Санкт-Петербургский государственный университет

Кафедра компьютерного моделирования и многопоточных систем

Лабораторная работа по дисциплине
Алгоритмы и структуры данных

”Исследование генетического алгоритма. Изучение различных кодировок генотипа.”

Выполнил:
Зайнуллин Мансур Альбертович
Группа: 23.Б16-пу

Руководитель:
Дик Александр Геннадьевич
ассистент кафедры компьютерного моделирования
и многопоточных систем

Санкт-Петербург
2024

Оглавление

1 Цель работы

Цель данной работы заключается в исследовании и сравнении двух основных способов кодирования генотипа хромосом в генетическом алгоритме: бинарного и вещественного. Исследование направлено на оценку их эффективности по критериям скорости сходимости, точности решения и устойчивости к локальным минимумам, что позволит определить наиболее подходящий метод для решения задач оптимизации.

2 Описание алгоритма ГА

Генетический алгоритм (ГА) — это метод оптимизации, основанный на принципах естественного отбора и генетики. Он используется для решения сложных задач, где традиционные методы могут быть неэффективны. Основные этапы работы ГА включают:

1. **Инициализация:**
    - Создание начальной популяции случайных решений, представленных в виде генотипов. Это начальное множество потенциальных решений, известных как ”особи”.
2. **Оценка приспособленности:**
    - Оценка качества каждого решения с помощью функции приспособленности. Чем лучше решение, тем выше его приспособленность.
3. **Селекция:**
    - Выбор лучших особей для создания следующего поколения. Используются методы, такие как турнирная селекция, чтобы особи с более высокой приспособленностью имели больший шанс быть выбранными.
4. **Кроссовер (Скрещивание):**
    - Процесс обмена генетической информацией между двумя родителями для создания потомков. Это эмулирует биологический процесс рекомбинации ДНК.
    - **Типы кроссовера:**
        - **Одноточечный:** Выбор одной точки разрыва для обмена генами.
        - **Двухточечный:** Выбор двух точек разрыва для обмена сегментами генов.

- **Случайный:** Случайный выбор генов от каждого родителя.
- **Арифметический:** Для вещественного кодирования, где каждый ген потомка — это среднее арифметическое генов родителей.

5. **Мутация:**

   - Внесение случайных изменений в генотипы для поддержания генетического разнообразия. Это позволяет избежать застревания в локальных оптимумах.

6. **Эволюция:**

   - Алгоритм повторяет процесс до достижения оптимального решения. Условия остановки могут включать достижение заданного уровня приспособленности или максимальное число поколений.

2.1 Модификации кроссовера

Кроссовер является ключевым этапом генетического алгоритма, который обеспечивает передачу и комбинирование генетической информации между родителями. Для повышения эффективности алгоритма применяются различные модификации кроссовера.

2.1.1 Одноточечный кроссовер

В одноточечном кроссовере выбирается одна точка разрыва, после которой гены родителей обмениваются между собой. Пример работы одноточечного кроссовера:

- Родители: 11001001 (1-й предок) 01010010 (2-й предок)
- Потомки: 11010010 (1-й потомок) 01001001 (2-й потомок)

В результате скрещивания особей с генотипами 201 и 82 получены особи с генотипами 210 и 73.

2.1.2 Двухточечный кроссовер

В двухточечном кроссовере выбираются две точки разрыва, между которыми происходит обмен генами между родителями. Пример работы двухточечного кроссовера:

- Родители: 11001001 (1-й предок) 01010010 (2-й предок)
- Потомки: 11010001 (1-й потомок) 01001010 (2-й потомок)

В результате скрещивания особей с генотипами 201 и 82 получены особи с генотипами 209 и 74.

2.1.3 Случайный кроссовер

В случайном кроссовере для каждого гена потомка случайно выбирается значение от одного из родителей. Пример работы случайного кроссовера:

- Родители: 11001001 (1-й предок) 01010010 (2-й предок)
- Потомки: 01010011 (1-й потомок) 11001000 (2-й потомок)

В результате скрещивания особей с генотипами 201 и 82 получены особи с генотипами 83 и 200.

2.1.4 Арифметический кроссовер

Арифметический кроссовер применяется для действительного (вещественного) кодирования. Каждый ген потомка вычисляется как взвешенная сумма соответствующих генов родителей. Формула для вычисления:

$$\text{child\_gene} = w_1 \cdot \text{parent1\_gene} + w_2 \cdot \text{parent2\_gene},$$

где w_1 и w_2 — веса, обычно равные 0.5, чтобы результат был средним между родителями.

Пример арифметического кроссовера:

- Родители: $(2.1, 3.5)$ (1-й предок) $(4.2, 1.8)$ (2-й предок)
- Потомки: $(3.15, 2.65)$

Этот метод используется для создания потомка, расположенного ”между” родителями в непрерывном пространстве решений.

3 Описание схемы пошагового выполнения алгоритма и блок-схемы

3.1 Описание схемы пошагового выполнения алгоритма

Алгоритм начинается с инициализации графического интерфейса и параметров. Программа ожидает взаимодействия с пользователем, в частности нажатия кнопки ”Старт”. После нажатия происходит проверка корректности введённых параметров. Если параметры корректны, алгоритм запускается в новом потоке.

Основной цикл алгоритма включает следующие шаги:

- **Инициализация популяции:** Создание начальной популяции особей.
- **Селекция:** Выбор родителей через турнирный отбор.
- **Кроссовер:** Создание потомков с использованием различных методов кроссовера.
- **Мутация:** Изменение генотипов потомков для поддержания разнообразия.
- **Оценка приспособленности:** Вычисление значений функции приспособленности для потомков.
- **Обновление GUI:** Отображение текущей популяции и лучшего результата.

Алгоритм завершается, когда достигается заданное количество поколений или пользователь останавливает выполнение. Результаты отображаются в интерфейсе.

3.2 Блок-схемы

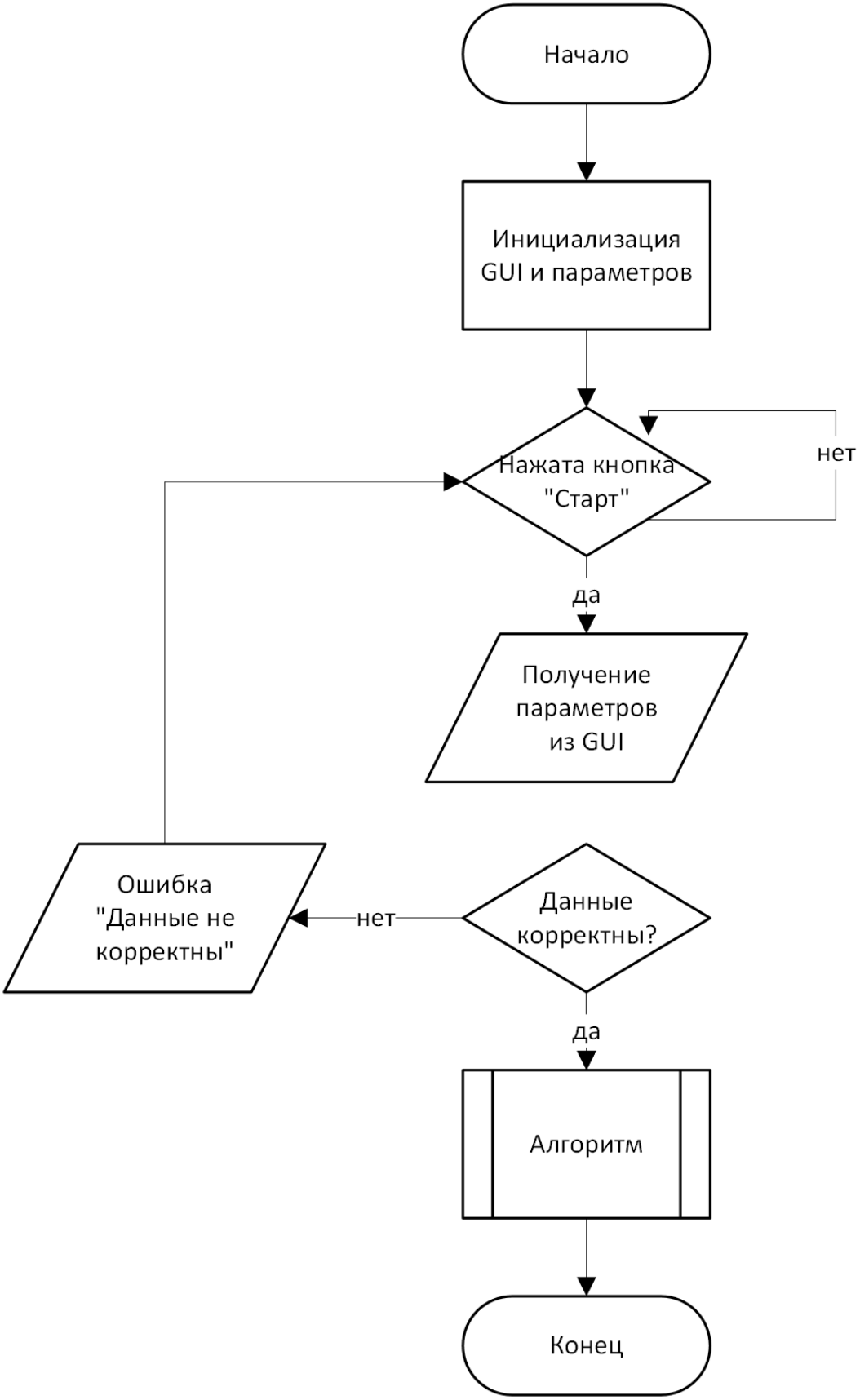


рис. 1 Основной поток программы

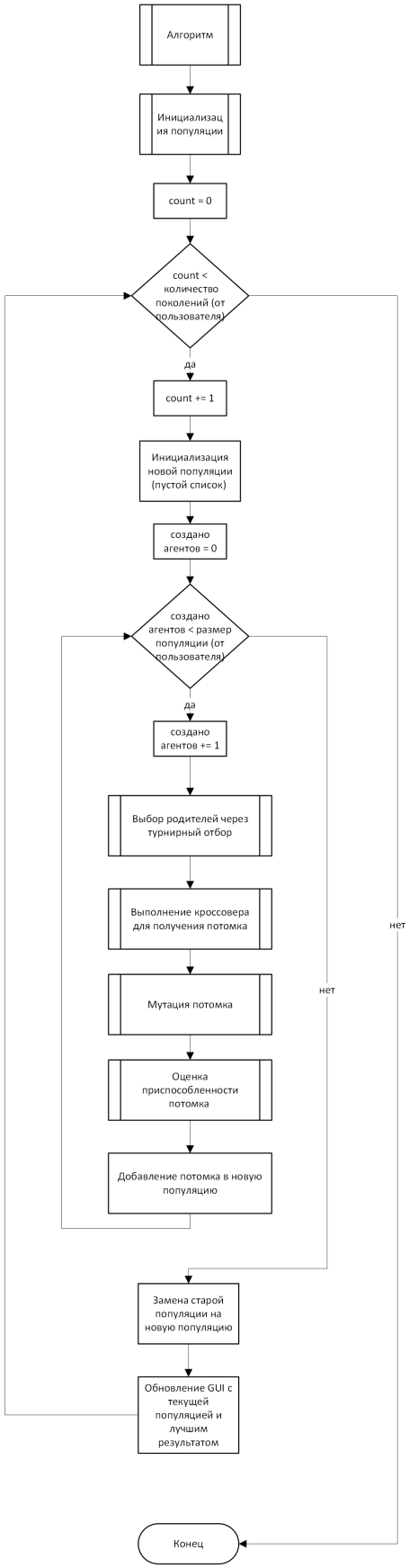


рис. 2 Поток выполнения алгоритма

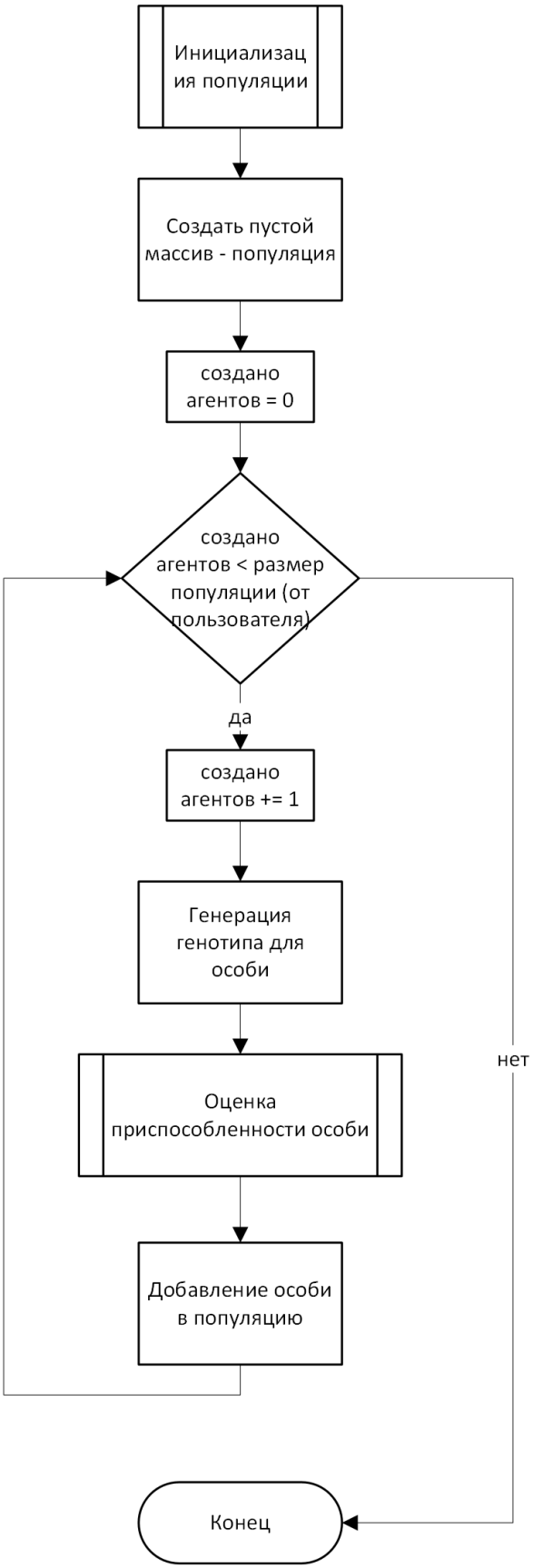


рис. 3 Инициализация популяции

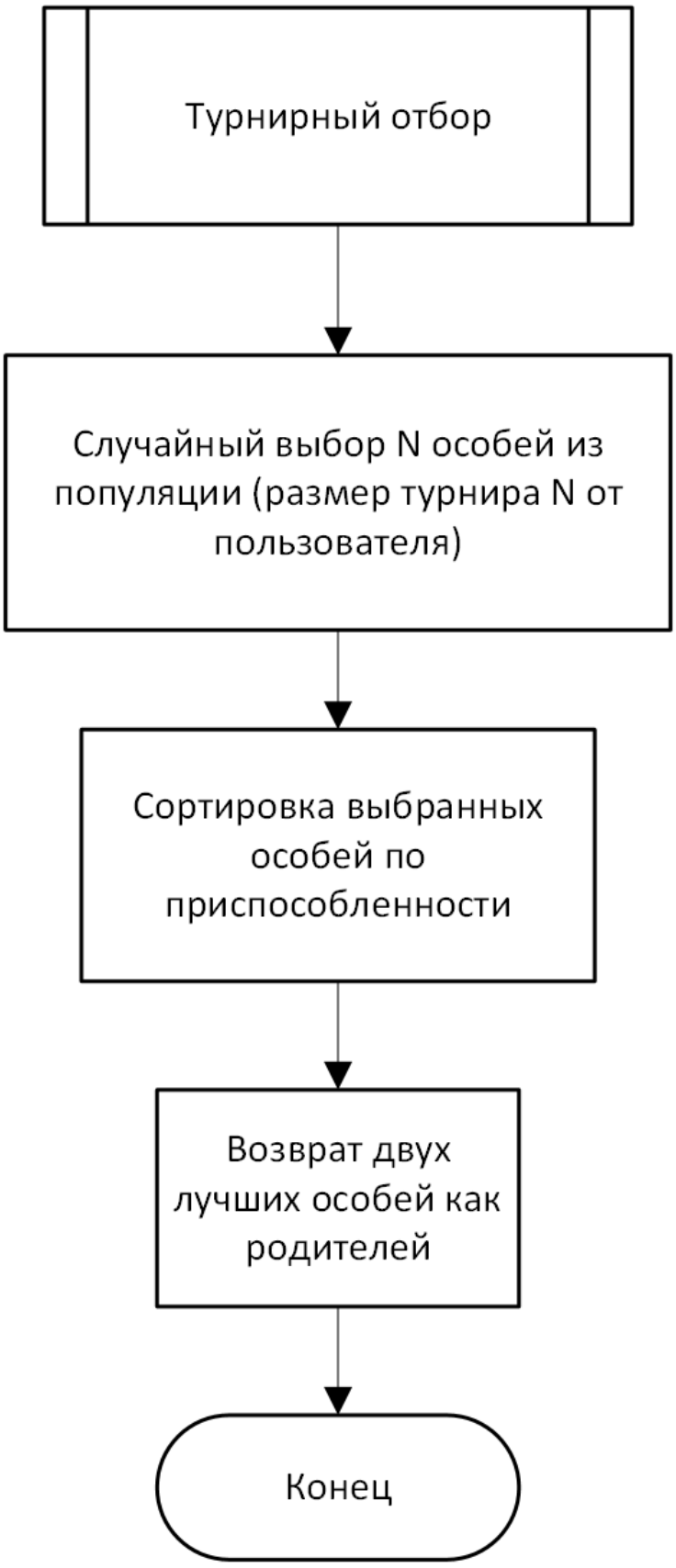


рис. 4 Турнирный отбор

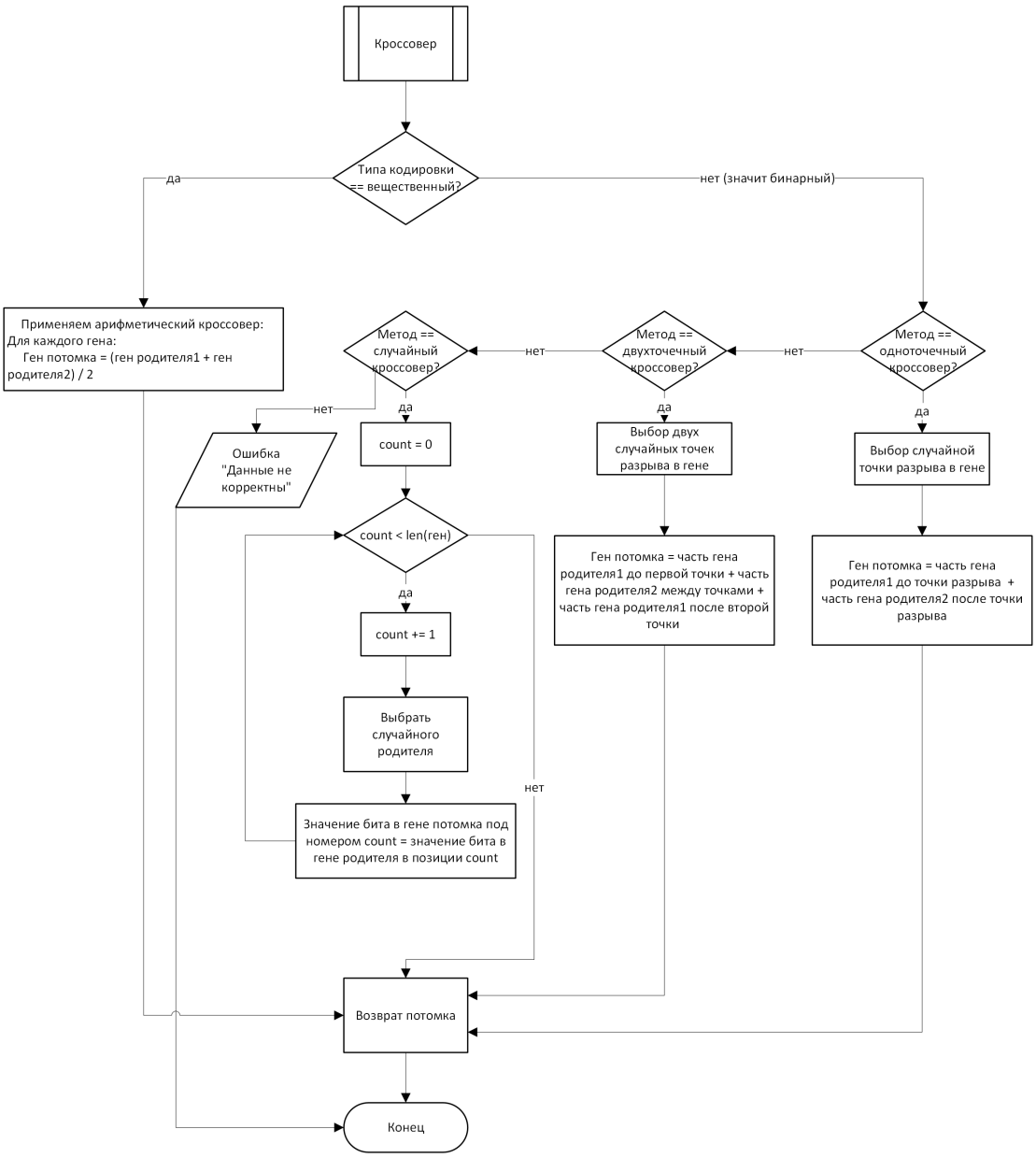


рис. 5 Операция кроссовера

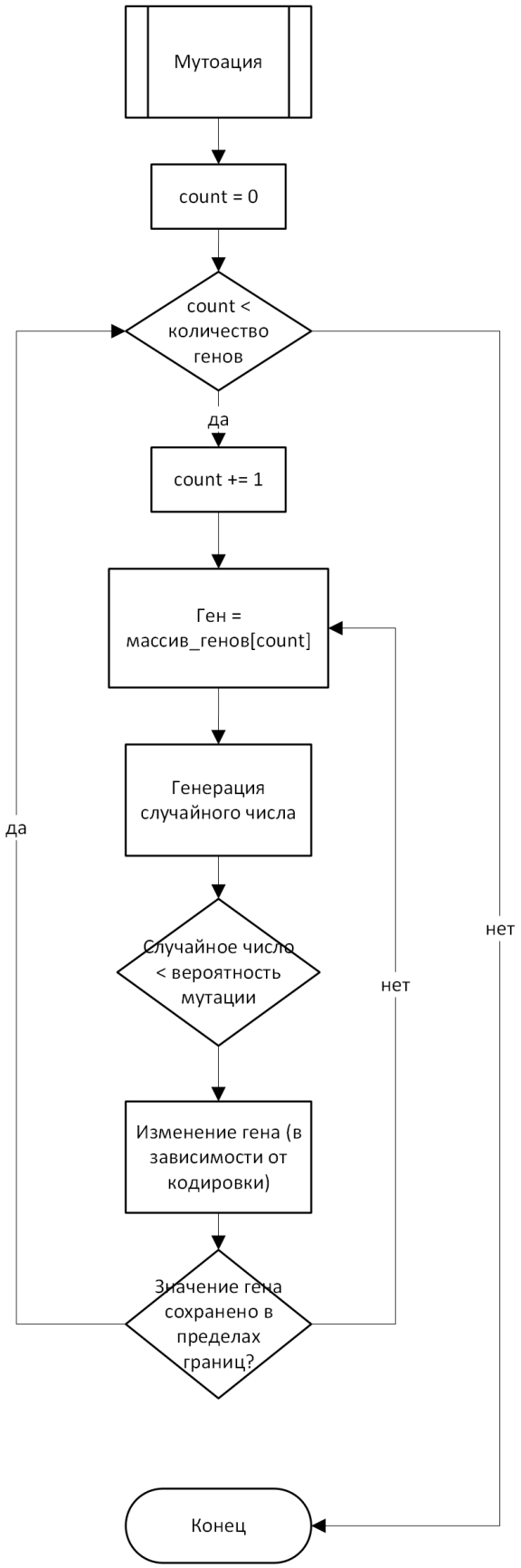


рис. 6 Операция мутации

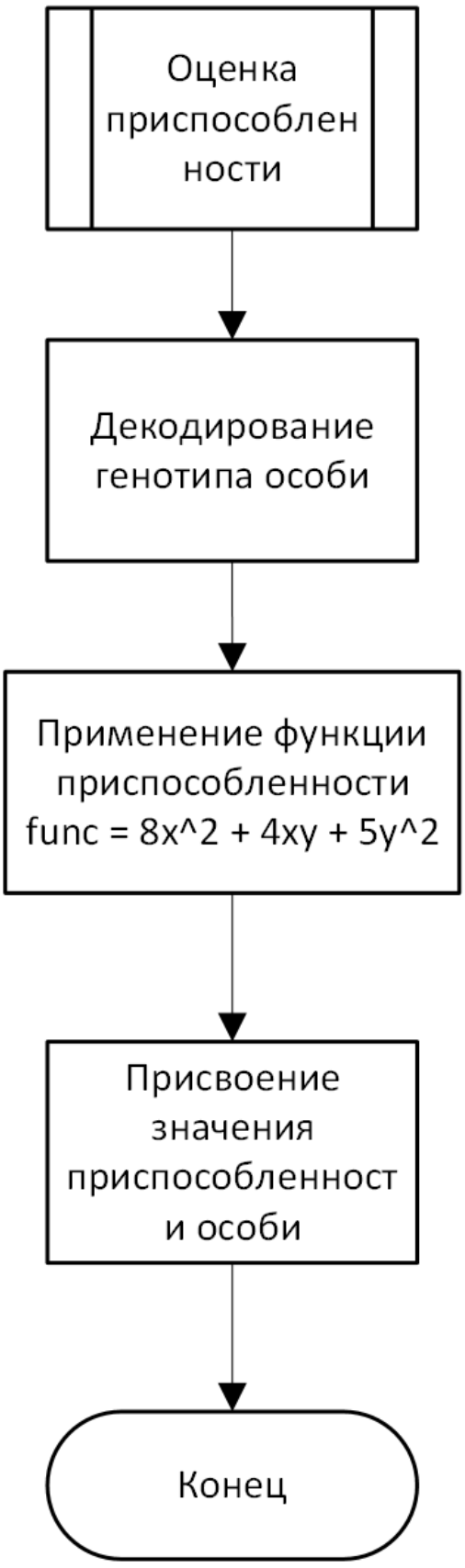


рис. 7 Оценка приспособленности

3.3 Листинг программы

```
import tkinter as tk
from tkinter import ttk, messagebox
from threading import Thread
import random

BINARY_CODING_ACCURACY = 1000 # Accuracy for binary encoding


class Individual:
    def __init__(self, genotype):
        self.genotype = genotype
        self.fitness = None # Fitness function value

    def __str__(self):
        return f"{self.genotype} | {self.fitness}"

    def __lt__(self, other):
        if isinstance(other, Individual):
            return self.fitness < other.fitness
        return NotImplemented

    def __le__(self, other):
        if isinstance(other, Individual):
            return self.fitness <= other.fitness
        return NotImplemented

    def __eq__(self, other):
        if isinstance(other, Individual):
            return self.fitness == other.fitness
        return NotImplemented

    def __ne__(self, other):
        if isinstance(other, Individual):
            return self.fitness != other.fitness
        return NotImplemented

    def __gt__(self, other):
        if isinstance(other, Individual):
```

```
            return self.fitness > other.fitness
        return NotImplemented

    def __ge__(self, other):
        if isinstance(other, Individual):
            return self.fitness >= other.fitness
        return NotImplemented


def initialize_population(population_size, encoding, search_space):
    population = []
    for _ in range(population_size):
        genotype = generate_genotype(encoding, search_space)
        individual = Individual(genotype)
        fitness_function(individual, encoding, search_space)
        population.append(individual)
    return population


def generate_random_binary_for_objects(max_value, num_bits):
    if max_value < 0:
        raise ValueError("Number of objects must be a positive number.")

    check_num_bits = len(bin(max_value)[2:])

    if check_num_bits > num_bits:
        raise ValueError("Binary representation of max_value exceeds
            ↪ possible length. (check_num_bits > num_bits)")

    random_value = random.randint(0, max_value)
    binary_string = bin(random_value)[2:].zfill(num_bits)
    return binary_string


def is_point_within_bounds(point, search_space):
    for coordinate, bounds in zip(point, search_space):
        if not bounds[0] <= coordinate <= bounds[1]:
            return False
    return True
```

```
def generate_genotype(encoding, search_space):
    if encoding == 'binary':
        genotype = []
        for bounds in search_space:
            range_size = int((bounds[1] - bounds[0]) *
                ↪ BINARY_CODING_ACCURACY)
            num_objects = random.randint(0, range_size)
            num_bits_in_str = len(bin(range_size)[2:])
            gene = generate_random_binary_for_objects(num_objects,
                ↪ num_bits_in_str)
            genotype.append(gene)
    elif encoding == 'real':
        genotype = [random.uniform(bounds[0], bounds[1]) for bounds in
            ↪ search_space]
    else:
        raise ValueError("Unsupported encoding")
    return genotype


def fitness_function(individual, encoding, search_space):
    x, y = decode_genotype(individual.genotype, encoding, search_space)
    func_var = 8 * (x ** 2) + 4 * x * y + 5 * (y ** 2)
    individual.fitness = func_var


def decode_genotype(genotype, encoding, search_space):
    if encoding == "real":
        return genotype
    elif encoding == "binary":
        decoded_values = []
        for gen, bounds in zip(genotype, search_space):
            decoded_values.append(float(binary_string_to_int(gen)) /
                ↪ BINARY_CODING_ACCURACY + bounds[0])
        return decoded_values
    else:
        raise ValueError("Unsupported encoding")


def binary_string_to_int(binary_string):
```

```
    try:
        return int(binary_string, 2)
    except ValueError:
        raise ValueError(f"Invalid binary string: {binary_string}")


def tournament_selection(population, tournament_size):
    selected_individuals = random.sample(population, tournament_size)
    selected_individuals = sorted(selected_individuals, key=lambda agent:
        ↪ agent.fitness)
    return selected_individuals[0], selected_individuals[1]


def mutate(individual, mutation_rate, search_space, encoding):
    if encoding == 'real':
        for mutated_gene_idx in range(len(individual.genotype)):
            if random.random() < mutation_rate:
                rand = random.uniform(-0.1, 0.1)
                individual.genotype[mutated_gene_idx] += rand
                if rand >= 0:
                    individual.genotype[mutated_gene_idx] =
                        ↪ min(search_space[mutated_gene_idx][1],
                                                    individual.genotype[mutated
                else:
                    individual.genotype[mutated_gene_idx] =
                        ↪ max(search_space[mutated_gene_idx][0],
                                                    individual.genotype[mutated
    elif encoding == 'binary':
        for mutated_gene_idx in range(len(individual.genotype)):
            if random.random() < mutation_rate:
                max_bin_num_in_dec = int((search_space[mutated_gene_idx][1]
                    ↪ - search_space[mutated_gene_idx][0]) *
                    ↪ BINARY_CODING_ACCURACY)

                gene = individual.genotype[mutated_gene_idx]
                attempts = 0
                while True:
                    attempts += 1
                    mutated_bit_idx = random.randint(0, len(gene) - 1)
                    gene_list = list(gene)
```

```
                gene_list[mutated_bit_idx] = '0' if
                    ↪ gene_list[mutated_bit_idx] == '1' else '1'
                new_gene = ''.join(gene_list)
                if int(new_gene, 2) <= max_bin_num_in_dec:
                    individual.genotype[mutated_gene_idx] = new_gene
                    break
                if attempts > 100:
                    break


def crossover(parent1, parent2, encoding, search_space,
    ↪ crossover_type=None):
    if encoding == "real":
        child_genotype = [(parent1.genotype[i] + parent2.genotype[i]) / 2
            ↪ for i in range(len(parent1.genotype))]
        return Individual(child_genotype)
    elif encoding == "binary":
        child_genotype = []

        for gene1, gene2, bounds in zip(parent1.genotype, parent2.genotype,
            ↪ search_space):
            max_bin_num_in_dec = int((bounds[1] - bounds[0]) *
                ↪ BINARY_CODING_ACCURACY)
            gene_length = len(gene1)
            attempts = 0
            while True:
                attempts += 1
                if crossover_type == 'Single-point crossover':
                    crossover_point = random.randint(1, gene_length - 1)
                    child_gene = gene1[:crossover_point] +
                        ↪ gene2[crossover_point:]
                elif crossover_type == 'Two-point crossover':
                    if gene_length < 3:
                        child_gene = gene1 # If gene is too short, skip
                            ↪ crossover
                    else:
                        point1 = random.randint(1, gene_length - 2)
                        point2 = random.randint(point1 + 1, gene_length - 1)
                        child_gene = gene1[:point1] + gene2[point1:point2] +
                            ↪ gene1[point2:]
```

```
                elif crossover_type == 'Random crossover':
                    child_gene = ''
                    for bit1, bit2 in zip(gene1, gene2):
                        child_gene += random.choice([bit1, bit2])
                else:
                    crossover_point = random.randint(1, gene_length - 1)
                    child_gene = gene1[:crossover_point] +
                        ↪ gene2[crossover_point:]

                if int(child_gene, 2) <= max_bin_num_in_dec:
                    child_genotype.append(child_gene)
                    break
                if attempts > 100:
                    child_gene = bin(random.randint(0,
                        ↪ max_bin_num_in_dec))[2:].zfill(gene_length)
                    child_genotype.append(child_gene)
                    break
            return Individual(child_genotype)
        else:
            raise ValueError("Unsupported encoding")


class GeneticAlgorithmGUI:
    def __init__(self, master):
        self.master = master
        self.master.title("Genetic Algorithm Optimization")
        self.master.configure(bg="#FFFAA0") # Pastel-yellow background
        self.master.geometry("1200x800")

        self.population_size_var = tk.IntVar(value=100)
        self.mutation_rate_var = tk.DoubleVar(value=0.05)
        self.tournament_size_var = tk.IntVar(value=10)
        self.encoding_var = tk.StringVar(value='real')
        self.generations_var = tk.IntVar(value=20)
        self.iteration_count = 0

        # New variables for search bounds
        self.min_x_var = tk.DoubleVar(value=-10)
        self.max_x_var = tk.DoubleVar(value=10)
        self.min_y_var = tk.DoubleVar(value=-10)
```

```
        self.max_y_var = tk.DoubleVar(value=10)

        self.running = False
        self.stop_requested = False

        self.crossover_type_var = tk.StringVar(value='Single-point
            ↪ crossover')

        self.create_widgets()

    def create_widgets(self):
        style = ttk.Style()
        style.configure('TFrame', background='#FFFAA0')
        style.configure('TLabel', background='#FFFAA0')
        style.configure('TLabelframe', background='#FFFAA0')
        style.configure('TLabelframe.Label', background='#FFFAA0')
        style.configure('TButton', background='#FFFAA0')

        # Top toolbar with buttons
        toolbar = ttk.Frame(self.master)
        toolbar.pack(side=tk.TOP, fill=tk.X)

        start_button = ttk.Button(toolbar, text="Start",
            ↪ command=self.start_algorithm)
        start_button.pack(side=tk.LEFT, padx=5, pady=5)

        stop_button = ttk.Button(toolbar, text="Stop",
            ↪ command=self.stop_algorithm)
        stop_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Main frame
        main_frame = ttk.Frame(self.master)
        main_frame.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        # Parameter frame on the left
        parameter_frame = ttk.LabelFrame(main_frame, text="Genetic
            ↪ Algorithm Parameters")
        parameter_frame.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)

        ttk.Label(parameter_frame, text="Population size:").pack(anchor='w')
```

```
population_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.population_size_var)
population_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Number of
    ↪ generations:").pack(anchor='w')
generations_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.generations_var)
generations_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Mutation
    ↪ probability:").pack(anchor='w')
mutation_rate_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.mutation_rate_var)
mutation_rate_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Tournament size:").pack(anchor='w')
tournament_size_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.tournament_size_var)
tournament_size_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Encoding type:").pack(anchor='w')
self.encoding_combobox = ttk.Combobox(parameter_frame,
    ↪ textvariable=self.encoding_var,
                                      values=['real', 'binary'])
self.encoding_combobox.pack(fill='x', pady=5)
self.encoding_combobox.bind('<<ComboboxSelected>>',
    ↪ self.on_encoding_change)

self.crossover_type_label = ttk.Label(parameter_frame,
    ↪ text="Crossover type:")
self.crossover_type_combobox = ttk.Combobox(parameter_frame,
    ↪ textvariable=self.crossover_type_var,
                                        values=['Single-point
                                            ↪ crossover', 'Two-point
                                            ↪ crossover',
                                                'Random crossover'])

# New fields for search bounds
ttk.Label(parameter_frame, text="Minimum x value:").pack(anchor='w')
```

```
min_x_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.min_x_var)
min_x_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Maximum x value:").pack(anchor='w')
max_x_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.max_x_var)
max_x_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Minimum y value:").pack(anchor='w')
min_y_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.min_y_var)
min_y_entry.pack(fill='x', pady=5)

ttk.Label(parameter_frame, text="Maximum y value:").pack(anchor='w')
max_y_entry = ttk.Entry(parameter_frame,
    ↪ textvariable=self.max_y_var)
max_y_entry.pack(fill='x', pady=5)

# Result frame on the right
result_frame = ttk.LabelFrame(main_frame, text="Results")
result_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True,
    ↪ padx=10, pady=10)

# Table for displaying population
columns = ("Individual Number", "Genome", "Fitness Value")
self.population_tree = ttk.Treeview(result_frame, columns=columns,
    ↪ show="headings")
for col in columns:
    self.population_tree.heading(col, text=col)
self.population_tree.pack(fill=tk.BOTH, expand=True)

# Scrollbar for table
scrollbar = ttk.Scrollbar(self.population_tree, orient="vertical",
    ↪ command=self.population_tree.yview)
self.population_tree.configure(yscrollcommand=scrollbar.set)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# Frame for iteration display
iteration_frame = ttk.Frame(self.master)
```

```
        iteration_frame.pack(side=tk.BOTTOM, fill=tk.X, padx=10, pady=10)

        ttk.Label(iteration_frame, text="Number of
            ↪ iterations:").pack(side=tk.LEFT)
        self.iteration_label = ttk.Label(iteration_frame, text="0")
        self.iteration_label.pack(side=tk.LEFT)

        # Frame for best result
        best_result_frame = ttk.LabelFrame(self.master, text="Best Result")
        best_result_frame.pack(side=tk.BOTTOM, fill=tk.X, padx=10, pady=10)

        ttk.Label(best_result_frame, text="Point coordinates:").grid(row=0,
            ↪ column=0, sticky="w")
        self.best_result_label = ttk.Label(best_result_frame, text="")
        self.best_result_label.grid(row=0, column=1, sticky="w")

        ttk.Label(best_result_frame, text="Function value:").grid(row=1,
            ↪ column=0, sticky="w")
        self.best_fitness_label = ttk.Label(best_result_frame, text="")
        self.best_fitness_label.grid(row=1, column=1, sticky="w")

        # Initial visibility state of elements
        self.on_encoding_change()

    def on_encoding_change(self, event=None):
        if self.encoding_var.get() == 'binary':
            self.crossover_type_label.pack(anchor='w')
            self.crossover_type_combobox.pack(fill='x', pady=5)
        else:
            self.crossover_type_label.pack_forget()
            self.crossover_type_combobox.pack_forget()

    def start_algorithm(self):
        if not self.running:
            try:
                population_size = int(self.population_size_var.get())
                generations = int(self.generations_var.get())
                mutation_rate = float(self.mutation_rate_var.get())
                tournament_size = int(self.tournament_size_var.get())
                encoding = self.encoding_var.get()
```

```
            min_x = float(self.min_x_var.get())
            max_x = float(self.max_x_var.get())
            min_y = float(self.min_y_var.get())
            max_y = float(self.max_y_var.get())

            # Check bounds validity
            if min_x >= max_x or min_y >= max_y:
                messagebox.showerror("Error", "Minimum values must be
                    ↪ less than maximum values.")
                return

            search_space = [(min_x, max_x), (min_y, max_y)]
        except ValueError:
            messagebox.showerror("Error", "Please enter valid parameter
                ↪ values.")
            return

        self.running = True
        self.stop_requested = False
        Thread(target=self.run_genetic_algorithm, args=(population_size,
            ↪ generations, mutation_rate,
                                                        tournament_size,
                                                            ↪ encoding,
                                                            ↪ search_space)).start()

def run_genetic_algorithm(self, population_size, generations,
    ↪ mutation_rate, tournament_size, encoding,
                              search_space):
    try:
        population = initialize_population(population_size, encoding,
            ↪ search_space)

        if encoding == 'binary':
            crossover_type = self.crossover_type_var.get()
        else:
            crossover_type = None

        for generation in range(generations):
            if self.stop_requested:
                break
```

```
            new_population = []

            for _ in range(population_size):
                parent1, parent2 = tournament_selection(population,
                    ↪ tournament_size)

                child = crossover(parent1, parent2, encoding,
                    ↪ search_space, crossover_type)

                mutate(child, mutation_rate, search_space, encoding)

                fitness_function(child, encoding, search_space)

                new_population.append(child)

            population = new_population

            self.update_population_table(population, encoding,
                ↪ search_space)

            best_individual = min(population, key=lambda agent:
                ↪ agent.fitness)
            decoded_result = decode_genotype(best_individual.genotype,
                ↪ encoding, search_space)
            self.update_best_result_label(decoded_result,
                ↪ best_individual.fitness)

            self.iteration_count += 1
            self.iteration_label.config(text=str(self.iteration_count))

        self.running = False
    except Exception as e:
        print(e)
        messagebox.showerror("Error", f"An error occurred: {str(e)}")
        self.running = False

def stop_algorithm(self):
    self.stop_requested = True
```

```
    def update_population_table(self, population, encoding, search_space):
        for item in self.population_tree.get_children():
            self.population_tree.delete(item)

        for i, individual in enumerate(population, start=1):
            decoded_genome = decode_genotype(individual.genotype, encoding,
                ↪ search_space)
            fitness_value = round(individual.fitness, 6)
            self.population_tree.insert("", "end", values=(i,
                ↪ decoded_genome, fitness_value))

    def update_best_result_label(self, decoded_result, fitness):
        self.best_result_label.config(text=f"{decoded_result}")
        self.best_fitness_label.config(text=f"{round(fitness, 6)}")


if __name__ == "__main__":
    root = tk.Tk()
    app = GeneticAlgorithmGUI(root)
    root.mainloop()
```

4 Формализация задачи

4.1 Задача оптимизации

Оптимизируемая функция:

$$f(x, y) = 8x^2 + 4xy + 5y^2$$

Задача состоит в нахождении пары (x, y), минимизирующей $f(x, y)$, при этом x и y находятся в диапазоне:

$$x, y \in [x_{\text{min}}, x_{\text{max}}] \quad \text{и} \quad [y_{\text{min}}, y_{\text{max}}]$$

Методы оптимизации включают использование бинарной и вещественной кодировки, а также модифицированных методов кроссовера (одноточечного, двухточечного, случайного и арифметического).

4.2 Спецификация программы

Table 1: Спецификация функций программы (часть 1)

Имя функции	**Тип возвращаемого значения**	**Описание функции**
initialize_population	list (Individual)	Инициализация начальной популяции для генетического алгоритма. Возвращает список особей (экземпляров класса Individual).
generate_random_binary_for_objects	str	Генерация случайного бинарного числа для кодирования указанного числа объектов. Возвращает строку с бинарным числом.
is_point_within_bounds	bool	Проверяет, находится ли точка в заданных границах. Возвращает True, если точка в границах, иначе False.
generate_genotype	list	Генерация генотипа в соответствии с выбранной кодировкой. Возвращает список генов.
fitness_function	None	Вычисляет значение функции приспособленности для особи и сохраняет его в атрибуте fitness объекта Individual.
decode_genotype	list	Декодирует генотип в зависимости от типа кодировки (бинарный или вещественный). Возвращает декодированный список значений.

Table 2: Спецификация функций программы (часть 2.1)

Имя функции	**Тип возвращаемого значения**	**Описание функции**
binary_string_to_int	int	Конвертирует бинарную строку в десятичное число.
tournament_selection	tuple (Individual, Individual)	Выполняет турнирный отбор двух особей из популяции. Возвращает двух родителей.
mutate	None	Выполняет мутацию генотипа особи. Изменяет генотип в зависимости от вероятности мутации и типа кодировки.
crossover	Individual	Выполняет операцию кроссовера между двумя родительскими особями. Возвращает новую особь (потомка).
on_encoding_change	None	Обновляет виджеты GUI в зависимости от выбранного типа кодировки (бинарный или вещественный).

Table 3: Спецификация функций программы (часть 2.2)

Имя функции	**Тип возвращаемого значения**	**Описание функции**
start_algorithm	None	Запускает выполнение генетического алгоритма в отдельном потоке, инициализируя параметры из пользовательского интерфейса.
run_genetic_algorithm	None	Основной цикл генетического алгоритма: выполняет селекцию, кроссовер, мутацию и обновление популяции на каждом поколении.
stop_algorithm	None	Прерывает выполнение генетического алгоритма.
update_population_table	None	Обновляет таблицу с популяцией в GUI.
update_best_result_labels	None	Обновляет метки GUI, отображающие лучший результат и значение функции приспособленности.

5 Контрольный пример и результаты тестирования

5.1 Описание контрольного примера

В рамках исследования был проведен контрольный пример, целью которого было оценить эффективность различных методов кроссовера в генетическом алгоритме. Параметры тестирования включали:

- Размер популяции: 100
- Вероятность мутации: 0.05
- Размер турнира: 10
- Диапазон значений x и y: от -10 до 10

На рисунке ниже представлено окно программы, демонстрирующее результаты тестирования:

5.1.1 Установка и настройка окружения

1. Установите **Git** и **Python 3.9**:
    - Git: `https://git-scm.com`
    - Python 3.9: `https://www.python.org`
2. Откройте терминал и клонируйте репозиторий с помощью команды:

    ```
    git clone https://github.com/MansurYa/labs-for-algorithms-an
    ```

    Это создаст копию проекта в текущем каталоге терминала.
3. Перейдите в директорию проекта:

    ```
    cd labs-for-algorithms-and-data-structures/Lab4
    ```

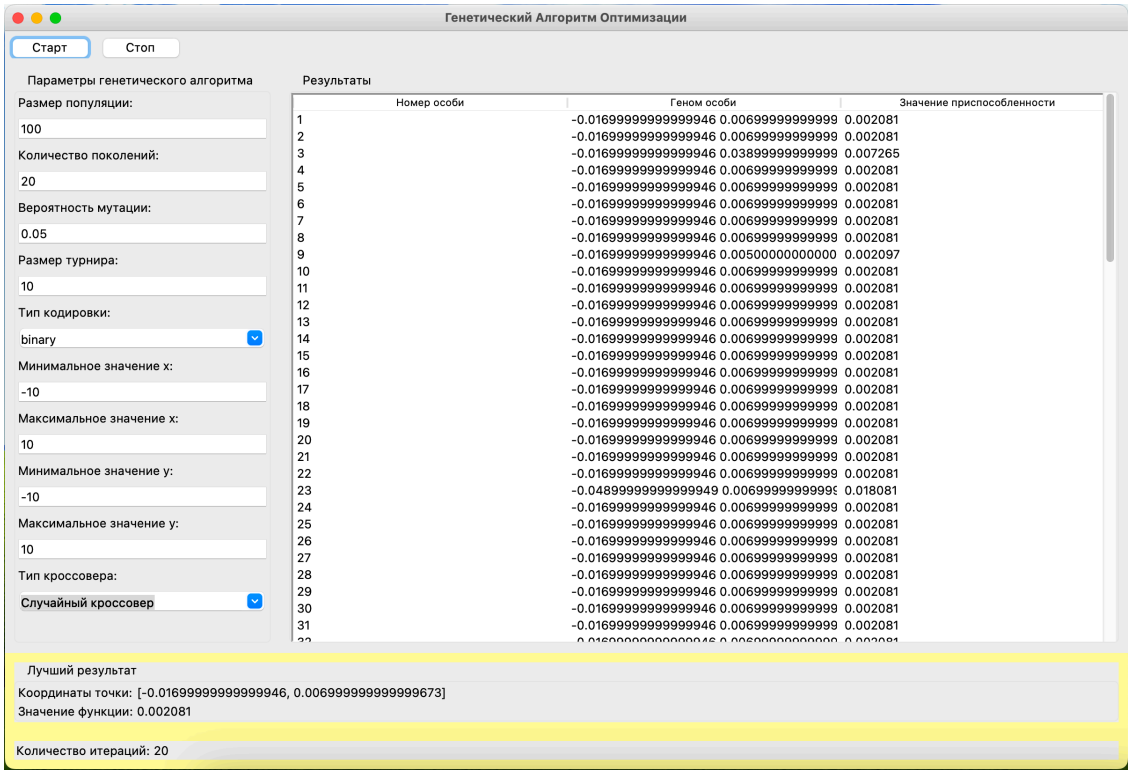
5.1.2 Запуск программы и выполнение задач

1. Запустите программу с помощью Python:

```
python3 main.py
```

2. Нажмите на кнопку ”Старт”: cd labs-for-algorithms-and-data-structures/Lab4

5.2 Контрольный пример



(рис. 8 Скриншот программы)

5.3 Результаты тестирования программы

Тестирование проводилось с использованием двух методов кроссовера: вещественного арифметического и бинарного одноточечного. Результаты представлены в таблицах ниже.

Table 4: Вещественный арифметический кроссовер

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	600	(-8.2e-04, -8.5e-04)	6e-06
10	1100	(-2.6e-06, -1.9e-06)	≈0.0
20	2100	(-2.2e-10, -9.1e-9)	≈0.0

Table 5: Бинарный одноточечный кроссовер

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	600	(0.25, -0.17699)	0.479
10	1100	(-0.01699, -0.27299)	0.393
20	2100	(0.003, -0.01699)	0.00131

6 Анализ и улучшение алгоритма

6.1 Анализ результатов работы алгоритма

Результаты показывают, что вещественный арифметический кроссовер обеспечивает более высокую точность, особенно при увеличении числа поколений. Бинарный одноточечный кроссовер также демонстрирует хорошие результаты, но с меньшей точностью. Выбор метода кроссовера должен основываться на специфике задачи и требованиях к точности.

6.2 Модернизация кроссовера

В рамках работы были реализованы следующие модификации:

- **Двухточечный кроссовер**: Этот метод увеличивает генетическое разнообразие, что помогает избежать застревания в локальных оптимумах. Он показал улучшение точности по сравнению с одноточечным кроссовером.
- **Случайный кроссовер**: Способствует разнообразию, выбирая гены случайным образом от каждого родителя. Это позволяет поддерживать разнообразие в популяции и улучшает результаты в сложных задачах.

6.3 Процесс отладки

В процессе отладки использовались графики, показывающие сходимость алгоритма. Это позволило выявить узкие места и оптимизировать производительность. Например, увеличение числа поколений положительно сказалось на точности решений.

Table 6: Результаты для двухточечного кроссовера

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	600	(-0.021, -0.028)	0.005
10	1100	(0.01299, -0.028)	0.0038
20	2100	(0.0039, -0.01699)	0.0013

Table 7: Результаты для случайного кроссовера

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	600	(0.06, 0.07)	0.008
10	1100	(0.005, -0.019)	0.0016
20	2100	(0.003, -0.016)	0.0013

6.4 Результаты тестирования

6.4.1 Двухточечный кроссовер

6.4.2 Случайный кроссовер

6.5 Выводы и рекомендации

На основе проведенного анализа рекомендуется использовать вещественный кроссовер для задач, требующих высокой точности. Бинарные методы подходят для быстрого получения результатов. Увеличение числа поколений и поддержание генетического разнообразия положительно сказываются на качестве решений. Эти выводы помогут в дальнейшем улучшении и адаптации генетического алгоритма для различных задач.

7 Результаты тестирования программы

7.1 Введение

7.2 Сравнительный анализ методов кроссовера

На основе проведенного тестирования можно сделать следующие выводы:

1. **Вещественный арифметический кроссовер**:
    - Показал высокую точность, особенно при увеличении числа поколений. Значение функции в лучшей точке стремится к нулю, что указывает на близость к оптимальному решению.
    - Рекомендуется для задач, где требуется высокая точность.
2. **Бинарные методы кроссовера**:
    - **Одноточечный кроссовер**: Обеспечивает быструю сходимость, но точность ниже, чем у вещественного кроссовера.
    - **Двухточечный кроссовер**: Улучшает распределение генетического материала, что способствует более точному решению.
    - **Случайный кроссовер**: Поддерживает генетическое разнообразие, что помогает избежать локальных минимумов.

8 Выводы по работе

В ходе исследования были изучены два основных способа кодирования генотипа хромосом в генетическом алгоритме: вещественный и бинарный. Генетический алгоритм продемонстрировал свою эффективность в оптимизации для обоих типов геномов.

1. **Вещественный геном**:
    - Обеспечивает высокую точность, особенно при увеличении числа поколений.
    - Предпочтителен в задачах, где требуется высокая точность в оптимальной точке.
2. **Бинарный геном**:
    - Обеспечивает стабильные результаты даже при ограниченном количестве итераций.
    - Подходит для быстрого получения приемлемых решений, но ограничивает точность из-за дискретного характера.
3. **Влияние параметров**:
    - Увеличение размера популяции и числа итераций положительно влияет на качество решения для обоих типов геномов.
4. **Модернизации кроссовера**:
    - Двухточечный кроссовер улучшает точность и устойчивость к локальным минимумам.
    - Случайный кроссовер поддерживает генетическое разнообразие, что способствует исследованию пространства решений.
5. **Рекомендации**:
    - Выбор метода кодирования и кроссовера должен основываться на специфике задачи.

- Для задач, требующих высокой точности, рекомендуется использовать вещественный геном и двухточечный кроссовер.
- Для быстрого получения результатов и поддержания разнообразия можно использовать бинарный геном и случайный кроссовер.

9 Полезные ссылки

- Репозиторий со всеми лабораторными работами по предмету ”Алгоритмы и структуры данных” в СПбГУ (группа БД 2022)
- Репозиторий текущей лабораторной работы №3