование комбинаторных объектов» Нижегородского государственного педагогического университета имени Козьмы Минина (учебное пособие, 2017г.), «Теория кодирования» Нижегородского государственного архитектурно-строительного университета (учебное пособие, 2023г.), «Теория кодирования» Нижегородского государственного университета им. Н.И. Лобачевского (учебное пособие, 2008г.), «Теория кодирования» Московского государственного университета.

Это показывает, что алгоритм Ал.А. Маркова преподается во многих университетах на курсах, связанных с теорией кодирования. Целью данной работы является разработка приложения для распознавания взаимной однозначности алфавитного кодирования с помощью алгоритма Ал.А. Маркова для наглядного объяснения работы алгоритма Маркова студентам, проверки знаний студентов и составления тестов по теме взаимной однозначности алфавитного кодирования.

2. Основные понятия алфавитного кодирования

Введем понятие алфавитного кодирования:

Введем обозначения $A : \{\alpha_1, \ldots, \alpha_m\}$ - некоторый алфавит с основанием m, $B : \{\beta_1, \ldots, \beta_r\}$ - кодовый алфавит с основанием r. A^* и B^* - множества всех слов из букв алфавитов A и B, соответственно. Теперь каждой букве α_i из алфавита A сопоставим слово из букв кодового алфавита B, которое обозначим как u_i. Тогда каждому слову $\alpha_{i_1} \ldots \alpha_{i_s} \in A^*$ соответствует одно и только одно слово $u_{i_1} \ldots u_{i_s} \in B^*$. Система слов $\mathcal{U} : \{u_1, \ldots, u_m\}$ - отображение множества A^* в множество B^*. Это отображение будем называть алфавитным кодированием.

Алфавитное кодирование задается следующей схемой:

$$f : \begin{cases} \alpha_1 \to u_1, \\ \alpha_2 \to u_2, \\ \quad \ldots \\ \alpha_m \to u_m. \end{cases} \qquad u_i \in \{0, \ldots, 9\}^* \text{ для всех } i = \overline{1, m}\,.$$

Коды u_i, $i = \overline{1, m}$, будем называть элементарными, а их набор $\mathcal{U} : \{u_1, \ldots, u_m\}$ - кодом.

Различные сообщения будут кодироваться разными последовательностями, если отображение f - инъективно. Выполнение этого требования обеспечивает однозначность декодирования сообщений. Такое кодирование будем называть взаимно-однозначным.

Пример взаимно-однозначного алфавитного кодирования:

$$f_1 : \begin{cases} \alpha_1 \to 1, \\ \alpha_2 \to 0. \end{cases}$$

Схема f_1 задает взаимно-однозначное кодирование. Любое сообщение декодируется однозначно.

Пример алфавитного кодирования, не являющегося взаимно-однозначным:

$$f_2 : \begin{cases} \alpha_1 \to 1, \\ \alpha_2 \to 11. \end{cases}$$

Рассмотрим слово $\beta = 111$. Его можно декодировать тремя разными способами: $\alpha_1\alpha_2$, или $\alpha_2\alpha_1$, или $\alpha_1\alpha_1\alpha_1$, следовательно кодирование, задаваемое схемой f_2 не является взаимно-однозначным.

Таким образом, сложно переоценить важность взаимной однозначности в теории кодирования. Для непосредственной проверки взаимной однозначности необходимо в общем случае проверить бесконечное множество слов.

Проблема распознавания взаимной однозначности алфавитного кодирования решена нижегородским математиком Александром Александровичем Марковым (1963 г.).

3. Алгоритм Ал.А.Маркова распознавания взаимной-однозначности алфавитного кодирования

Для рассмотрения алгоритма нам нужно ввести некоторые понятия:

Пусть слово β имеет вид $\beta_1\beta_2$. Тогда β_1 называется префиксом слова β, а β_2 - суффиксом. При этом, если $0 < |\beta_1| < |\beta|$, то β_1 называется собственным префиксом слова β, а если $0 < |\beta_2| < |\beta|$, то β_2 - собственный суффикс.

Рассматривается некоторый код $\mathcal{U} = (u_1, u_2, \ldots, u_m)$. Составляется множество S_1 слов, обладающих следующим свойством: слово β является собственным префиксом некоторого элементарного кода u_i и одновременно собственным суффиксом некоторого u_j (i индекс может быть равным j). Введем обозначение $S = S_1 \cup \{\lambda\}$, где λ - пустое слово.

Далее сопоставляем коду $\mathcal{U}$ ориентированный граф G, который строится следующим образом: вершинами графа являются элементы множества S, а ребро между вершинами α и β присутствует в графе, если существует элементарный код u_i и последовательность элементарных кодов $P = u_{i_1}u_{i_2}\ldots u_{i_k}$, такие, что $u_i = \alpha u_{i_1}u_{i_2}\ldots u_{i_k}\beta$. Заметим, что P может быть пустой, если $\alpha \neq \lambda$ и $\beta \neq \lambda$. Представление элементарного кода u_i в виде $u_i = \alpha u_{i_1}u_{i_2}\ldots u_{i_k}\beta$ будем называть разложением элементарного кода u_i. Ребру (α, β) в графе G сопоставим метку $\alpha u_{i_1}u_{i_2}\ldots u_{i_k}\beta$.

Петля у вершины λ присутствует в графе тогда и только тогда, когда существует u_i и последовательнось элементарных кодов $u_{i_1}u_{i_2}\ldots u_{i_k}$, таких, что $u_i = u_{i_1}u_{i_2}\ldots u_{i_k}$ ($k \geqslant 2$).

Теорема. Алфавитный код $\mathcal{U}$ является взаимно-однозначным тогда и только тогда, когда в графе G отсутствует ориентированные циклы, проходящие через вершину λ.

Доказательство.

Необходимость. Докажем необходимость от обратного: допустим, что в ориентированном графе G существует цикл, проходящий через вершину λ.

1. Если есть петля в вершине λ, то из алгоритма построения графа G следует, что существует кодовое слово $u_i = \lambda u_{i_1}\ldots u_{i_k}\lambda$ ($k \geqslant 2$). Тогда эта последовательность декодируется неоднозначно, так как $u_i = u_{i_1}\ldots u_{i_k}$.

То есть мы нашли пример слова, которое декодируется двумя разнымы способами, сдедовательно код U не является взаимно-однозначным.

2. Пусть в графе G нет петли в вершине λ, тогда существует ориентированный цикл, начинающийся и заканчивающийся в вершине λ: $\lambda\beta_1\beta_2\ldots\beta_n\lambda$, причем $\beta_j \neq$

$\lambda, j \in [1, n]$.

Для удобства обозначим λ как β_0, тогда рассматриваемый ориентированный цикл будет иметь следующий вид: $\beta_0\beta_1 \dots \beta_n\beta_0$, $\beta_j \neq \beta_0$ для $j \in [1, n]$.

По принципу построения графа G получим, что $\forall j\ \exists P_j : \beta_j P_j \beta_{j+1}$ - кодовое слово и $P_j = u_{j_1}u_{j_2}\dots u_{j_k}$ (P_j может быть и пустым словом (λ), кроме $j = 0, n$.

Рассмотрим слово $\beta_0 P_0 \beta_1 P_1 \beta_2 \dots \beta_{n-1} P_{n-1} \beta_n P_n \beta_0$. Его можно разбить на кодовые слова двумя разными способами:

1 способ: $\beta_0 P_0 \beta_1 \mid P_1 \mid \beta_2 P_2 \beta_3 \mid P_3 \mid \dots \mid P_{n-1} \mid \beta_n P_n \beta_0$.
2 способ: $P_0 \mid \beta_1 P_1 \beta_2 \mid P_2 \mid \beta_3 P_3 \mid \dots \mid \beta_{n-1} P_{n-1} \beta_n \mid P_n$.

Получили два различных разбиения для рассматриваемого слова (в случае четного n).

При нечетном n - аналогично существует 2 различных разбиения. Получается что рассматриваемое слово в любом случае имеет 2 расшифровки, следовательно код $\mathcal{U}$ не является взаимно-однозначным.

Достаточность.

Пусть код $\mathcal{U}$ не является взаимно-однозначным. Будем рассматривать минимальное по длине слово, допускающее две или более расшифровок. Для проведения доказательства нам не нужен конкретный пример слова, поэтому обозначим его как отрезок. А два варианта разбиения - как точки черного и синего цвета (номера 1 и 2 соответсвенно).

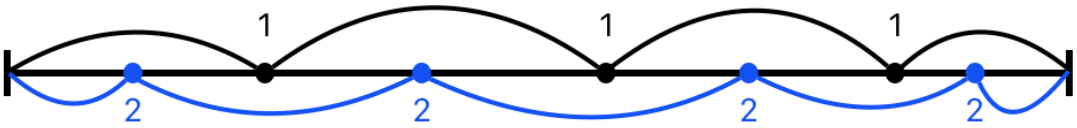

Рис. 1. Пример представления слова для хода доказательства.

Точки 1 и 2 не могут совпасть, так как если бы они совпали, то существовало бы более короткое слово, допускающее несколько расшифровок (а мы уже взяли минимальное по длине слово).

Рассмотрим слова, заключенные между точками разных разбиений:

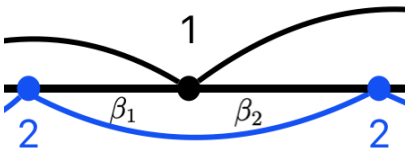

Рис. 2. Обозначение слов, заключенных между двумя разными разбиениями.

Можно заметить, что слова, заключенные между двумя разбиениями (обозначены как β_1 и β_2 являются одновременно собственными суффиксами и собственными

префиксами некоторых элементарных кодов, следовательно β_1 и β_2 являются вершинами ориентированного графа G.

Далее рассмотрим несколько вариантов расположения разбиений 1 и 2 относительно друг друга и дадим интерпретацию этих ситуаций на графе.

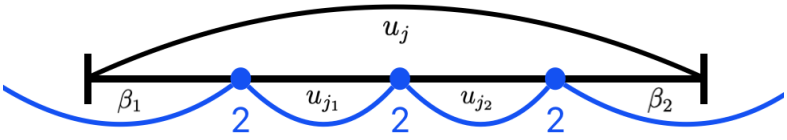

Рис. 3. Между точками разбиения 1 лежит несколько точек разбиения 2 и 2 неполных куска.

β_j и β_{j+1} являются вершинами в графе G, они соеденены ребром с меткой $u_{j_1}u_{j_2}$, так как $u_j = \beta_j u_{j_1} u_{j_2} \beta_{j+1}$.

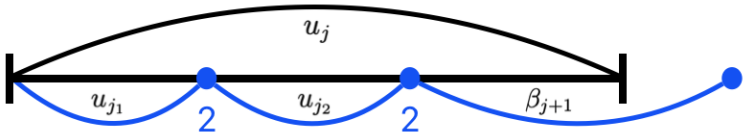

Рис. 4. Между точками разбиения 1 сначала лежат несколько точек 2, потом лежит неполный кусок разбиения 2.

β_{j+1} является вершиной графа. Также в графе присутствует ребро (λ, β_{j+1}) с меткой $u_{j_1}u_{j_2}$, так как $u_j = \lambda u_{j_1} u_{j_2} \beta_{j+1}$.

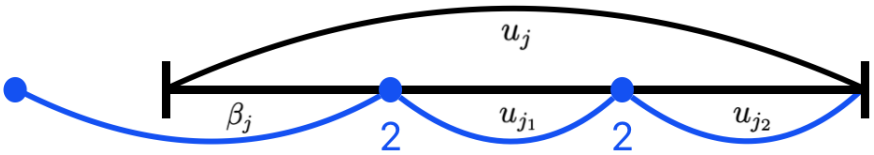

Рис. 5. Между точками разбиения 1 сначала лежит неполный кусок разбиения 2, потом лежат несколько точек 2.

β_j является вершиной графа. Также в графе присутствует ребро (β_j, λ) с меткой $u_{j_1}u_{j_2}$, так как $u_j = \lambda u_{j_1} u_{j_2} \beta_{j+1}$.

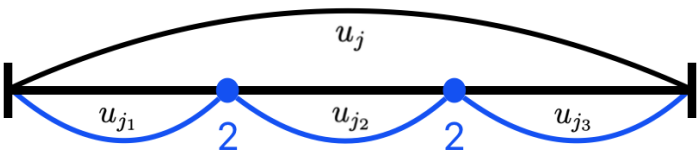

Рис. 6. Между точками разбиения 1 лежат несколько точек разбиения 2.

В этом случае в графе G существует петля в вершине λ, так как $u_j = \lambda u_{j_1} u_{j_2} u_{j_3} \lambda$.

Таким образом, если код $\mathcal{U}$ не является взаимно однозначным, то слово, допускающее две расшифровки, соответствует ориентированному циклу, начинающемуся и заканчивающемуся в вершине λ: $\lambda\beta_1\beta_2\ldots\beta_n\lambda$. ■

4. Примеры работы алгоритма Маркова

4.1. Пример 1

Пусть $\mathcal{U} = \{1, 100, 0001, 010, 0010\}$. Выясним, является ли код $\mathcal{U}$ взаимно-однозначным.
Построим множества S_1 и $S : S_1 = \{1, 0, 10, 01, 00, 001\}$, $S = \{1, 0, 10, 01, 00, 001, \lambda\}$.
Выпишем все нетривиальные разложения для элементарных кодов u_i :

Для u_1 нет нетривиальных разложений.
$u_2 = 100 = \lambda u_1 00 = 1\lambda 00 = 10\lambda 0$,
$u_3 = 0001 = 0\lambda 001 = 00\lambda 01$,
$u_4 = 010 = 0u_1 0 = 0\lambda 10 = 01\lambda 0$,
$u_5 = 0010 = 00u_1 0 = 0u_4\lambda = 00\lambda 10 = 001\lambda 0$.

Построим граф G:

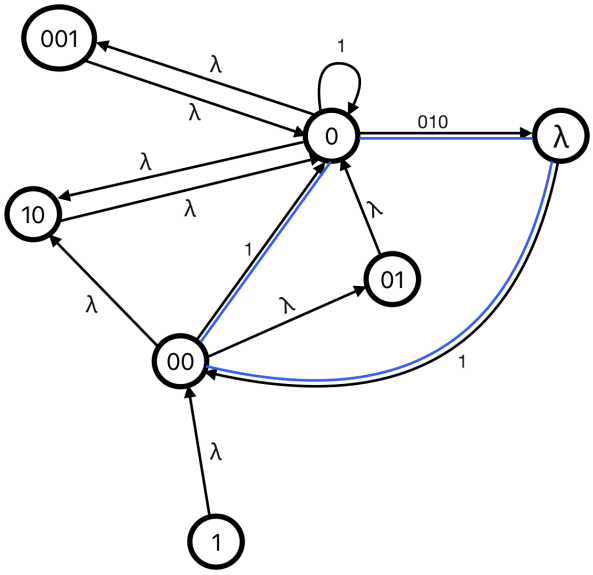

Рис. 7. Граф G, соответствующий коду $\mathcal{U} = \{1, 100, 0001, 010, 0010\}$.

В графе присутствует ориентированный цикл, проходящий через вершину λ, следовательно, код $\mathcal{U}$ не является взаимно-однозначным.

Построим по графу последовательность, допускающую две расшифровки. Начиная с вершины λ, будем приписывать друг к другу последовательности, соответствующие вершинам и ребрам графа, вдоль найденного цикла.

Слово $\gamma = 10010010$ допускает две расшифровки: $b_2 b_1 b_5$ и $b_1 b_5 b_4$.

4.2. Пример 2

Пусть $\mathcal{U} = \{1, 01, 10, 100, 001, 010\}$. Выясним, является ли код $\mathcal{U}$ взаимно-однозначным.
Построим множества S_1 и S : $S_1 = \{1, 0, 10, 01\}$, $S = \{1, 0, 10, 01, \lambda\}$.
Выпишем все нетривиальные разложения для элементарных кодов u_i :

Для u_1 нет нетривиальных разложений.
$u_2 = 01 = 0u_1\lambda = 0\lambda 1$,
$u_3 = 10 = \lambda u_1 0 = 1\lambda 0$,
$u_4 = 100 = \lambda u_3 0 = 10\lambda 0$,
$u_5 = 001 = 0u_2\lambda = 0\lambda 01$,
$u_6 = 010 = \lambda u_2 0 = 0u_3\lambda = 01\lambda 0 = 0\lambda 10$.

Построим граф G:

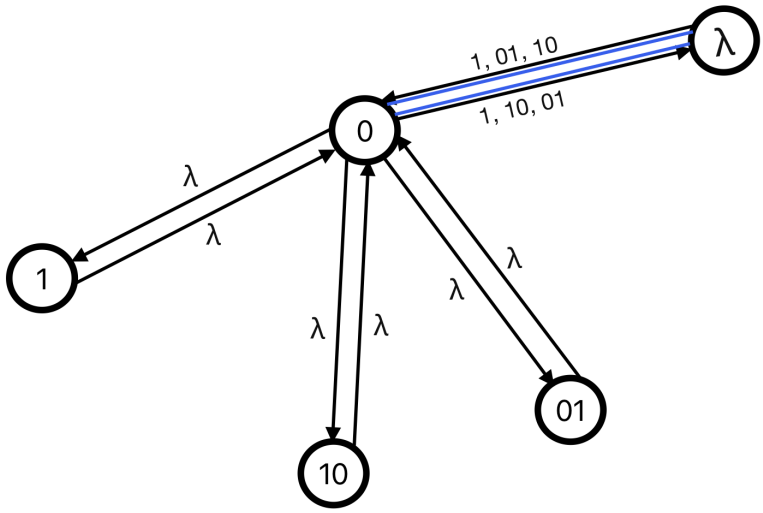

Рис. 8. Граф G, соответсвующий коду $\mathcal{U} = \{1, 01, 10, 100, 001, 010\}$.

В графе присутствует ориентированный цикл, проходящий через вершину λ, следовательно, код $\mathcal{U}$ не является взаимно-однозначным.

Построим по графу последовательность, допускающую две расшифровки. Начиная с вершины λ, будем приписывать друг к другу последовательности, соответствующие вершинам и ребрам графа, вдоль найденного цикла.

Слово $\gamma = 101$ допускает три расшифровки: b_3b_1, b_1b_2 и b_6.

4.3. Пример 3

Пусть $\mathcal{U} = \{10, 01, 100, 0100, 0000\}$. Выясним, является ли код $\mathcal{U}$ взаимно-однозначным.
Построим множества S_1 и S : $S_1 = \{1, 0, 00, 000\},\ S = \{1, 0, 00, 000, \lambda\}$.
Выпишем все нетривиальные разложения для элементарных кодов u_i :

$u_1 = 10 = 1\lambda 0,$
$u_2 = 01 = 0\lambda 1,$
$u_3 = 100 = \lambda u_1 0 = 1\lambda 00,$
$u_4 = 0100 = \lambda u_2 00 = 0u_1 0 = 0u_3\lambda,$
$u_5 = 00\lambda 00 = 0\lambda 000 = 000\lambda 0.$

Построим граф G:

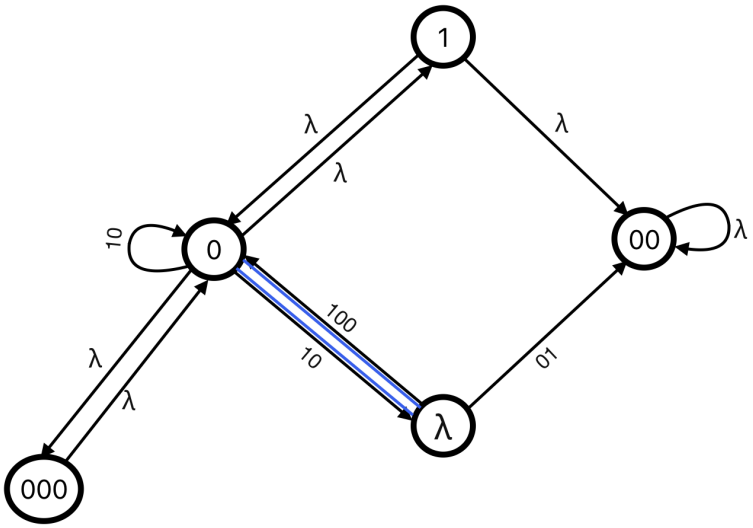

Рис. 9. Граф G, соответствующий коду $\mathcal{U} = \{10, 01, 100, 0100, 0000\}$.

В графе присутствует ориентированный цикл, проходящий через вершину λ, следовательно, код $\mathcal{U}$ не является взаимно-однозначным.

Построим по графу последовательность, допускающую две расшифровки. Начиная с вершины λ, будем приписывать друг к другу последовательности, соответствующие вершинам и ребрам графа, вдоль найденного цикла.

Слово $\gamma = 100100$ допускает две расшифровки: $b_3 b_3$ и $b_1 b_4$.

5. Разработка приложения

5.1. Архитектура программы

Программа написана на языке программирования Python с использованием модульного подхода, который обеспечивает ясность структуры, простоту поддержки и масштабируемость. Структура программы имеет следующий вид:

1. gui.py (Реализация графического интерфейса пользователя):
    - Этот модуль отвечает за визуальное взаимодействие пользователя с программой.
    - Отделение графического интерфейса позволяет легко вносить изменения в дизайн или добавлять новые элементы без изменения основной логики программы.
2. logic.py (Реализация алгоритма А.А Маркова):
    - В этом модуле реализована основная логика программы.
    - Такой подход позволяет легко модифицировать или тестировать логику независимо от интерфейса.
    - Это способствует повторному использованию кода, если потребуется реализовать другие интерфейсы.
3. main.py (Точка входа в программу):
    - Этот файл является основным скриптом, который объединяет интерфейс и логику.

Для поддержания кроссплатформенности приложения и удобства его запуска, воспользуемся библиотекой *PyInstaller*, которая создает запускаемые файлы для операционных систем Windows и MacOS. Это позволит запускать программу на разных платформах без необходимости открывать Python.

Рассмотрим структуру и содержание каждого файла программы более детально.

5.2. Программная реализация алгоритма А.А Маркова

Основная идея алгоритма следующая: на вход подается некоторый код: $\mathcal{U} = (u_1, u_2, \ldots, u_m)$, в результате работы алгоритма рисуется граф G, вершинами которого являются элементы множества S. Если код $\mathcal{U}$ не является взаимно-однозначным и, соответственно, в графе G присутствуют ориентированные циклы, проходящие через вершину λ, то ребра минимального по длине ориентированного цикла обозначаются синим цветом.

В процессе работы алгоритма строится множество S, элементы которого являются собственными суффиксами и собственными префиксами элементарных кодов.

Основная функция "markov_alg"с реализацией алгоритма Маркова получает на вход строку проверямого кода. Функция возвращает сам граф, переменную-флаг - обозначение является ли код взаимно-однознычным или нет, множество S, множества искривленных и прямых ребер графа и соответсвующие им цвета, имена ребер, а так же слово, допускающее две расшифровки.

Рассмотрим реализацию более детально.

Для начала нам нужно получить множество S.

Заполняем массив V строками, представляющими собой элементарные коды $u_1, u_2, \ldots, u_m$ с помощью методов *replace* и *split*, примененным к строке *Code*, переданной в функцию.

```
V = []

str = input("Input the alphabet: ")
V = str.replace(',', '').split(' ')
```

Рис. 10. Запись кода в переменную V.

Теперь нам надо составить множество $S1$. Для каждого слова проверяем каждый собственный суффикс: если его уже нет в множестве $S1$ и если он является так же и собственным постфиксом для какого-нибудь элементарного кода $u_1, u_2, \ldots, u_m$, тогда добавляем этот собственный суффикс в множество $S1$.

Далее добавляем пустое слово λ в множество $S1$ и получаем множество S из вершин которого будет строится граф G.

```
S1 = []
for code in V:
    n = len(code)
    for i in range(1,n):
        substr = code[:i]
        if substr not in S1 and isSuffAndPostf(V, substr):
            S1.append(substr)


Sl = S1 + ['λ']
S = S1 + ['']
print("S = {", Sl, "}")
```

Рис. 11. Построения множеств S_1 и S.

Проверка на то, что подстрока является и собственным суффиксом, и собственным префиксом какого либо кода из массива V (множества $\mathcal{U}$) происходит в функции "isSuffandPostf"с помощью встронных функций "startswith"и "endswith".

```
def isSuffAndPostf(code, substr):
    isSuf = False
    isPostf = False

    for word in code:
        if word.startswith(substr, 0, -1) == True:
            isSuf = True
        if word.endswith(substr, 1) == True:
            isPostf = True

    if isSuf == True and isPostf == True:
        return True
    else:
        return False
```

Рис. 12. Поиск элементов множества S_1.

Так как нам надо отслеживать наличие ориентированных циклов, проходящих через вершину λ, создадим ориентированный граф G и добавим вершины из множества S, используя библиотеку "networkx".

```
G = nx.DiGraph()

for each in Sl:
    G.add_node(each)
```

Рис. 13. Добавление вершин в граф G.

Вершины α и β графа G соединяются ориентированным ребром (α, β), если существует элементарный код u_i и последовательность элементарных кодов $P = u_{i_1}u_{i_2}\dots u_{i_k}$, такие, что $u_i = \alpha u_{i_1}u_{i_2}\dots u_{i_k}\beta$. Представление элементарного кода u_i в виде $u_i = \alpha u_{i_1}u_{i_2}\dots u_{i_k}\beta$ будем называть разложением элементарного кода u_i.

Для поиска разложений элементарных кодов реализуем рекурсивную функцию "findElementaryCecompositions":

```
def findElementaryDecompositions(target, arr, prefix="", result=[], count=0):
    if not target:
        result.append(prefix)
        return
    for word in arr:
        if target.startswith(word):
            findElementaryDecompositions(target[len(word):], arr, prefix + word, result)
            count += 1
```

Рис. 14. Функция для поиска разложений элементраных кодов.

Для добавления всех нужных ребер в наш граф G реализовали цикл:

```
for word in V:
    for start in S:
        for end in S:
            if start != '' and end != '' and start + end == word:
                G.add_edge(start, end, name='λ', color='black')
            elif word.startswith(start, 0, len(word)-1) and word.endswith(end, 1):
                counter = 0
                nameOfEdge = []
                findElementaryDecompositions(word[len(start):len(word)-len(end)], V, "", nameOfEdge, counter)
                if len(nameOfEdge) != 0 and nameOfEdge[0] != '':
                    if start == end and start != '':
                        G.add_edge(start, end, name = nameOfEdge[0], color='black')
                    elif start == '' and start == end:
                        if counter > 1:
                            G.add_edge('λ', 'λ', name=nameOfEdge[0], color='black')
                    else:
                        if start == '':
                            G.add_edge('λ', end, name=nameOfEdge[0], color='black')
                        elif end == '':
                            G.add_edge(start, 'λ', name=nameOfEdge[0], color='black')
                        else:
                            G.add_edge(start, end, name=nameOfEdge[0], color='black')
```

Рис. 15. Цикл для добавления ребер в граф G.

Далее находим все циклы графа с помощью функции "simple_cycles", встроенной в библиотеку networkx. Потом находим наименьшей из циклов, проходящий через вершину λ. Если такой цикл существует, то выводим его в консоль, если нет - присваиваем переменной "flag"единицу.

```
cycles = nx.simple_cycles(G)

cycleThrowLambda = None
lenmax = 10000

for cycle in cycles:
    if 'λ' in cycle:
        if len(cycle) < lenmax:
            cycleThrowLambda = cycle
            lenmax = len(cycle)

if cycleThrowLambda != None:
    print("Найден цикл:", cycleThrowLambda)
else:
    flag = 1
```

Рис. 16. Поиск минимального по длине цикла, проходящего через вершину λ.

Теперь, в случае, если код не является взаимно-однозначным($flag = 0$), мы можем получить слово, допускающее две расшифровки:

```
word = ""
edge_labels = nx.get_edge_attributes(G, 'name')


if flag == 0:
    n = len(cycleThrowLambda)

    if n != 2:
        word += cycleThrowLambda[0]
        for i in range(0, n-1):
            word += edge_labels[(cycleThrowLambda[i], cycleThrowLambda[i+1])] + cycleThrowLambda[i+1]

        word += edge_labels[(cycleThrowLambda[n-1], cycleThrowLambda[0])]

    else:
        word += cycleThrowLambda[0] + edge_labels[(cycleThrowLambda[0],
                                                   cycleThrowLambda[1])] + cycleThrowLambda[1] + edge_labels[(cycleThrowLambda[1],
                                                                                                              cycleThrowLambda[0])]
word = word.replace('λ', '')
```

Рис. 17. Поиск слова, допускающего больше одной расшифровки.

Создадим множества(вектора) изогнутых и прямых ребер. Это разделение необходимо для корректного отображения имен ребер:

```
curveEdges = []
straightEdges = []

for u,v in G.edges():
    if (u,v) in G.edges() and (v,u) in G.edges() and edge_labels[(u,v)] != edge_labels[(v,u)]:
        if (u,v) not in curveEdges:
            curveEdges.append((u,v))
        if (v,u) not in curveEdges:
            curveEdges.append((v,u))
    elif (u,v) in G.edges():
        if (u,v) not in straightEdges:
            straightEdges.append((u,v))
    elif (v,u) in G.edges():
        if (v,u) not in straightEdges:
            straightEdges.append((v,u))
```

Рис. 18. Создание двух типов ребер для корректного изображения графа.

Далее, создадим вектор цветов ребер(синий - если ребро входит в искомый цикл, черный - если нет):

```
if flag == 0:
    edge_colors1 = ['blue' if u in cycleThrowLambda and v in cycleThrowLambda
                    and u!= v and ((u,v) in curveEdges or (v,u) in curveEdges)
                    else 'black' for (u, v) in curveEdges]
    edge_colors2 = ['blue' if u in cycleThrowLambda and v in cycleThrowLambda
                    and u!= v and ((u,v) in straightEdges or (v,u) in straightEdges)
                    else 'black' for (u, v) in straightEdges]
else:
    edge_colors1 = ['black' for (u, v) in curveEdges]
    edge_colors2 = ['black' for (u, v) in straightEdges]


return G, flag, Sl, curveEdges, straightEdges, edge_colors1, edge_colors2, edge_labels, word
```

Рис. 19. Создание двух цветов ребер для выделения цикла через λ.

5.3. Реализация пользовательского интерфейса

Для разработки GUI, я воспользовался библиотекой *customtkinter* - используется для создания современного и настраиваемого графического интерфейса(надстройка над встроенной в питон библиотеки *tkinter*).

Для отрисовки графов удобно использовать библиотеку *matplotlib*, а также *matplotlib.backends.backend_tkagg* - для возможности встраивать графики *matplotlib* в интерфейс *tkinter*.

В данной ситуации удобно воспользоваться объектно-ориентированным подходом. Создадим три класса:

- App - главный класс приложения, объединяющий Input_View и Graph_frame в единый интерфейс.
- Input_View - основной пользовательский интерфейс для взаимодействия с алгоритмом. Содержит поле для ввода данных или загрузки кода из файла, кнопки для запуска обработки и очистки текста, отображение результатов работы алгоритма, включая множество S, информацию о взаимной однозначности кода и возможных проблемных словах.
- Graph_frame - отвечает за отображение графа, используя библиотеку *matplotlib*. Граф представлен в виде узлов и рёбер, с различными стилями (кривые, прямые) и подписями.

Далее, рассмотрим реализацию более подробно:
Реализация класса App, который создает главное окно и две frame:

```
class App(CTk.CTk):

    def __init__(self):
        super().__init__()
        CTk.set_appearance_mode("system")

        self.geometry("980x549+200+150")
        self.title("Алгоритм Маркова распознавания взаимной однозначности алфавитного кодирования")

        self.input_frame = Input_View(master=self)
        self.input_frame.place(relx = 0.005, rely =0.005, relwidth = 0.41, relheight = 0.99)


        self.graph_frame = Graph_frame(master=self)
        self.graph_frame.place(relx = 0.415, rely = 0.005, relwidth=0.58, relheigh = 0.99)

        self.input_frame.graphic = self.graph_frame
```

Рис. 20. Создания класса App.

Реализация класса $Graph\_frame$ в котором происходит отрисовка графа при нажатии кнопки START:

Для начала, напишем инициализатор:

```
class Graph_frame(CTk.CTkFrame):

    def __init__(self, master):
        super().__init__(master)

        self.graphic = CTk.CTkFrame(self)
        self.graphic.pack(expand=True, fill=BOTH)
```

Рис. 21. Создание класса Graph_frame.

Для отрисовки самого графа напишем функцию $draw$. Создаем отдельную функцию, для возможности вызывать ее из другого класса, в котором будет определена кнопка START. При визуализации графа используются методы библиотек $matplotlib$ и $networkx$:

```
def draw(self, G, curveEdges, straightEdges, edge_colors1, edge_colors2, edge_labels):

    for widget in self.graphic.winfo_children():
        widget.destroy()
    fig, ax = plt.subplots(figsize=(4, 4), dpi=115)
    pos = nx.circular_layout(G)

    nx.draw(G, pos, with_labels=True, node_size=1000, node_color="white",
            edgecolors='black', font_size=12, font_weight="bold", width=0, arrowsize=0.01, ax=ax)
    nx.draw_networkx_edges(G, pos,edgelist=curveEdges, connectionstyle="arc3,rad=0.15",
                           edge_color=edge_colors1, arrowsize=13, node_size=1000, ax=ax)
    nx.draw_networkx_edges(G, pos, edgelist=straightEdges, edge_color=edge_colors2,
                           width=1, arrowsize=13, node_size=1000, ax=ax)

    nx.draw_networkx_edge_labels(G, pos, edge_labels={(u,v):edge_labels[(u,v)] for (u,v) in curveEdges},
                                 connectionstyle="arc3,rad=0.16", ax=ax)
    nx.draw_networkx_edge_labels(G, pos, edge_labels={(u,v):edge_labels[(u,v)] for (u,v) in straightEdges},
                                 connectionstyle="arc3", ax=ax)

    canvas = FigureCanvasTkAgg(fig, master=self.graphic)
    canvas.draw()
    canvas.get_tk_widget().pack(fill="both", expand=True)
```

Рис. 22. Функция draw для отрисовки графа G.

Теперь рассмотрим реализацию класса $Input\_frame$. Определение всех виджетов тривиально, поэтому не будем включать их в рассмотрение. Более детально стоит рассмотреть работу фунции $start$, которая вызывается после нажатия одноименной кнопки:

Сначала строка с кодом записывается в переменную Vиз поля для ввода или из файла. Далее вызывается функция $markov\_alg$, определенная в logic.py.

```
def start(self, master):
    if self.flag == 0:
        V = self.entry.get()
    else:
        V = self.content

    G, fl, Sl, cE, sE, ec1, ec2, el, w = markov_alg(V)
```

Рис. 23. Функция, вызываемая при нажатии кнопки Start.

Далее, в зависимости от того, выводится решение о взаимной однозначности кода(с красным фоном - если код не взаимно-однозначный, и зеленым - в противоположном случае):

```
textt = None
colour = None
if fl == 1:
    textt="Код является взаимно-однозначным."
    colour="#4E6C50"
else:
    textt="Код не является взаимно-однозначным."
    colour="#EA5455"

if (len(self.framesol.winfo_children())>1):

    self.framesol.winfo_children()[1].destroy()
    self.frames.winfo_children()[1].destroy()

self.labels2 = CTk.CTkLabel (self.frames,
                             text="S = {" + ", ".join(Sl) + "}", font=("Arial", 16), fg_color="transparent")
self.labels2.place(relx=0.01, rely=0.5)

self.labelsol2 = CTk.CTkLabel (self.framesol,
                               text=textt, font=("Arial", 16), fg_color=colour, corner_radius=5)
self.labelsol2.place(relx=0.01, rely=0.5)
```

Рис. 24. Отрисовка поля с выводом о взаимной однозначности.

Затем поле, предназначенное для вывода слова, допускающего две расшифровки, заполняется, если код не является взаимно-однозначным и остается прозрачным в обратном случае:

```
if fl == 0:
    for widget in self.framew.winfo_children():
        widget.destroy()

    self.framew.configure(fg_color="#2B2B2B")

    self.labelw = CTk.CTkLabel(self.framew,
                               text="Слово, допускающее две расшифровки:", font=("Arial", 15))
    self.labelw.place(relx=0.01, rely=0.01)

    self.labelw2 = CTk.CTkLabel (self.framew, text=w, font=("Arial", 16))
    self.labelw2.place(relx=0.01, rely=0.5)

else:
    for widget in self.framew.winfo_children():
        widget.destroy()
    self.framew.configure(fg_color="transparent")
```

Рис. 25. Вывод слова, допускающего две расшифровки.

И, наконец, вызывается функция для отрисовки графа:

```
self.graphic.draw(G, cE, sE, ec1, ec2, el)
self.flag = 0
self.entry.insert(0, "")
```

Рис. 26. Вызов функции для отрисовки графа.

5.4. Файл для запуска всего приложения(main.py)

В main файле происходит создание объекта класса *App* и, таким образом, запуск всего приложения:

```
import gui


if __name__ == '__main__':

    app = gui.App()
    app.mainloop()
```

Рис. 27. Точка входа программы.

6. Тестирование приложения

Теперь, когда мы создали кроссплатформенное приложение для определения взаимной однозначности алфавитного кодирования с графическим интерфейсом, приступим к его тестированию.

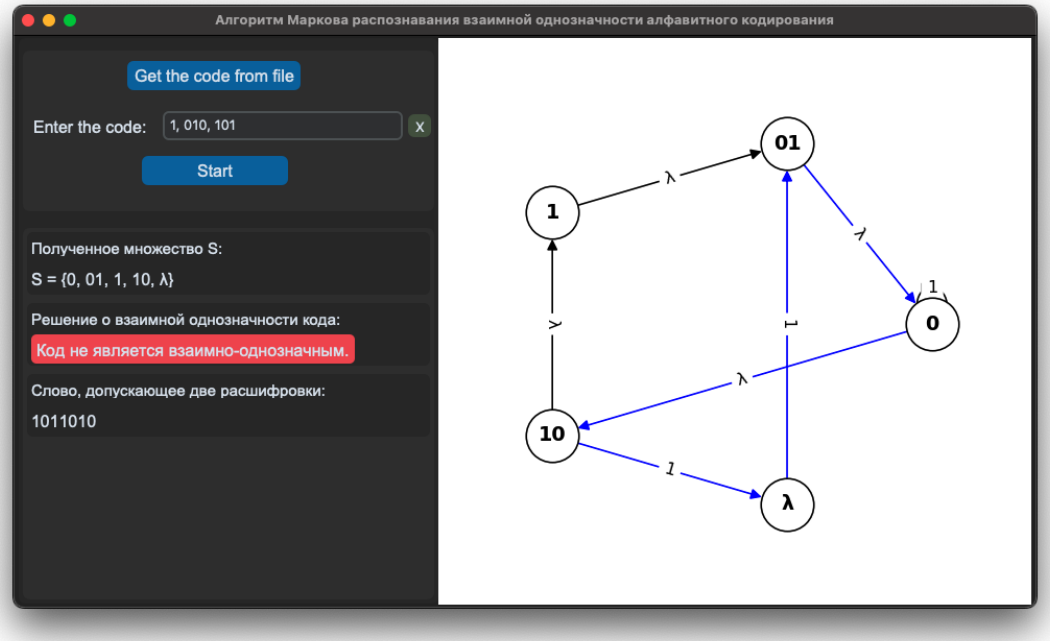

Рис. 28. Код $\mathcal{U} = \{1, 010, 101\}$ - не взаимно-однозначный.

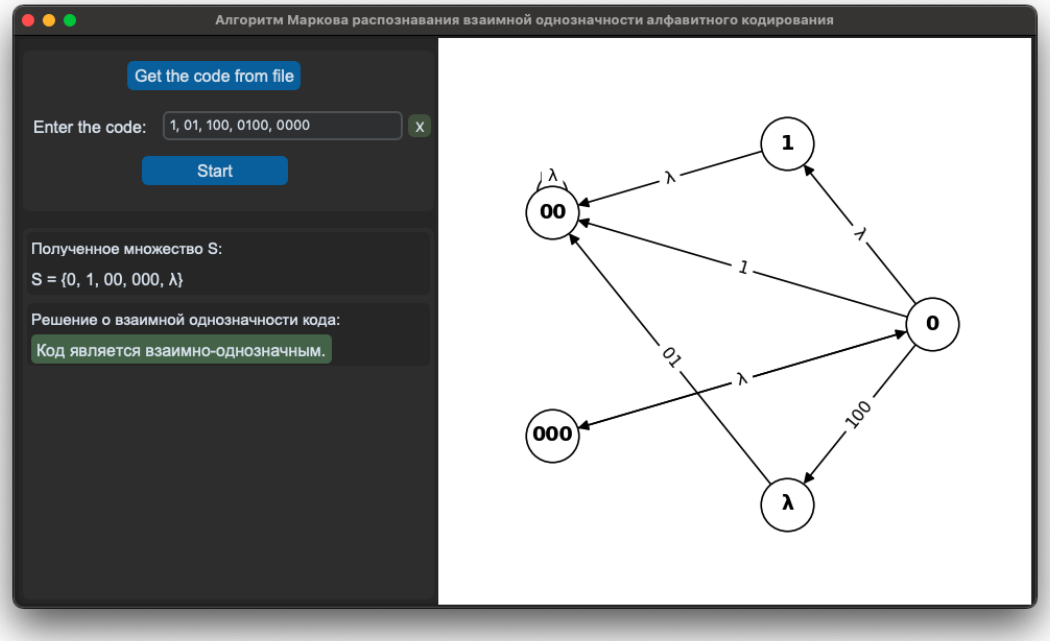

Рис. 29. Код $\mathcal{U} = \{1, 01, 100, 0100, 0000\}$ - взаимно-однозначный.

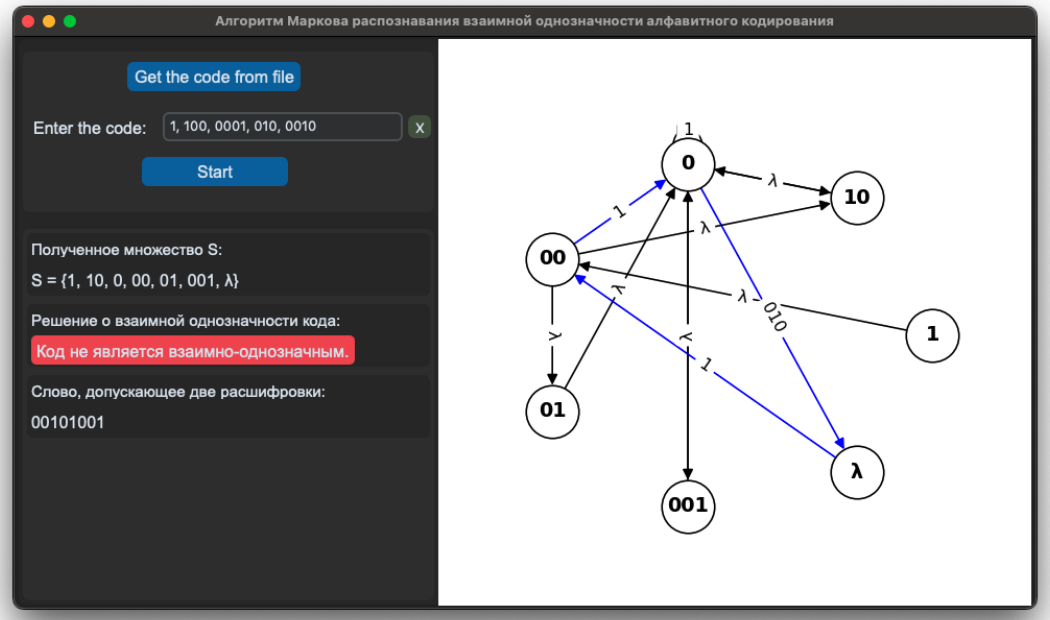

Рис. 30. Код $\mathcal{U} = \{1, 100, 0001, 010, 0010\}$ - не взаимно-однозначный.

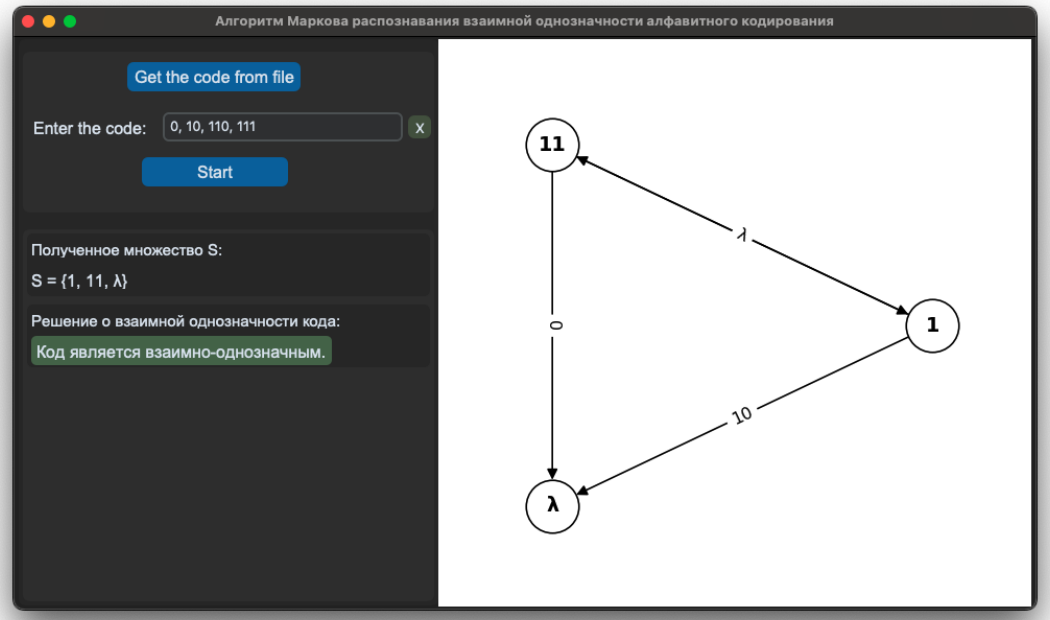

Рис. 31. Код $\mathcal{U} = \{0, 10, 110, 111\}$ - взаимно-однозначный.

В результате тестирования программы можно сделать вывод о том, что разработанное приложение функционирует корректно, успешно выполняя задачу проверки взаимной однозначности алфавитного кодирования. Программа корректно обрабатывает входные данные, визуализирует графы и предоставляет точные результаты анализа, что свидетельствует о её надежности и соответствии поставленным требованиям.

7. Руководство пользователя

При запуске программы открывается главное окно:

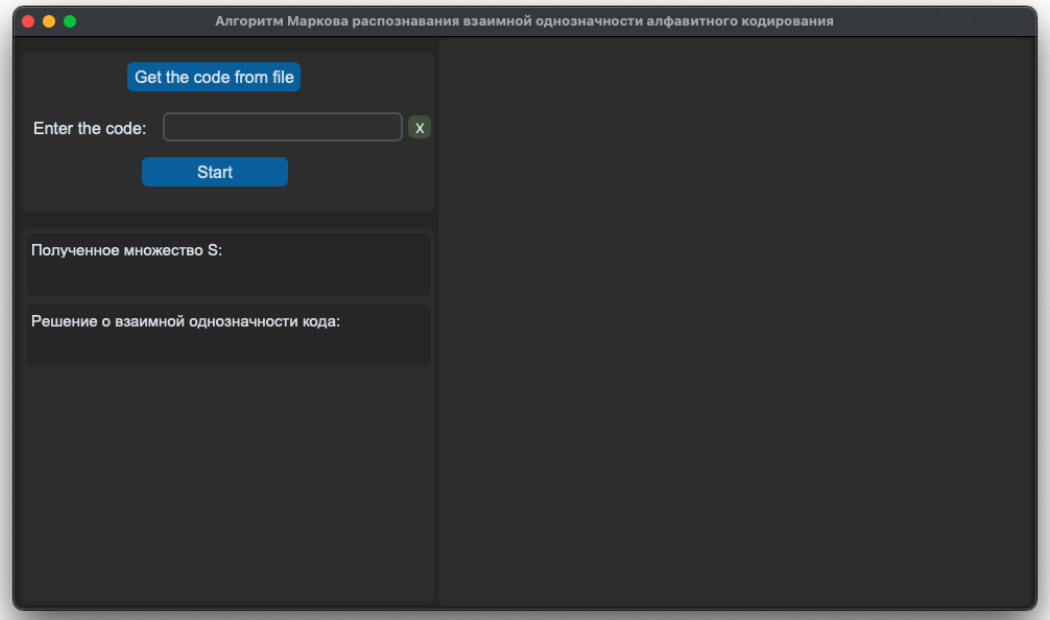

Рис. 32. Главное окно программы.

Пользователь может ввести код $\mathcal{U}$ двумя способами:

1. Ручной ввод: код должен иметь следующий вид - 1, 010, 101 (с запятой и пробелом после каждого кода)

2. Ввести код из файла: для этого надо нажать кнопку „Get the code from file“. После нажатия откроется проводник с возможностью выбрать .txt файл.

После нажатия кнопки „Start“ начинается проверка взаимной однозначности введенного кода.

Для удобного и быстрого очищения поля ввода кода с правой стороны от него есть маленькая кнопка (крестик). При нажатии на неё поле очищается.

8. Вывод

В рамках выполнения выпускной квалификационной работы было разработано приложение для проверки взаимной однозначности алфавитного кодирования с использованием алгоритма А.А. Маркова. В ходе работы были реализованы основные аспекты:

- Архитектура программы: модульная структура, обеспечивающая удобство сопровождения и масштабируемость.
- Реализация алгоритма Маркова: построение графа и проверка на наличие ориентированных циклов, подтверждающих нарушение взаимной однозначности.
- Пользовательский интерфейс: разработан графический интерфейс на базе библиотеки customtkinter для удобного взаимодействия пользователя с программой.
- Визуализация графов: использована библиотека matplotlib для наглядного отображения результатов.
- Расширение программы для случая общего алфавитного кодирования.

Результаты тестирования показали, что приложение корректно выполняет поставленную задачу, визуализирует графы и предоставляет точные результаты анализа, что подтверждает практическую применимость программы.

9. Список литературы

1. Марков Ал.А. Об алфавитном кодировании. I // Докл. АН СССР. — 1960. — Т. 132. No 3. — С. 521–523.

2. Смирнова Т.Г. О некоторых результатах Ал.А. Маркова в теории алфавитного кодирования // Материалы XIV Международного семинара «Дискретная математика и ее приложения» имени академика О. Б. Лупанова (Москва, МГУ, 20–25 июня 2022 г.). М.: ИПМ им. Келдыша. 2022. С. 279-282.

3. Л.П. Жильцова, Т.Г. Смирнова. Основы теории графов и теории кодирования в примерах и задачах. 2008. С. 38-45.

4. Ю.И. Галушкина, А.Н. Марьямов. Конспект лекций по дискретной математике. 2007. С. 104-113.

5. Н.Ю. Прокопенко. Теория кодирования. 2023. С. 23-28.

6. М.А.Иорданский, О.В.Смышляева. Кодирование комбинаторных объектов. 2017. С. 9-11.

7. Networkx doucmentation.

8. Customtkinter documentation.

9. Matplotlib documentation.

10. Приложения

Файл main.py:

```
import logic
import gui

if __name__ == '__main__':

    app = gui.App()
    app.mainloop()
```

Файл logic.py:

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque
import time

def isSuffAndPostf(code, substr):  # substr -
                                   word[:i]
    isSuf = False
    isPostf = False

    for word in code:
        if word.startswith(substr, 0, -1) == True:
            isSuf = True
        if word.endswith(substr, 1) == True:
            isPostf = True

    if isSuf == True and isPostf == True:
        return True
    else:
        return False

def findElementaryDecompositions(target, arr, flag, prefix="",
   result=[], count=0):
    if not target:
        result.append(prefix)
        if count > 1:
            flag[0] = True
        return
    for word in arr:
```

```
        if target.startswith(word):
            findElementaryDecompositions(target[len(word):], arr
                , flag, prefix + word, result, count+1)

def markov_alg(Code):
    flag = 0

    V = Code.replace(',', '').split('␣')
    print(V)

    start_time = time.perf_counter()
    S1 = []
    for code in V:
        n = len(code)
        for i in range(1,n):
            substr = code[:i]
            if substr not in S1 and isSuffAndPostf(V, substr):
                S1.append(substr)
    end_time = time.perf_counter()
    print(f"S_creation␣time:␣{end_time␣-␣start_time:.6f}␣seconds
        ")

    Sl = S1 + ['  ']
    S = S1 + ['']
    print("S␣=␣{", Sl, "}")

    G = nx.DiGraph()

    for each in Sl:
        G.add_node(each)

    start_time2 = time.perf_counter()
    for word in V:
        for start in S:
            for end in S:
                if start != '' and end != '' and start + end ==
                    word:
                    G.add_edge(start, end, name='  ', color='
                        black')
                elif word.startswith(start, 0, len(word)-1) and
```

```
                    word.endswith(end, 1):
                        counter = 0
                        nameOfEdge = []
                        flag2 = [False]
                        findElementaryDecompositions(word[len(start)
                            :len(word)-len(end)], V, flag2, "",
                            nameOfEdge, counter)
                        if len(nameOfEdge) != 0 and nameOfEdge[0] !=
                            '':
                            if start == end and start != '':
                                G.add_edge(start, end, name =
                                    nameOfEdge[0], color='black')
                            elif start == '' and start == end:
                                if flag2[0]:
                                    G.add_edge('  ', '  ', name=
                                        nameOfEdge[0], color='black')
                            else:
                                if start == '':
                                    G.add_edge('  ', end, name=
                                        nameOfEdge[0], color='black')
                                elif end == '':
                                    G.add_edge(start, '  ', name=
                                        nameOfEdge[0], color='black')
                                else:
                                    G.add_edge(start, end, name=
                                        nameOfEdge[0], color='black')
end_time2 = time.perf_counter()
print(f"Edge_adding_time: {end_time2 - start_time2:.6f}
    seconds")

start_time3 = time.perf_counter()
cycles = nx.simple_cycles(G)
end_time3 = time.perf_counter()
print(f"Cycles_finding_time: {end_time3 - start_time3:.6f}
    seconds")
# print(sorted(nx.simple_cycles(G)))

cycleThrowLambda = None
lenmax = 10000
```

```
for cycle in cycles:
    if '  ' in cycle:
        if len(cycle) < lenmax:
            cycleThrowLambda = cycle
            lenmax = len(cycle)

start_time4 = time.perf_counter()
if cycleThrowLambda != None:
    ind = cycleThrowLambda.index('  ')
    ctlt = deque(cycleThrowLambda)
    ctlt.rotate(len(cycleThrowLambda) - ind)
    cycleThrowLambda = list(ctlt)
    print("                 ␣          :", cycleThrowLambda)

else:
    flag = 1

end_time4 = time.perf_counter()
print(f"Cycle_through_lambda_finding_time:␣{end_time4␣-␣
   start_time4:.6f}␣seconds")

word = ""
edge_labels = nx.get_edge_attributes(G, 'name')


if flag == 0:
    n = len(cycleThrowLambda)

    if n != 2:
        word += cycleThrowLambda[0]
        for i in range(0, n-1):
            word += edge_labels[(cycleThrowLambda[i],
               cycleThrowLambda[i+1])] + cycleThrowLambda[i
               +1]

        word += edge_labels[(cycleThrowLambda[n-1],
           cycleThrowLambda[0])]

    else:
        word += cycleThrowLambda[0] + edge_labels[(
```

```
            cycleThrowLambda[0], cycleThrowLambda[1])] +
            cycleThrowLambda[1] + edge_labels[(
            cycleThrowLambda[1], cycleThrowLambda[0])]
word = word.replace('  ', '')



curveEdges = []
straightEdges = []

for u,v in G.edges():
    if (u,v) in G.edges() and (v,u) in G.edges() and
       edge_labels[(u,v)] != edge_labels[(v,u)]:
        if (u,v) not in curveEdges:
            curveEdges.append((u,v))
        if (v,u) not in curveEdges:
            curveEdges.append((v,u))
    elif (u,v) in G.edges():
        if (u,v) not in straightEdges:
            straightEdges.append((u,v))
    elif (v,u) in G.edges():
        if (v,u) not in straightEdges:
            straightEdges.append((v,u))

if flag == 0:
    edge_colors1 = ['blue' if u in cycleThrowLambda and v in
        cycleThrowLambda
                    and u!= v and ((u,v) in curveEdges or (v
                        ,u) in curveEdges)
                    else 'black' for (u, v) in curveEdges]
    edge_colors2 = ['blue' if u in cycleThrowLambda and v in
        cycleThrowLambda
                    and u!= v and ((u,v) in straightEdges or
                        (v,u) in straightEdges)
                    else 'black' for (u, v) in straightEdges
                        ]
else:
    edge_colors1 = ['black' for (u, v) in curveEdges]
    edge_colors2 = ['black' for (u, v) in straightEdges]
```

```
    return G, flag, Sl, curveEdges, straightEdges, edge_colors1,
        edge_colors2, edge_labels, word
```

Файл gui.py:

```
from logic import *
import customtkinter as CTk
from tkinter import *
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from tkinter import filedialog
import time

class Graph_frame(CTk.CTkFrame):

    def __init__(self, master):
        super().__init__(master)

        self.graphic = CTk.CTkFrame(self)
        self.graphic.pack(expand=True, fill=BOTH)

    def draw(self, G, curveEdges, straightEdges, edge_colors1,
       edge_colors2, edge_labels):

        for widget in self.graphic.winfo_children():
            widget.destroy()
        fig, ax = plt.subplots(figsize=(4, 4), dpi=115)
        pos = nx.circular_layout(G)

        nx.draw(G, pos, with_labels=True, node_size=1000,
           node_color="white",
                edgecolors='black', font_size=12, font_weight="
                   bold", width=0, arrowsize=0.01, ax=ax)
        nx.draw_networkx_edges(G, pos,edgelist=curveEdges,
           connectionstyle="arc3,rad=0.15",
                                edge_color=edge_colors1,
                                   arrowsize=13, node_size=1000,
                                   ax=ax)
        nx.draw_networkx_edges(G, pos, edgelist=straightEdges,
           edge_color=edge_colors2,
                                width=1, arrowsize=13, node_size
                                   =1000, ax=ax)
```

```
        nx.draw_networkx_edge_labels(G, pos, edge_labels={(u,v):
            edge_labels[(u,v)] for (u,v) in curveEdges},
                                     connectionstyle="arc3,rad
                                         =0.16", ax=ax)
        nx.draw_networkx_edge_labels(G, pos, edge_labels={(u,v):
            edge_labels[(u,v)] for (u,v) in straightEdges},
                                     connectionstyle="arc3", ax=
                                         ax)

        canvas = FigureCanvasTkAgg(fig, master=self.graphic)
        canvas.draw()
        canvas.get_tk_widget().pack(fill="both", expand=True)

class Input_View(CTk.CTkFrame):
    def __init__(self, master):
        super().__init__(master)

        self.flag = 0

        self.frame_1 = CTk.CTkFrame(self, height=20)
        self.frame_1.pack(padx = 4, pady = 12, ipadx = 8, ipady
            = 10, side = TOP)

        self.open_button = CTk.CTkButton(self.frame_1, text="Get
            the code from file", font=("Arial", 16), command=
            self.open_file)
        self.open_button.grid(row = 0, stick='w', columnspan=2,
            padx=100, pady=10)

        self.label = CTk.CTkLabel(self.frame_1, text="Enter the
            code:", font=("Arial", 16), )
        self.label.grid(row = 1, column = 0, stick = 'w', padx
            =10, pady=10)

        self.entry = CTk.CTkEntry(self.frame_1, width=230)
        self.entry.grid(row = 1, column = 1, padx = 5,stick = 'w
            ', pady = 10)

        self.clear_button = CTk.CTkButton(self.frame_1, height
```

```
        =20, width=20, text='X',
                                                fg_color="#4B5945",
                                                    hover_color="#66785
                                                    F", command=self.
                                                    clear_text)
        self.clear_button.grid(row = 1, column = 3, stick = 'e')


        self.button = CTk.CTkButton(self.frame_1, text="Start",
            font=("Arial", 16), command=lambda: self.start(master
            ))
        self.button.grid(row = 2, columnspan=2, padx=10, pady=5)

        self.frame_2 = CTk.CTkFrame(self)
        self.frame_2.pack(padx = 4, pady = 4, fill='both', side=
            TOP, expand=True)

        self.frames = CTk.CTkFrame(self.frame_2, height=60,)
        self.frames.pack(padx = 4, pady = 4, fill='x', side=TOP)

        self.labels = CTk.CTkLabel(self.frames, text="
                                    ␣                    ␣S:", font=("
            Arial", 15))
        self.labels.place(relx=0.01, rely=0.01)

        self.framesol = CTk.CTkFrame(self.frame_2, height=60,)
        self.framesol.pack(padx=4, pady=4, fill='x', side=TOP)

        self.labelsol = CTk.CTkLabel(self.framesol, text="
                            ␣  ␣                ␣
                                        ␣          :", font=("Arial",
             15))
        self.labelsol.place(relx=0.01, rely=0.01)

        self.framew = CTk.CTkFrame(self.frame_2, height=60,
            fg_color="transparent")
        self.framew.pack(padx=4, pady=4, fill='x', side=TOP)

    def start(self, master):
        # if self.flag == 0:
```

```
#       V = self.entry.get()
# else:
#       V = self.content

V = self.entry.get()

start_time = time.perf_counter()
G, fl, Sl, cE, sE, ec1, ec2, el, w = markov_alg(V)
end_time = time.perf_counter()
print(f"Execution time: {end_time - start_time:.6f}
    seconds")

textt = None
colour = None
if fl == 1:
    textt="       .
    colour="#4E6C50"
else:
    textt="       .
    colour="#EA5455"

if (len(self.framesol.winfo_children())>1):

    self.framesol.winfo_children()[1].destroy()
    self.frames.winfo_children()[1].destroy()

self.labels2 = CTk.CTkLabel (self.frames,
                             text="S = {" + ",".join(Sl
                                ) + "}", font=("Arial",
                                16), fg_color="
                                transparent")
self.labels2.place(relx=0.01, rely=0.5)

self.labelsol2 = CTk.CTkLabel (self.framesol,
                               text=textt, font=("Arial"
                                  , 16), fg_color=colour
                                  , corner_radius=5)
self.labelsol2.place(relx=0.01, rely=0.5)
```

```
        if fl == 0:
            for widget in self.framew.winfo_children():
                widget.destroy()

            self.framew.configure(fg_color="#2B2B2B")

            self.labelw = CTk.CTkLabel(self.framew,
                                       text="           ,␣

                                         ␣      ␣

                                         :", font=("Arial", 15)
                                         )
            self.labelw.place(relx=0.01, rely=0.01)

            self.labelw2 = CTk.CTkLabel (self.framew, text=w,
               font=("Arial", 16))
            self.labelw2.place(relx=0.01, rely=0.5)

        else:
            for widget in self.framew.winfo_children():
                widget.destroy()
            self.framew.configure(fg_color="transparent")

        self.graphic.draw(G, cE, sE, ec1, ec2, el)
        self.flag = 0
        self.entry.insert(0, "")

    def open_file(self):

        self.file_path = filedialog.askopenfilename(
            title="                ␣        ",
            filetypes=(("                ␣          ", "*.txt"
               ), ("      ␣          ", "*.*"))
        )

        if self.file_path:
            with open(self.file_path, 'r', encoding='utf-8') as
               file:
```

```
                self.content = file.read()
                # self.content = self.content[:-1]
                self.flag = 1
                self.entry.delete(0, CTk.END)
                self.entry.insert(0, self.content)

        print(self.content)


    def clear_text(self):
        self.entry.delete(0, CTk.END)


class App(CTk.CTk):

    def __init__(self):
        super().__init__()
        CTk.set_appearance_mode("system")

        self.geometry("980x549+200+150")
        self.title("                    ␣              ␣
                                        ␣                  ␣
                                        ␣                      ␣
                                    ")

        self.input_frame = Input_View(master=self)
        self.input_frame.place(relx = 0.005, rely =0.005,
           relwidth = 0.41, relheight = 0.99)


        self.graph_frame = Graph_frame(master=self)
        self.graph_frame.place(relx = 0.415, rely = 0.005,
           relwidth=0.58, relheigh = 0.99)

        self.input_frame.graphic = self.graph_frame
```