Оглавление

Цель работы:

Исследование вероятностного алгоритма двоичной экспоненциальной отсрочки.

Задача:

Написать моделирующую программу для вероятностного алгоритма двоичной экспоненциальной отсрочки.

Ход работы:

1. Описание алгоритма

Абоненты узнают о событии в канале только в окне, в котором они передавали. В вероятностном варианте каждый абонент меняет вероятность передачи в соответствии событием, которое произошло в канале при его передаче по следующему правилу:

$$P_{t+1} = \begin{cases} max\left(\frac{p_t}{2}, P_{min}\right), \text{при "конфликте " в канале} \\ p_{max}, \text{при "успехе" в канале} \end{cases}$$

Где p_t – вероятность, с которой абонент передавал при t-ой передаче;

P_{min} – минимальная вероятность передачи;

p_{max} – максимальная вероятность передачи.

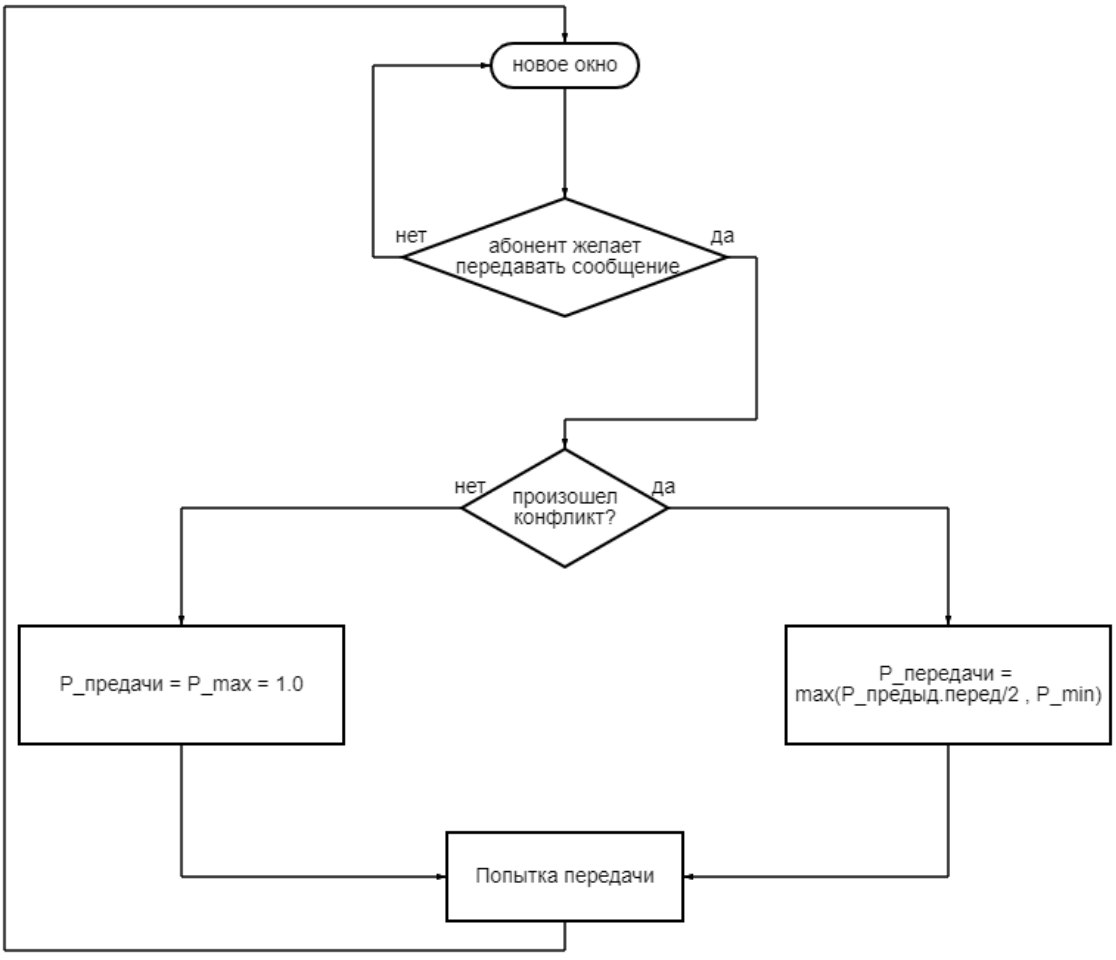

Рисунок 1. Алгоритм действий в одном временном окне

2. Дополнительное задание:

Необходимо было вычислить критическую минимальную вероятность, путем

моделирования работы системы с фиксированной критической входной интенсивностью потока (lambda = 2.0), перебирая при этом минимальную вероятность. Построив график зависимости минимальной вероятности от выходной интенсивности потока, желаем результат не был достигнут из-за особенностей программной реализации (было тяжело подобрать необходимые параметры, результат или не достигал необходимых значений, либо переходил за пределы этих значений)

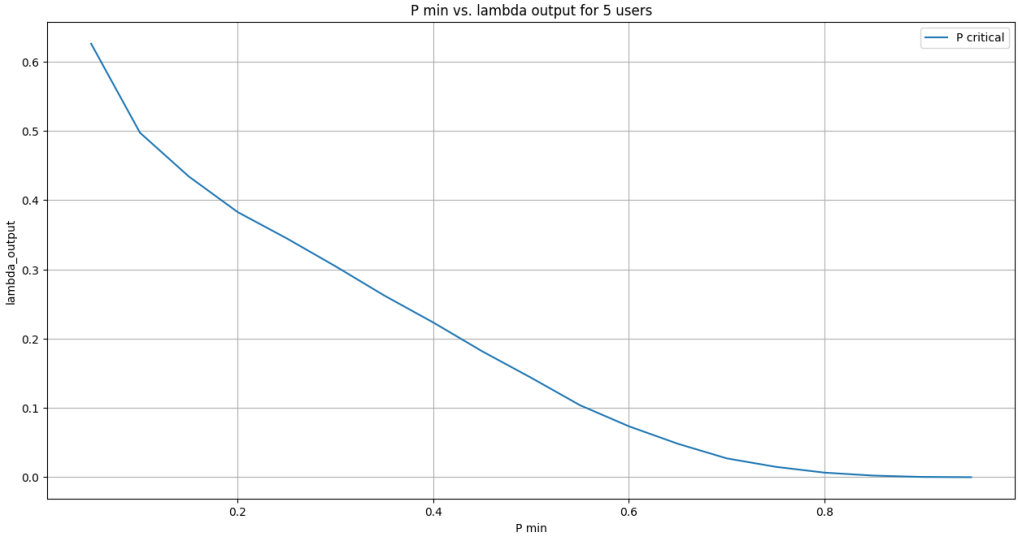

Рисунок 2. График зависимости интенсивности выходного потока от P_min при 5 пользователях в окне

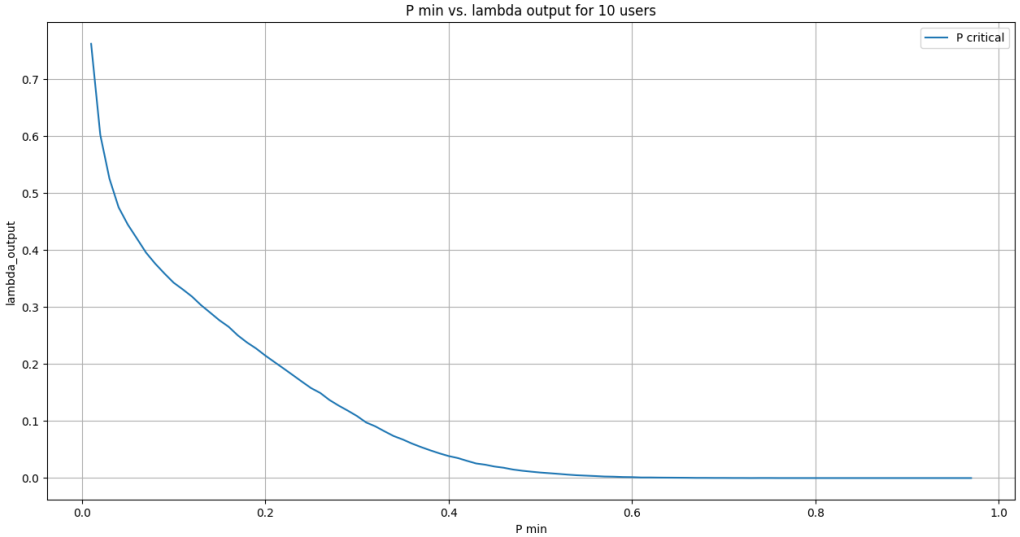

Рисунок 3. График зависимости интенсивности выходного потока от P_min при 10 пользователях в окне

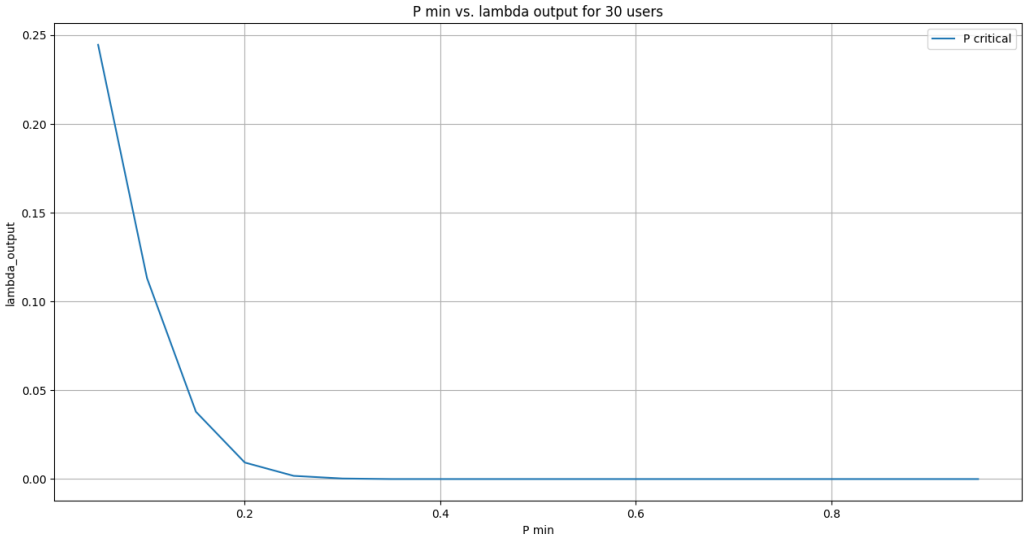

Рисунок 4. График зависимости интенсивности выходного потока от P_min при 30 пользователях в окне

Теоретически значение критической минимальной вероятности должно стремиться к $\frac{1}{M}$, где M – число абонентов в окне.

3. Графики работы программы:

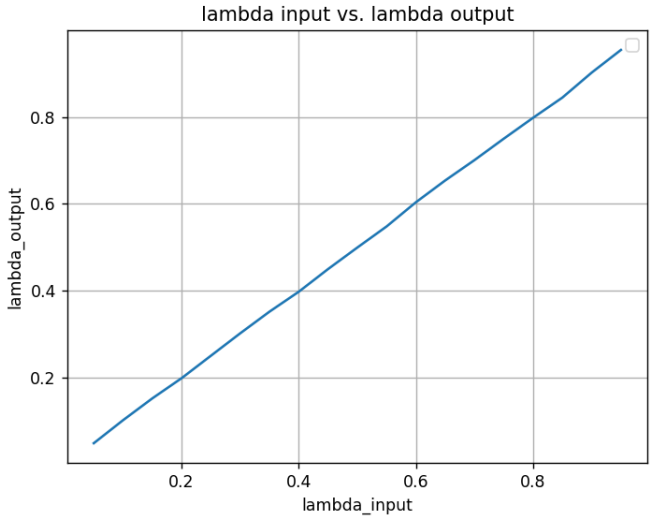

Рисунок 5. График зависимости интенсивности входного потока от выходного потока для 1 пользователя

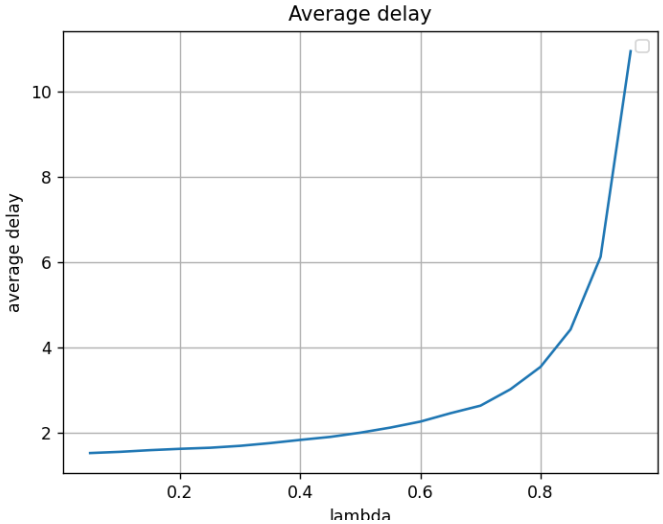

Рисунок 6.график средней задержки для 1 пользователя

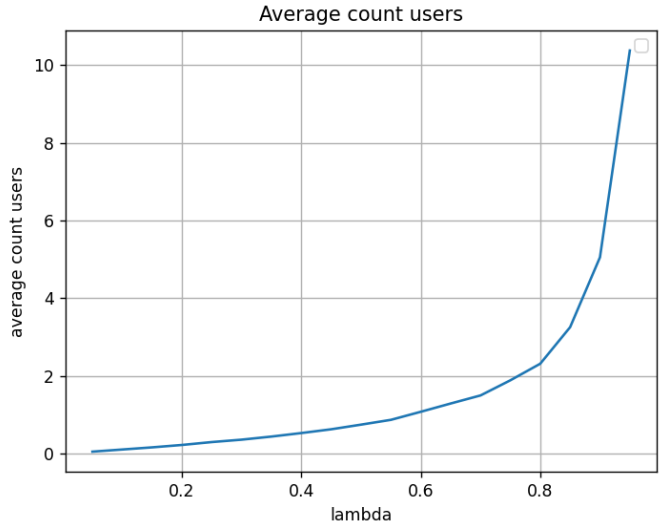

Рисунок 7. График среднего числа сообщений для 1 абонента

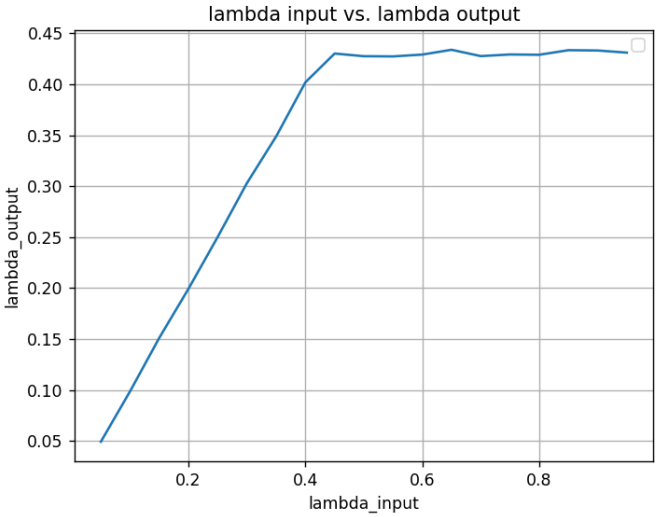

Рисунок 8. График зависимости интенсивности входного потока от выходного потока для 5 пользователей

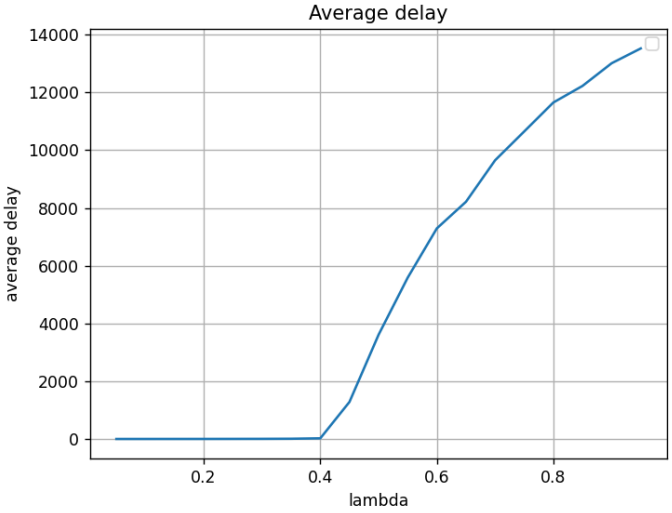

Рисунок 9. график средней задержки для 5 пользователей

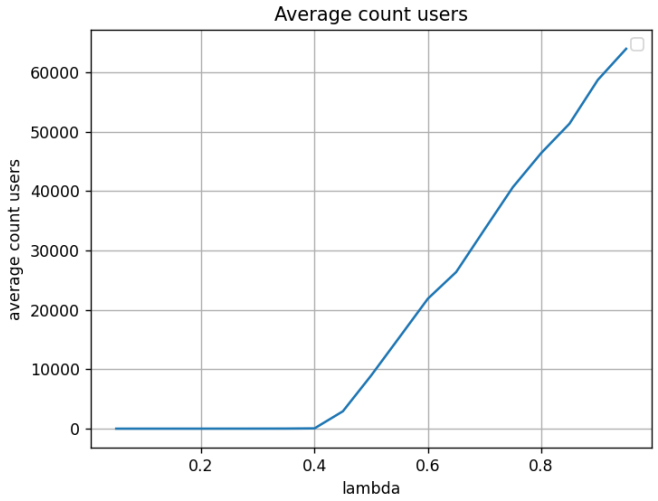

Рисунок 10. График среднего числа сообщений для 5 абонентов

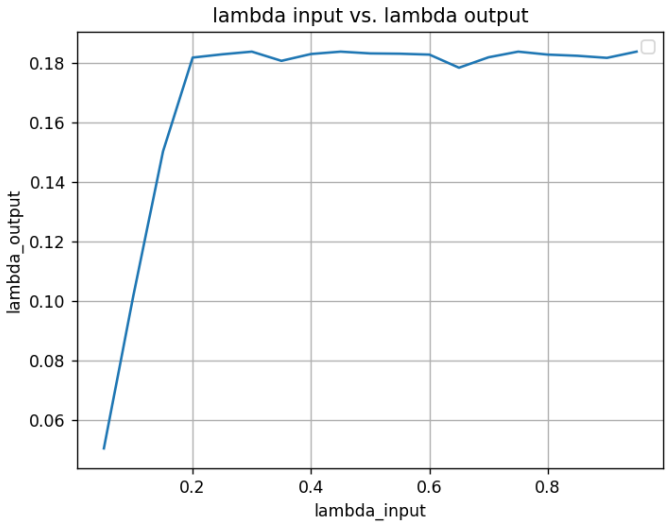

Рисунок 11. График зависимости интенсивности входного потока от выходного потока для 15 пользователей

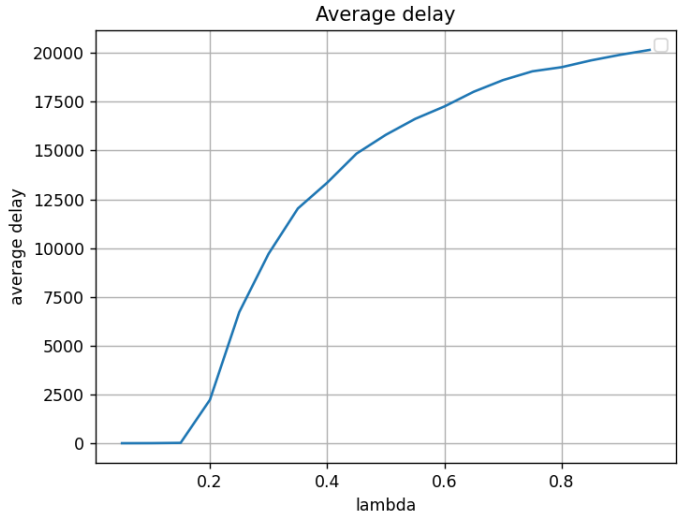

Рисунок 12. график средней задержки для 15 пользователей

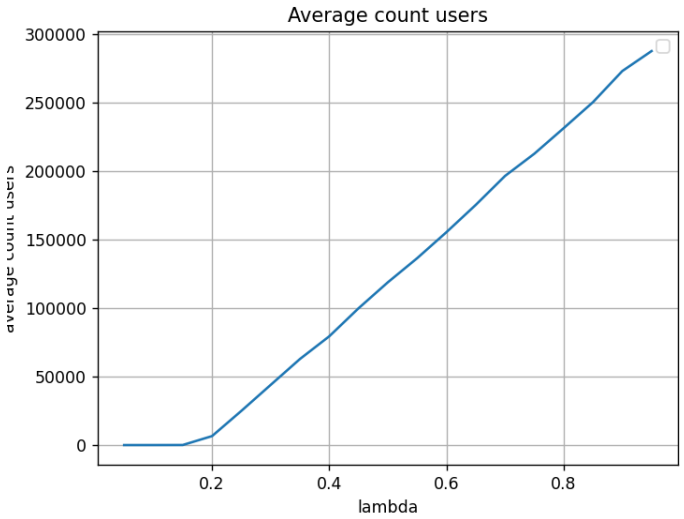

Рисунок 13 . График среднего числа сообщений для 15 абонентов

Выводы

Таким образом, в ходе выполнения данной лабораторной работы, была рассмотрена и изучена работа алгоритма двоичной экспоненциальной отсрочки. Так же были выявлены следующие выводы по построенным графикам: после преодоления отметки критической интенсивности потока, система перестает работать корректно.

Листинг кода

```
    import math
from alive_progress import alive_bar
import queue
import random
from pathlib import Path
from message import Message
from user import User
import graphics

MAX_TIME = 50000

file_practice_D = None
file_theoretic_D = None
file_practice_N = None
file_theoretic_N = None
file_lambda = None
# queue_messages = queue.Queue()
count_users_in_system = 0

count_users_in_one_window = 10
users_in_one_window = []
current_sending_user = 0

my_lambda_out = 0


def init_files():
    global file_practice_D
    global file_practice_N
    global file_theoretic_D
    global file_theoretic_N
    global file_lambda
    file_path_practic_D = Path(Path.cwd().parent, "outputData",
"10average_delay.txt")
    # file_path_theoretic_D = Path(Path.cwd().parent,
"outputData", "synch_theoretic_D.txt")
    file_path_practic_N = Path(Path.cwd().parent, "outputData",
"10average_N.txt")
    # file_path_theoretic_N = Path(Path.cwd().parent,
"outputData", "synch_theoretic_N.txt")
    file_path_lambda = Path(Path.cwd().parent, "outputData",
"10lambda.txt")

    file_practice_D = open(file_path_practic_D, 'w')
    # file_theoretic_D = open(file_path_theoretic_D, 'w')
    file_practice_N = open(file_path_practic_N, 'w')
    # file_theoretic_N = open(file_path_theoretic_N, 'w')
    file_lambda = open(file_path_lambda, 'w')


def get_queue_size(my_lambda):
```

```
    L = math.exp(-my_lambda)
    # L = math.pow(10, -5)
    p = 1.0
    k = 0
    while True:
        k += 1
        p *= random.random()
        if p < L:
            break
    return k - 1


def create_queue_messages(my_lambda, t, index_user):
    size_queue = get_queue_size(my_lambda /
count_users_in_one_window)

    values = []
    for i in range(size_queue):
        values.append(random.random())
    values.sort()
    for i in range(size_queue):
        bf = Message(values[i] + t)
        # queue_messages.put(bf)

users_in_one_window[index_user].add_new_message_in_queue(bf)
    global count_users_in_system
    count_users_in_system +=
users_in_one_window[index_user].get_size_queue()
    # return queue_messages


def check_conflict():
    count_users_sending = 0
    users_sending = []
    for i in range(count_users_in_one_window):
        if users_in_one_window[i].choose_pass_or_dont_pass():
            users_sending.append(i)
            count_users_sending += 1
    if count_users_sending > 1:  # событие "конфликт"
        for index in users_sending:
            users_in_one_window[index].was_conflict()
        return True
    elif count_users_sending == 1:  # событие "успех"
        global current_sending_user
        # print(users_sending)
        current_sending_user = users_sending[0]
        users_in_one_window[current_sending_user].was_success()
        return False
    elif count_users_sending == 0:  # событие "пусто"
        return True


def clear_data_for_all_users():
```

```
    for i in range(count_users_in_one_window):
        # print(f"clear queues ")
        # print(f"size[{i}] before =
{users_in_one_window[i].get_size_queue()}")
        users_in_one_window[i].set_start_P()
        users_in_one_window[i].clear_queue()
        # print(f"size[{i}] after =
{users_in_one_window[i].get_size_queue()}\n")


def add_users_in_system():
    global users_in_one_window
    global count_users_in_one_window
    for _ in range(count_users_in_one_window):
        users_in_one_window.append(User())


def create_queue_for_all_users(my_lambda, t):
    for i in range(count_users_in_one_window):
        create_queue_messages(my_lambda, t, i)


def print_p_users():
    for i in range(count_users_in_one_window):
        print(f"{i} user")
        users_in_one_window[i].print_P()


def set_P_min_for_all_users(p_min: int):
    for i in range(count_users_in_one_window):
        users_in_one_window[i].set_min_p(p_min)


def find_critical_P_min():
    add_users_in_system()
    file_path = Path(Path.cwd().parent, "outputData",
                     f"critical_P_min
{count_users_in_one_window} users lambda=0.33.txt")
    p_min = 0.05
    with open(file_path, 'w') as file:
        with alive_bar(int(1 / 0.05 - 1), dual_line=True) as
bar:
            bar.text = '\t-> working, please wait...'
            while p_min < 1:
                set_P_min_for_all_users(p_min)
                simulate_messaging(0.31)
                file.write(f"{round(p_min, 4)}
{round(get_lambda_out(), 4)}\n")
                p_min += 0.05
                bar()


def simulate_messaging(my_lambda):
```

```
    # print(my_lambda)
    clear_data_for_all_users()
    t = 0
    global my_lambda_out
    sent_messages = []
    # print(len(users_in_one_window))
    while t < MAX_TIME:
        # print(f"t = {t}")
        # print_p_users()
        # print(users_in_one_window[0].get_queue_empty())
        # if not users_in_one_window[0].get_queue_empty():
        if not check_conflict():  # если не произошел конфликт
            buffer_message =
users_in_one_window[current_sending_user].get_one_message()
            # print(current_sending_user)
            buffer_message.exit_time = t + 1
            sent_messages.append(buffer_message)
            my_lambda_out += 1

        create_queue_for_all_users(my_lambda, t)
        # count_users_in_system += queue_messages.qsize()

        t += 1

    global count_users_in_system
    count_users_in_system /= MAX_TIME
    my_lambda_out /= MAX_TIME
    return sent_messages


def get_average_practical_delay(my_lambda):
    delay = 0
    sent_message = []
    global count_users_in_system
    count_users_in_system = 0
    sent_message = simulate_messaging(my_lambda)
    for i in range(len(sent_message)):
        delay += sent_message[i].get_delta()

    # print("delay = ", delay)
    # average_delay = delay / len(sent_message)
    # print("average delay = ", average_delay)
    # print("\n")
    return delay / len(sent_message)


def get_average_theoretical_delay(my_lambda):
    d = (my_lambda * (2 - my_lambda)) / (2 * (1 - my_lambda))
    return d / my_lambda + 0.5


def get_average_count_users(my_lambda):
    global count_users_in_system
```

```
    return count_users_in_system


def get_average_theoretical_count_users(my_lambda):
    n = (my_lambda * (2 - my_lambda)) / (2 * (1 - my_lambda))
    return n


def get_lambda_out():
    global my_lambda_out
    return my_lambda_out


def make():
    init_files()
    my_lambda = 0.05
    # print("synchronous system")
    count_step = int(1 / 0.05 - 1)
    add_users_in_system()
    with alive_bar(count_step, dual_line=True) as bar:
        bar.text = '\t-> The synchronous system working, please
wait...'
        while my_lambda < 1:
            # print("lambda = ", my_lambda)
            file_practice_D.write(f"{round(my_lambda, 3)}
{round(get_average_practical_delay(my_lambda), 4)}\n")
            # file_theoretic_D.write(f"{round(my_lambda, 3)}
{round(get_average_theoretical_delay(my_lambda), 4)}\n")
            file_practice_N.write(f"{round(my_lambda, 3)}
{round(get_average_count_users(my_lambda), 4)}\n")
            # file_theoretic_N.write(
            #     f"{round(my_lambda, 3)}
{round(get_average_theoretical_count_users(my_lambda), 4)}\n")
            file_lambda.write(f"{round(my_lambda, 3)}
{round(get_lambda_out(), 4)}\n")
            my_lambda += 0.05
            bar()
    file_practice_D.close()
    file_practice_N.close()
    # file_theoretic_D.close()
    # file_theoretic_N.close()
    file_lambda.close()


if __name__ == "__main__":
    make()
    # graphics.draw_all_graphics()
    # find_critical_P_min()
    # add_users_in_system()
    # simulate_messaging(0.95)
```

class User

```
import queue
import random
# import binary_exponential_delay_algorithm

from message import Message


class User:
    __P_previous = 1.0  # P_(t)
    __P_current = 1.0  # P_(t-1)
    __P_min = 0.15
    __P_max = 1.0
    __P_most_min = 1.0
    __queue_message = queue.Queue()

    def __init__(self, P_previous=1.0, P_current=1.0):
        self.__P_previous = self.__P_max
        self.__P_current = self.__P_max

    def set_min_p(self, p_min):
        self.__P_min = p_min

    # @property
    # def P_current(self):
    #     return self.__P_current
    #
    # @property
    # def P_privious(self):
    #     return self.__P_previous
    def get_minimal_P(self):
        return self.__P_most_min

    def set_start_P(self):
        self.__P_previous = self.__P_max
        self.__P_current = self.__P_max

    def get_one_message(self):
        return self.__queue_message.get()

    def set_queue(self, q: queue.Queue()):
        # self.__queue_message.queue.clear()
        self.__queue_message = q

    def add_new_message_in_queue(self, message: Message):
        self.__queue_message.put(message)

    def choose_pass_or_dont_pass(self):
        if random.random() <= self.__P_current and
self.__queue_message.qsize() > 0:
            # self.__P_previous = self.__P_current
            return True
        else:
```

```
            return False

    def was_conflict(self):
        self.__P_current = max(self.__P_current / 2,
self.__P_min)

    def was_success(self):
        # self.__P_previous = self.__P_current
        self.__P_current = self.__P_max

    def clear_queue(self):
        self.__queue_message.queue.clear()

    def get_queue_empty(self):
        return self.__queue_message.empty()

    def get_size_queue(self):
        return self.__queue_message.qsize()

    def print_P(self):
        print(f"P_previous = {self.__P_previous}"
              f"\nP_current = {self.__P_current}"
              f"\nP_min = {self.__P_min}"
              f"\nP_max = {self.__P_max}\n")
    # def create_queue(self, size_queue: int):
    #     values = []
    #     for i in range(size_queue):
    #         values.append(random.random())
    #     values.sort()
    #     for i in range(size_queue):
    #         bf = Message(values[i])
    #         self.__queue_message.put(bf)
    #
    # def print_all_queue(self):
    #     i = 0
    #     print(self.__queue_message.empty())
    #     while not self.__queue_message.empty():
    #         self.__queue_message.get().print()
    #         i += 1


def create_q(size_queue: int):
    values = []
    qu = queue.Queue()
    for i in range(size_queue):
        values.append(random.random())
    values.sort()
    for i in range(size_queue):
        bf = Message(values[i])
        qu.put(bf)
    return qu
```

```
if __name__ == '__main__':
    us = User()
    print(us.P_current, us.P_privious)
    # us.create_queue(10)
    # us.set_queue(create_q(10))
    print(us.get_queue_empty())
    us.print_all_queue()
    print(us.choose_pass_or_dont_pass())
```