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

ОТЧЕТ

по практической работе №1

по дисциплине Учебная практика

по теме: Генетический алгоритм поиска МОД.

Емельянчик А.А.

Студенты гр. 2304 _____ Федоров Е. Жилин Д.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы.

Решение задачи нахождения минимального остовного дерева путем использования генетического алгоритма.

Задание.

Задача поиска МОД

Дан взвешенный неориентированный граф (ребра имеют вес больше 0). Необходимо найти минимальное остовное дерево для данного графа.

План решения задачи.

Генотип:
Генотип будет представлять собой множество ребер, образующих остовное дерево графа.
Каждый индивид в популяции будет представлен таким множеством ребер.

Первая популяция:
Изначальная популяция создается путем создания деревьев при помощи модифицированного алгоритма Прима. Его суть в том, что на каждом шаге вместо наименьшего ребра выбирается случайное.

Скрещивание:
Для скрещивания создается подграф путем соединения двух деревьев, затем при помощи рандомизированного алгоритма Прима находится случайное остовное дерево в данном подграфе.

Мутации:
В дереве удаляется случайное ребро, образуя две компоненты связности. Затем ищется другое ребро, способное объединить эти компоненты связности.

Приспособленность:
Приспособленность особи (остовного дерева) определяется на основе веса (суммы весов ребер) этого дерева.

Отбор родителей:
При помощи ранжированного отбора находится набор родителей.

Реализация алгоритма.

Первая популяция:
Функция self.random_prim() реализует алгоритм Прима для построения случайного остовного дерева:
Если передан аргумент subgraph_set, то на его основе создается подграф subgraph. Иначе, используется весь граф self.graph.
Инициализируется пустое множество tree_set, которое будет хранить ребра построенного остовного дерева.
Выбирается случайная начальная вершина curr из графа.
Множество connected хранит вершины, уже включенные в остовное дерево.
Список curr_edges хранит все ребра, инцидентные текущей вершине curr.
Пока количество вершин в connected не равно общему количеству вершин в графе:
Выбирается случайное ребро edge из curr_edges.
Если второй конец ребра edge[1] уже включен в connected, то ребро пропускается.
Иначе, вершина edge[1] добавляется в connected, ребро edge добавляется в tree_set.
Для новой вершины curr добавляются все ее инцидентные ребра в curr_edges.
Функция возвращает множество tree_set, содержащее ребра построенного остовного дерева.

Скрещивание:

Функция crossover реализует процесс скрещивания двух родительских особей.
Она объединяет множества ребер двух родителей в одно множество subgraph, а затем вызывает функцию random_prim() для построения нового остовного дерева на основе этого подграфа.
Таким образом, новое потомство наследует признаки от обоих родителей.

Мутация:

Функция mutate реализует процесс мутации особи.
Она принимает множество ребер, образующих остовное дерево, и выполняет следующие действия:
Создает копию подграфа, соответствующего входному множеству ребер.
Удаляет одно случайное ребро из этого подграфа.
Находит две компоненты связности, образовавшиеся после удаления ребра.
Перебирает все ребра графа в случайном порядке и добавляет первое ребро, которое соединяет две компоненты.
Возвращает новое множество ребер, представляющее модифицированное остовное дерево.
Таким образом, мутация заключается в замене одного ребра в остовном дереве на другое, которое восстанавливает связность дерева.

Алгоритм:

Функция algorithm() реализует основной цикл генетического алгоритма.
Она создает начальную популяцию из 100 случайных остовных деревьев, используя функцию random_prim().
Затем в цикле выполняет следующие шаги:
Сортирует популяцию по весу остовных деревьев.
Вычисляет модифицированную функцию приспособленности, основанную на весе деревьев.
Выбирает 50 родителей с вероятностью, пропорциональной их модифицированной приспособленности.
Создает 100 новых особей путем скрещивания родителей и применения мутации.
Заменяет текущую популяцию на новую.

Алгоритм продолжается до тех пор, пока не будет найдено остовное дерево с минимальным весом (или достигнуто максимальное количество итераций).

Прототип GUI.

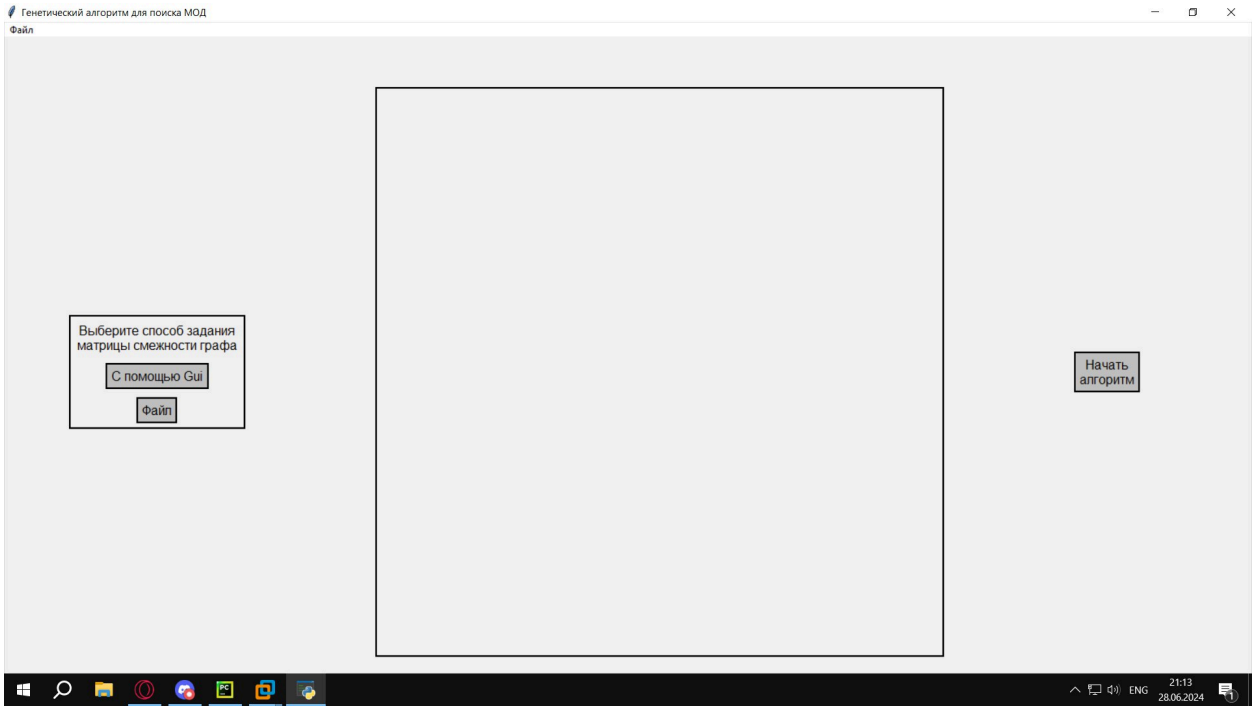

Рис. 1 - Стартовое окно

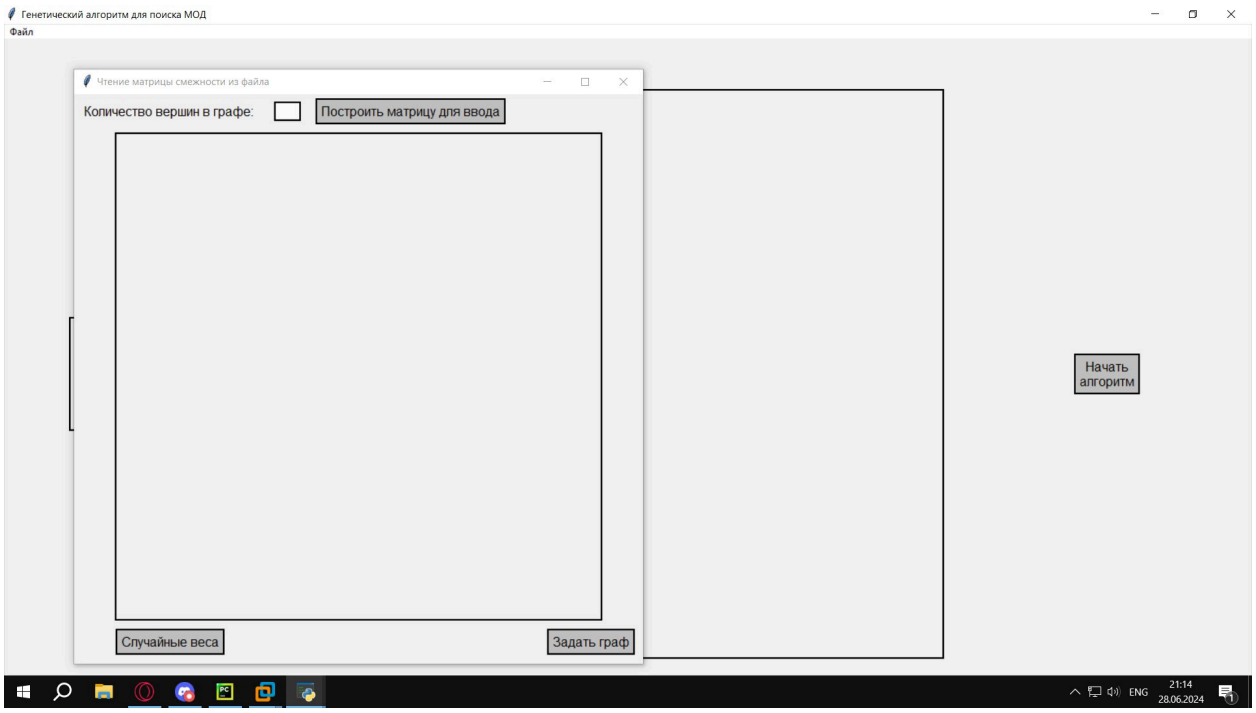

Рис. 2 - Окно ввода размера графа

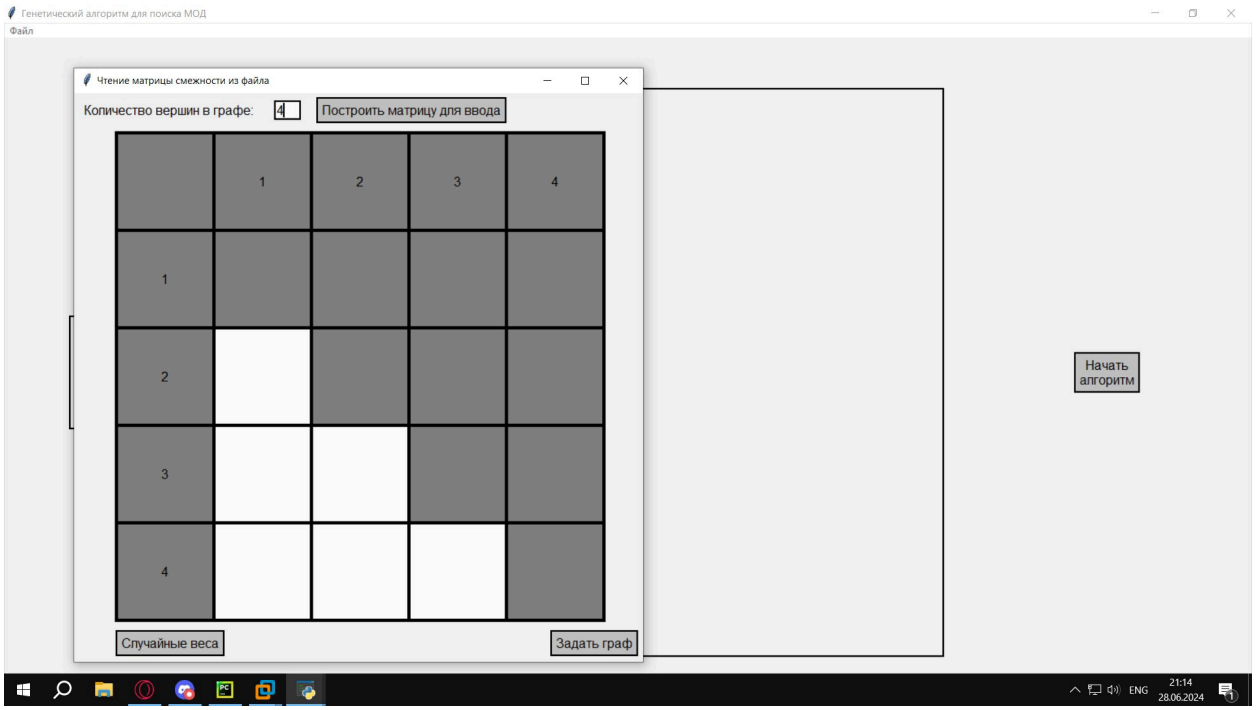

Рис. 3 - Ввод матрицы смежности

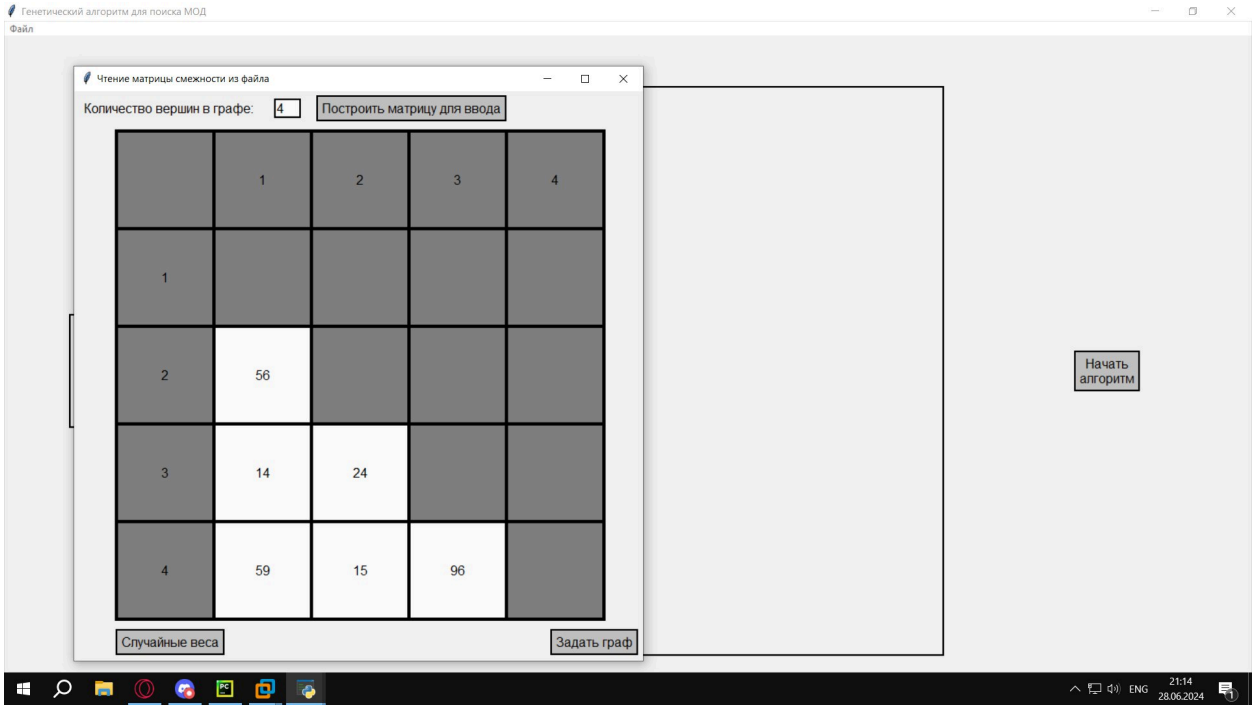

Рис. 4 - Введенная матрица смежности

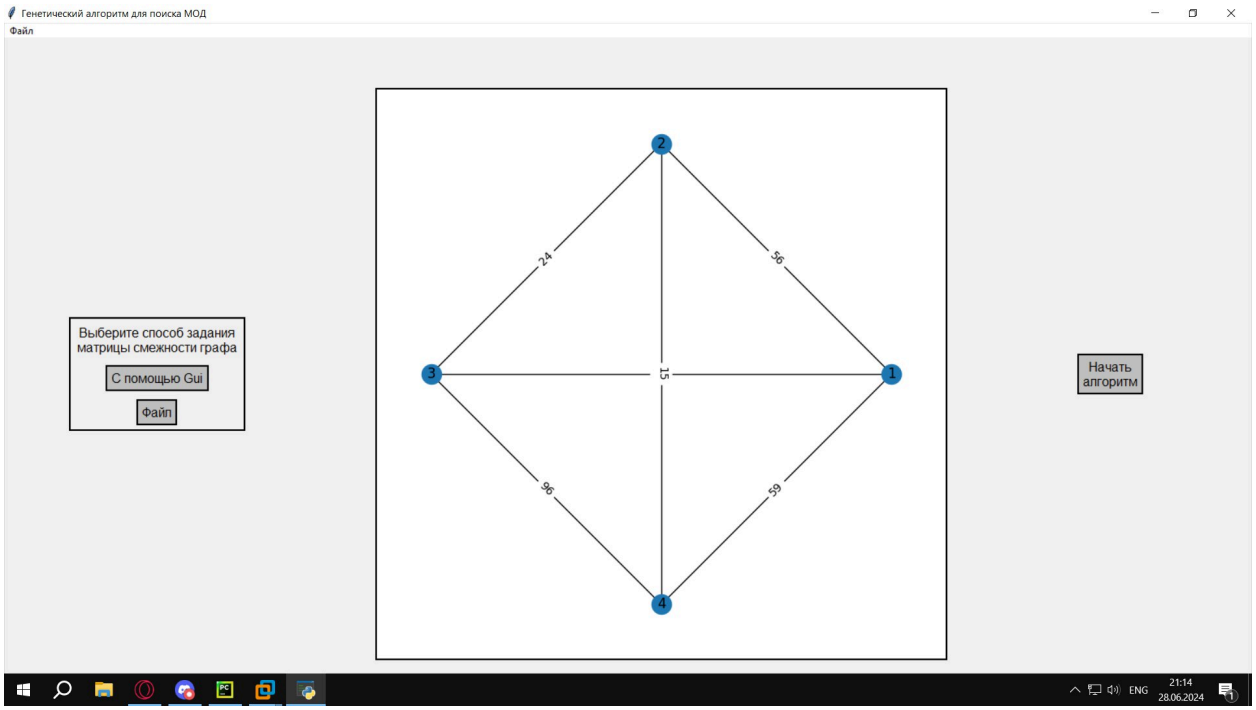

Рис. 5 - Изображение исходного графа

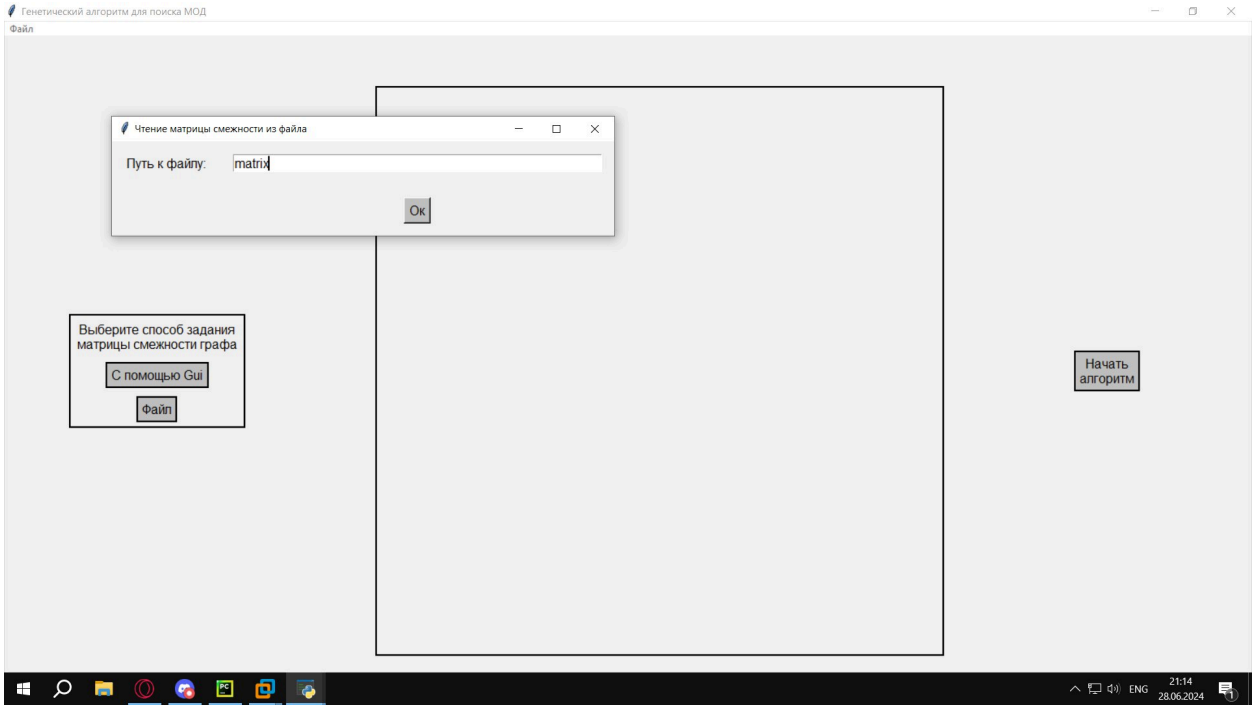

Рис. 6 - Окно ввода пути к файлу

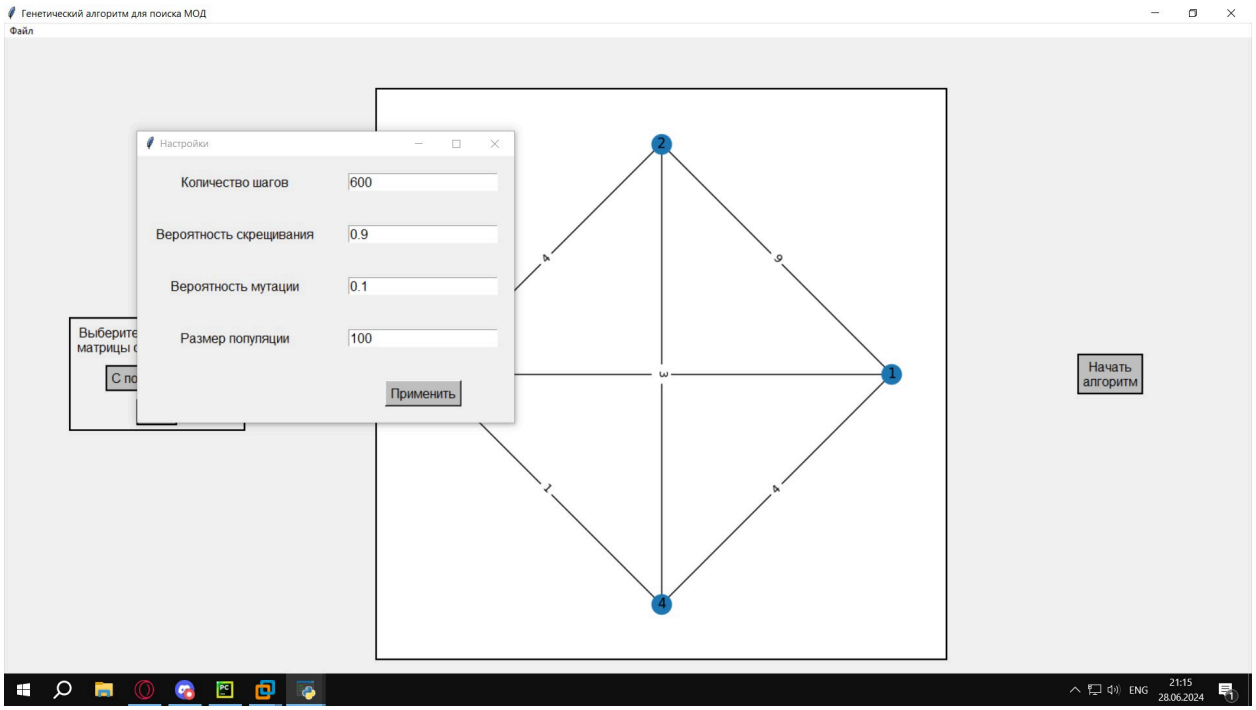

Рис. 7 - Окно настроек

Вывод.

Было спланировано решение задачи, частично были реализованы генетический алгоритм поиска МОД и GUI.

Итерация 2.

Обновление кода.

Был связан алгоритм с графическим интерфейсом. Для отображения результатов работы алгоритма был добавлен класс VisualisatorGA.

Создан новый класс, отвечающий за визуализацию результатов работы алгоритма. В конструкторе класса вызывается метод *algorithm()*, а сам обьект этого класса вызывается при нажатии на кнопку “Начать алгоритм” в графическом интерфейсе.

Было добавлено сохранение трех лучших результатов из каждого поколения.

Лучшие решения поколения: В каждом из count_gener поколений сохраняются лучшие три решения (остовные деревья) вместе с их весами:

```
best_solutions[i] = generation[:3]
best_weights[i] = tuple(map(self.tree_weight,
best_solutions[i]))
```

Здесь generation сортируется по весу деревьев с помощью функции tree_weight. Таким образом, best_solutions[i] содержит три решения с минимальными весами в текущем поколении. Лучише решения показываются при помощи *show_best_solution()*

Был изменен класс Solver таким образом, чтобы настройки, выбранные в GUI применялись к алгоритму.

В класс Solver были добавлены поля, соответствующие размеру популяции, количеству поколений, вероятности мутации и вероятности скрещивания, их использует метод *algorithm()*, который теперь возвращает лучшие решения и их веса.

Для графика используются значения лучших особей популяции.

Построение графика использует те же лучшие решения, что и их визуализация. График строится при помощи функции *show_graphic()*

Графический интерфейс.

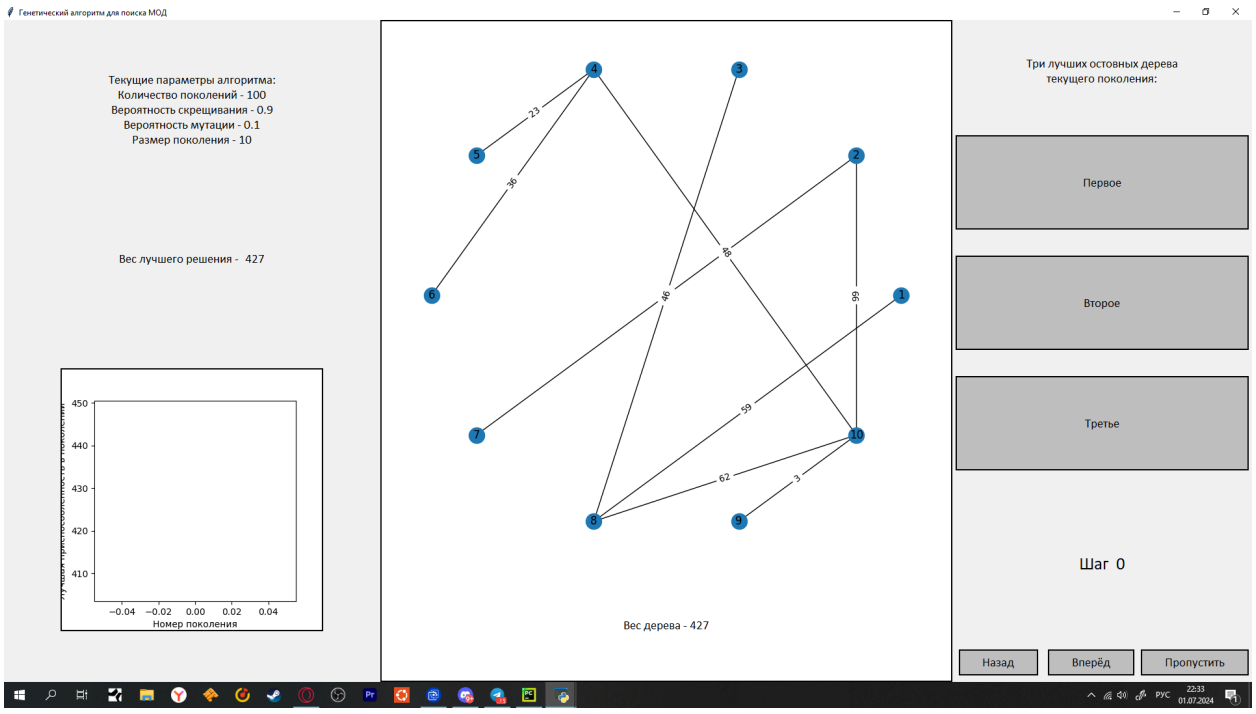

Рис.8 - Визуализация алгоритма.

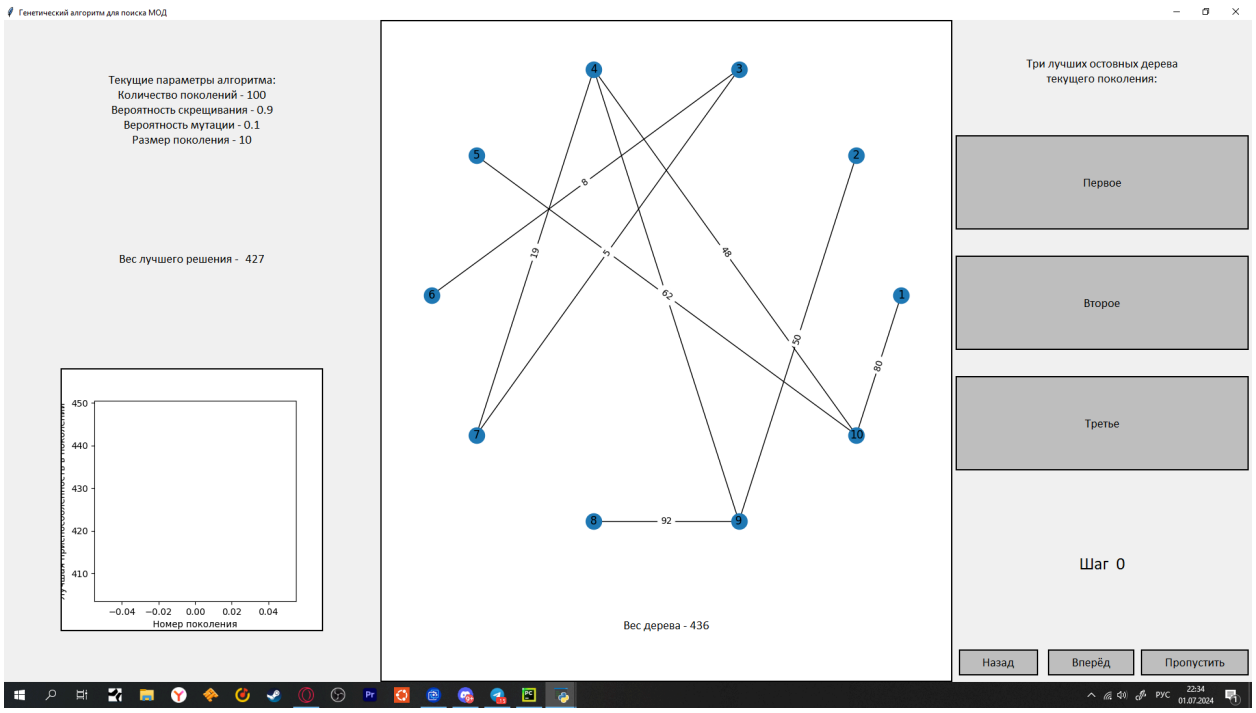

Рис. 9 - Третье лучшее решение нулевого шага.

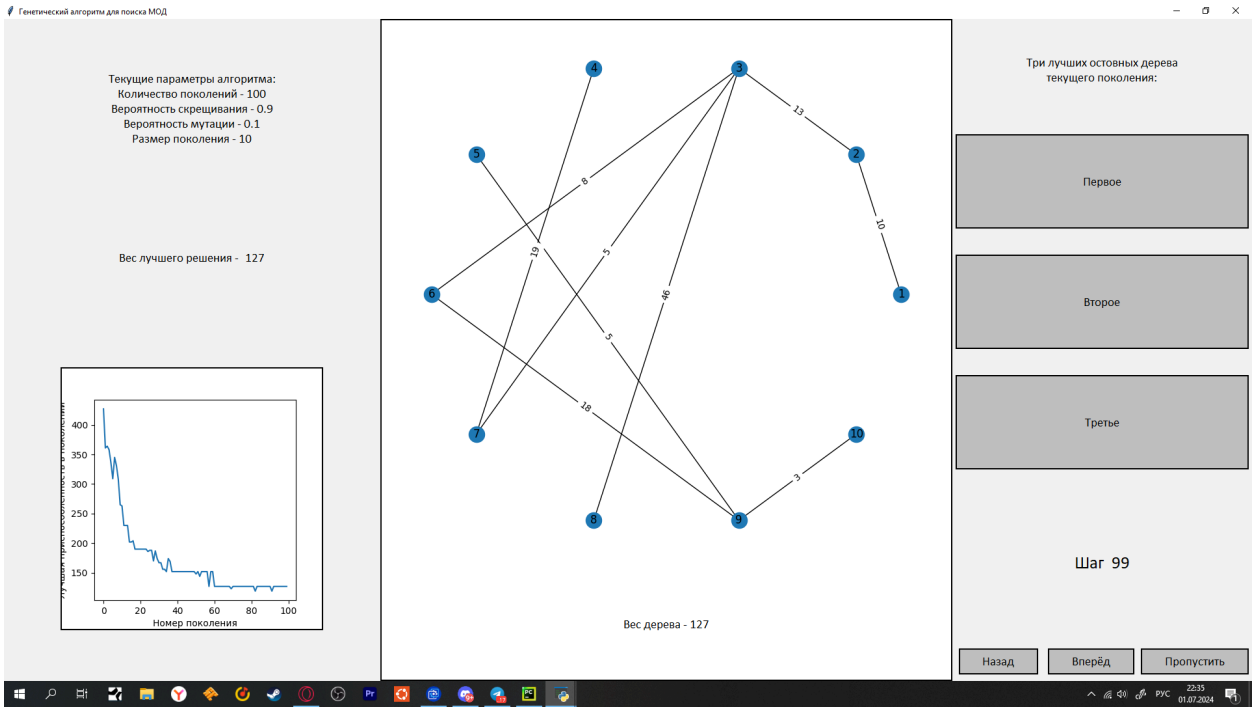

Рис. 10 - Конечный результат работы алгоритма.