Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Институт информатики и вычислительной техники

Кафедра прикладной математики и кибернетики

# Лабораторная работа №6 по дисциплине

**Прикладная стеганография**

Выполнил:

студент гр.МГ-411 Шевельков П.С.

ФИО студента

«15» мая 2025 г.

Новосибирск 2025 г.

# Задание на лабораторную работу:

Написать программу, которая использует выходные текстовые файлы из программного средства стегоанализа, реализованного в задании №5 для подсчета ошибки 1 и 2 рода. Заполненные контейнеры взять из результатов работ №3-4.

Отчет по работе должен содержать описание формата вывода данных стегоанализа, подсчет ошибки 1 и 2 рода и таблицу сравнения методов стегоанализа. В таблице привести результаты стегоанализа при разном заполнении контейнеров, указав максимально возможную фактическую ёмкость контейнера и % заполнения стегоконтейнера. Например, 50% заполненный стегоконтейнер при последовательном заполнении; при рассеянном заполнении; также для 100%. Привести ссылку на исходники.

# Результаты работы программы:

# 

рисунок 1. Интерфейс программы.

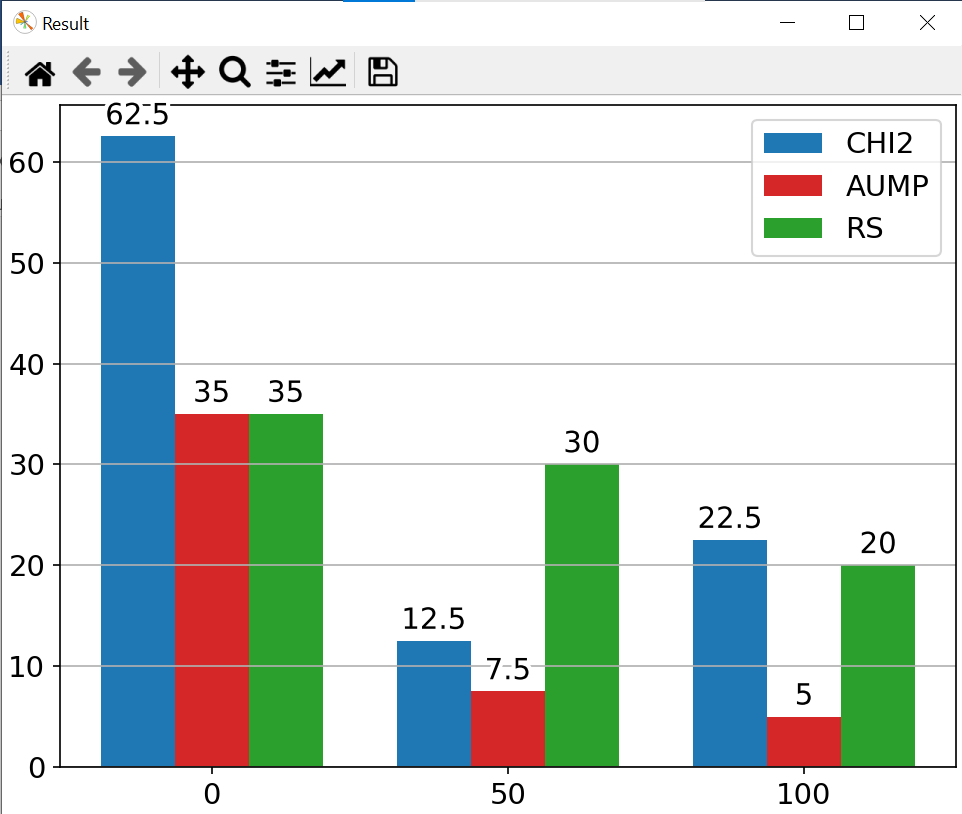


рисунок 2. Результат работы для выборки из 40 контейнеров.

**Описание программы по анализу изображений с внедрёнными сообщениями**

1. Загрузка набора изображений (в формате BMP).
2. Внедрение случайного битового сообщения в изображения с помощью одного из трёх методов LSB.
3. Проведение анализа изображений с различной степенью внедрения (0%, 50%, 100%).
4. Расчёт статистических метрик: Chi², AUMP, RS.
5. Отображение результатов в виде текстового отчёта и графика.
6. Возможность сохранить результаты в файл.

**Используемые методы внедрения (LSB):**

В проекте реализованы и сравниваются следующие методы внедрения сообщений:

1. LSB Pattern Matching (LSBSeq)

Последовательное внедрение битов в младшие биты пикселей.

Использует 3 младших бита на пиксель.

Простая и предсказуемая стратегия.

1. LSB Sequential (LSBMessage)

Вставка сообщений с учётом конкретной структуры изображения.

Предположительно, внедрение более устойчиво к обнаружению.

1. LSB Scaled

Масштабирует изображение перед внедрением.

Используется альтернативная стратегия внедрения и анализа.

Программа внедряет случайную битовую последовательность в каждое изображение в объёме:

* 0% (контрольная группа),
* 50%,
* 100% от максимально возможной вместимости.

Для каждой степени внедрения проводится анализ:

* Chi² — проверка распределения значений пикселей.
* AUMP — анализ шаблонов (параметризуемый).
* RS-анализ — метод, чувствительный к модификациям LSB.

Результаты нормализуются и отображаются в процентах.

**Архитектура программы:**

**1. MainWindow (главное окно):**

* Интерфейс на PyQt6.
* Содержит:
* Виджет для отображения изображений.
* Панель управления (выбор метода, параметры анализа).
* Отображение результатов.
* Поддерживает асинхронную обработку изображений через поток AnalyseWorker.

2. **AnalyseWorker:**

* Запускается в отдельном потоке.
* Осуществляет вставку сообщений и анализ изображений параллельно (через multiprocessing.Pool).
* Генерирует отчёт по каждому уровню внедрения (0%, 50%, 100%).

**3. LSBSeq, LSBMessage, LSBScaledMessage:**

* Реализация различных LSB-методов внедрения.
* Все используют библиотеку bitarray для работы с битовыми сообщениями.

**4. control.\*:**

* Модули анализа: ChiSquaredAnalysis, RSAnalysis, AUMPAnalysis.
* Модуль PlotBuilder — построение графиков результатов анализа.

**Выводы:**

* Программа позволяет визуализировать и количественно оценить изменения в изображениях в результате внедрения скрытых сообщений.
* Используемые статистические методы позволяют определить степень внедрения, что может быть полезно для задач стеганализa.
* Сравнение методов показывает, какие из них более устойчивы к обнаружению.

## **Ссылка на программу:**

<https://github.com/bothyD/steganograf>

**Листинг:**

import PIL.Image

from bitarray import bitarray

class LSBMessage:

    @staticmethod

    def inject\_message(

        img\_in: PIL.Image.Image, message\_bits: bitarray

    ) -> PIL.Image.Image:

        img = img\_in.copy()

        pixels\_in = img\_in.load()

        pixels = img.load()

        if pixels\_in is None or pixels is None:

            raise BaseException("pixels\_in is None")

        msg\_index = 0

        def index\_in():

            return msg\_index < len(message\_bits) - 1

        def mi():

            try:

                return (message\_bits[msg\_index] << 1) | message\_bits[msg\_index + 1]

            except BaseException:

                return 0b100

        for x in range(img.size[0]):

            for y in range(img.size[1]):

                if not index\_in():

                    break

                byte = pixels\_in[x, y]

                clp = (byte & 0b11000000) >> 6

                cmp = (byte & 0b01100000) >> 5

                crp = (byte & 0b00110000) >> 4

                if mi() == clp:

                    byte |= 1 << 2

                    msg\_index += 2

                else:

                    byte &= ~(1 << 2)

                if mi() == cmp:

                    byte |= 1 << 1

                    msg\_index += 2

                else:

                    byte &= ~(1 << 1)

                if mi() == crp:

                    byte |= 1

                    msg\_index += 2

                else:

                    byte &= ~(1)

                pixels[x, y] = byte

            else:

                continue

            break

        return img

    @staticmethod

    def extract\_message(img\_in: PIL.Image.Image, message\_bit\_len: int) -> bitarray:

        pixels\_in = img\_in.load()

        msg\_index = 0

        message\_bits = []

        if pixels\_in is None:

            raise BaseException("pixels\_in is None")

        for x in range(img\_in.size[0]):

            for y in range(img\_in.size[1]):

                if msg\_index >= message\_bit\_len:

                    break

                byte = pixels\_in[x, y]

                # Извлекаем старшие биты (по 2 бита каждый)

                clp = (byte & 0b11000000) >> 6

                cmp = (byte & 0b01100000) >> 5

                crp = (byte & 0b00110000) >> 4

                # Извлекаем флаги (младшие 3 бита)

                clp\_flag = (byte >> 2) & 1

                cmp\_flag = (byte >> 1) & 1

                crp\_flag = byte & 1

                # Если флаг установлен — значит соответствующий старший блок был частью сообщения

                if clp\_flag and msg\_index + 2 <= message\_bit\_len:

                    message\_bits.extend([(clp >> 1) & 1, clp & 1])

                    msg\_index += 2

                if cmp\_flag and msg\_index + 2 <= message\_bit\_len:

                    message\_bits.extend([(cmp >> 1) & 1, cmp & 1])

                    msg\_index += 2

                if crp\_flag and msg\_index + 2 <= message\_bit\_len:

                    message\_bits.extend([(crp >> 1) & 1, crp & 1])

                    msg\_index += 2

                if msg\_index >= message\_bit\_len:

                    break

            else:

                continue

            break

        return bitarray(message\_bits)

    @staticmethod

    def get\_max\_capacity(img\_in: PIL.Image.Image, message\_bits: bitarray):

        img = img\_in.copy()

        pixels\_in = img\_in.load()

        pixels = img.load()

        if pixels\_in is None or pixels is None:

            raise BaseException("pixels\_in is None")

        msg\_index = 0

        def index\_in():

            return msg\_index < len(message\_bits) - 1

        def mi():

            try:

                return (message\_bits[msg\_index] << 1) | message\_bits[msg\_index + 1]

            except BaseException:

                return 0b100

        for x in range(img.size[0]):

            for y in range(img.size[1]):

                if not index\_in():

                    break

                byte = pixels\_in[x, y]

                clp = (byte & 0b11000000) >> 6

                cmp = (byte & 0b01100000) >> 5

                crp = (byte & 0b00110000) >> 4

                if mi() == clp:

                    byte |= 1 << 2

                    msg\_index += 2

                else:

                    byte &= ~(1 << 2)

                if mi() == cmp:

                    byte |= 1 << 1

                    msg\_index += 2

                else:

                    byte &= ~(1 << 1)

                if mi() == crp:

                    byte |= 1

                    msg\_index += 2

                else:

                    byte &= ~(1)

                pixels[x, y] = byte

            else:

                continue

            break

        return msg\_index

import PIL.Image

from bitarray import bitarray

import numpy

SCALE\_COEF = 2

class LSBScaledMessage:

    @staticmethod

    def scale\_image(img\_in: PIL.Image.Image) -> PIL.Image.Image:

        img\_out = PIL.Image.new("L", (img\_in.size[0] \* 2, img\_in.size[1] \* 2))

        pixels\_in = img\_in.load()

        pixels\_out = img\_out.load()

        if pixels\_in is None or pixels\_out is None:

            raise BaseException("pixel arrays are none")

        BLOCKS = img\_in.size[0]

        for m in range(BLOCKS - 1):

            for n in range(BLOCKS - 1):

                x = m \* SCALE\_COEF

                y = n \* SCALE\_COEF

                pixels\_out[x, y] = pixels\_in[m, n]

                pixels\_out[x + 1, y] = (

                    pixels\_in[m, n] + pixels\_in[m + 1, n]

                ) // SCALE\_COEF

                pixels\_out[x, y + 1] = (

                    pixels\_in[m, n] + pixels\_in[m, n + 1]

                ) // SCALE\_COEF

                pixels\_out[x + 1, y + 1] = (

                    SCALE\_COEF \* pixels\_in[m, n]

                    + (pixels\_in[m + 1, n] + pixels\_in[m, n + 1]) // SCALE\_COEF

                ) // (SCALE\_COEF + 1)

        # Fill bottom border

        for m in range(BLOCKS):

            n = BLOCKS - 1

            for x in range(m \* SCALE\_COEF, (m + 1) \* SCALE\_COEF):

                for y in range(n \* SCALE\_COEF, (n + 1) \* SCALE\_COEF):

                    pixels\_out[x, y] = pixels\_in[m, n]

        # Fill right border

        for n in range(BLOCKS):

            m = BLOCKS - 1

            for x in range(m \* SCALE\_COEF, (m + 1) \* SCALE\_COEF):

                for y in range(n \* SCALE\_COEF, (n + 1) \* SCALE\_COEF):

                    pixels\_out[x, y] = pixels\_in[m, n]

        return img\_out

    @staticmethod

    def inject\_message(

        img\_in: PIL.Image.Image, message\_bits: bitarray

    ) -> PIL.Image.Image:

        img\_out = img\_in.copy()

        pixels\_in = img\_in.load()

        pixels\_out = img\_out.load()

        if pixels\_in is None or pixels\_out is None:

            raise BaseException("pixel arrays are none")

        # positions = []

        message\_bits\_index = 0

        # Going by blocks

        for x in range(0, img\_out.size[0] - SCALE\_COEF \* 2, SCALE\_COEF):

            for y in range(0, img\_out.size[1] - SCALE\_COEF \* 2, SCALE\_COEF):

                bit\_counts = tuple(

                    int(

                        numpy.log2(

                            max(

                                numpy.abs(

                                    pixels\_out[x + x\_i, y + y\_i] - pixels\_out[x, y]

                                ),

                                1,

                            )

                        )

                    )

                    for y\_i in range(SCALE\_COEF)

                    for x\_i in range(SCALE\_COEF)

                )

                for y\_i in range(SCALE\_COEF):

                    for x\_i in range(SCALE\_COEF):

                        message\_bits\_next\_index = (

                            message\_bits\_index + bit\_counts[x\_i + y\_i \* SCALE\_COEF]

                        )

                        val = sum(

                            el << index

                            for index, el in enumerate(

                                reversed(

                                    message\_bits[

                                        message\_bits\_index:message\_bits\_next\_index

                                    ]

                                )

                            )

                        )

                        # if bit\_counts[x\_i + y\_i \* SCALE\_COEF] > 0:

                        #     positions.append(tuple([x + x\_i, y + y\_i, val]))

                        pixels\_out[x + x\_i, y + y\_i] += val

                        if message\_bits\_next\_index >= len(message\_bits):

                            break

                        message\_bits\_index = message\_bits\_next\_index

                    # broke a leg, falling down the stairs

                    else:

                        continue

                    break

                else:

                    continue

                break

            else:

                continue

            break

        # print(positions)

        return img\_out

    @staticmethod

    def extract\_message(img\_in: PIL.Image.Image, message\_bit\_len: int) -> bitarray:

        pixels\_in = img\_in.load()

        if pixels\_in is None:

            raise BaseException("pixels array is none")

        message\_bits: list[int] = []

        # positions = []

        for x in range(0, img\_in.size[0] - SCALE\_COEF \* 2, SCALE\_COEF):

            for y in range(0, img\_in.size[1] - SCALE\_COEF \* 2, SCALE\_COEF):

                vals = (

                    0,

                    pixels\_in[x + 1, y]

                    - (pixels\_in[x, y] + pixels\_in[x + SCALE\_COEF, y]) // SCALE\_COEF,

                    pixels\_in[x, y + 1]

                    - (pixels\_in[x, y] + pixels\_in[x, y + SCALE\_COEF]) // SCALE\_COEF,

                    pixels\_in[x + 1, y + 1]

                    - (

                        SCALE\_COEF \* pixels\_in[x, y]

                        + (pixels\_in[x + SCALE\_COEF, y] + pixels\_in[x, y + SCALE\_COEF])

                        // SCALE\_COEF

                    )

                    // (SCALE\_COEF + 1),

                )

                bit\_counts = tuple(

                    int(

                        numpy.log2(

                            max(

                                numpy.abs(

                                    pixels\_in[x + x\_i, y + y\_i]

                                    - vals[x\_i + y\_i \* SCALE\_COEF]

                                    - pixels\_in[x, y]

                                ),

                                1,

                            )

                        )

                    )

                    for y\_i in range(SCALE\_COEF)

                    for x\_i in range(SCALE\_COEF)

                )

                vals = tuple(

                    [int(i) for i in f"{val:0{bit\_counts[index]}b}"]

                    if bit\_counts[index] > 0

                    else []

                    for index, val in enumerate(vals)

                )

                # for y\_i in range(SCALE\_COEF):

                #     for x\_i in range(SCALE\_COEF):

                #         if bit\_counts[x\_i + y\_i \* SCALE\_COEF] > 0:

                #             positions.append(

                #                 tuple([x + x\_i, y + y\_i, vals[x\_i + y\_i \* SCALE\_COEF]])

                #             )

                for val in vals:

                    message\_bits += val

                    # message\_bits += (

                    #     val[min(0, message\_bit\_len - len(message\_bits)) :: -1]

                    # )[::-1]

                    if len(message\_bits) >= message\_bit\_len:

                        diff = len(message\_bits) - message\_bit\_len

                        # print(diff)

                        # vals = [x for x in vals if len(x) > 0]

                        message\_bits = (

                            message\_bits[: len(message\_bits) - len(val)]

                            + val[::-1][: len(val) - diff][::-1]

                        )

                        # print(val)

                        # print(val[::-1][: len(val) - diff :][::-1])

                        break

                    else:

                        continue

                    break

                else:

                    continue

                break

            else:

                continue

            break

        # print(positions)

        return bitarray(message\_bits)

    @staticmethod

    def get\_max\_capacity(img\_in: PIL.Image.Image) -> int:

        pixels\_in = img\_in.load()

        if pixels\_in is None:

            raise BaseException("pixels\_in is none")

        capacity = 0

        for x in range(0, img\_in.size[0] - SCALE\_COEF, SCALE\_COEF):

            for y in range(0, img\_in.size[1] - SCALE\_COEF, SCALE\_COEF):

                bit\_counts = tuple(

                    int(

                        numpy.log2(

                            max(

                                numpy.abs(

                                    pixels\_in[x + x\_i, y + y\_i] - pixels\_in[x, y]

                                ),

                                1,

                            )

                        )

                    )

                    for y\_i in range(SCALE\_COEF)

                    for x\_i in range(SCALE\_COEF)

                )

                capacity += sum(bit\_counts)

        return capacity

import PIL.Image

from bitarray import bitarray

class LSBSeq:

    @staticmethod

    def inject\_message(img\_in: PIL.Image.Image, message: bitarray) -> PIL.Image.Image:

        img = img\_in.copy()

        pixels = img.load()

        last\_x = None

        last\_y = None

        msg\_index = 0

        max\_valid = len(message) - len(message) % 3

        if pixels is None:

            raise BaseException("pixels\_in is None")

        for x in range(img.width):

            for y in range(img.height):

                if msg\_index >= max\_valid:

                    last\_x = x

                    last\_y = y

                    break

                byte = pixels[x, y]

                for index in range(3):

                    if message[msg\_index + index]:

                        byte |= 1 << index

                    else:

                        byte &= ~(1 << index)

                pixels[x, y] = byte

                msg\_index += 3

            else:

                continue

            break

        if last\_x is not None and last\_y is not None:

            last\_bits\_len = len(message) % 3

            byte = pixels[last\_x, last\_y]

            for index in range(last\_bits\_len):

                if message[-(last\_bits\_len - index)]:

                    byte |= 1 << index

                else:

                    byte &= ~(1 << index)

            pixels[last\_x, last\_y] = byte

        return img

    @staticmethod

    def get\_max\_capacity(img\_in: PIL.Image.Image) -> int:

        return img\_in.width \* img\_in.height // 3

import PIL.Image

import numpy

import scipy

class AUMPAnalysis:

    \_\_WALL = 1

    @staticmethod

    def analyze(image: PIL.Image.Image, block\_size: int, parameters: int) -> bool:

        pixels = numpy.array(image, dtype=numpy.floating)

        scipy.io.savemat("array.mat", {"X": pixels})

        try:

            return (

                AUMPAnalysis.\_\_aump(pixels, block\_size, parameters)

                > AUMPAnalysis.\_\_WALL

            )

        except BaseException:

            return False

    @staticmethod

    def \_\_aump(X, m, d):

        Xpred, \_, w = AUMPAnalysis.\_\_pred\_aump(X, m, d)

        r = X - Xpred

        Xbar = X + 1 - 2 \* (X % 2)

        beta = numpy.sum(w \* (X - Xbar) \* r)

        return beta

    @staticmethod

    def \_\_pred\_aump(X, m, d):

        sig\_th = 1

        q = d + 1

        Kn = X.size // m

        Y = numpy.zeros((m, Kn))

        S = numpy.zeros\_like(X)

        Xpred = numpy.zeros\_like(X)

        x1 = numpy.linspace(1, m, m) / m

        H = numpy.vander(x1, q, increasing=True)

        for i in range(m):

            aux = X[:, i::m]

            Y[i, :] = aux.flatten()

        p = numpy.linalg.lstsq(H, Y, rcond=None)[0]

        Ypred = H @ p

        for i in range(m):

            Xpred[:, i::m] = Ypred[i, :].reshape(X[:, i::m].shape)

        sig2 = numpy.sum((Y - Ypred) \*\* 2, axis=0) / (m - q)

        sig2 = numpy.maximum(sig\_th\*\*2, sig2)

        Sy = numpy.ones((m, 1)) \* sig2

        for i in range(m):

            S[:, i::m] = Sy[i, :].reshape(X[:, i::m].shape)

        s\_n2 = Kn / numpy.sum(1.0 / sig2)

        w = numpy.sqrt(s\_n2 / (Kn \* (m - q))) / S

        return Xpred, S, w

import PIL.Image

import scipy.stats

import numpy

class ChiSquaredAnalysis:

    \_\_MIN\_BIN = 5

    \_\_WALL = 0.5

    @staticmethod

    def analyze(image: PIL.Image.Image, block\_size: int = 128) -> bool:

        pixels = image.load()

        if pixels is None:

            raise BaseException

        outs = []

        for x\_max in range(0, image.width, block\_size):

            for y\_max in range(0, image.height, block\_size):

                x\_min = min(0, x\_max - block\_size)

                y\_min = min(0, y\_max - block\_size)

                distribution\_actual: list[int] = [0] \* 8

                for x in range(x\_min, x\_max):

                    for y in range(y\_min, y\_max):

                        distribution\_actual[pixels[x, y] & 0b111] += 1

                distribution\_mean: list[int] = [0] \* len(distribution\_actual)

                for index in range(0, len(distribution\_actual) - 1, 2):

                    mean = (

                        distribution\_actual[index] + distribution\_actual[index + 1]

                    ) / 2

                    if mean.is\_integer():

                        distribution\_mean[index] = distribution\_mean[index + 1] = int(

                            mean

                        )

                    elif distribution\_actual[index] < distribution\_actual[index + 1]:

                        distribution\_mean[index] = int(mean)

                        distribution\_mean[index + 1] = int(mean) + 1

                    else:

                        distribution\_mean[index] = int(mean) + 1

                        distribution\_mean[index + 1] = int(mean)

                # print(distribution\_actual)

                # print(distribution\_mean)

                # combine bins

                index = 0

                ChiSquaredAnalysis.\_\_MIN\_BIN = numpy.average(distribution\_actual)

                while index < len(distribution\_actual):

                    if distribution\_actual[index] < ChiSquaredAnalysis.\_\_MIN\_BIN:

                        val\_sum = distribution\_actual[index]

                        stop = index

                        for jindex in range(index + 1, len(distribution\_actual)):

                            val\_sum += distribution\_actual[jindex]

                            if val\_sum >= ChiSquaredAnalysis.\_\_MIN\_BIN:

                                stop = jindex

                                break

                        else:

                            stop = len(distribution\_actual) - 1

                        distribution\_actual[index : stop + 1] = [

                            sum(distribution\_actual[index : stop + 1])

                        ]

                        distribution\_mean[index : stop + 1] = [

                            sum(distribution\_mean[index : stop + 1])

                        ]

                    index += 1

                if distribution\_actual[-1] < ChiSquaredAnalysis.\_\_MIN\_BIN:

                    distribution\_actual[-2:] = [sum(distribution\_actual[-2:])]

                    distribution\_mean[-2:] = [sum(distribution\_mean[-2:])]

                del index

                for x in distribution\_actual:

                    if x == 0:

                        break

                else:

                    if len(distribution\_actual) == 2:

                        if distribution\_actual[0] == 0 or distribution\_actual[1] == 0:

                            outs.append(1)

                            continue

                    if len(distribution\_actual) == 1:

                        outs.append(1)

                        continue

                    outs.append(

                        scipy.stats.chisquare(

                            f\_obs=distribution\_actual, f\_exp=distribution\_mean, ddof=1

                        )[1]

                    )

        # print(outs)

        # print(numpy.average(outs))

        # outs = [x for x in outs if x > 0.000001]

        # return numpy.average(outs) > 0.5

        return numpy.average(outs) < ChiSquaredAnalysis.\_\_WALL

import PIL.Image

import numpy

class RSAnalysis:

    ANALYSIS\_COLOR\_GRAYSCALE = -1

    ANALYSIS\_COLOR\_RED = 0

    ANALYSIS\_COLOR\_GREEN = 1

    ANALYSIS\_COLOR\_BLUE = 2

    def \_\_init\_\_(self, m: int, n: int):

        self.\_\_mMask = [[0] \* (m \* n), [0] \* (m \* n)]

        k: int = 0

        for i in range(n):

            for j in range(m):

                if ((j % 2) == 0 and (i % 2) == 0) or ((j % 2) == 1 and (i % 2) == 1):

                    self.\_\_mMask[0][k] = 1

                    self.\_\_mMask[1][k] = 0

                else:

                    self.\_\_mMask[0][k] = 0

                    self.\_\_mMask[1][k] = 1

                k += 1

        self.\_\_mM = m

        self.\_\_mN = n

    # colorfull images are not supported currently

    # def analyze(self, image: Image.Image, color: int, overlap: bool) -> list[float]:

    def analyze(

        self,

        image: PIL.Image.Image,

        color: int = ANALYSIS\_COLOR\_GRAYSCALE,

        overlap: bool = True,

    ) -> bool:

        imgx: int = image.width

        imgy: int = image.height

        startx: int = 0

        starty: int = 0

        block: list[int] = [0] \* (self.\_\_mM \* self.\_\_mN)

        numregular: float = 0

        numsingular: float = 0

        numnegreg: float = 0

        numnegsing: float = 0

        numunusable: float = 0

        numnegunusable: float = 0

        variationB: float

        variationP: float

        variationN: float

        pixels = image.load()

        if pixels is None:

            raise BaseException("pixels is none")

        while startx < imgx and starty < imgy:

            for m in range(2):

                k: int = 0

                for i in range(self.\_\_mN):

                    for j in range(self.\_\_mM):

                        block[k] = pixels[startx + j, starty + i]

                        k += 1

                variationB = self.\_\_getVariation(block, color)

                block = self.\_\_flipBlock(block, self.\_\_mMask[m])

                variationP = self.\_\_getVariation(block, color)

                block = self.\_\_flipBlock(block, self.\_\_mMask[m])

                self.\_\_mMask[m] = self.\_\_invertMask(self.\_\_mMask[m])

                variationN = self.\_\_getNegativeVariation(block, color, self.\_\_mMask[m])

                self.\_\_mMask[m] = self.\_\_invertMask(self.\_\_mMask[m])

                if variationP > variationB:

                    numregular += 1

                if variationP < variationB:

                    numsingular += 1

                if variationP == variationB:

                    numunusable += 1

                if variationN > variationB:

                    numnegreg += 1

                if variationN < variationB:

                    numnegsing += 1

                if variationN == variationB:

                    numnegunusable += 1

            if overlap:

                startx += 1

            else:

                startx += self.\_\_mM

            if startx >= (imgx - 1):

                startx = 0

                if overlap:

                    starty += 1

                else:

                    starty += self.\_\_mN

            if starty >= (imgy - 1):

                break

        totalgroups: float = numregular + numsingular + numunusable

        allpixels: list[float] = self.\_\_getAllPixelFlips(image, color, overlap)

        x: float = self.\_\_getX(

            numregular,

            numnegreg,

            allpixels[0],

            allpixels[2],

            numsingular,

            numnegsing,

            allpixels[1],

            allpixels[3],

        )

        epf: float

        ml: float

        if 2 \* (x - 1) == 0:

            epf = 0

        else:

            epf = abs(x / (2 \* (x - 1)))

        if x - 0.5 == 0:

            ml = 0

        else:

            ml = abs(x / (x - 0.5))

        results: list[float] = [0] \* 28

        results[0] = numregular

        results[1] = numsingular

        results[2] = numnegreg

        results[3] = numnegsing

        results[4] = abs(numregular - numnegreg)

        results[5] = abs(numsingular - numnegsing)

        results[6] = (numregular / totalgroups) \* 100

        results[7] = (numsingular / totalgroups) \* 100

        results[8] = (numnegreg / totalgroups) \* 100

        results[9] = (numnegsing / totalgroups) \* 100

        results[10] = (results[4] / totalgroups) \* 100

        results[11] = (results[5] / totalgroups) \* 100

        results[12] = allpixels[0]

        results[13] = allpixels[1]

        results[14] = allpixels[2]

        results[15] = allpixels[3]

        results[16] = abs(allpixels[0] - allpixels[1])

        results[17] = abs(allpixels[2] - allpixels[3])

        results[18] = (allpixels[0] / totalgroups) \* 100

        results[19] = (allpixels[1] / totalgroups) \* 100

        results[20] = (allpixels[2] / totalgroups) \* 100

        results[21] = (allpixels[3] / totalgroups) \* 100

        results[22] = (results[16] / totalgroups) \* 100

        results[23] = (results[17] / totalgroups) \* 100

        results[24] = totalgroups

        results[25] = epf

        results[26] = ml

        results[27] = ((imgx \* imgy \* 3) \* ml) / 8

        return ml > 0.01

    def \_\_getX(

        self,

        r: float,

        rm: float,

        r1: float,

        rm1: float,

        s: float,

        sm: float,

        s1: float,

        sm1: float,

    ) -> float:

        x: float = 0

        dzero: float = r - s

        dminuszero: float = rm - sm

        done: float = r1 - s1

        dminusone: float = rm1 - sm1

        a: float = 2 \* (done + dzero)

        b: float = dminuszero - dminusone - done - (3 \* dzero)

        c: float = dzero - dminuszero

        if a == 0:

            x = c / b

        discriminant: float = b \* b - (4 \* a \* c)

        if discriminant >= 0:

            rootpos: float = ((-1 \* b) + numpy.sqrt(discriminant)) / (2 \* a)

            rootneg: float = ((-1 \* b) - numpy.sqrt(discriminant)) / (2 \* a)

            if numpy.abs(rootpos) <= numpy.abs(rootneg):

                x = rootpos

            else:

                x = rootneg

        else:

            cr = (rm - r) / (r1 - r + rm - rm1)

            cs = (sm - s) / (s1 - s + sm - sm1)

            x = (cr + cs) / 2

        if x == 0:

            ar = ((rm1 - r1 + r - rm) + (rm - r) / x) / (x - 1)

            as\_ = ((sm1 - s1 + s - sm) + (sm - s) / x) / (x - 1)

            if as\_ > 0 or ar < 0:

                cr = (rm - r) / (r1 - r + rm - rm1)

                cs = (sm - s) / (s1 - s + sm - sm1)

                x = (cr + cs) / 2

        return x

    def \_\_getAllPixelFlips(

        self, image: PIL.Image.Image, color: int, overlap: bool

    ) -> list[float]:

        allmask: list[int] = [1] \* (self.\_\_mM \* self.\_\_mN)

        imgx: int = image.width

        imgy: int = image.height

        startx: int = 0

        starty: int = 0

        block: list[int] = [0] \* (self.\_\_mM \* self.\_\_mN)

        numregular: float = 0

        numsingular: float = 0

        numnegreg: float = 0

        numnegsing: float = 0

        numunusable: float = 0

        numnegunusable: float = 0

        variationB: float

        variationP: float

        variationN: float

        pixels = image.load()

        if pixels is None:

            raise BaseException("pixels is none")

        while startx < imgx and starty < imgy:

            for m in range(2):

                k: int = 0

                for i in range(self.\_\_mN):

                    for j in range(self.\_\_mM):

                        block[k] = pixels[startx + j, starty + i]

                        k += 1

                block = self.\_\_flipBlock(block, allmask)

                variationB = self.\_\_getVariation(block, color)

                block = self.\_\_flipBlock(block, self.\_\_mMask[m])

                variationP = self.\_\_getVariation(block, color)

                block = self.\_\_flipBlock(block, self.\_\_mMask[m])

                self.\_\_mMask[m] = self.\_\_invertMask(self.\_\_mMask[m])

                variationN = self.\_\_getNegativeVariation(block, color, self.\_\_mMask[m])

                self.\_\_mMask[m] = self.\_\_invertMask(self.\_\_mMask[m])

                if variationP > variationB:

                    numregular += 1

                if variationP < variationB:

                    numsingular += 1

                if variationP == variationB:

                    numunusable += 1

                if variationN > variationB:

                    numnegreg += 1

                if variationN < variationB:

                    numnegsing += 1

                if variationN == variationB:

                    numnegunusable += 1

            if overlap:

                startx += 1

            else:

                startx += self.\_\_mM

            if startx >= (imgx - 1):

                startx = 0

                if overlap:

                    starty += 1

                else:

                    starty += self.\_\_mN

            if starty >= (imgy - 1):

                break

        results: list[float] = [0] \* 4

        results[0] = numregular

        results[1] = numsingular

        results[2] = numnegreg

        results[3] = numnegsing

        return results

    @staticmethod

    def getResultNames() -> tuple[str, ...]:

        return (

            "Number of regular groups (positive)",

            "Number of singular groups (positive)",

            "Number of regular groups (negative)",

            "Number of singular groups (negative)",

            "Difference for regular groups",

            "Difference for singular groups",

            "Percentage of regular groups (positive)",

            "Percentage of singular groups (positive)",

            "Percentage of regular groups (negative)",

            "Percentage of singular groups (negative)",

            "Difference for regular groups %",

            "Difference for singular groups %",

            "Number of regular groups (positive for all flipped)",

            "Number of singular groups (positive for all flipped)",

            "Number of regular groups (negative for all flipped)",

            "Number of singular groups (negative for all flipped)",

            "Difference for regular groups (all flipped)",

            "Difference for singular groups (all flipped)",

            "Percentage of regular groups (positive for all flipped)",

            "Percentage of singular groups (positive for all flipped)",

            "Percentage of regular groups (negative for all flipped)",

            "Percentage of singular groups (negative for all flipped)",

            "Difference for regular groups (all flipped) %",

            "Difference for singular groups (all flipped) %",

            "Total number of groups",

            "Estimated percent of flipped pixels",

            "Estimated message length (in percent of pixels)(p)",

            "Estimated message length (in bytes)",

        )

    def \_\_getVariation(self, block: list[int], color: int) -> float:

        var: float = 0

        color1: int

        color2: int

        for i in range(0, len(block), 4):

            color1 = self.\_\_getPixelColor(block[0 + i], color)

            color2 = self.\_\_getPixelColor(block[1 + i], color)

            var += numpy.abs(color1 - color2)

            color1 = self.\_\_getPixelColor(block[3 + i], color)

            color2 = self.\_\_getPixelColor(block[2 + i], color)

            var += numpy.abs(color1 - color2)

            color1 = self.\_\_getPixelColor(block[1 + i], color)

            color2 = self.\_\_getPixelColor(block[3 + i], color)

            var += numpy.abs(color1 - color2)

            color1 = self.\_\_getPixelColor(block[2 + i], color)

            color2 = self.\_\_getPixelColor(block[0 + i], color)

            var += numpy.abs(color1 - color2)

        return var

    def \_\_getNegativeVariation(

        self, block: list[int], color: int, mask: list[int]

    ) -> float:

        var: float = 0

        color1: int

        color2: int

        for i in range(0, len(block), 4):

            color1 = self.\_\_getPixelColor(block[0 + i], color)

            color2 = self.\_\_getPixelColor(block[1 + i], color)

            if mask[0 + i] == -1:

                color1 = self.\_\_invertLSB(color1)

            if mask[1 + i] == -1:

                color2 = self.\_\_invertLSB(color2)

            var += numpy.abs(color1 - color2)

            color1 = self.\_\_getPixelColor(block[1 + i], color)

            color2 = self.\_\_getPixelColor(block[3 + i], color)

            if mask[1 + i] == -1:

                color1 = self.\_\_invertLSB(color1)

            if mask[3 + i] == -1:

                color2 = self.\_\_invertLSB(color2)

            var += numpy.abs(color1 - color2)

            color1 = self.\_\_getPixelColor(block[3 + i], color)

            color2 = self.\_\_getPixelColor(block[2 + i], color)

            if mask[3 + i] == -1:

                color1 = self.\_\_invertLSB(color1)

            if mask[2 + i] == -1:

                color2 = self.\_\_invertLSB(color2)

            var += numpy.abs(color1 - color2)

            color1 = self.\_\_getPixelColor(block[2 + i], color)

            color2 = self.\_\_getPixelColor(block[0 + i], color)

            if mask[2 + i] == -1:

                color1 = self.\_\_invertLSB(color1)

            if mask[0 + i] == -1:

                color2 = self.\_\_invertLSB(color2)

            var += numpy.abs(color1 - color2)

        return var

    def \_\_getPixelColor(self, pixel: int, color: int) -> int:

        return pixel

    def \_\_flipBlock(self, block: list[int], mask: list[int]) -> list[int]:

        for i in range(len(block)):

            if mask[i] == 1:

                block[i] = self.\_\_negateLSB(block[i])

            elif mask[i] == -1:

                block[i] = self.\_\_invertLSB(block[i])

        return block

    def \_\_negateLSB(self, abyte: int) -> int:

        temp = abyte & 0xFE

        if temp == abyte:

            return abyte | 0x1

        else:

            return temp

    def \_\_invertLSB(self, abyte: int) -> int:

        if abyte == 255:

            return 256

        if abyte == 256:

            return 255

        return self.\_\_negateLSB(abyte + 1) - 1

    def \_\_invertMask(self, mask: list[int]) -> list[int]:

        return [x \* -1 for x in mask]

from ui.ImagesWidget import ImagesWidget

from ui.ButtonsWidget import ButtonsWidget

import control

import PIL.Image

import PIL.ImageQt

import PyQt6.QtWidgets

import PyQt6.QtGui

import PyQt6.QtCore

import multiprocessing

import pathlib

import json

import itertools

from bitarray import bitarray

import bitarray.util as bitutil

import enum

class AnalyseWorker(PyQt6.QtCore.QThread):

    \_\_POOL\_SIZE = 5

    \_\_AUMP\_KEY = "AUMP"

    \_\_CHI2\_KEY = "CHI2"

    \_\_RS\_KEY = "RS"

    \_\_RS\_FLIPPED\_PIXELS\_INDEX = 25

    finished = PyQt6.QtCore.pyqtSignal(dict)

    class Methods(enum.Enum):

        LSB\_PATTERN\_MATCHING = 0

        LSB\_SEQUENTIAL = 2

        LSB\_SCALED = 3

    def \_\_init\_\_(

        self,

        images: list[PIL.Image.Image],

        method: Methods,

        rs: tuple[int, int],

        aump: tuple[int, int],

    ):

        super().\_\_init\_\_()

        self.\_\_images = images

        self.\_\_processes = None

        self.\_\_canceled = False

        self.\_\_method = method

        self.\_\_rs = rs

        self.\_\_rs\_a = control.RSAnalysis(\*rs)

        self.\_\_aump = aump

    @staticmethod

    def insert\_message(

        image: PIL.Image.Image,

        message: bitarray,

        percent: int,

        method: Methods,

    ):

        if percent > 0:

            match method:

                case AnalyseWorker.Methods.LSB\_PATTERN\_MATCHING:

                    cap = control.LSBSeq.get\_max\_capacity(image)

                    return control.LSBSeq.inject\_message(

                        image, message[: cap \* percent // 100]

                    )

                case AnalyseWorker.Methods.LSB\_SEQUENTIAL:

                    cap = control.LSBMessage.get\_max\_capacity(image, message)

                    return control.LSBMessage.inject\_message(

                        image, message[: cap \* percent // 100]

                    )

                case AnalyseWorker.Methods.LSB\_SCALED:

                    cap = control.LSBScaledMessage.get\_max\_capacity(image)

                    return control.LSBScaledMessage.inject\_message(

                        control.LSBScaledMessage.scale\_image(image),

                        message[: cap \* percent // 100],

                    )

        if method == AnalyseWorker.Methods.LSB\_SCALED:

            return control.LSBScaledMessage.scale\_image(image)

        return image

    @staticmethod

    def analyze\_image(image: PIL.Image.Image, rs\_a, aump: tuple[int, int]):

        out = {}

        out[AnalyseWorker.\_\_CHI2\_KEY] = control.ChiSquaredAnalysis.analyze(image)

        out[AnalyseWorker.\_\_AUMP\_KEY] = control.AUMPAnalysis.analyze(image, \*aump)

        out[AnalyseWorker.\_\_RS\_KEY] = rs\_a.analyze(image)

        return out

    def cancel(self):

        self.\_\_canceled = True

        for proc in multiprocessing.active\_children():

            proc.kill()

    def run(self):

        bits = bitutil.urandom(500000)

        out = {}

        for insert\_rate in range(0, 101, 50):

            with multiprocessing.Pool(self.\_\_POOL\_SIZE) as pool:

                images = pool.starmap(

                    AnalyseWorker.insert\_message,

                    zip(

                        self.\_\_images,

                        itertools.repeat(bits),

                        itertools.repeat(insert\_rate),

                        itertools.repeat(self.\_\_method),

                    ),

                )

            with multiprocessing.Pool(self.\_\_POOL\_SIZE) as pool:

                out\_cur = pool.starmap(

                    AnalyseWorker.analyze\_image,

                    zip(

                        images,

                        itertools.repeat(self.\_\_rs\_a),

                        itertools.repeat(self.\_\_aump),

                    ),

                )

                out[insert\_rate] = {

                    k: sum([dic[k] for dic in out\_cur]) for k in out\_cur[0]

                }

                if insert\_rate > 0:

                    for key in out[insert\_rate].keys():

                        out[insert\_rate][key] = (

                            (len(self.\_\_images) - out[insert\_rate][key])

                            \* 100

                            / len(self.\_\_images)

                        )

                else:

                    for key in out[insert\_rate].keys():

                        out[insert\_rate][key] = (

                            out[insert\_rate][key] \* 100 / len(self.\_\_images)

                        )

            if self.\_\_canceled:

                self.finished.emit({})

                return

        # out = {k: [dic[k] for dic in out] for k in out[0]}

        # out = {

        #     key: {index \* 25: out[key][index] for index in range(len(out[key]))}

        #     for key in out.keys()

        # }

        self.finished.emit(out)

class MainWindow(PyQt6.QtWidgets.QMainWindow):

    \_\_WINDOW\_TITLE = "Steganography Task 6"

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle(self.\_\_WINDOW\_TITLE)

        self.populate()

        self.connect\_buttons()

        self.apply\_styles()

    def apply\_styles(self):

        self.setStyleSheet("""

            QMainWindow {

                background-color: #F9FBFD;

            }

            QTextBrowser {

                border: 1px solid #CCC;

                border-radius: 6px;

                background-color: #FFF;

                padding: 10px;

                font-family: Consolas, monospace;

                font-size: 13px;

            }

        """)

    def show\_progress\_indicator(self):

        self.\_\_progress\_indicator = PyQt6.QtWidgets.QProgressDialog(self)

        self.\_\_progress\_indicator.setWindowModality(

            PyQt6.QtCore.Qt.WindowModality.WindowModal

        )

        self.\_\_progress\_indicator.setRange(0, 0)

        self.\_\_progress\_indicator.setAttribute(

            PyQt6.QtCore.Qt.WidgetAttribute.WA\_DeleteOnClose

        )

        self.\_\_progress\_indicator.setWindowTitle("Processing...")

        self.\_\_progress\_indicator.setLabelText("Analyzing images, please wait...")

        self.\_\_progress\_indicator.show()

    def hide\_progress\_indicator(self):

        self.\_\_progress\_indicator.close()

    def connect\_buttons(self):

        self.\_\_worker\_thread = None

        def open\_images\_event():

            paths = PyQt6.QtWidgets.QFileDialog.getOpenFileNames(

                parent=self, caption="Select images to open", filter="BMP (\*.bmp)"

            )[0]

            if paths:

                self.\_\_images\_before\_widget.images = [PIL.Image.open(x) for x in paths]

        def analyze\_images\_event():

            def worker\_finished(out: dict):

                self.hide\_progress\_indicator()

                self.\_\_result\_text\_browser.setText(json.dumps(out, indent=2))

                control.PlotBuilder.build\_plot("Result", out)

            if not self.\_\_images\_before\_widget.images:

                PyQt6.QtWidgets.QMessageBox.warning(self, "Warning", "No images loaded!")

                return

            self.show\_progress\_indicator()

            self.\_\_worker\_thread = AnalyseWorker(

                self.\_\_images\_before\_widget.images,

                self.\_\_buttons\_widget.method\_dropdown.currentData(),

                (

                    self.\_\_buttons\_widget.rs\_m\_spinbox.value(),

                    self.\_\_buttons\_widget.rs\_n\_spinbox.value(),

                ),

                (

                    self.\_\_buttons\_widget.aump\_block\_size\_spinbox.value(),

                    self.\_\_buttons\_widget.aump\_parameter\_spinbox.value(),

                ),

            )

            self.\_\_worker\_thread.finished.connect(worker\_finished)

            self.\_\_progress\_indicator.canceled.connect(self.\_\_worker\_thread.quit)

            self.\_\_worker\_thread.start()

        def save\_results\_event():

            try:

                pathlib.Path("out.json").write\_text(

                    self.\_\_result\_text\_browser.toPlainText()

                )

                PyQt6.QtWidgets.QMessageBox.information(self, "Success", "Results saved to out.json")

            except Exception as e:

                PyQt6.QtWidgets.QMessageBox.critical(self, "Error", f"Failed to save results:\n{e}")

        self.\_\_buttons\_widget.open\_images\_button.clicked.connect(open\_images\_event)

        self.\_\_buttons\_widget.analyze\_images\_button.clicked.connect(analyze\_images\_event)

        self.\_\_buttons\_widget.save\_resuts\_button.clicked.connect(save\_results\_event)

    def populate(self):

        self.\_\_central\_widget = PyQt6.QtWidgets.QWidget(self)

        self.\_\_main\_layout = PyQt6.QtWidgets.QVBoxLayout(self.\_\_central\_widget)

        self.\_\_main\_layout.setContentsMargins(15, 15, 15, 15)

        self.\_\_main\_layout.setSpacing(15)

        self.\_\_central\_widget.setLayout(self.\_\_main\_layout)

        self.setCentralWidget(self.\_\_central\_widget)

        self.\_\_upper\_layout\_widget = PyQt6.QtWidgets.QWidget(self)

        self.\_\_upper\_layout = PyQt6.QtWidgets.QHBoxLayout(self.\_\_upper\_layout\_widget)

        self.\_\_upper\_layout.setSpacing(20)

        self.\_\_upper\_layout\_widget.setLayout(self.\_\_upper\_layout)

        self.\_\_main\_layout.addWidget(self.\_\_upper\_layout\_widget, stretch=1)

        self.\_\_images\_before\_widget = ImagesWidget(self.\_\_upper\_layout\_widget)

        self.\_\_buttons\_widget = ButtonsWidget(self.\_\_upper\_layout\_widget)

        self.\_\_upper\_layout.addWidget(self.\_\_images\_before\_widget, stretch=3)

        self.\_\_upper\_layout.addWidget(self.\_\_buttons\_widget, stretch=1)

        self.\_\_result\_text\_browser = PyQt6.QtWidgets.QTextBrowser(self.\_\_central\_widget)

        self.\_\_result\_text\_browser.setMinimumHeight(150)

        self.\_\_main\_layout.addWidget(self.\_\_result\_text\_browser)

        # Заполнить комбобокс методами с удобочитаемыми названиями

        self.\_\_buttons\_widget.method\_dropdown.clear()

        for method in AnalyseWorker.Methods:

            # Отображать название метода красиво

            self.\_\_buttons\_widget.method\_dropdown.addItem(method.name.replace('\_', ' ').title(), method)

import matplotlib

import matplotlib.patheffects

import matplotlib.pyplot as plt

import numpy

class PlotBuilder:

    \_\_COLORS = ("tab:blue", "tab:red", "tab:green")

    @staticmethod

    def build\_plot(title: str, data, save\_path: str = "report/plot.png"):

        matplotlib.rcParams.update({"font.size": 14})

        xt = data.keys()

        data = list(data.values())

        data = {k: [x[k] for x in data] for k in data[0].keys()}

        x = numpy.arange(len(xt))

        width = 0.25

        multiplier = 0

        fig, ax = plt.subplots(constrained\_layout=True)

        for index, (attribute, measurement) in enumerate(data.items()):

            offset = width \* multiplier

            rects = ax.bar(

                x + offset,

                measurement,

                width,

                label=attribute,

                color=PlotBuilder.\_\_COLORS[index],

            )

            ax.bar\_label(rects, padding=3)

            multiplier += 1

        for text in ax.texts:

            text.set\_path\_effects(

                [matplotlib.patheffects.withStroke(linewidth=4, foreground="w")]

            )

        ax.set\_xticks(x + width, [str(x) for x in xt])

        ax.grid(axis="y")

        ax.legend()

        if fig.canvas.manager is not None:

            fig.canvas.manager.set\_window\_title(title)

        # 💾 Сохраняем график перед отображением

        fig.savefig(save\_path, dpi=300, bbox\_inches="tight")

        plt.show()