МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №5

по дисциплине: Основы искусственного интеллекта
тема: «Решение оптимизационных задач эволюционно-генетическими алгоритмами»

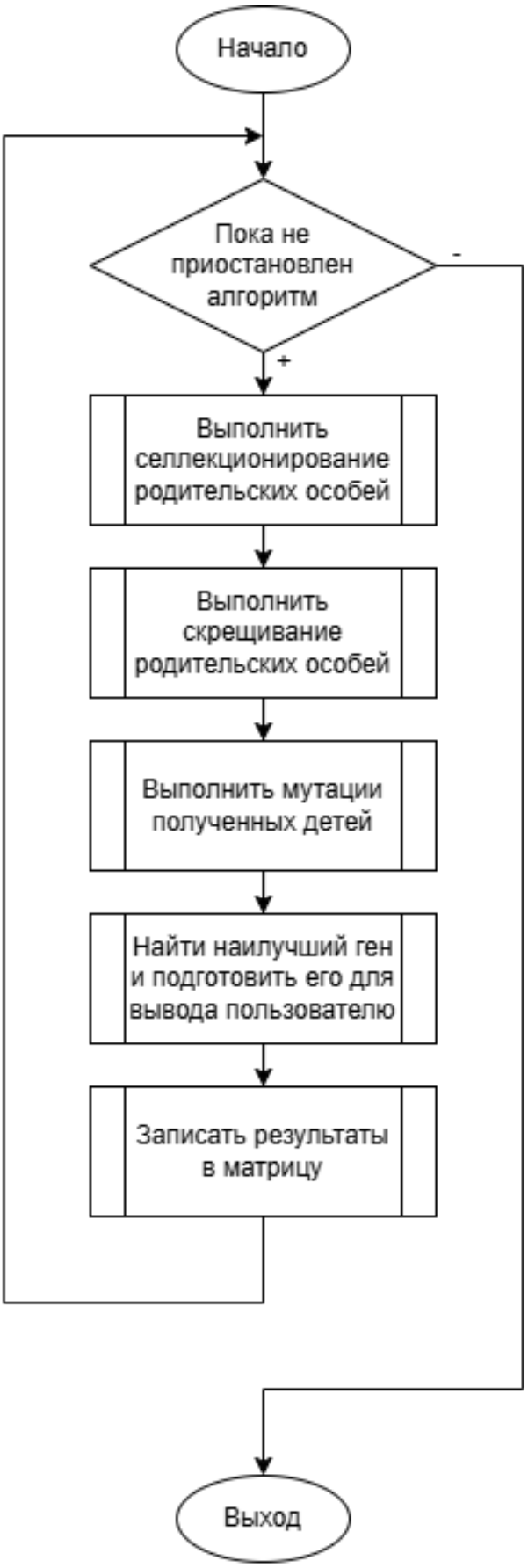
Выполнил: ст. группы ПВ-223
Игнатьев Артур Олегович

Проверили:
пр. Твердохлеб Виталий Викторович

Белгород 2025 г.

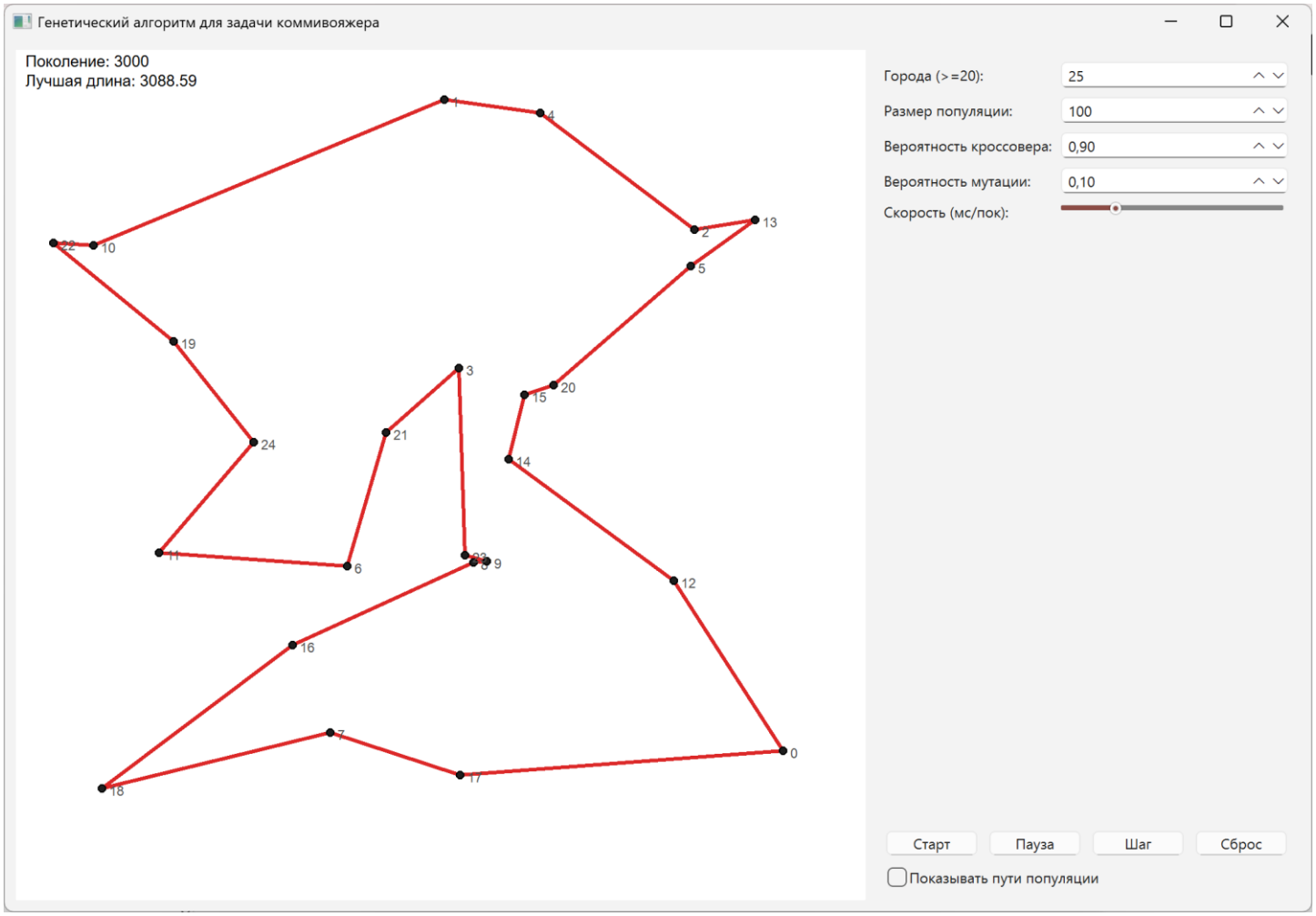
Цель работы: изучение особенностей генетических алгоритмов необходимых при решении оптимизационных задач.

Ход работы



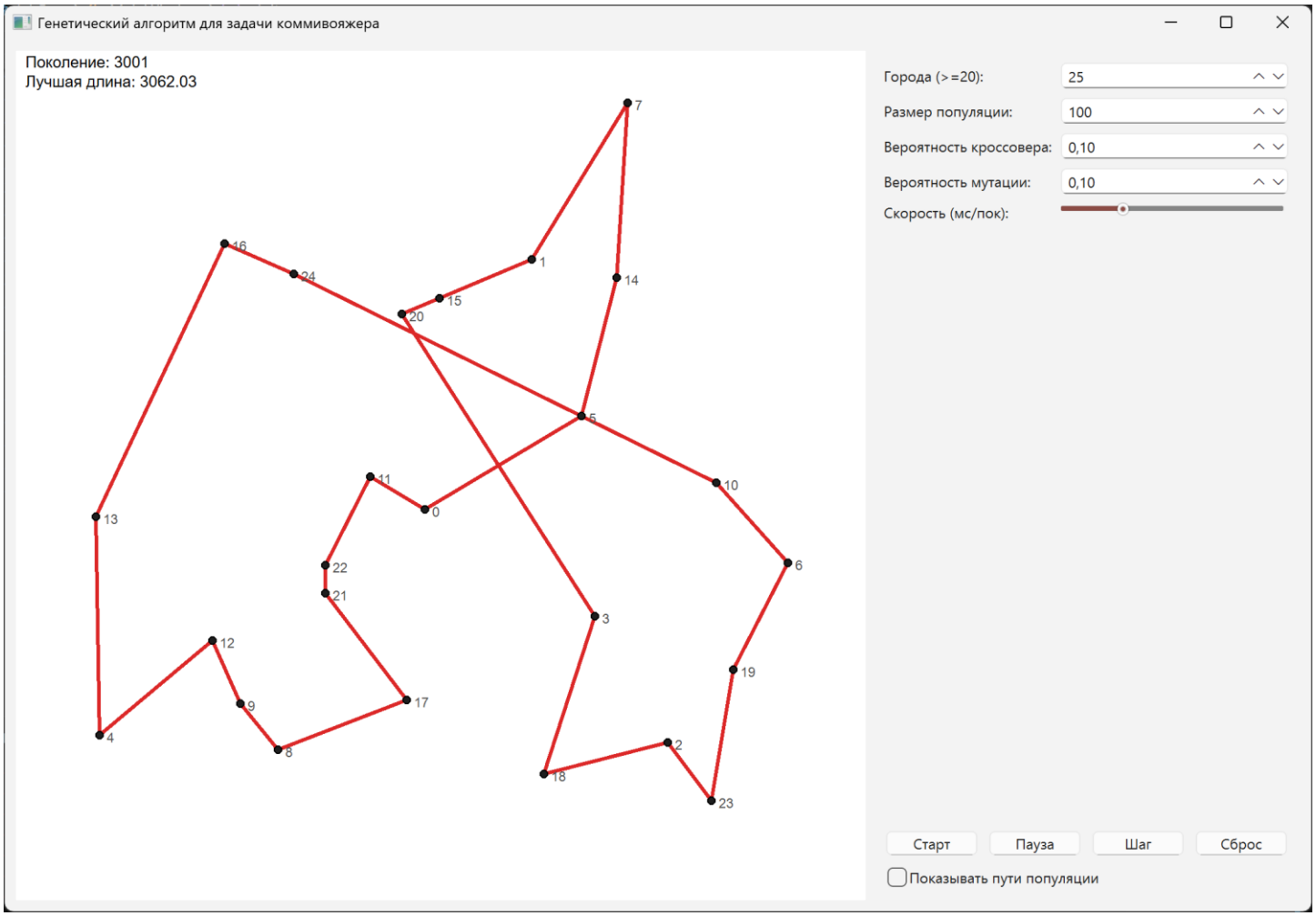
Демонстрация работы программы

Тест №1: обычный сценарий



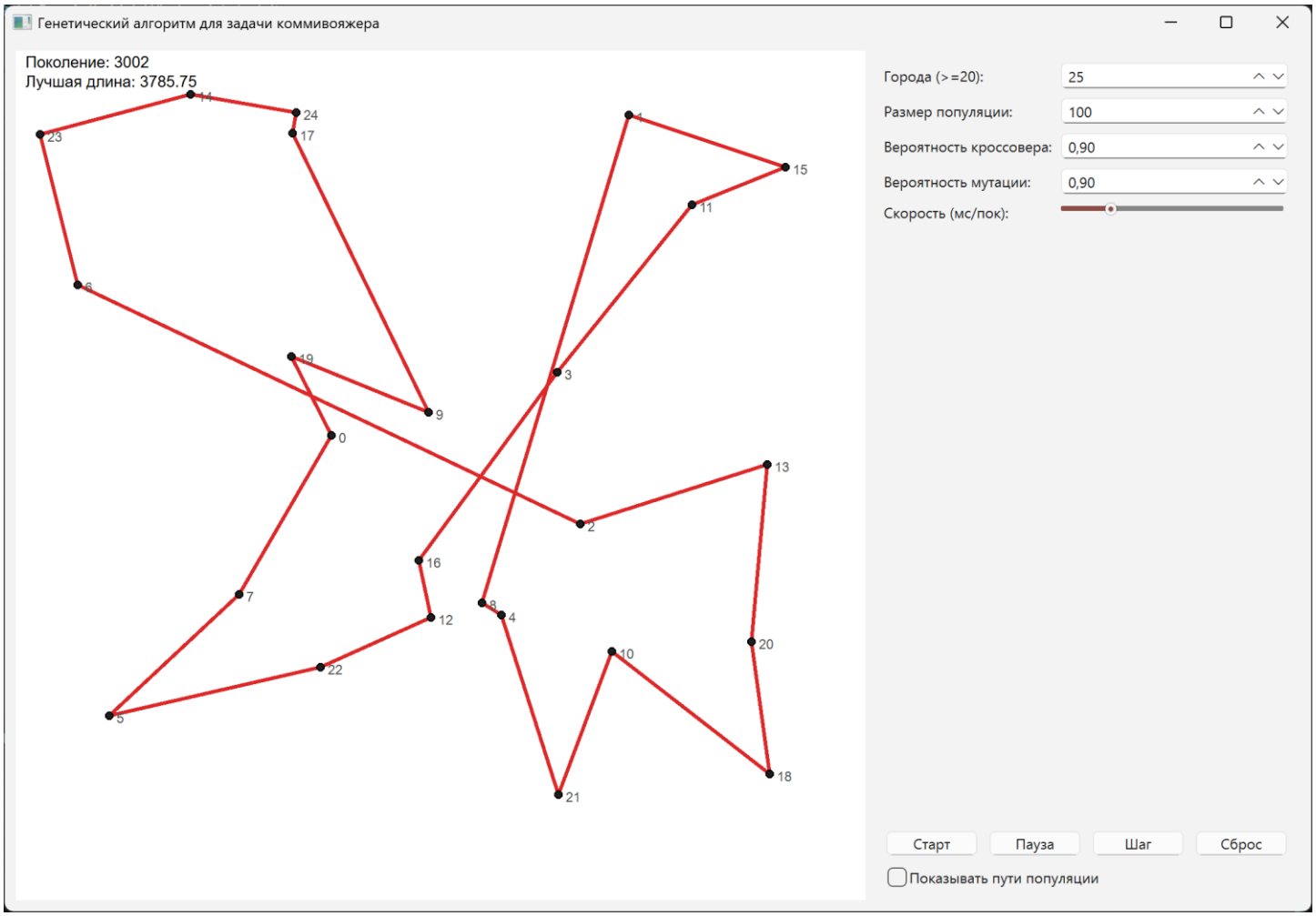
Алгоритм быстро находит оптимальное или близкое решение, доходит до локального экстремума.

Тест №2: низкое разнообразие



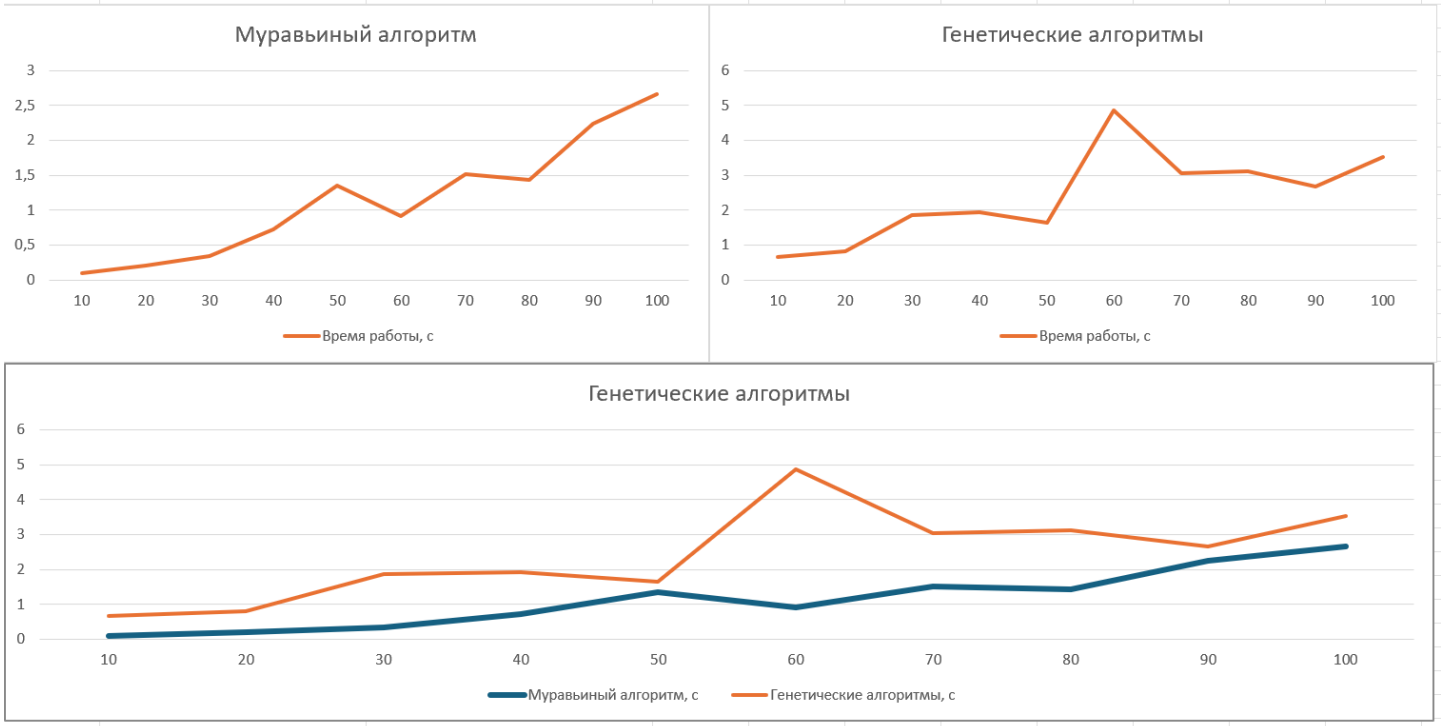
Медленный поиск, решения далеки от оптимальных из-за малого разнообразия генов.

Тест №3: слишком много мутаций



Алгоритм не сходится, работает как случайный перебор, решения нестабильны.

Тест №4: сравнение времени работы с муравьиным алгоритмом



Муравьиный алгоритм (МА) работает быстрее генетического (ГА), потому что он быстрее находит хорошее решение. В МА муравьи оставляют феромоны на удачных путях, как следы, которые помогают другим муравьям быстро сосредоточиться на лучших маршрутах. Из-за этого алгоритм сходится всего за 18–145 итераций, даже с 100 муравьями. А в ГА популяция эволюционирует через мутации и кроссовер, что требует много времени — 100–2000+ поколений, чтобы улучшить решения шаг за шагом.

Кроме того, каждый шаг МА проще в вычислениях: муравьи выбирают города по вероятностям, и обновление феромонов занимает меньше ресурсов, даже если городов много. В ГА же на каждом поколении нужно проверять приспособленность (длину пути) для всей популяции, что медленнее. В итоге МА тратит 0.08–1.52 секунды, а ГА — 0.40–1.35, хотя оба находят похожие пути. Для больших задач МА выгоднее, но ГА проще настроить.

Вывод: в ходе лабораторной работы изучили особенности генетических алгоритмов необходимых при решении оптимизационных задач.

Код программы:

```
import sys
import math
import random
import time
from dataclasses import dataclass, field
from typing import List, Tuple
import numpy as np
from PyQt6 import QtCore, QtGui, QtWidgets

# ---------- Параметры по умолчанию ----------
DEFAULT_NUM_CITIES = 25  # Не менее 20
DEFAULT_POP_SIZE = 100
DEFAULT_GENERATIONS = 10000
DEFAULT_CROSSOVER_PROB = 0.8
DEFAULT_MUTATION_PROB = 0.05
DEFAULT_MAX_COORD = 700 - 40  # Для canvas

# Размеры GUI
WINDOW_W = 1000
WINDOW_H = 700
CANVAS_W = 700
CANVAS_H = 700
MARGIN = 20

# ---------- Структуры данных ----------
@dataclass
class City:
    x: float
    y: float

@dataclass
class Chromosome:
    path: List[int] = field(default_factory=list)
    fitness: float = float('inf')  # Длина тура, минимизировать

# ---------- Реализация генетического алгоритма ----------
class GeneticTSP:
    def __init__(self, num_cities=DEFAULT_NUM_CITIES, pop_size=DEFAULT_POP_SIZE,
                 crossover_prob=DEFAULT_CROSSOVER_PROB, mutation_prob=DEFAULT_MUTATION_PROB,
                 max_coord=DEFAULT_MAX_COORD):
        self.num_cities = num_cities
        self.pop_size = pop_size
        self.crossover_prob = crossover_prob
        self.mutation_prob = mutation_prob
        self.max_coord = max_coord
        self.cities: List[City] = []
        self.distance = np.zeros((num_cities, num_cities), dtype=float)
        self.population: List[Chromosome] = []
        self.best_fitness = float('inf')
        self.best_path: List[int] = []
        self.generation = 0
        self._init_cities()
        self._init_population()

    def _init_cities(self):
        self.cities = []
        for i in range(self.num_cities):
            x = random.uniform(MARGIN, self.max_coord - MARGIN)
            y = random.uniform(MARGIN, self.max_coord - MARGIN)
            self.cities.append(City(x, y))
        # Вычисляем расстояния
        for i in range(self.num_cities):
            for j in range(self.num_cities):
```

```
                if i != j:
                    dx = self.cities[i].x - self.cities[j].x
                    dy = self.cities[i].y - self.cities[j].y
                    self.distance[i, j] = math.hypot(dx, dy)

    def _init_population(self):
        self.population = []
        for _ in range(self.pop_size):
            path = list(range(self.num_cities))
            random.shuffle(path)
            chrom = Chromosome(path=path)
            chrom.fitness = self._compute_fitness(path)
            self.population.append(chrom)
        self._update_best()
        self.generation = 0

    def _compute_fitness(self, path: List[int]) -> float:
        length = 0.0
        for i in range(len(path) - 1):
            length += self.distance[path[i], path[i+1]]
        # Замыкаем цикл
        length += self.distance[path[-1], path[0]]
        return length

    def _update_best(self):
        for chrom in self.population:
            if chrom.fitness < self.best_fitness:
                self.best_fitness = chrom.fitness
                self.best_path = chrom.path.copy()

    # Отбор: рулетка на основе обратной приспособленности
    def _select_parent(self) -> Chromosome:
        total_inverse = sum(1.0 / chrom.fitness for chrom in self.population)
        r = random.uniform(0, total_inverse)
        cumulative = 0.0
        for chrom in self.population:
            cumulative += 1.0 / chrom.fitness
            if cumulative >= r:
                return chrom
        return self.population[-1]  # Fallback

    # Скрещивание: Order Crossover (OX)
    def _order_crossover(self, parent1: List[int], parent2: List[int]) -> Tuple[List[int],
List[int]]:
        size = len(parent1)
        start, end = sorted(random.sample(range(size), 2))
        child1 = [-1] * size
        child2 = [-1] * size
        # Копируем сегмент
        child1[start:end+1] = parent1[start:end+1]
        child2[start:end+1] = parent2[start:end+1]
        # Заполняем оставшиеся элементы из другого родителя
        def fill_child(child, parent, seg_start, seg_end):
            p_idx = (seg_end + 1) % size
            c_idx = (seg_end + 1) % size
            while c_idx != seg_start:
                if parent[p_idx] not in child[seg_start:seg_end+1]:
                    child[c_idx] = parent[p_idx]
                    c_idx = (c_idx + 1) % size
                p_idx = (p_idx + 1) % size
            # Заполняем оставшиеся элементы (не должно происходить)
            missing = [i for i in range(size) if i not in child]
            for idx in range(size):
                if child[idx] == -1:
                    child[idx] = missing.pop(0)
        fill_child(child1, parent2, start, end)
        fill_child(child2, parent1, start, end)
```

```
        return child1, child2

    # Мутация: поменять местами два города
    def _mutate(self, path: List[int]):
        if random.random() < self.mutation_prob:
            i, j = random.sample(range(self.num_cities), 2)
            path[i], path[j] = path[j], path[i]

    # Один шаг эволюции
    def evolve(self):
        new_population = []
        # Элитизм: сохраняем лучшего
        best_chrom = min(self.population, key=lambda c: c.fitness)
        new_population.append(Chromosome(path=best_chrom.path.copy(),
fitness=best_chrom.fitness))
        while len(new_population) < self.pop_size:
            parent1 = self._select_parent()
            parent2 = self._select_parent()
            if random.random() < self.crossover_prob:
                child1_path, child2_path = self._order_crossover(parent1.path, parent2.path)
            else:
                child1_path = parent1.path.copy()
                child2_path = parent2.path.copy()
            self._mutate(child1_path)
            self._mutate(child2_path)
            child1 = Chromosome(path=child1_path, fitness=self._compute_fitness(child1_path))
            child2 = Chromosome(path=child2_path, fitness=self._compute_fitness(child2_path))
            new_population.append(child1)
            if len(new_population) < self.pop_size:
                new_population.append(child2)
        self.population = new_population[:self.pop_size]
        self._update_best()
        self.generation += 1

# ---------- PyQt GUI ----------
class TSPWidget(QtWidgets.QWidget):
    def __init__(self, ga: GeneticTSP, parent=None):
        super().__init__(parent)
        self.ga = ga
        self.setMinimumSize(CANVAS_W, CANVAS_H)
        self.setMaximumSize(CANVAS_W, CANVAS_H)
        self.setSizePolicy(QtWidgets.QSizePolicy.Policy.Fixed,
QtWidgets.QSizePolicy.Policy.Fixed)
        self.show_population = False

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.fillRect(self.rect(), QtGui.QColor('white'))
        if self.show_population:
            for chrom in self.ga.population:
                path = chrom.path
                pen = QtGui.QPen(QtGui.QColor(150, 150, 255, 80), 1)
                painter.setPen(pen)
                for i in range(len(path) - 1):
                    a = self.ga.cities[path[i]]
                    b = self.ga.cities[path[i + 1]]
                    painter.drawLine(int(a.x), int(a.y), int(b.x), int(b.y))
                # Закрываем цикл
                a = self.ga.cities[path[-1]]
                b = self.ga.cities[path[0]]
                painter.drawLine(int(a.x), int(a.y), int(b.x), int(b.y))
        # Рисуем лучший путь
        if self.ga.best_path:
            pen = QtGui.QPen(QtGui.QColor(220, 40, 40), 3)
            painter.setPen(pen)
            bp = self.ga.best_path
            for i in range(len(bp) - 1):
```

```
                a = self.ga.cities[bp[i]]
                b = self.ga.cities[bp[i + 1]]
                painter.drawLine(int(a.x), int(a.y), int(b.x), int(b.y))
            # Закрываем цикл
            a = self.ga.cities[bp[-1]]
            b = self.ga.cities[bp[0]]
            painter.drawLine(int(a.x), int(a.y), int(b.x), int(b.y))
        # Рисуем города
        for i, c in enumerate(self.ga.cities):
            r = 6
            painter.setBrush(QtGui.QBrush(QtGui.QColor(30, 30, 30)))
            painter.setPen(QtGui.QPen(QtGui.QColor(0, 0, 0), 1))
            painter.drawEllipse(int(c.x - r/2), int(c.y - r/2), r, r)
            # Рисуем индекс
            painter.setPen(QtGui.QPen(QtGui.QColor(80, 80, 80), 1))
            painter.setFont(QtGui.QFont("Arial", 8))
            painter.drawText(int(c.x + 6), int(c.y + 6), str(i))
        # Рисуем информационный текст
        painter.setPen(QtGui.QPen(QtGui.QColor(0, 0, 0), 1))
        painter.setFont(QtGui.QFont("Arial", 10))
        painter.drawText(8, 14, f"Поколение: {self.ga.generation}")
        painter.drawText(8, 30, f"Лучшая длина: {self.ga.best_fitness:.2f}")

class ControlPanel(QtWidgets.QWidget):
    def __init__(self, tsp_widget: TSPWidget, ga: GeneticTSP, parent=None):
        super().__init__(parent)
        self.tsp_widget = tsp_widget
        self.ga = ga
        # Таймер для эволюции
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.on_timer)
        self.timer_interval = 100  # ms
        self.timer.setInterval(self.timer_interval)
        # Элементы пользовательского интерфейса
        layout = QtWidgets.QVBoxLayout()
        self.setLayout(layout)
        form = QtWidgets.QFormLayout()
        layout.addLayout(form)
        # Параметры
        self.cities_spin = QtWidgets.QSpinBox()
        self.cities_spin.setRange(20, 100)
        self.cities_spin.setValue(self.ga.num_cities)
        self.pop_spin = QtWidgets.QSpinBox()
        self.pop_spin.setRange(10, 500)
        self.pop_spin.setValue(self.ga.pop_size)
        self.cross_spin = QtWidgets.QDoubleSpinBox()
        self.cross_spin.setRange(0.0, 1.0)
        self.cross_spin.setValue(self.ga.crossover_prob)
        self.cross_spin.setSingleStep(0.05)
        self.mut_spin = QtWidgets.QDoubleSpinBox()
        self.mut_spin.setRange(0.0, 1.0)
        self.mut_spin.setValue(self.ga.mutation_prob)
        self.mut_spin.setSingleStep(0.01)
        self.speed_slider = QtWidgets.QSlider(QtCore.Qt.Orientation.Horizontal)
        self.speed_slider.setRange(10, 1000)
        self.speed_slider.setValue(self.timer_interval)
        form.addRow("Города (>=20):", self.cities_spin)
        form.addRow("Размер популяции:", self.pop_spin)
        form.addRow("Вероятность кроссовера:", self.cross_spin)
        form.addRow("Вероятность мутации:", self.mut_spin)
        form.addRow("Скорость (мс/пок):", self.speed_slider)
        # Кнопки
        row = QtWidgets.QHBoxLayout()
        self.start_btn = QtWidgets.QPushButton("Старт")
        self.pause_btn = QtWidgets.QPushButton("Пауза")
        self.step_btn = QtWidgets.QPushButton("Шаг")
        self.reset_btn = QtWidgets.QPushButton("Сброс")
```

```
        row.addWidget(self.start_btn)
        row.addWidget(self.pause_btn)
        row.addWidget(self.step_btn)
        row.addWidget(self.reset_btn)
        layout.addLayout(row)
        # Checkboxes
        self.show_pop = QtWidgets.QCheckBox("Показывать пути популяции")
        layout.addWidget(self.show_pop)
        # Сигналы
        self.start_btn.clicked.connect(self.start)
        self.pause_btn.clicked.connect(self.pause)
        self.step_btn.clicked.connect(self.step)
        self.reset_btn.clicked.connect(self.reset)
        self.speed_slider.valueChanged.connect(self.change_speed)
        self.show_pop.stateChanged.connect(self.toggle_population)

    def change_speed(self, val):
        self.timer_interval = val
        self.timer.setInterval(self.timer_interval)

    def toggle_population(self, state):
        self.tsp_widget.show_population = (state == QtCore.Qt.CheckState.Checked)
        self.tsp_widget.update()

    def start(self):
        # Применяем параметры
        if int(self.cities_spin.value()) != self.ga.num_cities or int(self.pop_spin.value()) !=
self.ga.pop_size:
            self.ga.num_cities = int(self.cities_spin.value())
            self.ga.pop_size = int(self.pop_spin.value())
            self.ga._init_cities()
            self.ga._init_population()
            self.tsp_widget.repaint()  # Немедленная перерисовка после изменения параметров
        self.ga.crossover_prob = float(self.cross_spin.value())
        self.ga.mutation_prob = float(self.mut_spin.value())
        self.timer.start()

    def pause(self):
        self.timer.stop()

    def reset(self):
        self.timer.stop()
        self.ga._init_cities()
        self.ga._init_population()
        self.tsp_widget.repaint()  # Используем перекраску для немедленного обновления

    def step(self):
        self.ga.evolve()
        self.tsp_widget.update()

    def on_timer(self):
        self.ga.evolve()
        self.tsp_widget.update()

class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Генетический алгоритм для задачи коммивояжера")
        self.resize(WINDOW_W, WINDOW_H)
        self.ga = GeneticTSP(num_cities=DEFAULT_NUM_CITIES, pop_size=DEFAULT_POP_SIZE,
                             crossover_prob=DEFAULT_CROSSOVER_PROB,
mutation_prob=DEFAULT_MUTATION_PROB)
        self.tsp_widget = TSPWidget(self.ga)
        self.control_panel = ControlPanel(self.tsp_widget, self.ga)
        central = QtWidgets.QWidget()
        layout = QtWidgets.QHBoxLayout()
        central.setLayout(layout)
```

```
        layout.addWidget(self.tsp_widget)
        layout.addWidget(self.control_panel)
        self.setCentralWidget(central)

def main():
    app = QtWidgets.QApplication(sys.argv)
    win = MainWindow()
    win.show()
    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```