МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по учебной практике

Тема: Генетические алгоритмы

Кузнецова С.Е.
Максимова Е.Д.
Студентки гр. 3341 _____ Чинаева М.Р.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Разработать программу генетического алгоритма для решения задачи поиска клики наибольшего размера в графе. Реализовать генетический алгоритм для эффективного решения этой задачи. Предусмотреть возможность сохранения и загрузки данных. Создать графический интерфейс для визуализации работы алгоритма и управления его параметрами пользователем.

Задание №18.

Задача о клике.

В заданном графе G = (V,E) необходимо найти клику максимального размера.

Распределение ролей в бригаде.

Максимова Екатерина: Реализация GUI

Чинаева Маргарита: Реализация основного алгоритма и подготовка для связи с GUI

Кузнецова Светлана: Загрузка и выгрузка данных и вспомогательные функции для основного алгоритма (проверка смежности, прохождение по соседям, проверка подграфа, является ли он кликой и т.п.) + генерация случайных данных

ИТЕРАЦИЯ 1

План реализации генетического алгоритма:

В качестве хромосомы используем массив размером n (количество вершин), в котором возможны только 2 значения – 0 и 1. Где 0 – вершина не включена в клику, 1 – вершина включена в клику.

Сортируем вершины в порядке невозрастания степени, переназначаем номера вершин согласно их порядку в итоговом массиве и храним массив изначальных порядковых номеров. Например, степень 3 вершины больше степени 1, а степень 1 больше 2. Тогда храним «массив перехода» 312. Это нужно для того, чтобы при разрыве хромосомы вершины, наиболее вероятно находящиеся в одной клике, не разрывались. Наиболее важно это для вершин с большой степенью, так как они с большей вероятностью будут находиться в наибольшей клике.

Для генерации изначальной популяции используем «правило рулетки», в котором, чем больше степень у вершины, тем больше шанс случайно выбрать ее при составлении клики.

Алгоритм генерации одной хромосомы для начальной популяции:

1) Случайно выбираем вершину

2) Из ее соседей случайно выбираем следующую, которая при этом связанна со всеми находящимися в клике.

3) Повторяем, пока есть допустимые вершины.

Функция приспособленности – сумма единиц в хромосоме – количество вершин в клике.

Оператор выбора родителей – метод рулетки, в котором размеры максимального и минимального секторов отличаются не более, чем на заданное пользователем число процентов. Это нужно для разнообразия популяции, чтобы одна особь не породила слишком много потомков.

Скрещивание родителей происходит многоточечно с постепенным уменьшением количества разрезов, так как в начале алгоритма нужно наибольшее разнообразие особей, а при приближении к решению это уже менее важно.

Отбор особей в новую популяцию будет проводиться отбором вытеснением, в котором из всех особей с одинаковой приспособленностью будет отдаваться предпочтение с разными генотипами.

Для каждой популяции производим мутации. С заданной пользователем вероятностью выбирается будет ли мутирована данная хромосома. Затем с заданной пользователем вероятностью мутируется каждый ген в хромосоме.

После мутации пока хромосома не станет кликой, постепенно ее уменьшаем. Из множества вершин с минимальной степенью случайно удаляем одну, снова проверяем на клику, и так далее. Это нужно, так как в популяции должны быть только особи, которые являются кликами.

Генетический алгоритм останавливает работу, если было выполнено некоторое, заданное пользователем количество итераций «застоя», то есть лучший результат не менялся в течение нескольких итераций. А так как по свойству графов размер максимальной клики не превосходит наибольшей степени вершины в графе, то алгоритм также останавливает свою работу, если лучшая хромосома задает клику этого размера.

Загрузка/выгрузка, хранение данных

Граф хранится в виде списка смежности – в виде списка множеств длины n, где n – число вершин графа, и элемент с индексом v – это множество всех вершин графа, смежных с v. Таким образом, проверка смежности двух вершин происходит за O(1), проход по всем соседям вершины v – за O(deg(v)), где deg(v) – степень вершины v. Наличие множеств также гарантирует отсутствие дубликатов вершин.

Классы, реализуемые для хранения данных:

1. Класс Parameters – будет отвечать за хранение параметров генетического алгоритма (размер популяции, максимальное количество итераций, количество итераций в застое и т.д.), будет содержать функцию для загрузки параметров из json-файла.

2. Класс Graph – будет отвечать за хранение графа. Будет иметь функции загрузки графа из json-файла, генерации случайного графа,

перевода графа из матрицы смежности в список смежности (список множеств).

3. Класс Chromosome – будет отвечать за хранение хромосомы (потенциальной клики) и ее приспособленности. Будет иметь функцию для вычисления приспособленности.

4. Класс Population – будет хранить текущую популяцию (список хромосом текущего поколения), лучшее решение и среднюю приспособленность. Будет содержать функцию для обновления значений поколения: лучшего решения и средней приспособленности на каждом шаге.

5. Класс History – будет хранить промежуточные данные о ходе выполнения алгоритма – список номеров поколений, для которых записаны промежуточные данные, приспособленность лучшего решения на каждом поколении, среднюю приспособленность популяции на каждом поколении, лучшие решения на каждом поколении. Будет содержать функции для обновления данных, сохранения данных в json-файл.

6. Класс Manager – будет отвечать за загрузку/выгрузку всех данных из файла и хранить данные для работы алгоритма (экземпляры классов параметров, графа и истории). Будет содержать функции для загрузки всех данных из json-файла и сохранения их в json-файл.

Хранение входных и выходных данных:

1. Входные данные – параметры и граф будут храниться в json-файле с соответствующими ключами.

2. Выходные данные – историю эволюции – список лучших и средних приспособленностей для каждого поколения, список лучших решений – также будут храниться в json-файле.

Прототип графического интерфейса (GUI).

В графическом интерфейсе предусмотрено 3 основных окна:

Главное окно. Слева расположен виджет с визуаизацией графа, под которым расположен блок с проигрывателем, отображением размера текущей лучшей клики, кнопками сохранения и загрузки. Также на виджете распологается маленькая кнопка для вывода всей популяции на текущем шаге алгоритма. Справа — столбец текстовых полей с соответствующими параметрами, вводимыми пользователем и кнопка с вызовом окна для ввода матрицы.

Параметры для ввода пользователем:

- Population size - размер популяции.
- Max generations amount - максимальное количество поколений
- Stagnation limit — максимальное количество итераций в застое
- Mutation rate - вероятноть мутации одного гена.
- Gene mutation - вероятность скрещивания.
- Fitness scale - масштабирование (в процентах) приспособленности хромосом при селекции родителей.
- Max cut points - максимальное количество точек разреза при кроссинговере.
- Decr m.prob step (Decrease mutation probability step) - шаг (как количество поколений) для уменьшения вероятности мутации.
- Reduce cuts step шаг (как количество поколений) для уменьшения количества точек при кроссинговере.

Окно с графиком изменения лучший и средней приспособленности в зависимости от поколения. Появляется при нажатии кнопки запуска алгоритма

Окно ввода матрицы смежности. Пользователь вводит размер матрицы в соответсвующее поле. После нажатия кнопки заполнения матрицы создается таблица для ввода, где доступны ячейки выше диагонали (остальные значения заполняются симметрично). Также предусмотрена кнопка рандомной генерации. Визуализация графа и возможность запуска алгоритма на нем появляется после нажатия кнопки «Create graph».

При возникновении ошибок, связанных с вводом пользователя появляется всплывающее окно с описание ошибки.

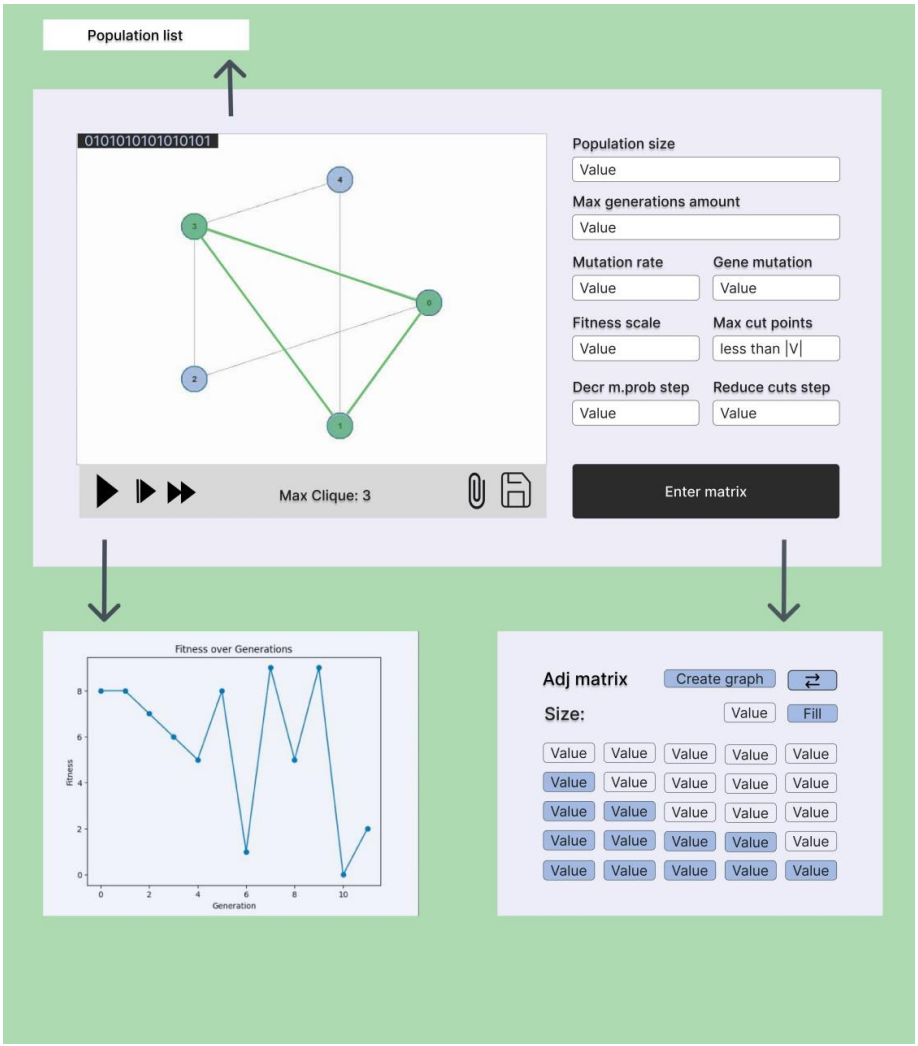

Рисунок 1 - Макет GUI

ИТЕРАЦИЯ 2

Описание текущей реализации генетического алгоритма:

Класс *GeneticAlgorithm* - реализует генетический алгоритм для поиска максимальной клики в графе. Использует эволюционные методы для нахождения оптимального решения задачи о максимальной клике.

Поля класса:

Graph graph - Граф, в котором ищется максимальная клика

Parameters params - Параметры работы генетического алгоритма

float current_mutation_prob_chrom - Текущая вероятность мутации хромосомы

float current_mutation_prob_gene - Текущая вероятность мутации отдельного гена

int current_crossover_points - Текущее количество точек кроссовера

int generation - Номер текущего поколения

int stagnation_count - Счетчик поколений без улучшения лучшего решения

int best_fitness - Найденный максимальный размер клики

list[int] best_chromosome - Лучшая найденная хромосома

int n - Количество вершин в графе

int max_degree_original - Максимальная степень вершины в исходном графе

Основные методы:

__init__(self, graph: Graph, params: Parameters) – Инициализирует алгоритм с заданным графом и параметрами, выполняет преобразование графа и генерацию начальной популяции.

generate_chromosome(self) -> List[int] – Генерирует случайную хромосому, представляющую клику, использует жадный алгоритм с случайным выбором вершин среди вершин одной степени.

generate_initial_population(self) -> List[Individual] – Создает начальную популяцию заданного размера.

select_parents(self) -> List[Individual] – Выбирает родителей для скрещивания методом рулетки с масштабированием приспособленности.

crossover(parent1, parent2) -> Tuple[List[int], List[int]] - Выполняет кроссовер с несколькими точками разрыва, попеременно копируя сегменты хромосом родителей.

mutate_and_repair(chromosome) -> List[int] – Применяет мутации и восстанавливает хромосому до валидной клики.

select_new_population(current_pop, offspring) -> List[Individual] – Формирует новую популяцию с сохранением разнообразия, выбирая особи, максимально отличающиеся от уже выбранных.

_update_best_solution() – Обновляет лучшее решение при обнаружении улучшения.

_reduce_parameters() – Уменьшает параметры алгоритма на заданный процент от начальных значений.

should_stop() -> bool – Проверяет условия остановки алгоритма.

next_generation() – Выполняет одну полную итерацию генетического алгоритма:

1. Выбор родителей
2. Генерация потомков
3. Формирование новой популяции
4. Обновление лучшего решения и списка поколений
5. Уменьшение параметров (при необходимости)

get_best_solution() -> Tuple[int, List[int]] – Возвращает лучшее решение в исходной нумерации вершин.

_hamming_distance() – Вычисляет нормализованное расстояние Хэмминга

Пример работы алгоритма см в приложение А.

Обработка и хранение данных, вспомогательные функции

Были реализованы классы для хранения данных, использующихся в работе алгоритма, и вспомогательные функции.

1. Класс *Graph* – отвечает за работу с графом, его загрузкой, имеет вспомогательные функции для работы с графом

Поля класса:

list[set[int]] adj_list – начальный граф в виде списка смежности (списка множеств)

int n – его размер (количество вершин)

list[set[int]] transformed_adj – преобразованный граф в виде списка смежности – граф с переназначенными вершинами, отсортированными по убыванию степеней в исходном графе

list old_to_new – список для преобразования старых индексов (вершин графа) в новые

list new_to_old – список для преобразования новых индексов (вершин графа) в старые

Основные методы:

from_adj_matrix(matrix: list[list[int]]) -> 'Graph' – строит граф в виде списка смежности из матрицы смежности

to_adj_matrix(self) -> list[list[int]] – преобразует текущий граф в матрицу смежности 0/1

load_from_matrix_file(file_path: str) -> 'Graph' – загружает граф через json-файл, в котором граф представлен как матрица смежности (список списков 0/1)

save_to_matrix_file(self, file_path: str) -> None – сохраняет граф в JSON-файл как матрицу смежности 0/1

random_graph(n: int, p: float) -> 'Graph' – генерирует случайный список смежности размера n с вероятностью p для каждого ребра i – j и строит по нему граф

transform_by_degree(self) – преобразует граф, переупорядочивая вершины по убыванию степени. Создает списки для отображения старых индексов в новые, новых в старые

transform_to_original(self, sorted_chromosome: list[int]) -> list[int] – преобразует хромосому из преобразованной нумерации в исходную нумерацию вершин графа

transform_to_sorted(self, original_chromosome: list[int]) -> list[int] – преобразует хромосому из исходной нумерации в преобразованную

get_subgraph(self, chromosome: list[int]) -> list[set[int]] – строит подграф на основе хромосомы (для исходного графа). В дальнейшем будет необходима для построения подграфа в GUI.

all_neighbors(self, v: int) -> list[int] – возвращает список соседей вершины v в преобразованном графе

has_edge(self, u: int, v: int) -> bool – проверяет наличие ребра между u и v в преобразованном графе

degree_in_subgraph(self, included: list[int]) – вычисляет степени вершин в подграфе (в преобразованном графе)

is_clique(self, chromosome: list[int]) -> bool – проверяет, задают ли включенные в хромосому вершины клику в преобразованном графе

repair_chromosome(self, chromosome: list[int]) -> list[int] – пока включенные вершины не образуют клику, удаляет случайную вершину минимальной степени в подграфе. Возвращает новую хромосому

load_from_file(file_path: str) -> 'Graph' – загружает граф через json-файл, в котором граф представлен в виде "вершина: список соседей"

2. Класс *Parameters* – отвечает за хранение параметров генетического алгоритма, их загрузкой из файла.

Поля класса:

int population_size – Размер популяции

int max_generations – Максимальное количество итераций

int stagnation_limit – Количество итераций в застое

float max_mutation_prob_gene – Максимальная вероятность мутации 1 гена

float max_mutation_prob_chrom – Максимальная вероятность мутации хромосомы

int fitness_scaling_percent – Масштабирование приспособленности хромосом при селекции родителей (в процентах)

int max_crossover_points – Максимальное количество точек разреза

int decrease_percent – Процент уменьшения вероятности мутации и точек разреза

int decrease_step – Шаг (количество поколений) уменьшения точек разреза, вероятности мутации гена и хромосомы

Методы класса:

from_graph(self, n: int) -> 'Parameters' – вычисляет и возвращает параметры по умолчанию для данного графа

validate_parameters(data: dict, n: int) -> None – проверяет словарь параметров на наличие всех требуемых ключей, правильные типы и корректность значений

3. Класс *Indididual* – отвечает за хранение конкретной особи, заданной хромосомой.

Поля класса:

list[int] chromosome – Бинарный вектор, задающий хромосому

float fitness – Размер клики

Методы класса:

evaluate(self) – вычисляет приспособленность особи как размер клики, считая, что заданная хромосома всегда задает клику

4. Класс *Population* – отвечает за работу с популяцией, хранит все необходимые параметры и имеет вспомогательные методы для работы с популяцией

Поля класса:

list[Individual] individuals – список особей

Individual best – лучшая особь в популяции

float avg_fitness – средняя приспособленность

Методы класса:

get_len(self) – Подсчет количества особоей в популяции

update_stats(self) – Пересчитывает статистики популяции

select_best(self, n: int) -> list[Individual] – Возвращает n лучших особей

get_fitnesses(self) -> list[float] – Возвращает список приспособленностей всех особей

add_individuals(self, new_individuals: list[Individual]) – Добавляет новых особей в популяцию

5. Класс *History* – отвечает за хранение промежуточных данных работы алгоритма для дальнейшей работы с ними, имеет вспомогательные методы

Поля класса:

list[float] best_fitness – Лучшие приспособленности на каждом поколении

list[float] avg_fitness – Средние приспособленности на каждом поколении

Методы класса:

record(self, generation: int, population: Population) – Пересчитывает параметры популяции и сохраняет их в историю

save_to_json(self, path: str) – Сохраняет историю работы алгоритма в результирующий json-файл

Финальная разработка визуальной части GUI

Используемые библиотеки: tkinter (gui в целом), networkX (визуализация графа), matplotlib (график приспособленности), numpy.

В рамках доработки графического интерфейса был частично изменен дизайн. Реализована двухоконная система на базе классов *MainApp* (главное окно) и *MatrixWindow* (окно матрицы смежности), обеспечивающая удобное взаимодействие с программой. Основное окно приложения, реализованное в

классе *MainApp*, теперь содержит все ключевые элементы управления: панель параметров алгоритма, область визуализации графа через networkx, график изменения показателей приспособленности (*FitnessPlotWidget*), список найденных решений (*SolutionListWidget*) и элементы управления воспроизведением процесса поиска решений.

Класс *MatrixWindow* отвечает за работу с матрицей смежности, предоставляя интерактивный редактор с автоматической проверкой вводимых значений через механизмы класса *Validator*. Пользователь может кликать по ячейкам таблицы, переключая значения между 0 и 1, при этом система автоматически обеспечивает симметричность матрицы благодаря синхронизации между элементами Button и Label.

Интерактивные возможности интерфейса значительно расширены за счет класса *ZoomableWidget*, который обеспечивает функцию масштабирования с привязкой к положению курсора. Для навигации по объемным спискам решений в классе SolutionListWidget добавлены скроллбары. Элементы управления воспроизведением в MainApp позволяют как запускать алгоритм полностью, так и анализировать его работу пошагово.

Система активно проверяет корректность вводимых данных через методы класса *Validator*. Для матрицы смежности автоматически проверяется соответствие требованиям (симметричность, нулевая диагональ). При обнаружении ошибок выводятся сообщения через messagebox.

Кратко перечислим ключевые методы для классов:

Класс *Validator* (Валидация данных)

Методы:

1. *validate_graph_exists(graph)*

   Проверяет существование графа (не None). Возвращает True/False, показывает ошибку если графа нет.

2. *validate_graph_for_save(adj_matrix)*

   Проверяет возможность сохранения матрицы (не None). Возвращает True/False, показывает ошибку если матрицы нет

3. *validate_matrix_size(size, max_size=15)*

   Проверяет размер матрицы (не больше max_size). Возвращает скорректированный размер, показывает предупреждение.

4. *validate_cell_value(value)*

   Проверяет значение ячейки (должно быть 0 или 1). Возвращает число или None при ошибке.

5. *validate_matrix(matrix)*

   Проверяет матрицу на: квадратность, симметричность, нули на диагонали. Возвращает (bool, сообщение)

Класс *FileManager* (Работа с файлами)

Методы:

1. *get_save_filename(title, defaultextension, filetypes)*

   Открывает диалог сохранения файла. Возвращает путь к файлу.

2. *get_open_filename(title, filetypes)*

   Открывает диалог выбора файла. Возвращает путь к файлу.

3. *save_matrix_to_file(filename, matrix)*

   Сохраняет матрицу в текстовый файл. Возвращает True/False, показывает статус.

4. *load_matrix_from_file(filename)*

   Загружает матрицу из файла. Возвращает (матрица, сообщение).

Класс *UIManager* (Создание UI-элементов)

Методы:

1. *create_frame(parent, **kwargs)*

   Создает фрейм с настройками.

2. *create_label(parent, text, **kwargs)*

   Создает текстовую метку.

3. *create_button(parent, text, command, **kwargs)*

   Создает кнопку с обработчиком.

4. *create_entry(parent, **kwargs)*

   Создает поле ввода.

5. *configure_grid_frame(frame, rows, cols)*

   Настраивает сетку фрейма.

6. *create_table_header(parent, text, row, col, **kwargs)*

   Создает заголовок таблицы.

Класс *ZoomableWidget* (Управление масштабированием)

Методы:

1. *_zoom_to_cursor(event, scale_factor)*

   Масштабирует график/граф относительно курсора.

2. *_on_zoom(event)*

   Обрабатывает событие колеса мыши (Windows/Mac).

3. *_on_zoom_linux(event, direction)*

   Обрабатывает событие кнопок мыши (Linux).

4. *reset_zoom()*

   Сбрасывает масштаб к исходному.

5. *save_original_limits()*

   Сохраняет исходные границы осей.

Класс *FitnessPlotWidget* (График приспособленности)

Методы:

1. *_create_plot(self)*

   Создает фигуру и оси для графика, а также инициализирует холст для отображения графика в Tkinter.

2. *draw_empty(self)*

   Вызывает метод для рисования пустого графика.

3. *draw_fitness(self, generations: list[int] = [], best: list[float] = [], avg: list[float] = [])*

Очищает оси графика и рисует линии для лучших и средних значений фитнеса, если они предоставлены.

4. *update_fitness_plot(self, best_fitness: list[float], avg_fitness: list[float])*

   Обновляет график фитнеса, принимая списки лучших и средних значений фитнеса и вызывая метод для их отображения.

Класс *MainApp* (Главное окно)

Ключевые методы:

1. *update_graph(adj_matrix)*

   Обновляет граф по новой матрице смежности.

2. *_draw_graph(highlight_clique=False)*

   Отрисовывает граф с подсветкой клики.

3. *run_algorithm()*

   Запускает генетический алгоритм (заглушка).

4. *save_graph(), load_adjacency_matrix()*

   Работа с файлами через FileManager.

Класс *MatrixWindow* (Окно матрицы)

Методы:

1. *_initialize_window(self)*

   Настраивает свойства окна: заголовок "Adjacency Matrix", размеры и стиль.

2. *_initialize_variables(self)*

   Инициализирует переменные: size_var (размер матрицы) и entries (список для хранения ячеек).

3. *_create_ui(self)*

   Координирует создание всех элементов UI: верхней панели, блока выбора размера и таблицы.

4. *create_matrix_table(self)*

Очищает старую таблицу и создаёт новую на основе выбранного размера.

5. *toggle_cell(self, i, j)*

   Меняет значение ячейки (0 ↔ 1) и синхронизирует противоположную (j, i) для симметрии.

6. *_generate_random_matrix(self, size)*

   Возвращает случайную матрицу через RandomGenerator.generate_random_matrix.

7. *_handle_large_matrix(self, matrix)*

   Для матриц >15x15 сохраняет данные и выводит их в текстовом формате.

8. *_handle_small_matrix(self, matrix)*

   Для матриц ≤15x15 обновляет таблицу UI.

9. *_update_matrix_display(self, matrix)*

   Визуально обновляет значения ячеек в таблице на основе переданной матрицы.

10. *create_graph(self)*

    Преобразует текущую матрицу в граф и передаёт её в окно для визуализации.

    Интерфейс см в приложении Б.

ИТЕРАЦИЯ 3

Описание текущей реализации генетического алгоритма:

По итогам тестирования в алгоритм были внесены изменения:

1. Теперь при любом выборе случайного элемента из множества элементов с заданными весами, веса сначала масштабируются, аналогично выбору родителя. Например, ранее при генерации начальной популяции веса вершин были равны их степеням, что приводило к слишком частому выбору

больших вершин и на тестах вершины с небольшой степенью почти никогда не выбирались

2. Изменился алгоритм отбора в новую популяцию, ранее после совместной сортировки потомков и родителей выбирался первый элемент из отсортированного списка, это приводило к тому, что, так как сортировка устойчивая, то всегда выбирался один и тот же элемент. Это вредило алгоритму, ведь далее мы выбираем хромосому с наибольшим расстоянием хемминга от уже выбранных, а уже выбранная хромосома могла быть близка к тем, которые входят в наибольшую клику. Также теперь выбирается не просто последний элемент с наибольшим расстоянием от уже выбранных, а случайный элемент из множества элементов с равным друг другу наибольшим расстоянием до уже выбранных.

Также была произведена связь алгоритма и гуи, для этого в классе MainApp были добавлены или обновлены следующие методы:

update_graph(self, adj_matrix: np.ndarray) -> None – обновляет текущий граф приложения на основе переданной матрицы смежности. Передает граф в менеджер алгоритмов, автоматически сбрасывает состояние алгоритма и адаптирует параметры по умолчанию в соответствии с размером нового графа.

run(self, n: int) -> None – основной метод выполнения генетического алгоритма. Поддерживает три режима работы:

n=1: выполнение одного поколения

n>1: выполнение N поколений

n=-1: выполнение до завершения алгоритма

Перед запуском проверяет наличие графа и сбрасывает алгоритм при изменении параметров. Загружает параметры из интерфейса, выполняет алгоритм через AlgorithmManager, обновляет интерфейс (список решений, график фитнеса, визуализацию клики) и обрабатывает ошибки выполнения.

run_algorithm(self) -> None – запускает одно поколение генетического алгоритма. Вызывает основной метод выполнения с параметром n=1.

step_algorithm(self) -> None – выполняет пять последовательных поколений генетического алгоритма. Вызывает основной метод выполнения с параметром n=5.

end_algorithm(self) -> None – запускает выполнение генетического алгоритма до его полного завершения (до достижения критериев остановки). Вызывает основной метод выполнения с параметром n=-1. Автоматически определяет момент завершения вычислений и фиксирует итоговый результат.

reset_algorithm(self) -> None – полностью сбрасывает состояние алгоритма и связанные элементы интерфейса.

Также были рассчитаны и добавлены значения по умолчанию, которые лучше остальных показали себя на тестах:

population_size – количество вершин, деленное на 10, но не менее 5 и не более 30

max_generations – количество вершин, умноженное на 3, но не менее 150 и не более 500

stagnation_limit –в 10 раз меньше, чем максимальное количество итераций, но не менее 20 и не более 50

max_mutation_prob_gene - 0,1

max_mutation_prob_chrom – 0,4, если в популяции меньше 8 особей и 0,3, если больше

fitness_scaling_percent –по умолчанию 40

max_crossover_points - количество вершин, деленное на 5, но не менее 3 и не больше количества вершин

decrease_percent – 20

decrease_step – 20

Обработка и хранение данных, вспомогательные функции

Были реализован класс, отвечающий за связь генетического алгоритма с GUI, между различными компонентами, хранение графа, параметров и промежуточных данных и работы с ними, а также некоторые вспомогательные функции

1. Класс *AlgoritmManager* – отвечает за координацию работы генетического алгоритма, управление состоянием, работу с графами, параметрами и промежуточными данными

Поля класса:

Optional[Graph] graph – граф, хранящийся во время работы алгоритма (объект класса Graph)

Optional[Parameters] params – параметры, использующиеся для работы алгоритма (объект класса Parameters)

Optional[GeneticAlgorithm] algorithm – объект класса GeneticAlgorithm, в котором реализуется работа генетического алгоритма

Optional[History] history – объект класса History, который хранит историю работы алгоритма (лучшую и среднюю приспособленность на каждом шаге)

bool is_initialized – флаг, указывающий на то, готов ли алгоритм к выполнению (введены ли данные: параметры и граф)

bool is_completed – флаг, указывающий на то, завершил ли алгоритм работу (достиг ли условия остановки (макс. поколений/застой))

Основные методы:

load_graph_from_matrix(self, file_path: str) -> None – загружает граф из JSON-файла с матрицей смежности

generate_random_graph(self, n: int) -> None – генерирует случайный неориентированный граф с n вершинами

save_graph_as_matrix(self, file_path: str) -> None – сохраняет текущий граф в JSON-файл как матрицу смежности

set_graph(self, graph: Graph) -> None – устанавливает граф для алгоритма

set_parameters(self, params: Parameters) -> None – устанавлиает параметры алгоритма

load_and_validate_parameters(self, data: dict) -> None – проверяет параметры на корректность. Использует graph.n для проверки параметров

_check_initialization(self) -> None – проверяет, можно ли инициализировать алгоритм

initialize_algorithm(self) -> None – инициализирует алгоритм с текущими графом и параметрами

_check_ready(self) -> None – проверяет, готов ли алгоритм к выполнению

step(self) -> Tuple[List[int], List[List[int]]] – выполняет одну итерацию алгоритма

step_n(self, n: int = 5) -> Tuple[List[int], List[List[int]]] – выполняет N итераций алгоритма

run_until_completion(self) -> Tuple[List[int], List[List[int]]] – выполняет алгоритм до завершения

_get_current_state(self) -> Tuple[List[int], List[List[int]]] – возвращает текущее состояние алгоритма

plot_history(self) -> plt.Figure – строит график эволюции на основе истории

reset_algorithm(self) -> None – Сбрасывает алгоритм до начального состояния. Сохраняет текущие граф и параметры

Методы класса:

record(self, generation: int, population: Population) – Пересчитывает параметры популяции и сохраняет их в историю

plot_fitness(self) – Строит график динамики лучшей и средней приспособленности по поколениям

save_to_json(self, path: str) – Сохраняет историю работы алгоритма в результирующий json-файл

Были обновлены функции GUI для загрузки из JSON-файла графа, заданного в виде матрицы смежности и передаче его менеджеру для дальнейшей работы, и сохранения графа в JSON-файл в виде матрицы смежности

Были обновлены методы из класса *MainApp*:

load_adjacency_matrix(self) – загрузка матрицы смежности из файла

save_graph(self) – сохранение графа в файл

И проверка корректности матрицы, вводимой из файла с помощью функции *parse_matrix_from_file(filename)*

Также, была реализована установка параметров по умолчанию в GUI в зависимости от количества вершин графа с использованием класса *ParameterConfig* и функции *update_defaults_based_on_graph_size(cls, graph_size)*

Разработка GUI

Была проведена подготовка к связыванию GUI с алгоритмом. Класс *MainApp* разбит на классы, чтобы упростить расширение и изменение системы и ее связи алгоритмом.

Текущий набор классов:

1. *GraphVisualizer* — класс отрисовки графа.
2. *LeftPanel* — левая панель с параметрами, кнопками для вызова установки по умолчанию и вызова окна ввода матрицы соответственно.
3. *ParameterConfig* — класс для работы с параметрами по умолчанию.
4. *FitnessPlotWidget* — класс для отрисовки графика функции приспособленности.
5. *SolutionListWidget* — класс для отображения текущей популяции.
6. *MatrixWindow* — класс окна ввода матрицы.

7. *ZoomableWidget* — класс для создания способности приближать/отдалять виджеты.
8. *FileManager* — класс для работы с файлами.
9. *Validator* — класс проверок данных.
10. *UIManager* — класс унифицированного создания визуальных элементов интерфейса (вызовов окна ошибки, информации и т. д.)
11. *Styles* и *Colors* — стили и цвета для элементов интерфейса соответственно.
12. *MainApp* — класс основного окна, связывает остальные его визуальные блоки, получает и передает на визуализацию результаты работы алгоритма.

Рассмотрим *MainApp* и вынесенные из него классы.

Класс *GraphVisualizer*

Ключевые методы:

1. *_build_graph_area(self)*

   Создает область для графа, инициализирует холст для отображения графа и кнопку сброса.
2. *_draw(self, highlight_clique: bool = False)*

   Отрисовывает граф на холсте, выделяя клик, если это указано.
3. *update_graph(self, graph, layout, clique=None)*

   Обновляет граф и его расположение, а также выделяет текущий клик, если он указан.
4. *reset_zoom(self)*

   Сбрасывает масштаб графа, используя функциональность виджета зума.

Класс *SolutionListWidget*

Методы:

1. *_create_listbox_frame(self)*

Создает фрейм для размещения списка решений.

2. *update_solution_list(self, solutions, best_index)*

   Обновляет список решений в пользовательском интерфейсе, форматируя и добавляя новые решения.

Класс *LeftPanel*

Ключевые методы:

1. *_create_parameter_fields(self)*

   Создает поля ввода для параметров, используя значения по умолчанию из конфигурации.

2. *_create_parameter_buttons(self, open_matrix_callback)*

   Создает кнопки для установки значений по умолчанию и открытия матрицы.

3. *validate_and_get_parameters(self) -> Optional[dict]*

   Проверяет параметры на корректность с пмощью Valdator

4. *set_default_values(self)*

   Устанавливает значения по умолчанию для всех полей ввода, основываясь на размере графа.

5. *reset_parameters_changed_flag(self)*

   Сбрасывает флаг изменения параметров.

6. *has_parameters_changed(self) -> bool*

   Возвращает состояние флага изменения параметров.

Класс *MainApp*

Ключевые методы:

1. *_initialize_window(self)*

   Инициализирует окно приложения с заголовком и размерами.

2. *_initialize_state(self)*

   Инициализирует состояние приложения, включая граф, матрицу смежности и параметры.

3. *_bind_events(self)*

   Привязывает события к окну, такие как изменение размера.

4. *create_widgets(self)*

   Создает все виджеты приложения, включая панели и области графа.ъ

5. *_update_ui_after_algorithm(self, n)*

   Обновляет пользовательский интерфейс после выполнения алгоритма.

6. *run(self, n)*

   Запускает алгоритм с указанным количеством шагов.

Новые классы состоят из ранее реализованных методов. Изменены или добавлены методы получения данных, их проверки и изменения для получения от алгоритма или для передачи ему. Например, как инструмент добавлены метод *ndarray_to_list* и другие методы преобразования формата данных.

Изменена некоторая часть визуального отображения данных: убрано подсвечивание лучшего решения, оно всегда находится на вершине списка решений, при загрузке и сохранении по умолчанию установлено отображение файлов формата json.

Демонстарция работы приложения

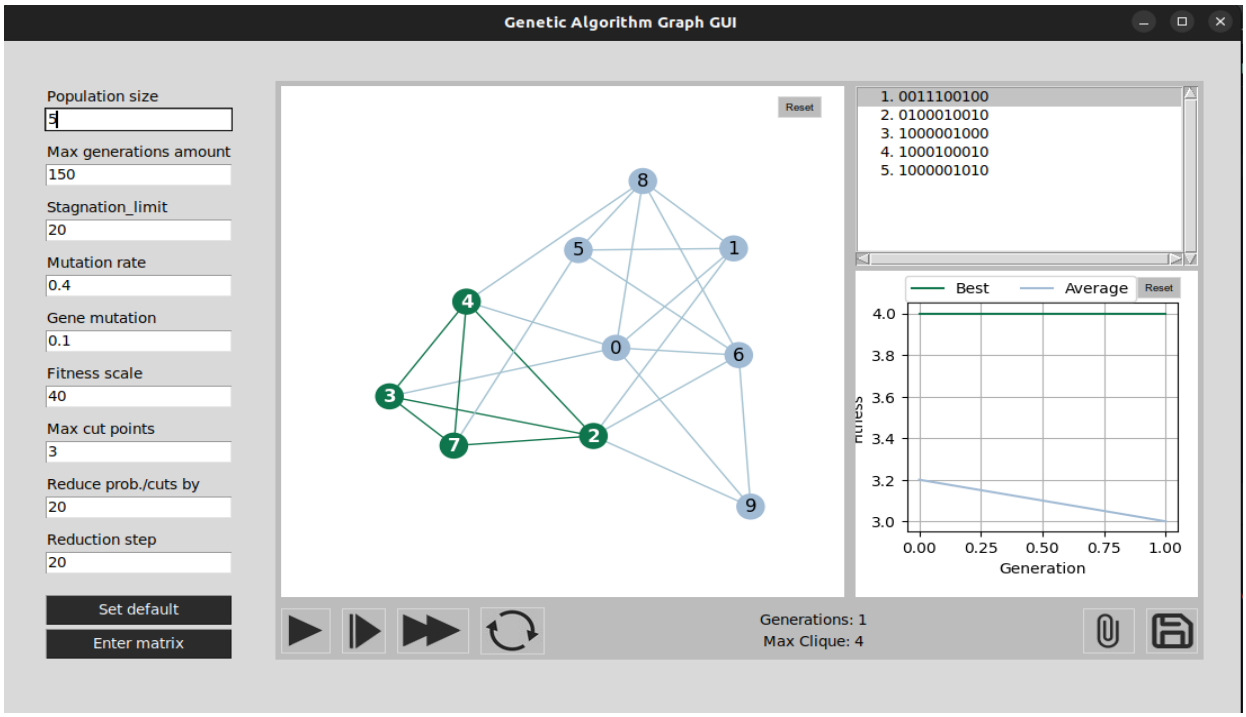

Рисунок 1 – первая итерация алгоритма на случайно сгенерированной матрице и настройках по умолчанию

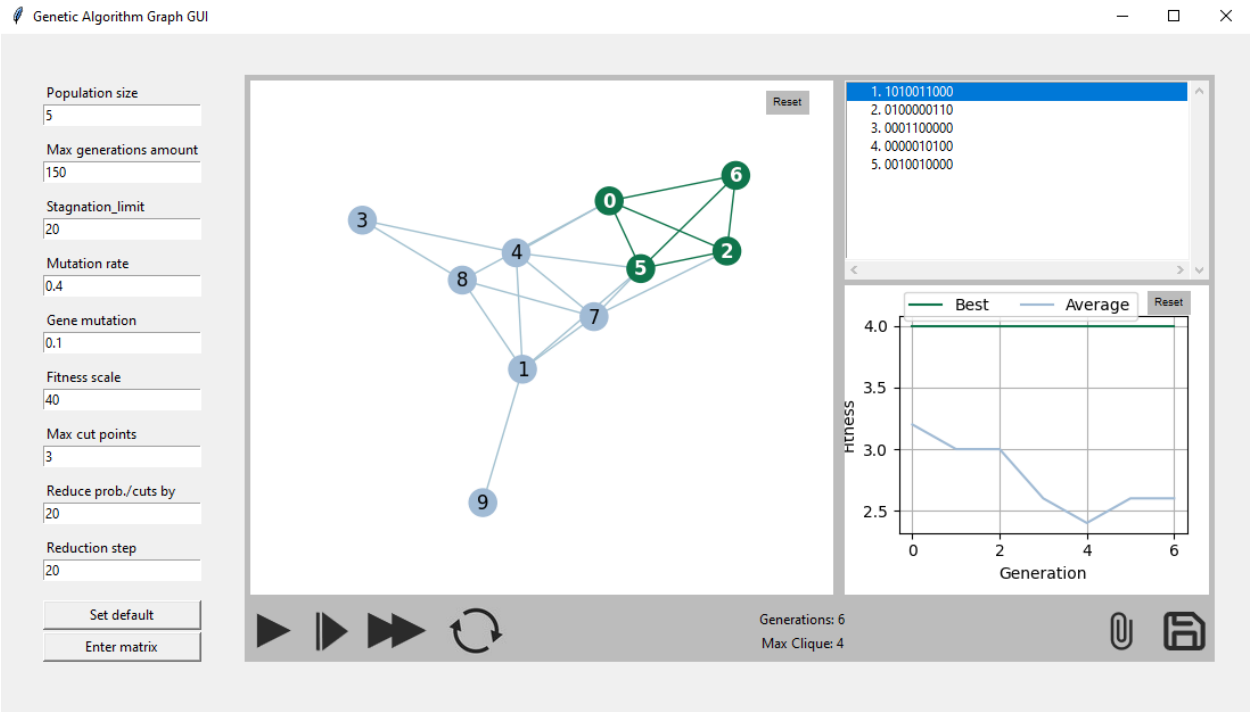

Рисунок 2 – выполнение на 5 шагов алгоритма вперед

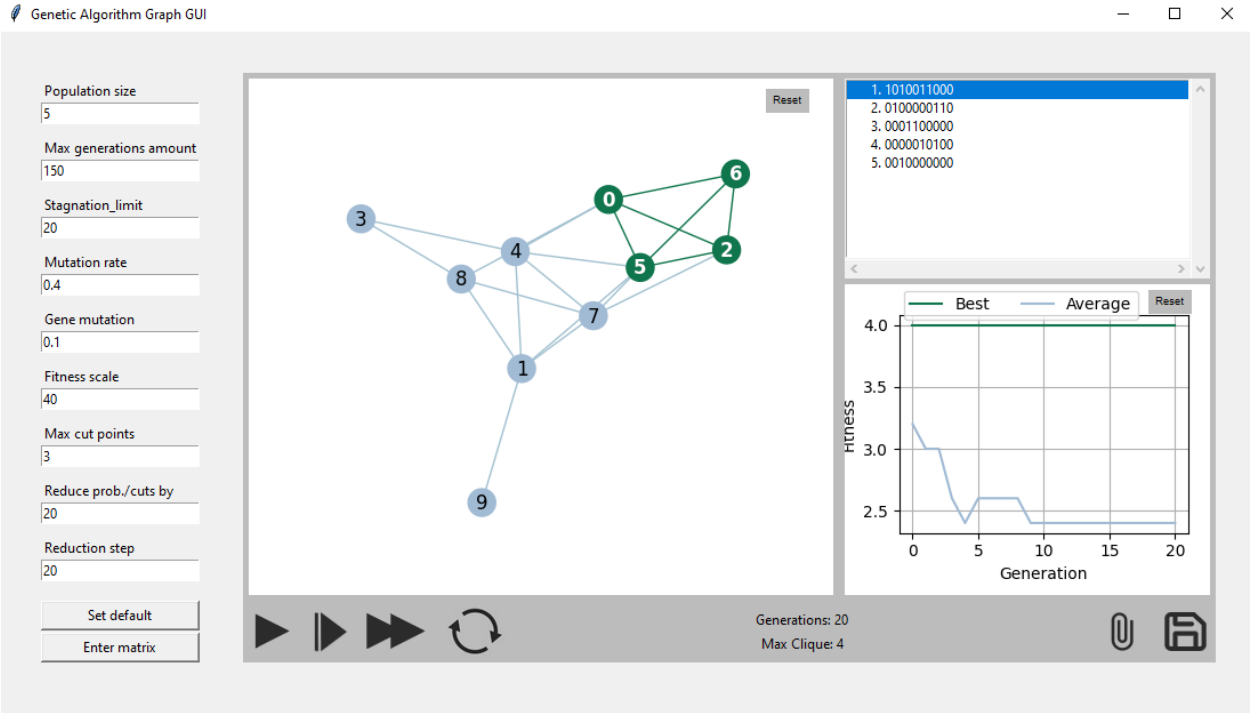

Рисунок 3 – выполнение алгоритма до конца

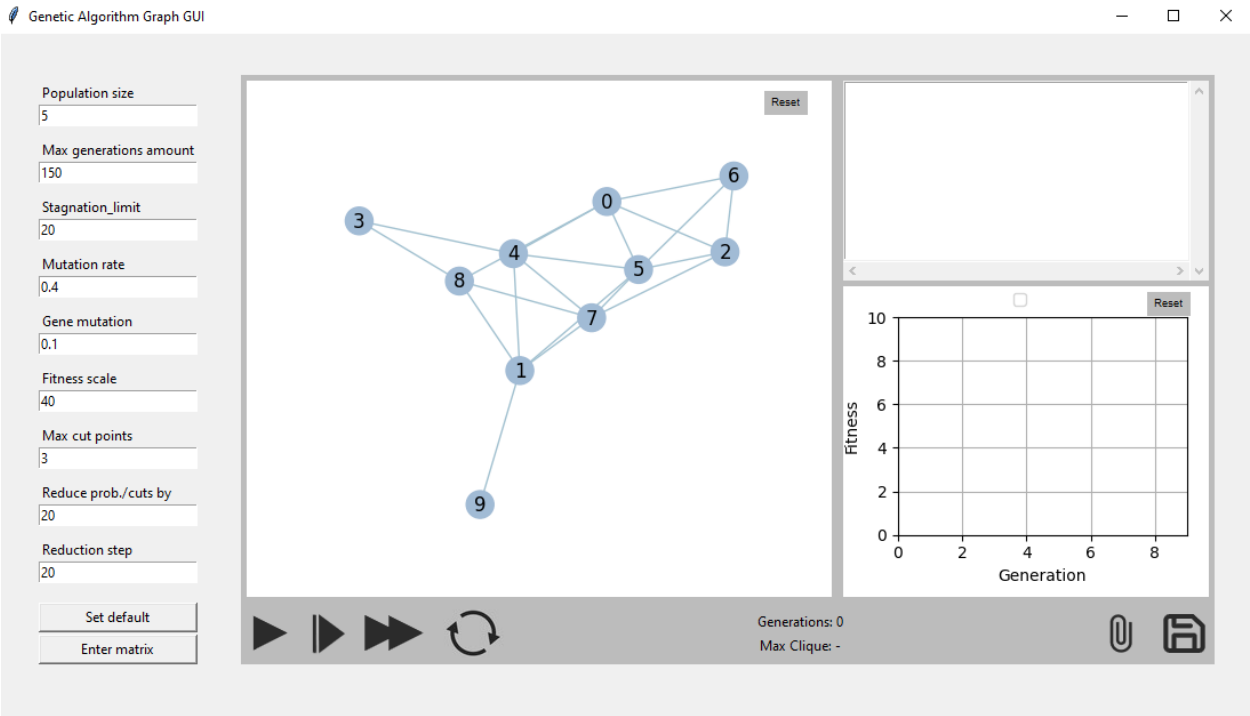

Рисунок 4 – сброс алгоритма с сохранением данных

Выводы.

Реализована программа по решению задачи о поиске клики максимального размера.

Проанализированы и выбраны оптимальные подходы к представлению решений, отбору, скрещиванию и мутации. Решение написано на Python с соблюдением требования самостоятельной реализации генетического алгоритма без использования специализированных библиотек.

Разработанное приложение предоставляет удобный инструмент для исследования работы алгоритма: визуализацию текущего лучшего решения, отслеживание динамики приспособленности популяции, а также возможность изменять параметры для улучшения результатов. Интерфейс написан с использование tkinter, интегрирова визуализация графов через networkX и отображение статистики работы алгоритма с помощью Mmatplotlib. Предусмотрена возможность сохранения и загрузки данных в формате json.

Программа успешно решает задачу поиска клики максимального размера. Полученный результат демонстрирует, что генетические алгоритмы являются действенным подходом к решению сложных комбинаторных задач.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ АЛГОРИТМА

graph1.json:

```
{
    "0": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    "1": [0, 2, 3, 4, 5, 6, 7, 11, 12],
    "2": [0, 1, 3, 4, 5, 6, 7, 13, 14],
    "3": [0, 1, 2, 4, 5, 6, 7, 15],
    "4": [0, 1, 2, 3, 5, 6, 7, 16],
    "5": [0, 1, 2, 3, 4, 6, 7, 17],
    "6": [0, 1, 2, 3, 4, 5, 7, 18],
    "7": [0, 1, 2, 3, 4, 5, 6, 19],
    "8": [0, 9, 10, 11, 20, 21],
    "9": [0, 8, 12, 13, 20, 21],
    "10": [0, 8, 14, 15, 20],
    "11": [1, 8, 16, 17, 20],
    "12": [1, 9, 18, 19, 20],
    "13": [2, 9, 16, 17, 20],
    "14": [2, 10, 18, 19, 20],
    "15": [3, 10, 16, 18, 20],
    "16": [4, 11, 13, 15, 20],
    "17": [5, 11, 13, 19, 20],
    "18": [6, 12, 14, 15, 20],
    "19": [7, 12, 14, 17, 20],
    "20": [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    "21": [8, 9, 22],
    "22": [21, 23],
    "23": [22, 24],
    "24": [23, 25],
    "25": [24, 26],
    "26": [25]
}
```

Вывод при тесте:

```
Начало работы генетического алгоритма
Размер графа: 27 вершин
Параметры алгоритма:
  Размер популяции: 5
  Макс. поколений: 100
  Лимит застоя: 20
  Начальные точки кроссовера: 5

Начальная популяция:
  Особа 1: 01111111100000000000000000 (размер: 8)
  Особа 2: 01111111100000000000000000 (размер: 8)
  Особа 3: 10000000000010000010000000 (размер: 3)
  Особа 4: 01111111100000000000000000 (размер: 8)
  Особа 5: 10000000001001000000000000 (размер: 3)

Поколение 1:
  Лучший размер клики: 8
  Застой: 1 / 20
  Параметры: точки=5, мутация хромосомы=0.300, мутация гена=0.100
  Текущая популяция:
```

```
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000010000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 10000000001001000000000000 (размер: 3)
    Особа 5: 10000000000000100010000000 (размер: 3)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]

Поколение 2:
  Лучший размер клики: 8
  Застой: 2 / 20
  Параметры: точки=5, мутация хромосомы=0.300, мутация гена=0.100
  Текущая популяция:
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000010000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 10000000001001000000000000 (размер: 3)
    Особа 5: 10000000000000100010000000 (размер: 3)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]

Поколение 3:
  Лучший размер клики: 8
  Застой: 3 / 20
  Параметры: точки=5, мутация хромосомы=0.300, мутация гена=0.100
  Текущая популяция:
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000010000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 10000000001001000000000000 (размер: 3)
    Особа 5: 00000000000000000000000010 (размер: 1)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]

…
Сокращение вывода
…

Поколение 17:
  Лучший размер клики: 8
  Застой: 17 / 20
  Параметры: точки=4, мутация хромосомы=0.270, мутация гена=0.090
  Текущая популяция:
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000010000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 01000000000100000000000000 (размер: 2)
    Особа 5: 10000000001001000000000000 (размер: 3)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]

Поколение 18:
  Лучший размер клики: 8
  Застой: 18 / 20
  Параметры: точки=4, мутация хромосомы=0.270, мутация гена=0.090
  Текущая популяция:
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000010000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 01000000000100000000000000 (размер: 2)
    Особа 5: 10000000001001000000000000 (размер: 3)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]
```

```
Поколение 19:
  Лучший размер клики: 8
  Застой: 19 / 20
  Параметры: точки=4, мутация хромосомы=0.270, мутация гена=0.090
  Текущая популяция:
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000100000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 01000000000100000000000000 (размер: 2)
    Особа 5: 10000000001001000000000000 (размер: 3)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]

Поколение 20:
  Лучший размер клики: 8
  Застой: 20 / 20
  Параметры: точки=3, мутация хромосомы=0.240, мутация гена=0.080
  Текущая популяция:
    Особа 1: 01111111100000000000000000 (размер: 8)
    Особа 2: 10000000000010000100000000 (размер: 3)
    Особа 3: 00000101100000000000000000 (размер: 3)
    Особа 4: 01000000000100000000000000 (размер: 2)
    Особа 5: 10000000001001000000000000 (размер: 3)
  Лучшая клика: вершины [0, 1, 2, 3, 4, 5, 6, 7]


Результат работы алгоритма:
Найдена клика размера 8
Вершины в клике: [0, 1, 2, 3, 4, 5, 6, 7]
Всего поколений: 20
Клика валидна
Причина остановки: Превышен лимит застоя
```