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

ОТЧЕТ

по летней практике

по дисциплине «Генетические алгоритмы»

Тема: Задача о вершинном покрытии графа

Студент гр. 3341 _____ Кудин А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Цель работы - создать интерактивный прототип генетического алгоритма для приближенного решения задачи минимального вершинного покрытия в графе.

Задание.

Для заданного неориентированного графа G = (V, E), необходимо найти множество вершин S, которое является вершинным покрытием графа и минимально.

Описание прототипа интерфейса

Прототип графического интерфейса реализован прямо в Jupyter Notebook с помощью ipywidgets – это позволяет быстро собирать контролы ввода и вывод прямо в ячейках без внешних окон. При старте экран делится на две вертикальные колонки: слева — панель настроек, справа — область вывода.
В панели настроек расположены поля для задания параметров алгоритма:

- „Вершины“ и „Рёбра“ (IntText) — задают размеры и плотность генерируемого графа,
- „Популяция“, „Поколения“ и „Мутация“ (IntText/FloatText) — основные параметры эволюции,
- „Селекция“ (Dropdown) — выбор между турнирным и рулеточным отбором,
- „Шагов пропустить“ (IntText) и кнопка «Пропустить N поколений» — для быстрой «перемотки» эволюции.

Чуть ниже находятся две основные кнопки: «Сгенерировать граф», которая сбрасывает состояние и создаёт новый случайный граф с заданными характеристиками, и «Следующий шаг», запускающая одну итерацию генетического алгоритма.

Ниже панели настроек размещён элемент Textarea, в котором в виде списка строк отображаются все рёбра текущего графа (каждое u—v на новой строке). Это помогает сразу увидеть структуру графа текстом, отдельно от картинки.

Справа — область output_area, куда выводятся текстовые отчёты (номер поколения, лучший вектор покрытия, размер покрытия, число покрытых рёбер и текущее значение fitness). Под текстом автоматически рисуются две визуализации:

1. Граф с подсвеченными зелёным цветом вершинами выбранного покрытия (функция visualize_graph).
2. Диаграмма динамики fitness по поколениям (функция plot_fitness).

Технологии и библиотеки для GUI

Для построения интерфейса был выбран Jupyter Notebook в сочетании с библиотекой ipywidgets. Этот стек даёт сразу готовые виджеты для числового ввода (IntText, FloatText), выпадающих списков (Dropdown), текстовых областей (Textarea) и кнопок (Button), причём все они автоматически располагаются в ячейках блокнота и умеют реагировать на события без перезагрузки страницы. Для компоновки элементов используются контейнеры HBox и VBox.

Для вывода и обновления визуализаций применяется модуль IPython.display (в частности, объект Output), который гарантирует, что текст и графики появятся именно в нужном месте интерфейса. Функции visualize_graph и plot_fitness рисуют граф и диаграмму при помощи networkx (для расчёта и отрисовки структуры графа) и matplotlib (для построения кривых динамики качества). Для расчёта статистики популяции и нормализации значений используется NumPy, который обрабатывает массивы fitness-значений и возвращает среднее и максимумы для построения графиков.

Реализация генетического алгоритма

В нашем прототипе каждая особь («хромосома») представлена простым бинарным вектором длины N, где N — число вершин графа. Единица в позиции i означает, что вершина i включена в текущее покрытие, ноль - что она в него не входит. Такое представление удобно тем, что все основные операции (скрещивание, мутация, оценка качества) сводятся к простым операциям над списками из нулей и единиц.

Для оценки качества решения служит fitness-функция, которая одновременно учитывает такие критерии: корректность покрытия (штраф за каждое непокрытое ребро), минимальность (размер самого покрытия). Конкретно мы вычисляем

- uncovered = число рёбер, у которых оба конца не включены в покрытие
- size = сумма единиц в векторе
- fitness = - (size + penalty × uncovered)

где penalty (обычно 10) гарантирует, что непокрытые рёбра будут жёстко наказываться, а знак «–» превращает задачу минимизации в задачу максимизации: чем ближе fitness к нулю, тем лучше решение.

Начальная популяция формируется случайным образом функцией generate_population, повторяющей generate_individual заданное число раз.

Селекция родителей реализована двумя вариантами. По умолчанию применяется турнирный отбор в парах: из двух случайно выбранных особей победителем становится та, у которой fitness выше. Второй вариант — «рулетка», где каждый индивид получает вероятность пропорционально сдвинутому в положительную область значению fitness. Пользователь может переключаться между этими методами прямо из интерфейса.

Скрещивание проводится в классическом формате одноточечного crossover: мы выбираем случайную точку разрыва и соединяем префикс одной хромосомы с суффиксом другой, получая «потомка». Мутация — побитовая: каждое значение вектора может инвертироваться с небольшой вероятностью (по

умолчанию 0.1), что предотвращает преждевременную сходимость и помогает выйти из локальных минимумов.

Основная эволюционная итерация (step_ga) включает в себя:

1. Вычисление fitness для всех особей.
2. Формирование нового поколения: для каждой позиции выбираются два родителя (турниром или рулеткой), из них создаётся ребёнок через crossover и затем модифицируется через mutate.
3. Обновление истории: в списке best_scores сохраняется наивысшее значение fitness текущей популяции, а в mean_scores — среднее, что позволяет отслеживать динамику сходимости.
4. Увеличение счётчика поколений.

Основные компоненты GA

Представление (кодирование) решения

Мы используем бинарную хромосому — список из нулей и единиц длины N, где N — количество вершин графа. Элемент со значением 1 означает, что соответствующая вершина включена в покрытие. Такое простое и компактное представление позволяет легко манипулировать решениями при скрещивании и мутации.

Fitness-функция

Чтобы алгоритм знал, какие решения лучше, мы подсчитываем два показателя:

- Размер покрытия (количество единиц),
- Число непокрытых рёбер (штраф за каждое ребро, концы которого оба не включены).

Итоговая оценка

fitness=−(размер+penalty×uncovered)

делает так, что решения с меньшими размерами и без ошибок получают более высокие (меньше по абсолютной величине, но ближе к нулю) значения

функции приспособленности.

Селекция

- Турнирная селекция: из двух случайно выбранных особей побеждает та, у которой fitness выше.
- Рулеточная селекция: каждый индивид получает вероятность выбора, пропорциональную его относительной силе (fitness после сдвига в положительную область).

  Выбор метода происходит через интерфейс, давая возможность сравнить их эффективность.

Скрещивание (crossover)

Применяется одноточечное скрещивание: случайно выбираем точку деления хромосомы и формируем потомка, объединяя первый сегмент от одного родителя с остатком от другого.

Мутация

Для поддержания разнообразия и предотвращения застревания в локальных оптимумах мы побитово инвертируем гены с небольшой вероятностью (обычно 10 %). Это позволяет вносить «свежие» вариации, даже если оператор скрещивания не создаёт новых сочетаний.

Параметры алгоритма

- Размер популяции — число особей в каждом поколении.
- Число поколений — сколько раз выполнить цикл селекции, скрещивания и мутаций.
- Вероятность мутации — как часто случайные гены меняются.
- Penalty — вес штрафа за непокрытые рёбра.

Интеграция алгоритма и интерфейса

Связующим звеном между вычислительной частью и пользовательским интерфейсом служит единый объект state и набор функций-обработчиков, привязанных к событиям виджетов. Когда пользователь нажимает «Сгенерировать граф», в обработчике initialize сначала считываются все параметры из полей ввода (число вершин, рёбер, размер популяции, поколений, вероятность мутации и способ селекции), затем вызывается generate_graph для построения нового networkx-графа и generate_population для старта популяции. Результат сохраняется в state, а текстовое поле edges_display наполняется списком рёбер.

Дальнейшие шаги эволюции реализуются в единой функции _do_steps(count), которую вызывают два обработчика: next_step (один шаг) и skip_steps (несколько шагов подряд). Внутри _do_steps вызывается step_ga нужное число раз, после чего из state извлекаются лучшие и средние значения fitness, а также текущая лучшая хромосома. Эти данные сразу выводятся текстом в область output_area, а затем — визуально: функциями visualize_graph и plot_fitness рисуются обновлённый граф с подсвеченным покрытием и график динамики качества.

Выводы.

В результате выполнения работы был получен прототип генетического алгоритма, предложенный прототип эффективно находит корректные вершинные покрытия и постепенно минимизирует их размер. Интерактивный GUI позволяет быстро менять параметры, выбирать метод селекции и следить за эволюцией решений шаг за шагом.

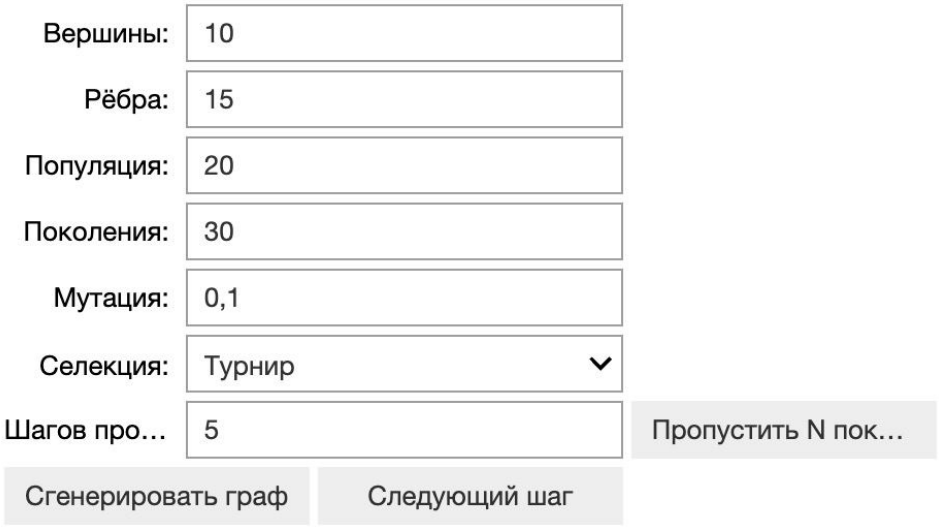

РИСУНОК № 1 – ПАНЕЛЬ ПАРАМЕТРОВ

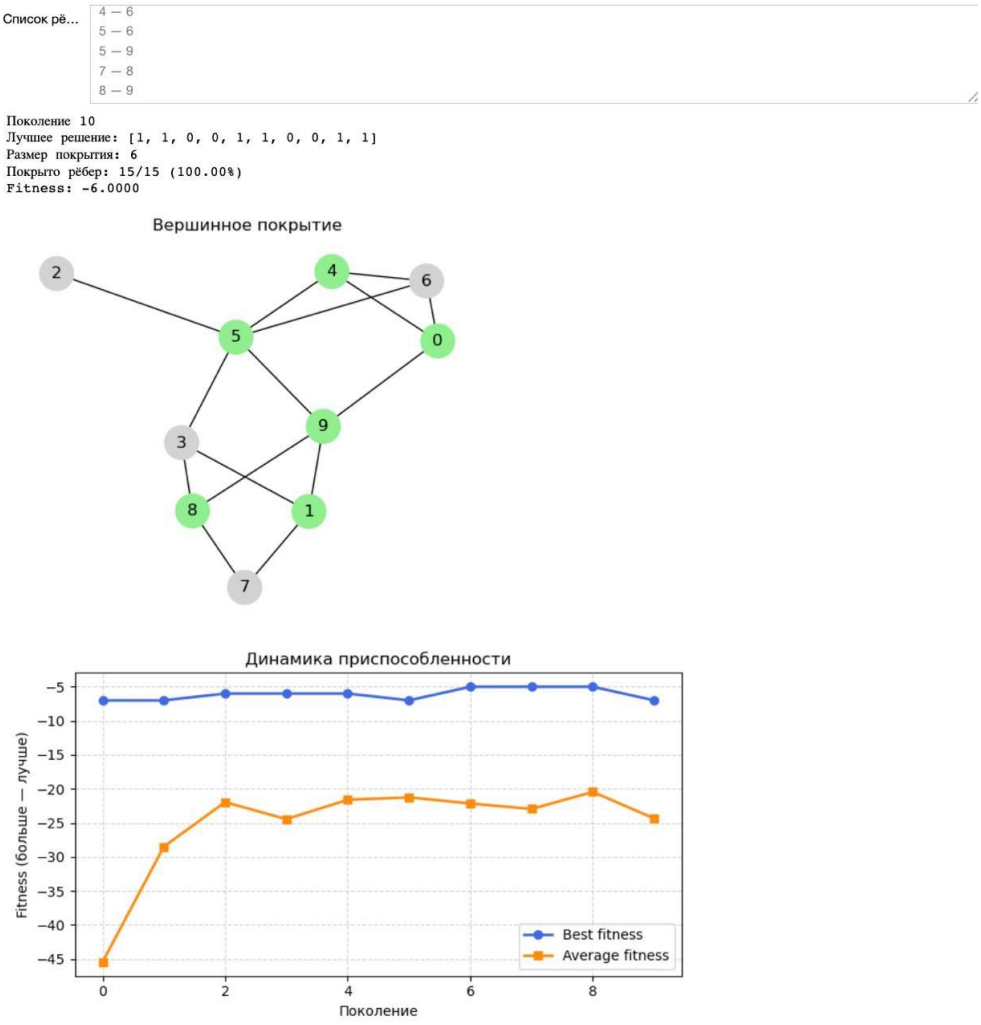

Рисунок № 2 – Пример работы алгоритма

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
import random
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import ipywidgets as widgets
from IPython.display import display, clear_output

state = {
    "graph": None,
    "edges": None,
    "population": None,
    "best_scores": [],
    "mean_scores": [],
    "generation": 0,
    "params": {
        "num_vertices": 10,
        "num_edges": 15,
        "pop_size": 20,
        "generations": 30,
        "mutation_rate": 0.1,
        "selection_method": "tournament"
    }
}

def generate_graph(num_vertices, num_edges):
    G = nx.Graph()
    G.add_nodes_from(range(num_vertices))
    while G.number_of_edges() < num_edges:
        u, v = random.sample(range(num_vertices), 2)
        G.add_edge(u, v)
    return G

def generate_individual(num_vertices):
    return [random.choice([0, 1]) for _ in range(num_vertices)]

def fitness(individual, edges):
    uncovered = sum(1 for u, v in edges if not (individual[u] or
individual[v]))
    penalty = 10
    size = sum(individual)
    return - (size + penalty * uncovered)

def generate_population(size, num_vertices):
    return [generate_individual(num_vertices) for _ in range(size)]

def crossover(p1, p2):
    point = random.randint(1, len(p1) - 1)
    return p1[:point] + p2[point:]

def mutate(ind, rate):
    return [(1 - g) if random.random() < rate else g for g in ind]

def tournament_selection(pop, scores):
    i1, i2 = random.sample(range(len(pop)), 2)
```

```
        return pop[i1] if scores[i1] > scores[i2] else pop[i2]

    def roulette_selection(pop, scores):
        min_score = min(scores)
        shifted = [s - min_score + 1e-6 for s in scores]
        total = sum(shifted)
        probs = [s / total for s in shifted]
        idx = np.random.choice(len(pop), p=probs)
        return pop[idx]

    def step_ga():
        pop = state["population"]
        edges = state["edges"]
        m_rate = state["params"]["mutation_rate"]
        method = state["params"]["selection_method"]
        scores = [fitness(ind, edges) for ind in pop]
        new_pop = []
        for _ in range(len(pop)):
            if method == 'roulette':
                p1 = roulette_selection(pop, scores)
                p2 = roulette_selection(pop, scores)
            else:
                p1 = tournament_selection(pop, scores)
                p2 = tournament_selection(pop, scores)
            child = crossover(p1, p2)
            child = mutate(child, m_rate)
            new_pop.append(child)
        state["population"] = new_pop
        state["best_scores"].append(max(scores))
        state["mean_scores"].append(np.mean(scores))
        state["generation"] += 1

    def visualize_graph(G, cover):
        pos = nx.spring_layout(G, seed=42)
        color_map = ['lightgreen' if cover[node] else 'lightgrey' for
node in G.nodes]
        plt.figure(figsize=(6, 5))
        nx.draw_networkx(G, pos, node_color=color_map, node_size=600,
with_labels=True)
        plt.title("Вершинное покрытие")
        plt.axis("off")
        plt.show()

    def plot_fitness():
        best = state["best_scores"]
        mean = state["mean_scores"]
        plt.figure(figsize=(7, 4))
        plt.plot(best, marker='o', label='Best fitness',
color='royalblue', linewidth=2)
        plt.plot(mean, marker='s', label='Average fitness',
color='darkorange', linewidth=2)
        plt.xlabel('Поколение')
        plt.ylabel('Fitness (больше — лучше)')
        plt.title("Динамика приспособленности")
        plt.grid(True, linestyle='--', alpha=0.5)
        plt.legend()
        plt.tight_layout()
        plt.show()
```

```
    def show_min_covers():
        pop = state["population"]
        min_size = min(sum(ind) for ind in pop)
        min_covers = [ind for ind in pop if sum(ind) == min_size]
        print(f"\nМинимальный размер покрытия: {min_size}")
        for cov in min_covers:
            print(cov)

    vertex_input                              =    widgets.IntText(value=10,
description="Вершины:")
    edge_input         = widgets.IntText(value=15, description="Рёбра:")
    pop_input                                 =    widgets.IntText(value=20,
description="Популяция:")
    gen_input                                 =    widgets.IntText(value=30,
description="Поколения:")
    mut_input                                =   widgets.FloatText(value=0.1,
description="Мутация:")
    skip_input            = widgets.IntText(value=5, description="Шагов
пропустить:")
    selection_dropdown = widgets.Dropdown(
        options=[('Турнир', 'tournament'), ('Рулетка', 'roulette')],
        value='tournament',
        description='Селекция:'
    )
    edges_display      = widgets.Textarea(
        value='', description='Список рёбер:',
        layout=widgets.Layout(width='100%', height='100px'),
        disabled=True
    )
    start_btn         = widgets.Button(description="Сгенерировать граф")
    step_btn           = widgets.Button(description="Следующий шаг")
    skip_btn                =  widgets.Button(description="Пропустить  N
поколений")
    output_area        = widgets.Output()
    def initialize(_):
        with output_area:
            clear_output()
            state["params"].update({
                "num_vertices":    vertex_input.value,
                "num_edges":       edge_input.value,
                "pop_size":        pop_input.value,
                "generations":     gen_input.value,
                "mutation_rate":   mut_input.value,
                "selection_method":selection_dropdown.value
            })
            G = generate_graph(vertex_input.value, edge_input.value)
            pop          =          generate_population(pop_input.value,
vertex_input.value)
            state.update({
                "graph":       G,
                "edges":       list(G.edges()),
                "population":  pop,
                "best_scores": [],
                "mean_scores": [],
                "generation":  0
            })
```

```
            edges_display.value = '\n'.join(f"{u} — {v}" for u, v in
state["edges"])
            print("Граф сгенерирован.")

    def _do_steps(count):
        for _ in range(count):
            step_ga()
        best = max(state["population"], key=lambda ind: fitness(ind,
state["edges"]))
        covered = sum(1 for u, v in state["edges"] if best[u] or best[v])
        total   = len(state["edges"])
        size    = sum(best)
        fit_val = fitness(best, state["edges"])
        print(f"Поколение {state['generation']}")
        print(f"Лучшее решение: {best}")
        print(f"Размер покрытия: {size}")
        print(f"Покрыто рёбер: {covered}/{total} ({covered/total:.2%})")
        print(f"Fitness: {fit_val:.4f}")
        visualize_graph(state["graph"], best)
        plot_fitness()

    def next_step(_):
        with output_area:
            clear_output()
            if state["generation"] < state["params"]["generations"]:
                _do_steps(1)
            else:
                print("Все поколения пройдены.")
                show_min_covers()

    def skip_steps(_):
        with output_area:
            clear_output()
            to_run                =                min(skip_input.value,
state["params"]["generations"] - state["generation"])
            if to_run <= 0:
                print("Все поколения пройдены.")
                show_min_covers()
            else:
                _do_steps(to_run)
                if                 state["generation"]                ==
state["params"]["generations"]:
                    show_min_covers()
    start_btn.on_click(initialize)
    step_btn.on_click(next_step)
    skip_btn.on_click(skip_steps)

    display(widgets.VBox([
        vertex_input, edge_input, pop_input, gen_input, mut_input,
        selection_dropdown,
        widgets.HBox([skip_input, skip_btn]),
        widgets.HBox([start_btn, step_btn]),
        edges_display,
        output_area
    ]))
```