**Введение**

В настоящее время электронные очереди прочно закрепились на пьедистале управления и распределения нагрузки между субъектами обслуживания. Мы постоянно сталкиваемся с ними в различных банках, юридических и правовых фирмах, почтовых отделениях и медицинских заведениях. Правильная организация и реализация электронной очереди способна увеличить пропускную способность исполнителей услуг, а так же исключить спорные моменты среди участников очередей. Использование такого вида управления, прежде всего, должно быть направлено на удовлетворенние посетителя и уменьшение его времени пребывания в очереди.

Следует отметить, что зачастую инженеры, которые внедряют данную технологию в компании, не уделяют должного внимания части, которая напрямую задействует потребителей услуг – распределение времени. В сзвязи с этим, руководители и исполнители вынуждены слушать претензии и смотреть на споры среди посетителей. Из-за неправильно выбраных технологических решений, практически никогда не достигается планируемый уровень посещаемости, и пропускной способности, что влечет за собой значительную потерю прибыли.

Изменив подход к распределению времени в очередях, уход от текущего вида буквенно-цифровых обозначений, использование прогнозируемого времени заявки при определенном ее типе, простая электронная очередь может превратиться в надежный и эффективный инструмент управления потоком клиентов. Важно помнить, что для многих клиентов, время, которое они тратят на получение учлуги, зачастую, имеет более высокую ценность, чем сама услуга. Что, в свою очередь, может побудить их перейти к конкурентам, где время получения услуги может быть значительно меньше из-за другого подхода к очереди. К тому же, чем больше удовлетворенных посетителей или клиентов покинут компанию, тем больше вероятность, что они вновь вернутся или дадут рекомендации своим знакомым.

Цель работы: Разработать новый алгоритм управления электронной очередью, который на основании результатов машинного обучения динамически перестраивает очередь.

Такой вариант будет интересен как пользователям, так и исполнителям, так как позволяет экономить время и увеличивает пропускную способность.

Пользователь выбирает из списка необходимую услугу и на квитке с номером видит прогнозируемое время обслуживания по данному типу услуги.

Объект исследования**:**процесс создания програмного обеспечения для управления электронными очередями.

Предмет исследования**:**программные инструменты для реализации модели электронной очереди.

Практическая значимость и актуальность и дипломной работы заключается в повышении эффективности оказания услуг и увеличении дохода организаций.

Гипотеза дипломной работы основана на том, что новый тип модели для электронных очередей может быть эффективным средством для развития бизнеса, если будут корректно собираться исходные данные, которые необходимы для анализа и использования машинного обучения.

Для достижения поставленной цели необходимо решить следующие **задачи:**

* изучить принципы работы электронных очередей;
* исследовать состояние и модели очередей в городских организациях;
* выбрать язык программирования;
* разработать по для управления электронной очередью;

Для решения поставленных задач применялись следующие методы исследования: анализ, сравнение, наблюдение, апробирование, анкетирование.

Структура работы обусловлена предметом, целью и задачами исследования. Работа состоит из введения, двух глав, заключения и списка литературы.

Введение раскрывает актуальность, определяют степень научной разработки темы, объект, предмет, цель, задачи и методы исследования, раскрывают теоретическую и практическую значимость работы.

Глава 1. Теоретические основы разработки электронных очередей, общие понятия и характеристики

* 1. Основные понятия

Система управления электронной очередью - комплекс, включающий в себя как програмную, так и аппаратную составляющую, позволяющий производить оптимизацию и формализацию управления потоком посетителей. Главной целью системы является распределение посетителей внутри организации и получение информации о наиболее востребованных типах услуг, времени их исполнения и различных статистик.

* 1. Сравнительный анализ систем электронных очередей

Для проведения сравнительного анализа систем электронных очередей, прежде всего, необходимо определить основные критерии сравнения. Для этого рассмотрим несколько, наиболее важных параметров ЭО.

1. Среднее время ожидания – среднее время, которое тратит 1 клиент, находясь в очереди в ожидании оказания услуги.
2. Максимальное время ожидания
3. Максимальная длина очереди

Так же, будем рассматривать сами модели очередей и их устройство, как они учитывают различные категории граждан и типы обращений.

Наиболее распространенной, в текущей момент, является модель очереди, имеющая тип FIFO – первый пришел, первый ушел. Данный тип характерен для различных почтовых отделений, организаций, занимающихся документооборотом, больниц и других гос учреждений. Приоритет имеют только льготне категории граждан. Для них создается отдельная очередь, которая помещаяется в начало основной. Для остальных пользователей используется обычный аналог живой очереди.

Для усовершенствования модели очереди и сокращения среднего времени ожидания и остальных показателей, я предлагаю использовать сегментную сортировку внутри очереди, которая будет основываться на истории заявок. Основным отличием от общепринятой системы является учет типа заявки и предполагаемое время ее исполнения.

Предполагается, что данный вид системы управления электронной очередью способен значительно сократить среднее время ожидания в очереди, максимальное время нахождения в ней и ее длину.

* 1. Средства разработки

Язык программирования - Python 3

Основные плюсы данного языка – простота работы с данными, возможность использования библиотек для машинного обучения, простые этапы для визуализации данных.

Так же, огромным плюсом является то, что данный язык кроссплатформенный, что позволяет запускать программу на различных ОС.

В качестве хранилища данных была выбрана база данных SQLite, так-как она способна в кротчайшие сроки предоставить нужный функционал для разработки.

В совокупности с базой данных SQLite, данный тандем способен полноценно продемонстрировать отличия моделей, их поведение, быстродействие и отобразить все данные на графиках.

Так-как и Python 3 и SQLite являются проектами с открытым исходным кодом, они могут использоваться без каких либо ограничений.

Глава 2. Разработка и программная реализация алгоритма построения электронной очереди.

2.1. Основная структура программы

Программа будет состоять из 2 основных разделов:

1) Конструктор очереди – Окно для формирования параметров очередей

2) Отчетный лист – Окно для вывода результатов

В проекте будут присутствовать 4 файла:

1. Библиотека для работы с базой данных
2. База данных
3. Основной скрипт для моделирования и оптимизации очередей
4. Файл логирования событий

Из основного скрипта, путем импорта библиотек, будет получен доступ к таблицам базы данных, что позволит оперировать объектами, обходя SQL запросы.

Основными сущностями являются:

1. Операция – основная сущность, представляющая из себя уникальный номер операции, ее категория, название и тип заявителя.
2. Льгота – сущность из справочника всех доступных льгот
3. Тип операции – сущность из справочника операций, которые доступны для исполнения

Для заполнения списка операций используются значения из справочников льгот и типов операций.

Основной модуль программы сочитает в себе как графическую, так и аналитическую части исполняемого ПО. Для возможности технической поддержки, дополнительно реализована возможность логирования событий, в частности, работа с базой данных.

Часто используемые самостоятельные подзадачи оформлены в виде классов и функций которые храниться в основном модуле.

**2.3. Разработка бизнес части**

Для эмулятора очереди использован подход универсальных точек исполнения. Они равнозначны и могут исполнять все категории заявок.

Для реализации очереди используется сегментированный список, степень сегментации влияет на количество заявок, которые находятся в одном кластере.

Если вместимость блока равна единице, то очередь работает как монотонная без сортировок

Когда размер кластера увеличивается, поступающие заявки заполняют ячейки и сортируются по прогнозируемогу времени. Упорядочивание внутри блоков гарантирует, что ни один клиент не будет постоянно двигаться в конец очереди.

Приложение содержит 2 оновных функциональных сервиса

1. Сервис анализа
2. Сервис оптимизации

2.3.1

Сервис анализа представляет собой модуль, который моделирует жизненный процесс электронной очереди на указанных параметрах.

После запуска происходит генерация потока посетителей, которые посетят организацию за указанный временной промежуток. Каждый объект очереди прихода состоит из типа посетителя, заявки, времени, которое должно пройти между заходящими людьми. Услуга генерируется по данным истории из базы данных.

Как только процесс заполнения входной очереди заканчивается, начинается обычный рабочий день. Постепенно посетители приходят и выбирают свои услуги, далее в работу включается алгоритм формирования и управления очередью. Перед попаданием в поток, заявка приобретает еще один параметр – время исполнения. Он рассчитывается из истории и времени выполнения таких же услуг, либо имеют такую же категорию. Когда сформированы новые структуры, очередь, если указан параметр сегментации, происходит сортировка внутри сегментов. После каждой новой заявки, сегментирование идет внутри последней части очереди. Данных подход гарантирует, что ни одна заявка не будет отодвигаться постоянно на последнее место. Если же сегментаия не указана, то распределение работает как в обычной системе FIFO, первый пришел первый ушел. Все кассы равнозначны, каждая может оказывать весь спектр услуг. В процессе работы словари накапливают данные, которые необходимы для рассчета выходных параметров.

Прогон осуществляется 1000 раз, выходные параметры все это время накапливаюстся и в конечном результате вычисляются средние значения, которые, уже, выводятся пользователю

2.3.2

Сервис оптимизации представляет собой модуль, который моделирует жизненный процесс электронной очереди в пределах указанных параметров.

Данный алгоритм последовательно проходит все сочетания входных данных и находит наиболее благоприятные значения параметров. Оптимизация может осуществляться по одному из нескольких выходных параметров, например, минимизация среднего времени ожидания, длины очереди и т.д

**2.3. Разработка дизайна приложения**

В качестве графического ядра приложения был выбран фреймворк tkinter, который предоставляет расширенный функцинал для создания окон и их наполнения.

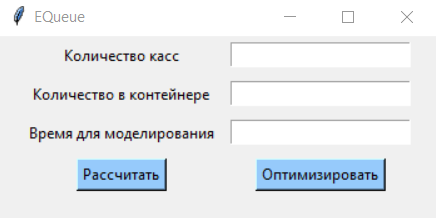
Режим дизайна был выбран макетный, он показывает примерное расположение объектов на фрейме и позволяет получить полноценный доступ к бизнес сервисам приложения.

На главном окне решено разместить 3 поля

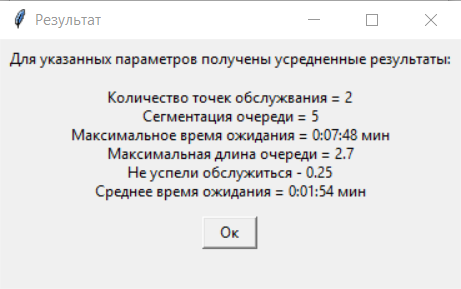
1. Количество касс – поле, которое отвечает за точное количество исполнителей в режиме анализа, или за верхнюю границу количества точек исполнения заявок в режиме оптимизации
2. Количество в контейнере - поле, которое отвечает за точное количество заявок в контейнере сепарации в режиме анализа, или за верхнюю границу числа операций в отдельном массиве в режиме оптимизации
3. Время для моделирования – количество минут для моделирования

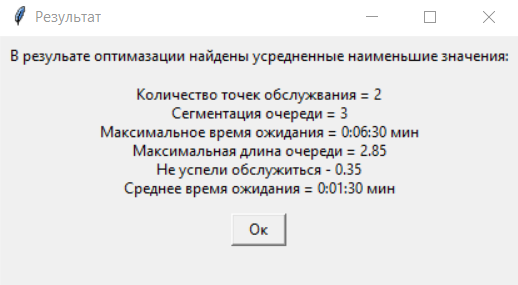
Ниже полей присутствуют 2 кнопки управления

1. Рассчитать – отвечает за запуск бизнес процесса анализа по точным параметрам
2. Оптимизировать - отвечает за запуск бизнес процесса оптимизации в указанных границах



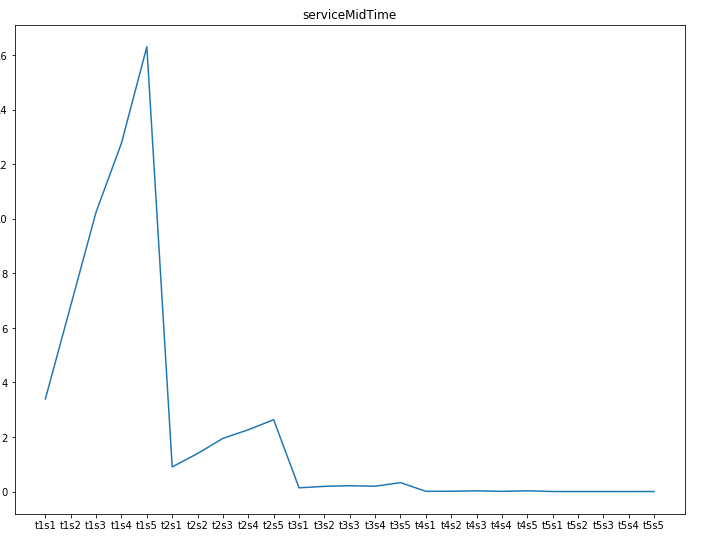
Окно отчета представляет собой список основных характеристик очереди, для режима анализа выводится результат по указанным параметрам, в режиме оптимизации по наилучшему.





Дополнительно, в режиме оптимизации выводится график зависимостей параметра от переменного количества точек обслуживания и сегментации

Ниже представлен график зависимостей среднего времени обслуживания от переменных.



Приложения

1. DatBaseConnector

import logging

import sqlite3

# add filemode="w" to overwrite

logging.basicConfig(filename="diplom.log", level=logging.INFO, format = "%(asctime)s %(levelname)s - %(message)s")

#logging.debug("")

#logging.info("")

#logging.error("")

def stopLogging():

logging.shutdown()

class Operation():

def \_\_init\_\_(self, stringFromDB):

self.id = stringFromDB[0]

self.operation\_name = stringFromDB[1]

self.operation\_type = stringFromDB[2]

self.operation\_time = stringFromDB[3]

self.concession\_grade = stringFromDB[4]

def getId(self):

return self.id

def getOperation\_name(self):

return self.operation\_name

def getOperation\_type(self):

return int(self.operation\_type)

def getOperation\_time(self):

return int(self.operation\_time)

def getConcession\_grade(self):

return int(self.concession\_grade)

def toString(self):

resStr = "id:" + str(self.id) + "; "

resStr += "operation\_name:" + str(self.operation\_name) + "; "

resStr += "operation\_type:" + str(self.operation\_type) + "; "

resStr += "operation\_time:" + str(self.operation\_time) + "; "

resStr += "concession\_grade:" + str(self.concession\_grade)

return resStr

class datBaseConnector():

def \_\_init\_\_(self):

logging.info("init connection")

self.creator()

#returnMode:

#fetchOne

#fetchAll

#strCostr

def execute(self, query, param = None, returnMode = ''):

logging.info(query + " " + (str(param)))

self.conn = sqlite3.connect('diplom.sqlite', timeout=100)

self.cursor = self.conn.cursor()

try:

if param == None:

self.cursor.execute(query)

else:

self.cursor.execute(query, param)

self.conn.commit()

except Exception as err:

logging.error('execute failed: %s : %s' % (query, str(err)))

result = None

if(returnMode == 'fetchOne'):

result = self.cursor.fetchone()

elif(returnMode == 'fetchAll'):

result = self.cursor.fetchall()

elif(returnMode == 'operations'):

result = self.getListOfOperations(self.cursor)

elif(returnMode == 'types'):

result = self.getTypes(self.cursor)

elif(returnMode == 'oper\_descr'):

result = self.getDictionaryOfOperationsAndDescriptions(self.cursor)

self.closeConnect()

return result

def creator(self):

self.execute("""CREATE TABLE `time\_records` (

` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,

`operation\_name` TEXT NOT NULL,

`operation\_type` INTEGER NOT NULL,

`operation\_time` BLOB NOT NULL,

`concession\_grade` INTEGER NOT NULL DEFAULT 0

);""")

self.execute("""CREATE TABLE `operation\_types` (

`operation\_number` INTEGER UNIQUE,

`operation\_name` TEXT NOT NULL UNIQUE,

PRIMARY KEY(`operation\_number`)

);""")

self.execute("""CREATE TABLE `concession\_grades` (

`concession\_grade` INTEGER NOT NULL UNIQUE,

`concession\_name` TEXT NOT NULL UNIQUE,

PRIMARY KEY(`concession\_grade`)

); """)

# выбор всех записей определенного типа

def selectByType(self, operationType):

stringToExec = "select \* from time\_records where operation\_type = " + str(operationType)

return self.execute(stringToExec, returnMode = 'operations')

def selectOperationTypes(self):

stringToExec = "select \* from operation\_types"

return self.execute(stringToExec, returnMode = 'types')

def selectUniqeDescriptions(self):

stringToExec = "select distinct \* from time\_records"

return self.execute(stringToExec, returnMode = 'oper\_descr')

def getListOfOperations(self, cursor):

operations = []

firstDat = cursor.fetchone()

if firstDat == None:

return None

operations.append(Operation(firstDat))

while firstDat is not None:

operations.append(Operation(firstDat))

firstDat = cursor.fetchone()

return operations

def getDictionaryOfOperationsAndDescriptions(self, cursor):

operationsAndDescriptions = {}

firstDat = cursor.fetchone()

if firstDat == None:

return None

operation = Operation(firstDat)

operationsAndDescriptions[operation.getOperation\_name()] = operation.getOperation\_type()

while firstDat is not None:

operation = Operation(firstDat)

operationsAndDescriptions[operation.getOperation\_name()] = operation.getOperation\_type()

firstDat = cursor.fetchone()

return operationsAndDescriptions

def getTypes(self, cursor):

types = []

firstDat = cursor.fetchone()

types.append(firstDat)

while firstDat is not None:

types.append(firstDat)

firstDat = cursor.fetchone()

return types

def closeConnect(self):

logging.info("closeConnect")

self.conn.close()

2. diplom

import random

import math

import statistics

from DatBaseConnector import datBaseConnector

from DatBaseConnector import Operation

import matplotlib.pyplot as plt

import numpy as np

from tkinter import tix as tk

from tkinter import \*

from tkinter import ttk

repeatCount = 20

incomeMidTime = 4

#Класс описывающий среднее свремя по категориям для отдельного типа обращения

class BaseTime():

def \_\_init\_\_(self, timeObject):

self.usualPerson = timeObject[0]

self.oldPerson = timeObject[1]

self.invalidPerson = timeObject[2]

self.timeList = timeObject

def getUsualPerson(self):

return self.usualPerson

def getOldPerson(self):

return self.oldPerson

def getInvalidPerson(self):

return self.invalidPerson

def getByNumber(self, number):

return self.timeList[number - 1]

#класс, хранящий словарь из типов операций и соответствующих им классов BaseTime

class BaseTimeInfo():

def \_\_init\_\_(self):

self.times = {}

def addNewCalc(self, listOfOperations):

operationTimes = [[], [], []]

for operation in listOfOperations:

operationTimes[operation.getConcession\_grade() - 1].append(operation.getOperation\_time())

def getMedian(timesList):

if(len(timesList) > 0):

return statistics.median(timesList)

return 1

self.times[listOfOperations[0].getOperation\_type()] = BaseTime(

[getMedian(operationTimes[0]),

getMedian(operationTimes[1]),

getMedian(operationTimes[2])])

#работа только с объектом типа Операция

def randZeroToOne():

return random.random()

#описывает элемент очереди - операция и время выполнения

class QueueUnit():

def \_\_init\_\_(self, operation, unitTime):

self.operation = operation

self.unitTime = unitTime

def getOperation(self):

return self.operation

def getUnitTime(self):

return self.unitTime

def increaseTime(self):

self.unitTime -= 1

if self.unitTime == 0:

return self.operation

return None

#Ксласс, описывающий заполнение входной очереди, на вход принимает время моделирования в минутах

class InputQueueEngine():

def \_\_init\_\_(self, limitModelingTime):

self.limitModelingTime = limitModelingTime

self.\_\_initParameters()

self.commingQueue = []

self.\_\_initCommingQueue(incomeMidTime)

# устанавливает исходные параметры

def \_\_initParameters(self):

# хранит в себе словарь из типа операции и времени для различных типов посетителей

self.timeBr = BaseTimeInfo()

# коннектор бд

self.connect = datBaseConnector()

# все уникальные операции, по которым есть информация в базе

self.uniqeOperations = self.connect.selectUniqeDescriptions()

self.\_\_fillTimeBr()

# заполняет словарь средних по времени для каждой операции

def \_\_fillTimeBr(self):

for element in self.connect.selectOperationTypes():

operations = self.connect.selectByType(element[0])

if operations != None:

self.timeBr.addNewCalc(operations)

# заполняет очередь через случайные заявки и случайных людей

def \_\_initCommingQueue(self, middleTime):

tempTime = 0

while tempTime < self.limitModelingTime:

randomTimeComming = self.\_\_getRandomCommingTime(middleTime)

self.commingQueue.append(QueueUnit(self.\_\_genRandomOperation(), randomTimeComming))

tempTime += randomTimeComming

def \_\_getRandomCommingTime(self, middleTime):

randomTimeComming = int(np.random.poisson(middleTime, 1))

while randomTimeComming <= 0:

randomTimeComming = int(np.random.poisson(middleTime, 1))

return randomTimeComming

# Возвращает случайный тип посетителя

def \_\_getRandomConcession(self):

return random.randint(1, 3)

# Возвращает случайно определенную операцию

def \_\_genRandomOperation(self):

randDescr = random.choice(list(self.uniqeOperations.keys()))

randomOperType = self.uniqeOperations[randDescr]

randConcession = self.\_\_getRandomConcession()

randomMidTime = self.timeBr.times[randomOperType].getByNumber(randConcession)

while(randomMidTime < 1):

randomMidTime = self.timeBr.times[randomOperType].getByNumber(1)

randomOperTime = int(random.gauss(randomMidTime, 3))

while(randomOperTime < 1):

randomOperTime = int(random.gauss(randomMidTime, 3))

return Operation([1,

randDescr,

randomOperType,

randomOperTime,

randConcession])

def tryToPopFromQueue(self):

if(self.inputQueueGet()):

return self.commingQueue.pop(0)

# Вернет операцию, если человек пришел, None если нет

def inputQueueGet(self):

if(len(self.commingQueue) > 0):

return self.commingQueue[0].increaseTime()

def printStat(self):

for unit in self.commingQueue:

print("элемент очереди " + unit.getOperation().toString())

print("имеет время прихода = " + str(unit.getUnitTime()))

#Класс описывающий кассу

class Till():

def \_\_init\_\_(self, tillNumber, serviceTime = 0):

self.tillNumber = tillNumber

self.isVacant = True

self.serviceTime = serviceTime

self.serviceCount = 0

def getTillNumber(self):

return self.tillNumber

def getIsVacant(self):

return self.isVacant

def getServiceCount(self):

return self.serviceCount

def getServiceTime(self):

return self.serviceTime

def initNewTime(self, operation):

self.serviceTime = operation.getOperation\_time()

self.changeStatus()

def \_\_increaseServiceCount(self):

self.serviceCount += 1

def changeStatus(self):

self.isVacant = not self.isVacant

return self.isVacant

def increaseServiceTime(self):

if(self.isVacant):

return True

else:

if(self.serviceTime > 0):

self.serviceTime -= 1

else:

self.changeStatus()

self.\_\_increaseServiceCount()

def toString(self):

print(str(self.tillNumber) + str(self.isVacant))

def getStat(self):

print("Статистика для " + str(self.tillNumber) + " кассы:")

print("Обслужано: " + str(self.serviceCount))

#Класс реализующий работу касс, инициализируется количеством касс

class TillEngine():

def \_\_init\_\_(self, tillCount):

self.tillCount = tillCount

self.listOfTills = [Till(x) for x in range(tillCount)]

# попытка операции попасть к оператору, результат - успешно/нет

def callFromTill(self, operation):

for till in self.listOfTills:

if(till.getIsVacant()):

till.initNewTime(operation)

return True

return False

# во всех точках обслуживания уменьшается время на 1

def increaseTimeAllTills(self):

for till in self.listOfTills:

till.increaseServiceTime()

def getStat(self):

for till in self.listOfTills:

till.getStat()

class QueueObject():

def \_\_init\_\_(self, queueUnit):

self.waitingTime = 0

self.queueUnit = queueUnit

def increaseTime(self):

self.waitingTime += 1

#основная модель очереди

class GeneralStandartQueue():

def \_\_init\_\_(self, separation):

self.generalQueue = []

self.separation = separation

self.waitingTimeArray = []

self.queueMaxLen = 1

def addOperationIntoQueue(self, queueUnit):

if(len(self.generalQueue) == 0):

listOp = []

listOp.append(QueueObject(queueUnit))

self.generalQueue.append(listOp)

else:

tempOp = self.generalQueue[len(self.generalQueue) - 1]

if(len(tempOp) == self.separation):

listOp = []

listOp.append(QueueObject(queueUnit))

self.generalQueue.append(listOp)

else:

tempOp.append(QueueObject(queueUnit))

actualLength = self.getLengthCode()

if (self.queueMaxLen < actualLength):

self.queueMaxLen = actualLength

self.sortInQueuePart(self.generalQueue[len(self.generalQueue) - 1])

def getLengthCode(self):

if ((self.generalQueue) == None):

return 0

lentemp = 0

for queueObjectList in self.generalQueue:

lentemp += len(queueObjectList)

return lentemp

def popOperationFromQueue(self):

popOper = None

if(len(self.generalQueue) > 0):

popOper = self.generalQueue[0].pop(0)

if(len(self.generalQueue[0]) == 0):

self.generalQueue.pop()

self.waitingTimeArray.append(popOper.waitingTime)

return popOper.queueUnit

def increaseAllTimes(self):

for arr in self.generalQueue:

for queueObject in arr:

queueObject.increaseTime()

def sortInQueuePart(self, queuePart):

tempTr = []

for party in queuePart:

tempTr.append(party.queueUnit.operation.getOperation\_time())

queuePart = sorted(queuePart, key=lambda queueUnit: queueUnit.queueUnit.operation.getOperation\_time())

tempTr = []

for party in queuePart:

tempTr.append(party.queueUnit.operation.getOperation\_time())

def getWaitingTimes(self):

return self.waitingTimeArray

def getMaxQueueLen(self):

return self.queueMaxLen

def getFinalLenState(self):

return len(self.generalQueue)

def getFirstOperation(self):

if(len(self.generalQueue) > 0):

if(len(self.generalQueue[0]) > 0):

return self.generalQueue[0][0].queueUnit

return None

def getStat(self):

print("Количество людей в очереди GeneralStandartQueue = " + str(len(self.generalQueue)))

# главный класс-модель работы почты

class PostModel():

def \_\_init\_\_(self, initTime, tillCount, separateValue):

self.initTime = initTime

self.inputQueueEngine = InputQueueEngine(initTime)

self.tillCount = tillCount

self.separateValue = separateValue

self.tillEngine = TillEngine(tillCount)

self.genQueue = GeneralStandartQueue(separateValue)

def start(self):

tempTime = 0

while(tempTime < self.initTime):

# попытка добавить операцию в очередь, проверка очереди прихода

tempOper = self.inputQueueEngine.tryToPopFromQueue()

# self.genQueue.getStat()

if(tempOper != None):

self.genQueue.addOperationIntoQueue(tempOper)

# попыка достать операцию и отправить на обслуживание

tempOper = self.genQueue.getFirstOperation()

if(tempOper):

if(self.tillEngine.callFromTill(tempOper.getOperation())):

self.genQueue.popOperationFromQueue()

# уменьшает все время на кассах

self.tillEngine.increaseTimeAllTills()

# добавляет время ожидания всем в очереди

self.genQueue.increaseAllTimes()

tempTime += 1

def getTillsStat(self):

self.tillEngine.getStat()

def getStatDict(self):

waitingTimes = self.genQueue.getWaitingTimes()

# print(waitingTimes)

return {'tillCount' : self.tillCount,

'separateValue' : self.separateValue,

'maxWaitingTime' : max(waitingTimes),

'minWaitingTime' : min(waitingTimes),

'maxQueueLen' : self.genQueue.getMaxQueueLen(),

'countIgnored' : self.genQueue.getFinalLenState(),

'serviceMidTime' : sum(waitingTimes) / len(waitingTimes)}

def getUserStat(self):

stat = self.getStatDict()

print("Максимальное время ожмдания = " + str(stat['maxWaitingTime']))

print("Минимальное время ожмдания = " + str(stat['minWaitingTime']))

print("Максимальная длина очереди = " + str(stat['maxQueueLen']))

print("Не успели обслужиться - " + str(stat['countIgnored']))

print("Среднее время ожидания = " + str(stat['serviceMidTime']))

#создание модели для почты, параметры - время, количество точек обслуживания

generalTimeToModel = 80

class OptimalParameters():

def \_\_init\_\_(self, modelTime, tillCountMax, separateValueMax):

self.modelTime = modelTime

self.tillCountMax = tillCountMax

self.separateValueMax = separateValueMax

self.listOfStat = []

def tryToOptimise(self):

tillCount = 1

while(tillCount <= self.tillCountMax):

separateValue = 1

while(format(separateValue, '.3f') <= format(self.separateValueMax, '.3f')):

objTemp = MultipleTesting(self.modelTime, tillCount, separateValue)

self.listOfStat.append(objTemp.getStat())

separateValue += 1

tillCount += 1

def drawPlotByName(self, namePar):

listToDraw = []

lableList = []

for oneStat in self.listOfStat:

listToDraw.append(oneStat[namePar])

lableList.append("t" + str(oneStat['tillCount']) + 's' + str(oneStat['separateValue']))

plt.plot(lableList, listToDraw)

plt.title(namePar)

plt.rcParams["figure.figsize"] = [12, 9]

plt.show()

def getOptimalParameters(self, optimalType):

self.tryToOptimise()

theMostOptimal = None

if(len(self.listOfStat) > 0):

theMostOptimal = self.listOfStat[0]

for statObj in self.listOfStat:

if(statObj[optimalType] < theMostOptimal[optimalType]):

theMostOptimal = statObj

return theMostOptimal

class MultipleTesting():

def \_\_init\_\_(self, modelTime, tillCount, separateValue):

self.modelTime = modelTime

self.tillCount = tillCount

self.separateValue = separateValue

self.resultArray = self.calculate(self.modelTime, self.tillCount, self.separateValue)

def calculate(self, modelTime, tillCount, separateValue):

resultArray = []

tempI = 0

while(tempI < repeatCount):

tempPosMod = PostModel(modelTime, tillCount, separateValue)

tempPosMod.start()

resultArray.append(tempPosMod.getStatDict())

tempI += 1

return resultArray

def getStat(self):

maxWaitingTime = 0

minWaitingTime = 0

maxQueueLen = 0

countIgnored = 0

serviceMidTime = 0

for singleStat in self.resultArray:

maxWaitingTime += singleStat['maxWaitingTime']

minWaitingTime += singleStat['minWaitingTime']

maxQueueLen += singleStat['maxQueueLen']

countIgnored += singleStat['countIgnored']

serviceMidTime += singleStat['serviceMidTime']

statsLen = len(self.resultArray)

return {'tillCount' : self.tillCount,

'separateValue' : self.separateValue,

'maxWaitingTime' : maxWaitingTime / statsLen,

'minWaitingTime' : minWaitingTime / statsLen,

'maxQueueLen' : maxQueueLen / statsLen,

'countIgnored' : countIgnored / statsLen,

'serviceMidTime' : serviceMidTime / statsLen}

class GeneralFrame(Frame):

def \_\_init\_\_(self, parent):

Frame.\_\_init\_\_(self, parent)

self.parent = parent

self.modelTime = StringVar()

self.tillCount = StringVar()

self.separateValue = StringVar()

self.initUI()

def initUI(self):

self.parent.title("SovaVolunteer")

self.pack(fill=BOTH, expand=1)

frame = Frame(self)

frame.pack()

labelName = Label(frame, text = 'Количество касс').grid(row = 0, column = 0, padx=5, pady=5)

POSField = Entry(frame, font='Arial 10', textvariable=self.tillCount)

POSField.grid(row = 0, column = 1, columnspan = 1, padx=5, pady=5)

labelName = Label(frame, text = 'Количество в контейнере').grid(row = 1, column = 0, padx=5, pady=5)

containerField = Entry(frame, font='Arial 10', textvariable=self.separateValue)

containerField.grid(row = 1, column = 1, columnspan = 1, padx=5, pady=5)

labelName = Label(frame, text = 'Время для моделирования').grid(row = 2, column = 0, padx=5, pady=5)

timeField = Entry(frame, font='Arial 10', textvariable=self.modelTime)

timeField.grid(row = 2, column = 1, columnspan = 1, padx=5, pady=5)

button\_calc = Button(frame, text = 'Рассчитать', command = self.calculateFunc)

button\_calc.config(background="#96cafb")

button\_calc.grid(row = 3, column = 0, columnspan = 1, padx=5, pady=5)

button\_optim = Button(frame, text = 'Оптимизировать', command = self.optimise)

button\_optim.config(background="#96cafb")

button\_optim.grid(row = 3, column = 1, columnspan = 1, padx=5, pady=5)

frame.mainloop()

def optimise(self):

tillCount = int(self.tillCount.get())

separateValue = int(self.separateValue.get())

modelTime = int(self.modelTime.get())

optimPar = OptimalParameters(modelTime, tillCount, separateValue)

self.formReportFrame(optimPar.getOptimalParameters('countIgnored'), 1)

optimPar.drawPlotByName('countIgnored')

def calculateFunc(self):

tillCount = int(self.tillCount.get())

separateValue = int(self.separateValue.get())

modelTime = int(self.modelTime.get())

objTemp = MultipleTesting(modelTime, tillCount, separateValue)

self.formReportFrame(objTemp.getStat())

def formReportFrame(self, result, optimMode = None):

resFrame = Toplevel()

resFrame.title("Результат")

if(optimMode):

labelTableName = Label(resFrame, text="В резульате оптимазации найдены наименьшие значения:")

else:

labelTableName = Label(resFrame, text="Для указанных параметров получены результаты:")

widthFrame = 170

if(labelTableName.winfo\_reqwidth() + 10 > widthFrame):

widthFrame = labelTableName.winfo\_reqwidth() + 10

resFrame.geometry(str(widthFrame) + "x200")

labelTableName.grid(row = 0, column = 0, columnspan = 3, padx=5, pady=5)

textToLable = "Количество точек обслужвания = " + str(result['tillCount'])

textToLable += "\nСегментация очереди = " + str(result['separateValue'])

textToLable += "\nМаксимальное время ожидания = " + str(result['maxWaitingTime'])

textToLable += "\nМинимальное время ожмдания = " + str(result['minWaitingTime'])

textToLable += "\nМаксимальная длина очереди = " + str(result['maxQueueLen'])

textToLable += "\nНе успели обслужиться - " + str(result['countIgnored'])

textToLable += "\nСреднее время ожидания = " + str(result['serviceMidTime'])

labelStat = Label(resFrame, text = textToLable).grid(row = 1, column = 0, rowspan = 6, columnspan = 3, padx=5, pady=5)

def okay():

resFrame.destroy()

button\_decine = Button(resFrame, text = ' Ок ', command = okay)

button\_decine.grid(row = 10, column = 1, padx=5, pady=5)

root=Tk()

var = StringVar()

root.geometry("350x300")

style = ttk.Style(root)

GeneralFrame(root)

root.mainloop()