МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ по учебной

практике

по дисциплине «Генетические алгоритмы и PSA»

Тема: Применение генетического алгоритма к задаче оптимизации с графическим интерфейсом.

Студент гр. 3384 Поздеев В. Д.

Студент гр. 3381 Марков М. М.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучить и реализовать генетические алгоритмы для решения задач оптимизации с интерактивным графическим интерфейсом, позволяющим настраивать параметры, визуализировать процесс эволюции решений и анализировать их качество по поколениям.

Задание.

Вариант 17. Задача минимизации задержек

Дано N задач, каждая из которых имеет свое время выполнения и дедлайн к которому она должна быть выполнена. Задача, это составить расписание с минимальными задержками. Задержка – количество времени, на которое выполнение задач превысило дедлайны.

Выполнение работы.

Этап 1. Формирование бригады, выбор ЯП и распределение ролей.

В рамках первого этапа проекта была сформирована бригада из двух человек. Выбран язык программирования Python. Распределение ролей было примерно следующим (в github репозитории по issues видно):

- Поздеев Вадим: разработка графического интерфейса (GUI), настройка визуализации хода выполнения алгоритма, построение графиков изменения приспособленности.
- Марков Марк: разработка базовых классов, парсеров данных, создание абстрактных классов и конкретных стратегий отбора, скрещивания и мутации.
- Вместе: работа над классом генетического алгоритма, интеграция компонентов, отладка и корректировка взаимодействия всех модулей, стили графического приложения.

Итерация 2. Демонстрация прототипа GUI и плана решения задачи (описание формата данных, используемых функций качества, и т.д.)

На данном этапе было реализовано следующее.

Backend

Файл *defaultClasses.py*

В *defaultClasses.py* сосредоточены базовые сущности, на которых строится вся логика работы генетического алгоритма.

Класс *Task.* Использует *@dataclass* для автоматического создания конструктора и методов сравнения/представления. При создании объекта через *__post_init__* ему назначается уникальный целочисленный идентификатор id (счетчик *_next_id*), после чего вызывается метод *_validate()*, проверяющий, что поля *time* и *deadline* неотрицательны. В случае нарушения этого условия генерируется *ValueError*, что предотвращает использование некорректных данных на ранней стадии.

Класс *ParamGeneticAlgorithm* также оформлен как *@dataclass* и хранит все важнейшие параметры для управления эволюцией:

- *crossover* — вероятность применения операции скрещивания;
- *mutation* — вероятность мутации каждой особи;
- *num_individuals* — размер популяции;
- *num_generations* — число поколений, через которые должен пройти алгоритм;
- *num_to_select* — сколько особей отбирать перед операцией скрещивания.

В методе *__post_init__* вызывается *_validate()*, который выбросит исключение при выходе любых параметров за логически допустимые границы (например, вероятность вне [0,1] или слишком маленькая популяция).

Класс *ScheduleInfo* представляет одну особь популяции — конкретный порядок выполнения задач (наша особь (хромосома) состоит из перестановок генов). В конструкторе он копирует переданный список индексов *order*, сохраняет ссылку на общий список *tasks* и сразу же вычисляет целевую функцию — суммарную задержку (*tardiness*) при помощи метода *_calculate_tardiness()*.

Этот метод итерируется по порядку индексов, накапливая общее время и добавляя к сумме только положительные отклонения от дедлайнов. Каждая особь также получает свой уникальный идентификатор id через счетчик *_next_id*, что облегчает отладку и визуализацию. Метод *copy()* создаёт глубокую копию особи, позволяя безопасно дублировать её при операциях отбора и скрещивания.

Перечисление *State* задаёт четыре логических этапа работы алгоритма:

- *INIT* — инициализация первоначальной популяции;
- *SELECTION* — этап отбора лучших особей для скрещивания;
- *CROSSBREEDING* — применение операции скрещивания;
- *MUTATION* — применение мутаций ко всем потомкам.

Это перечисление используется для управления потоком выполнения алгоритма и для маркировки каждого поколения.

Класс *GenerationState* обеспечивает хранение полного «снимка» одного поколения в любой из четырёх стадий. В нём содержатся:

- *population* — список всех объектов *ScheduleInfo*;
- *state* — значение из State, указывающее, на каком этапе находится поколение;
- *best* — ссылка на лучшую особь по минимальной задержке, вычисляемая в *__post_init__*;
- *average_tardiness* — среднее значение задержки по всем особям.

Кроме того, каждая инстанция *GenerationState* получает свой уникальный идентификатор id через внутренний счётчик *_next_id*, что позволяет сохранять историю поколений, перемещаться «шаг за шагом» вперёд и назад и отображать динамику изменения показателей на графиках.

В сумме эти классы образуют фундаментальную модель: *Task* описывает отдельно взятую задачу, *ParamGeneticAlgorithm* — параметры работы ГА, *ScheduleInfo* — одну упорядоченную комбинацию задач с вычисленной метрикой качества, а *GenerationState* — состояние всей популяции в конкретный момент

эволюции. Перечисление *State* контролирует порядок фаз алгоритма, обеспечивая возможность пошаговой демонстрации и анализа.

Файл *selection.py*

В *selection.py* содержится модуль, отвечающий за реализацию разнообразных методов отбора (selection) особей из текущей популяции перед операцией скрещивания.

Интерфейс *SelectionStrategy*. Абстрактный базовый класс для реализации различных стратегий отбора. Метод *select* принимает на вход объект *GenerationState* (с одновременным доступом к списку особей и метаданным поколения) и число *num_to_select* — сколько особей нужно отобрать. В результате возвращается новый экземпляр *GenerationState*, в котором содержится уже отобранная подмножество особей и метка *state=State.SELECTION*.

Класс *TournamentSelection*. Реализует метод турнира: для каждой из k позиций в списке отобранных особей случайным образом выбирается пара разных претендентов из всей популяции, сравниваются их значения задержки (*tardiness*), и победитель (особь с меньшей задержкой) копируется в итоговый список. По окончании отбора создаётся и возвращается новое состояние поколения, отмеченное как этап *SELECTION*.

Класс *RankSelection* опирается на ранжирование особей по возрастанию задержки. Вначале сортирует популяцию, затем формирует веса, обратно пропорциональные рангу (чем лучше особь, тем больше её вес). С помощью *random.choices* выбирает k особей с учётом этих весов, копирует каждую из них и формирует итоговый *GenerationState* с *state=State.SELECTION*. Такой подход устраняет зависимость от абсолютных величин задержек, смещая фокус на относительный рейтинг.

Класс *StochasticUniversalSampling*. Имитирует стохастический универсальный отсев, базируясь на обратных значениях задержек как показателях приспособленности. Сначала вычисляются вероятности выбора

каждой особи из популяции, нормированные к единице. Затем генерируется равномерно распределённая последовательность «указателей» в диапазоне [0,1), и для каждого указателя выбирается соответствующая особь по кумулятивному распределению. Это гарантирует более равномерное представление всех уровней приспособленности в выбранном наборе и снижает дисперсию выборки по сравнению с классическим отбором, основанным на обычной рулетке. Итоговый список копий отобранных особей упаковывается в новый *GenerationState* с тем же флагом *SELECTION*.

Во всех трёх реализациях ключевым моментом является создание глубоких копий (*.copy()*) исходных объектов *ScheduleInfo*, что предотвращает нежелательные изменения исходной популяции при последующих операциях скрещивания и мутации. Такой дизайн позволяет легко расширять набор методов отбора, просто добавляя новые классы, наследующие *SelectionStrategy* и реализующие метод *select*.

Файл *crossbreeding.py*

В *crossbreeding.py* содержится реализация механизма скрещивания (crossbreeding) для уже отобранных особей через базовый абстрактный класс *CrossbreedingStrategy*. Метод crossbreed принимает текущее состояние поколения (*GenerationState*), число потомков *num_to_produce* и вероятность применения операции rate. В результате он возвращает новый объект *GenerationState* со списком потомков и флагом *State.CROSSBREEDING*.

Единственная на данный момент конкретная стратегия — *OrderCrossbreeding*, адаптированный для задач на перестановках. Алгоритм работает так:

1. Перемешивание родителей. Из входного *state.population* создаётся копия списка, которая затем перемешивается, чтобы пары родителей формировались случайным образом.

2. Параллельная генерация потомков. Для каждой смежной пары *(p1, p2)* до достижения требуемого числа *num_to_produce* с вероятностью *rate* выполняется метод:
    - Случайным образом выбираются два индекса отрезка *[i1...i2]*.
    - У каждого потомка копируется подотрезок из одного родителя, а оставшиеся позиции заполняются генами второго родителя в их относительном порядке, пропуская уже скопированные.
    - Так создаются два потомка (с ролями родителей поменянными местами).
    - Каждый ребёнок упаковывается в *ScheduleInfo* для корректного вычисления задержки.
3. Копирование без изменений. Если метод не применяется (с вероятностью *1-rate*), то из родителей копируются их хромосомы напрямую, чтобы обеспечить сохранение текущих хороших решений.
4. Дополнение до полного размера. Если после первого прохода число потомков оказалось меньше *num_to_produce*, оставшиеся «слоты» заполняются случайными копиями из исходного набора родителей, чтобы гарантировать постоянный размер популяции.
5. Формирование нового поколения. Все созданные потомки (обрезанные до *num_to_produce* на случай избытка) упаковываются в новый экземпляр *GenerationState* с флагом *State.CROSSBREEDING*, который затем передаётся в последующий этап мутации или итерации алгоритма.

Такая реализация позволяет корректно работать с задачей хромосомы-перестановки, гарантируя, что потомки остаются допустимыми перестановками без повторяющихся или пропущенных генов. Благодаря единому интерфейсу *CrossbreedingStrategy* легко добавлять или менять другие методы скрещивания

(например, одноточечный или равномерный), при необходимости адаптируя их под специфику перестановок.

Файл *mutation.py*

В *mutation.py* содержится модуль, отвечающий за операцию мутации в хромосомах особей. Он организован по тому же принципу, что и отбор и скрещивание, через единый абстрактный интерфейс *MutationStrategy*. Принимает *state* — текущее состояние поколения (объект *GenerationState*) и *mutation_rate* — вероятность применения мутации к каждой особи. Метод возвращает новый *GenerationState* с флагом *State.MUTATION*.

Реализованы три стратегии мутации:

1. *NoMutation* — пассивный вариант, при котором популяция просто копируется без изменений. Используется для отладки или когда мутацию необходимо отключить. Возвращает новое состояние с тем же набором особей.
2. *SwapMutation* — классический оператор «обмен двух генов» для перестановок. Для каждой особи с вероятностью mutation_rate случайно выбираются две позиции в хромосоме и меняются местами. Гарантирует корректность перестановки (ни один элемент не пропадает и не дублируется).
3. *InversionMutation* — оператор инверсии: с заданной вероятностью выбирается случайный отрезок хромосомы и его содержимое переворачивается (*reversed*).

Каждая стратегия, после формирования нового списка индивидов (либо мутированных, либо нет), упаковывает результат в *GenerationState* с обновлённым списком *population* и помечает этап как *State.MUTATION*.

Файл *geneticAlgorithm.py*

В *geneticAlgorithm.py* сосредоточена основная логика работы самого генетического алгоритма. Класс *geneticAlgorithm* инкапсулирует весь цикл

эволюции популяции: инициализацию, пошаговое выполнение операций отбора, скрещивания и мутации, а также хранение истории промежуточных состояний.

При создании экземпляра алгоритма в конструкторе (*__init__*) устанавливаются стандартные параметры и стратегии:

- *self.params* хранит объект *ParamGeneticAlgorithm*, в котором задаются численные параметры алгоритма (вероятности операций, размеры популяции и число поколений).
- *self.selection*, *self.crossbreeding* и *self.mutation* получают начальные реализации стратегий (*TournamentSelection*, *OrderCrossbreeding* и *SwapMutation*).
- *self.history* служит для накопления всех промежуточных объектов *GenerationState*, что позволяет реализовать «шаг назад» и строить графики динамики.

Метод *set_tasks(self, tasks)* передаёт алгоритму заранее прочитанный список задач (объектов *Task*). После этого вызовом *change_params(...)* можно скорректировать вероятности мутации/скрещивания, размеры популяции и число поколений, при этом внутренний метод валидации проверит корректность введённых значений.

При *create_individuals()* формируется начальная популяция: для каждой особи создаётся случайная перестановка индексов задач, на основании которой строится объект *ScheduleInfo* с рассчитанной суммарной задержкой. Полученный список передаётся в новый *GenerationState* со статусом *State.INIT*.

Основные этапы алгоритма перечислены в методе *do_next()*, который в зависимости от текущего состояния (*INIT* → *SELECTION* → *CROSSBREEDING* → *MUTATION*) последовательно вызывает:

1. *do_selection()* – сохраняет текущее состояние в историю и заменяет его результатом стратегии отбора (*SelectionStrategy.select*), формируя новое поколение из *num_to_select* лучших или случайно выбранных особей.

2. *do_crossbreeding()* – добавляет в историю текущее состояние и создает потомков методом скрещивания (*CrossbreedingStrategy.crossbreed*) с заданной вероятностью.
3. *do_mutation()* – сохраняет состояние и мутирует всех особей (*MutationStrategy.mutate*), увеличивая счётчик пройденных поколений *self.iteration*.

Метод *finish()* доводит текущий процесс до конца (дожидается состояния *State.MUTATION* у начального поколения), а затем повторяет полные циклы до тех пор, пока не будет пройдено заданное число поколений. Это гарантирует, что по завершении работы в *self.history* окажутся все промежуточные состояния, а в *self.generationState* — итоговое.

Наконец, get_best() сканирует сохранённую историю и возвращает ту особь, у которой наименьшая задержка встречается за всё время выполнения алгоритма. Таким образом реализуется поиск глобально лучшего найденного решения.

Все взаимодействия между классами устроены по принципу «стратегии»: работа алгоритма остается неизменной, а конкретные правила отбора, скрещивания и мутации можно подменить, передав в *geneticAlgorithm* свои реализации интерфейсов *SelectionStrategy*, *CrossbreedingStrategy* или *MutationStrategy*.

Файл *parser.py*

В *parser.py* сосредоточена единая логика получения и валидации исходных задач из разных источников. Он умеет работать как с текстом, введённым в GUI (многострочная строка), так и с содержимым произвольного файла (любое расширение).

При вызове парсера исходная строка проверяется: если это путь к существующему файлу — он читается построчно, иначе обрабатываются строки переданной многострочной переменной. Каждая непустая строка сначала

очищается от пробелов, затем разбивается по любым комбинациям запятых и пробелов. Если после этого не получается ровно два токена, парсер сразу сообщает об ошибке с указанием номера строки и её содержимого. Далее оба токена пытаются превратить в целые числа; в случае неудачи генерируется исключение с пояснением, какой из элементов не соответствует формату. Наконец, по этим двум числам создаётся экземпляр задачи, и внутри своего конструктора он уже проверяет, что время и дедлайн неотрицательны.

Если после обработки всех строк не оказалось ни одной валидной задачи, парсер выбрасывает ошибку о пустом наборе данных. Таким образом, клиентские классы гарантированно получают корректный список объектов *Task* или сразу же получают подробное сообщение о том, где во входных данных ошибка.

Класс *RandomParser*, реализованный здесь же, позволяет сгенерировать заданное число задач со случайными параметрами. Он проверяет корректность переданных диапазонов времени и дедлайнов, после чего последовательно создаёт каждый объект *Task*, при необходимости смещая дедлайн вперёд, чтобы он был не меньше времени выполнения. В результате получается список валидированных случайных задач.

GUI

Файл *App.py*

Создан класс *App*, в котором хранится парсер, ссылка на состояние, ссылка на алгоритм. Для GUI используется паттерн состояния. Созданы различные методы, которые отвечают за настройку приложения и генетического алгоритма(параметры алгоритма, тип отбора, скрещивания, мутации). Также методы отвечающие за переключения состояния(clear_state, change_state) и запуск приложения.

В файле *graphs.py* хранятся виджеты отвечающие за графики(график среднего, график расписания, график задач).

Файл *startState.py*

Первое состояние это *StartState*. Внутри себя содержит главное окно *ttk.PannedWindow*, отвечающее за разделение на две части. Первая часть – это ещё одно окно *ttk.PanedWindow*, слева от которой находятся *importFrame*, отвечающее за ввод данных, а справа настройки генетического алгоритма, и генератора чисел. Вторая часть это *graph_container* и представляет собой график задач, которые вводятся. Также создается кнопка, отвечающая за переход в следующее состояние *WorkState*.

Файл *importFrame.py*

ImportFrame содержит внутри себя *ttk.Notebook*, позволяющий перемещаться по вкладкам ввода. Реализованы 3 вкладки.

Вкладка "Из файла" *file_input*, использующий *ttk.filedialog*, который позволяет выбирать нужный файл в диалоговом окне. Создается *ttk.Entry*, в который ничего нельзя писать и кнопка *ttk.Button*, которая меняет парсер, которая создает *ttk.filedialog*.

Вкладка "Ручной ввод" *manual_input*, использующий *scrolledText*, который создает удобное окно для ввода вручную, также добавляющая прокрутку, если пользователь напишет слишком много текста. В *manual_input* создается кнопка *ttk.Button*, которая применяет ввод и меняет парсер.

Вкладка "Генерация данных" *random_input*, использует созданные *RangeSettingsLine* и *SettingsLine* для того, чтобы пользователь мог настроить диапазоны времени выполнения задачи, дедлайнов и количество задач, которые будут создаваться. Также создана кнопка *ttk.Button*, которая меняет парсер и применяет ввод.

Файл *settingsFrame.py*

SettingLine представляет собой *ttk.Lable*, *ttk.Entry* и *ttk.Slider*. Принимает название настройки, поле для удобного ввода и слайдер, позволяющий менять *ttk.Entry*. Также есть методы для получения значения из ввода пользователя.

RangeSettingLine представляет собой два *SettingLine* и *ttk.Label*. Создаются два поля *settingLine* для минимума и максимума. Используются методы

validate_min, *validate_max*, которые не позволяют пользователю делать *max* меньше *min* и наоборот. Данный класс создает удобное поле в интерфейсе выбора диапазона значений. Также создан метод *get_range* для получения результата

DropdownSetting представляет собой *ttk.Combobox* и *ttk.Label*. Создаётся выпадающий список с заданными опциями и меткой слева. Класс позволяет выбирать значение из предопределённого списка. Метод *get_value* возвращает текущее выбранное значение.

SettingsFrame содержит *ttk.Notebook* для переключения между вкладками настроек. Реализованы 2 вкладки. Вкладка "Настройки алгоритма" использует *SettingLine* для параметров: размер популяции, количество поколений, размер отбора, вероятность мутации, вероятность скрещивания. Также использует *DropdownSetting* для выбора типа отбора (*TournamentSelection/RankSelection/StochasticUniversalSampling*), типа скрещивания (OrderCrossbreeding) и типа мутации (*NoMutation*/*SwapMutation*/*InversionMutation*). Вкладка "Настройки генерации" использует *SettingLine* для параметра *Seed*. Внизу фрейма созданы кнопки "Сохранить" (применяет настройки через *app.change_** методы) и "Сбросить" (восстанавливает значения по умолчанию).

Файл *workState.py*

WorkState содержит главное окно *ttk.PannedWindow* с горизонтальной ориентацией, разделяющее пространство на две равные части. В левой части размещается *ScheduleFrame* для отображения текущего расписания, в правой части – *GraphView* для визуализации графиков. Под основным окном создается панель управления с кнопками "Предыдущий шаг" (возврат на шаг назад в алгоритме), "Следующий шаг" (переход на шаг вперед) и "В конец" (завершение работы алгоритма). При нажатии любой кнопки происходит обновление обоих фреймов (*ScheduleFrame* и *GraphView*) для отражения изменений в состоянии алгоритма.

Файл *scheduleFrame.py*

ScheduleView содержит два фрейма: *gen_info* для информации о текущем поколении и *sched_info* для данных о расписании. В *gen_info* создаются три метки *ttk.Label* для отображения: типа состояния алгоритма (получаемого через *get_type*), средней задержки поколения и количества особей. В *sched_info* размещается график *ScheduleInfoGUI* и две метки *ttk.Label* для ID расписания и величины задержки. Класс реализует методы: *update_gen()* (обновляет данные поколения), *update_sched()* (обновляет график и данные расписания) и *update()* (синхронно обновляет оба блока).

ScrollableFrame представляет собой кастомный прокручиваемый контейнер, содержащий *tk.Canvas*, *ttk.Scrollbar* и внутренний фрейм *scrollable_frame*. При инициализации настраивается связь между холстом и скроллбаром: скроллбар управляет областью просмотра холста (*yscrollcommand*), а холст обновляет скроллбар при изменении содержимого. Все дочерние виджеты добавляются во внутренний фрейм *scrollable_frame*. Реализована поддержка прокрутки колесом мыши через привязку *<MouseWheel>*. Класс предоставляет метод *clear()* для полной очистки содержимого с автоматическим обновлением области прокрутки.

ScheduleSelection представляет собой прокручиваемый список расписаний на базе кастомного контейнера *ScrollableFrame*. Для каждого расписания в популяции создается кликабельный фрейм (*ttk.Frame*) с двумя метками*: ttk.Label* с ID расписания (жирный шрифт) и *ttk.Label* с величиной задержки (серый цвет). Все элементы фрейма привязаны к обработчику клика (*<Button-1>*), который вызывает метод *change_sched_view* для обновления основного вида расписания. Класс реализует метод *update()* для полной очистки и перестройки списка при изменении состояния алгоритма.

ScheduleFrame содержит *ttk.Notebook* для переключения между двумя вкладками: "Просмотр расписания" (размещает *ScheduleView*) и "Выбор расписания для просмотра" (размещает *ScheduleSelection*). Класс реализует метод *update()*, который синхронно обновляет обе вкладки при изменении состояния алгоритма. *ScheduleView* отображает детали текущего расписания, а

ScheduleSelection предоставляет прокручиваемый список всех расписаний в поколении для интерактивного выбора.

Файл *graphView.py*

GraphView содержит график *AverageTardinessGUI* для визуализации истории задержек и метку *ttk.Label*, отображающую текущую итерацию алгоритма в формате "Итерация: X/Y" (где X - текущий шаг, Y - общее число поколений). При инициализации график строится на основе истории генетического алгоритма (*app.genAlgorithm.history*). Метод update() перерисовывает график и обновляет текст метки актуальными значениями итерации, извлекая данные из параметров алгоритма (*app.genAlgorithm.params.num_generations*).

Выводы.

В ходе работы была разработана гибкая модульная архитектура для решения задач оптимизации при помощи генетического алгоритма. Реализованы все ключевые компоненты: парсеры входных данных (из файла, из текстовой строки и генерация случайных задач), базовые классы описания задачи и особи, три разновидности стратегий отбора, перекрёстка и мутации, а также «движок» самого алгоритма с возможностью пошагового выполнения и сохранения полной истории поколений.

Недоработки и планы по улучшению. Несмотря на то, что после завершения выполнения ГА приложение остаётся «живым», пока что не реализована логика повторного запуска алгоритма на тех же данных с изменёнными параметрами. GUI требует доработки: визуализация расписания сейчас неконсистентна — цвета повторяются и идентификаторы накладываются друг на друга, что затрудняет анализ. Планы включают обновление схемы раскраски, добавление возможности повторить решение с другими параметрами ГА.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ