



**Институт  
интеллектуальных кибернетических систем**  
**Кафедра №22 «Кибернетика»**

Направление подготовки 09.03.04 Программная инженерия

## **Пояснительная записка**

к учебно-исследовательской работе студента на тему:  
«Разработка системы, реализующей генетический алгоритм и применение её к  
расположению вершин графа на плоскости»

<p><b>Комиссия</b></p> <hr/> <p>(подпись)</p> <hr/> <p>(подпись)</p> <hr/> <p>(подпись)</p> <hr/> <p>(подпись)</p>	<hr/> <p>(ФИО)</p> <hr/> <p>(ФИО)</p> <hr/> <p>(ФИО)</p> <hr/> <p>(ФИО)</p>
--	---

## **Оглавление**

<b>Реферат .....</b>	<b>3</b>
<b>Введение .....</b>	<b>4</b>
<b>Раздел 1. Исследование современных генетических алгоритмов и сравнительный анализ методов укладки различных графов.....</b>	<b>6</b>
1.1. Формулировка оптимизационных задач .....	6
1.2. Описание методов решения тестовых задач .....	11
1.3. Анализ результатов решения тестовых задач с помощью выбранных методов 21	
Выводы .....	33
Цели и задачи на УИР .....	34
<b>Раздел 2. Разработка новых алгоритмов, адаптированных к поставленной задаче .</b>	<b>34</b>
2.1. Разработка генетических операторов .....	34
2.2. Разработка гибридных методов .....	39
Выводы .....	41
<b>Раздел 3. Проектирование системы, реализующей разработанные алгоритмы .....</b>	<b>41</b>
3.1. Разработка требований к создаваемой системе .....	41
3.2. Проектирование системы на основе требований .....	42
Выводы .....	44
<b>Раздел 4. Программная реализация системы и достигнутые показатели .....</b>	<b>45</b>
4.1. Программная реализация системы и алгоритмов .....	45
4.2. Достигнутые показатели .....	46
Выводы .....	50
<b>Заключение .....</b>	<b>51</b>
<b>Список литературы.....</b>	<b>53</b>
<b>Приложения.....</b>	<b>54</b>
Приложение 1. Программный код алгоритмов генерации графов.....	54
Приложение 2. Программный код алгоритма Фрюхтермана-Рейнгольда.....	56
Приложение 3. Программный код метода динамического программирования для решения задачи о рюкзаке 0-1.....	57
Приложение 4. Программный код метода ветвей и границ .....	58
Приложение 5. Программный код метода полного перебора для решения симметричной задачи о коммивояжере.....	60
Приложение 6. Программный код SGA.....	61
Приложение 7. Программный код SSGA.....	62
Приложение 8. Программный код NSGA2 и быстрой не-доминирующей сортировки 63	
Приложение 9. Программный код SPEA2.....	65

<b>Приложение 10.</b>	<b>Графики сравнения ГА для задачи укладки графов.....</b>	<b>67</b>
<b>Приложение 11.</b>	<b>Фронты Парето для задач из набора ZDT, построенные ГА .....</b>	<b>70</b>
<b>Приложение 12.</b>	<b>Программный код оператора консервативной нормальной мутации</b>	<b>75</b>

## **Реферат**

Пояснительная записка содержит 75 страниц. Количество источников – 16. Количество рисунков – 33. Количество таблиц – 11.

Ключевые слова: генетические алгоритмы, многокритериальная оптимизация, теория графов, укладка графов.

Целью данной работы является разработка системы, реализующей генетический алгоритм, и применение ее к расположению вершин графа на плоскости.

В первом разделе проводится исследование современных генетических алгоритмов и сравнительный анализ методов укладки различных графов.

Второй раздел посвящен теоретической разработке новых алгоритмов для расположения вершин графа на плоскости.

Третий раздел посвящен разработке требований к системе, реализующей разработанные алгоритмы. Обосновывается выбор инструментальных средств.

В четвертом разделе приводится состав и структура реализованного программного обеспечения, дается его краткая характеристика. Показываются достигнутые показатели, указываются использованные подходы к тестированию.

В заключении подводятся итоги разработки системы. Также в этом разделе представлены перспективы дальнейшей работы над проектом.

## Введение

Современные научные и технические задачи все чаще требуют поиска оптимальных решений в условиях многокритериальности и высокой вычислительной сложности. Как отмечают авторы [1], большинство таких задач относятся к классу комбинаторных оптимизационных проблем, для которых характерно наличие множества альтернативных решений различного качества. Традиционные методы, основанные на полном или направленном переборе, сталкиваются с фундаментальными ограничениями: рост вычислительных ресурсов экспоненциально зависит от размерности задачи, а жесткие требования к математическим моделям сужают область их применимости [1], [2]. Это особенно актуально для задач, связанных с теорией графов, где необходимо одновременно учитывать несколько критериев.

В этом контексте укладка графов на плоскости является одной из ключевых задач. Графы представляют собой фундаментальный математический инструмент для моделирования сложных систем, находящий широкое применение в биоинформатике (анализ белковых взаимодействий, генетических сетей), социологии (визуализация социальных связей), информационных технологиях (проектирование компьютерных сетей, баз данных, схемотехника) и анализе больших данных (представление зависимостей и структур). Эффективная визуализация графов значительно улучшает читаемость и интерпретацию данных, делая сложные взаимосвязи интуитивно понятными.

Одной из центральных проблем в укладке графов является минимизация количества пересечений ребер. Наличие пересечений существенно снижает информативность и эстетическую привлекательность визуализации, затрудняя восприятие структуры графа человеком. Более того, в прикладных областях, таких как проектирование печатных плат, отсутствие пересечений является функциональным требованием. Помимо минимизации пересечений, задача укладки графа часто включает оптимизацию других метрик, таких как равномерность распределения вершин, минимизация длин ребер, обеспечение симметрии и однородности плотности расположения вершин. Эти метрики формируют многокритериальную задачу оптимизации, требующую комплексных подходов.

В этом контексте генетические алгоритмы (ГА) демонстрируют значительный потенциал как универсальный инструмент управляемого перебора. Их эволюционная природа, основанная на механизмах селекции, скрещивания и мутации, позволяет эффективно исследовать обширные пространства решений [2]. Однако эффективность ГА существенно зависит от выбора операторов, параметров адаптации и способов учета множественности критериев. Модификации алгоритмов, такие как NSGA2 и SPEA2, предлагают решения для

многокритериальных задач, но их применимость к специфическим проблемам, таким как визуализация графов, требует дополнительного исследования и адаптации.

Целью данной работы является разработка и анализ гибридных генетических алгоритмов, ориентированных на эффективное решение задачи укладки невзвешенных, неориентированных графов на плоскости с учетом нескольких критериев оптимизации, включая минимизацию количества пересечений ребер. Для достижения поставленных целей будут разработаны специализированные генетические операторы, адаптированные под особенности задачи, и создана модульная программная система для проведения экспериментальных исследований.

# Раздел 1. Исследование современных генетических алгоритмов и сравнительный анализ методов укладки различных графов

В данном разделе работы представлено исследование современных ГА и иных методов решения тестовых задач. Также приведен анализ результатов решения тестовых задач с помощью выбранных методов в целях выявления наиболее перспективных подходов к решению поставленной задачи. Помимо этого, описываются основные задачи УИР, которые ставятся в процессе работы.

## 1.1. Формулировка оптимизационных задач

Следующие оптимизационные задачи будут использованы в качестве тестовых. Они послужат основой для последующего сравнительного анализа методов.

1. Задача о расположении вершин графа на плоскости.

Задача заключается в назначении каждой из  $N$  вершин невзвешенного, неориентированного графа с  $M$  ребрами координат  $(0 \leq x_i, y_i \leq 1)$  в двухмерном пространстве таким образом, чтобы оптимизировать выбранные метрики. Предложенные метрики описаны ниже.

- Количество пересечений ребер.

Определим функцию [3], вычисляющую ориентацию тройки точек (Counter-Clockwise):

$$ccw(p, q, r) \triangleq \text{sign} \left( (r_x - p_x)(q_y - p_y) - (q_x - p_x)(r_y - p_y) \right) \quad (1)$$

$$ccw(p, q, r) \begin{cases} > 0, \text{ если } r \text{ лежит слева от } \overrightarrow{pq} \\ = 0, \text{ коллинеарность} \\ < 0, \text{ справа} \end{cases} \quad )$$

Тогда, для двух неориентированных ребер  $e_1 = \{v_a, v_b\}$  и  $e_2 = \{v_c, v_d\}$  определим функцию пересечения:

$$cr(e_1, e_2) \triangleq [ccw(v_a, v_c, v_d) \neq ccw(v_b, v_c, v_d)] \wedge [ccw(v_a, v_b, v_c) \neq ccw(v_a, v_b, v_d)] \quad (2)$$

$$cr(e_1, e_2) = \begin{cases} 1, \text{ отрезки } \overline{v_a v_b} \text{ и } \overline{v_c v_d} \text{ пересекаются} \\ 0, \text{ иначе} \end{cases}$$

Тогда метрику количества пересечений ребер в графе определим так:

$$I(G) = \sum_{\substack{\{e_1, e_2\} \subseteq \binom{E}{2} \\ e_1 \cap e_2 = \emptyset}} cr(e_1, e_2), \quad (3)$$

где  $\binom{E}{2}$  – множество всех неупорядоченных пар ребер

- Равномерность расположения вершин.

Для области размещения с шириной  $W$  и высотой  $H$  определим желаемое расстояние между вершинами:

$$d_{des}(G) = \frac{2 \cdot \min(W, H)}{\sqrt{N}} \quad (4)$$

Среднее попарное расстояние между вершинами и стандартное отклонение расстояний:

$$\mu_d(G) = \frac{1}{N(N-1)} \sum_{\substack{i,j \in V \\ i \neq j}} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (5)$$

$$\sigma_d(G) = \sqrt{\frac{1}{N(N-1)} \sum_{\substack{i,j \in V \\ i \neq j}} \left( \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - \mu_d(G) \right)^2} \quad (6)$$

Тогда метрику равномерности расположения вершин определим так:

$$D(G) = \frac{|d_{des} - \mu_d(G)|}{d_{des}} + \sigma_d(G) \quad (7)$$

Таким образом, метрика учитывает как отклонение расположения вершин от масштаба, так и неравномерность распределения.

- Равномерность углов между смежными ребрами.

Соседи вершины  $v$  (смежные с ней):

$$N(v) = \{u \in V \mid \{u, v\} \in E\} \quad (8)$$

$\vec{uv}$  – вектор от вершины  $u$  к вершине  $v$ . Тогда множество величин углов между смежными ребрами в графе:

$$\Theta(G) = \bigcup_{v \in V} \left\{ \cos^{-1} \left( \frac{\vec{vu} \cdot \vec{vw}}{|\vec{vu}| \cdot |\vec{vw}|} \right) \mid u, w \in N(v), u \neq w \right\} \quad (9)$$

Среднее значение углов и стандартное отклонение углов:

$$\mu_\theta(G) = \frac{1}{|\Theta(G)|} \sum_{\theta \in \Theta(G)} \theta \quad (10)$$

$$\sigma_\theta(G) = \sqrt{\frac{1}{|\Theta(G)|} \sum_{\theta \in \Theta(G)} (\theta - \mu_\theta(G))^2} \quad (11)$$

Тогда метрику равномерности углов между смежными ребрами определим так:

$$A(G) = \sigma_\theta(G) \quad (12)$$

Для удобного сравнения методов решения этой задачи между собой и для работы алгоритмов, не поддерживающих многокритериальную оптимизацию, введем единую функцию «приспособленности», отображающую пространство решений в  $\mathbb{R}$ :

$$F_{fit} = (I + 1)(D + A) \quad (13)$$

Эта функция приоритизирует минимизацию количества пересечений ребер и решает проблему ненормированного диапазона этой метрики.

Методы будут тестироваться на двух типах графов: произвольных (ПР) и планарных (ПЛ). Псевдокод алгоритма генерации произвольного графа приведен ниже.

**Входные данные:**

- количество\_вершин: общее число вершин в графе
- количество\_ребер: желаемое число ребер

**Выходные данные:**

- Случайный неориентированный граф без петель и кратных ребер

**Шаги:**

**1. Проверка входных параметров:**

```
если количество_ребер < 0 или количество_ребер > количество_вершин ×
(количество_вершин - 1) / 2:
    завершить выполнение с ошибкой «Недопустимое количество ребер»
```

**2. Инициализация:**

```
ребра ← пустой список
множество_ребер ← пустое множество строк
```

**3. Генерация ребер:**

```
пока длина(ребра) < количество_ребер:
```

```
    вершина_1 ← случайное целое от 0 до количество_вершин - 1
    вершина_2 ← случайное целое от 0 до количество_вершин - 1
```

```
    если вершина_1 = вершина_2:
        продолжить (пропустить петлю)
```

```
    (меньшая, большая) ← отсортировать(вершина_1, вершина_2)
    ключ ← строка вида «меньшая–большая»
```

```
    если ключ уже содержится в множество_ребер:
        продолжить (пропустить кратное ребро)
```

```
    добавить ключ в множество_ребер
```

```
    добавить ребро (меньшая, большая) в список ребра
```

**4. Возврат результата:**

```
вернуть граф, содержащий количество_вершин, количество_ребер и список ребер
```

Этот алгоритм генерирует случайный неориентированный граф без петель и кратных ребер с заданным количеством вершин и ребер. С помощью вышеописанного алгоритма был получен граф с 50 вершинами и 150 ребрами (ПР50) (Рисунок 1). Каждой вершине графа назначены случайные координаты внутри единичного квадрата.

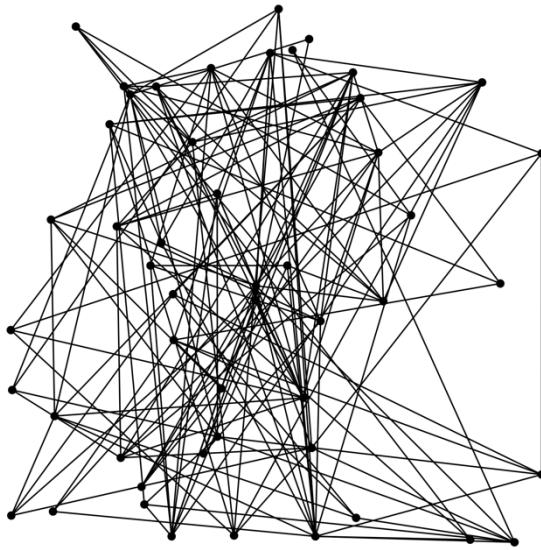


Рисунок 1 – Произвольный граф с 50 вершинами и 150 ребрами (ПР50)

Для генерации планарного графа предлагается воспользоваться методом триангуляции Делоне [4]. Псевдокод алгоритма генерации планарного графа приведен ниже.

**Входные данные:**

– количество\_вершин: число вершин графа

**Выходные данные:**

– Планарный граф с данным числом вершин, построенный на основе триангуляции Делоне

**Шаги:**

1. Сгенерировать множество случайных точек:

```
точки ← []
для i от 1 до количество_вершин:
    x ← случайное_число_от_0_до_1()
    y ← случайное_число_от_0_до_1()
    добавить (x, y) в точки
```

2. Построить триангуляцию Делоне по этим точкам:

```
триангуляция ← delaunay(точки)
если триангуляция неудачна:
    прервать выполнение с ошибкой
```

3. Извлечь уникальные ребра из триангуляции:

```
множество_ребер ← пустое множество
для каждого треугольника в триангуляции:
    для каждой пары соседних вершин (a, b) треугольника:
        если a > b: поменять a и b местами
        добавить (a, b) в множество_ребер
```

4. Преобразовать множество\_ребер в список ребер графа:

```
ребра ← []
для каждого (u, v) в множество_ребер:
    добавить ребро от u до v в ребра
```

5. Вернуть граф со следующими полями:

- количество\_вершин
- количество ребер = длина(ребра)

– список ребер

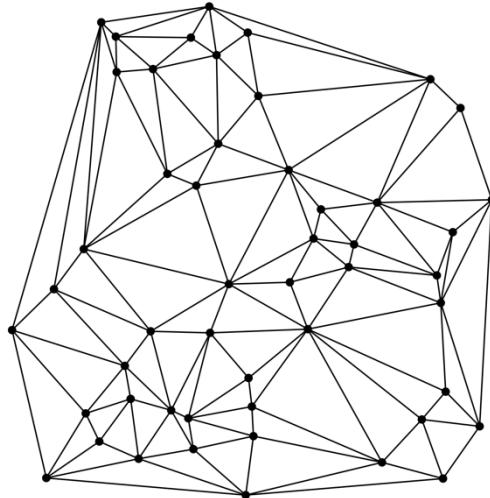


Рисунок 2 – Планарный граф с 50 вершинами (ПЛ50)

С помощью этого алгоритма был получен планарный граф с 50 вершинами (ПЛ50) (Рисунок 2).

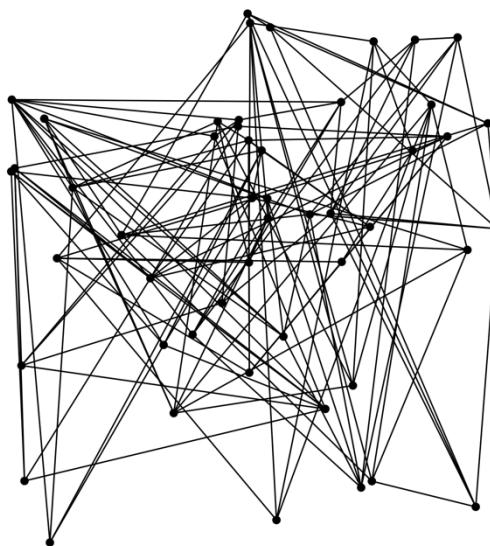


Рисунок 3 – Запутанный ПЛ50

Каждой вершине того же графа были назначены случайные координаты внутри единичного квадрата (Рисунок 3).

При использовании триангуляции Делоне, среднее количество ребер в планарном графе оказывается примерно равным трехкратному количеству вершин. Вследствие чего, для корректности сравнения методов укладки графов на плоскости, предлагается генерировать произвольные графы тоже с количеством ребер, равным количеству вершин, умноженному на 3.

## 2. Набор задач: ZDT1, ZDT2, ZDT3, ZDT4, ZDT6 (Zitzler–Deb–Thiele). [5]

Набор тестовых задач ZDT представляет собой широко используемую коллекцию эталонных задач для оценки эффективности алгоритмов многокритериальной оптимизации. Он включает шесть задач, каждая из которых имеет две цели и предназначена для минимизации. Задачи ZDT моделируют различные аспекты многокритериальных проблем, такие как форма фронта Парето (выпуклая, невыпуклая, разрывная), сложность ландшафта (наличие локальных оптимумов) и тип пространства решений (непрерывное или дискретное).

## 3. Задача о рюкзаке 0-1. [6]

Для  $\forall N, D \in \mathbb{N}; N, D > 1$ : дано  $N$  предметов,  $i$ -ый предмет имеет стоимость  $p_i > 0$  и требуемые ресурсы  $r_{ij} > 0, j \in [1, D - 1]$ . Необходимо выбрать из этих предметов такой набор, чтобы суммарная стоимость была максимальна, а суммарное количество каждого требуемого ресурса не превосходило заданных ограничений  $C_j, j \in [1, D - 1]$ . Каждый предмет есть в единственном экземпляре.

То есть, найти такой набор  $(x_1, \dots, x_N), x_i \in \{0, 1\}$ , при котором:

$$\begin{cases} \sum_{i=1}^N r_{ij} x_i \leq C_j, j \in [1, D - 1] \\ \sum_{i=1}^N p_i x_i \rightarrow \max \end{cases} \quad (14)$$

## 4. Симметричная задача о коммивояжере. [7]

Для данного полного взвешенного графа с  $N$  вершинами найти минимальный гамильтонов цикл. Каждой вершине в графе присвоены координаты  $(0 \leq x_i, y_i \leq 1)$  в двухмерном пространстве. Расстояния между вершинами вычисляются с помощью евклидовой метрики и, соответственно, симметричны.

### 1.2. Описание методов решения тестовых задач

Следующие методы будут использованы для решения тестовых задач.

#### 1. Алгоритм Фрюхтермана-Рейнгольда (ФР). [8]

Этот алгоритм предлагается использовать для укладки графов на плоскости. В его основе лежат силовые взаимодействия. Идея алгоритма заключается в моделировании сил между всеми парами вершин графа. В данной модели вершины представляют собой стальные кольца, а ребра — пружины, соединяющие эти кольца. Притягивающая сила действует подобно силе упругости пружины, а отталкивающая сила аналогична электрическому

отталкиванию между зарядами. Алгоритм стремится минимизировать суммарную энергию системы путем перемещения вершин и перераспределения сил между ними.

Ключевое отличие ФР от ГА заключается в его детерминированной природе и локальной оптимизации путем спуска по энергетическому ландшафту. Он эффективно находит локальные оптимумы, но может застревать в них для сложных графов.

Пусть  $G = (V, E)$  – неориентированный граф. Каждой вершине  $v \in V$  приписывается позиция на плоскости  $\vec{p}_v = (x_v, y_v) \in \mathbb{R}^2$ . Пусть  $k > 0$  – идеальное расстояние между вершинами, а  $T$  – текущая температура (ограничение максимального перемещения за итерацию). Причем, обычно  $k = \sqrt{\frac{W \cdot H}{|V|}}$ , где  $W$  и  $H$  – ширина и высота области размещения соответственно. Сила отталкивания действует между всеми парами вершин:

$$\begin{aligned} \forall u, v \in V, u \neq v: \\ \vec{D}_{uv} &= p_u - p_v \\ \vec{F}_{rep}(u, v) &= \frac{k^2}{\|\vec{D}_{uv}\|^2} \cdot \frac{\vec{D}_{uv}}{\|\vec{D}_{uv}\|} \end{aligned} \quad (15)$$

Сила притяжения действует по ребрам:

$$\begin{aligned} \forall \{u, v\} \in E: \\ \vec{D}_{uv} &= p_u - p_v \\ \vec{F}_{att}(u, v) &= -\frac{\|\vec{D}_{uv}\|^2}{k} \cdot \frac{\vec{D}_{uv}}{\|\vec{D}_{uv}\|} \end{aligned} \quad (16)$$

Тогда суммарная сила, действующая на вершину:

$$\vec{F}_u = \sum_{v \in V, v \neq u} \vec{F}_{rep}(u, v) + \sum_{v \in \mathcal{N}(u)} \vec{F}_{att}(u, v) \quad (17)$$

Где  $\mathcal{N}(u)$  – множество смежных  $u$  вершин. Позиция вершин обновляется следующим образом:

$$\Delta \vec{p}_u = \min(\|\vec{F}_u\|, T) \cdot \frac{\vec{F}_u}{\|\vec{F}_u\|}, \quad \vec{p}_u' = \vec{p}_u + \Delta \vec{p}_u \quad (18)$$

После каждого шага алгоритма  $T$  уменьшается на заданную величину  $\delta T$  – шаг охлаждения.

Псевдокод алгоритма приведен ниже.

**Входные данные:**

граф: множество вершин и ребер  
 позиции: начальные координаты всех вершин  
 ширина, высота: размеры области отображения  
 начальная\_температура: начальное значение температуры  
 количество\_шагов: количество итераций алгоритма  
 константа\_k: управляющая константа для силы притяжения и отталкивания

**Выходные данные:**

## Обновленные координаты вершин графа

Шаги:

1. Инициализация:

для каждой вершины  $i$ :  
смещение $[i] \leftarrow (0, 0)$

2. Вычисление отталкивающих сил:

для каждой пары различных вершин  $(i, j)$ :  
расстояние  $\leftarrow$  расстояние\_между( $i, j$ )  
сила  $\leftarrow$  (константа\_k) $^2 /$  расстояние $^2$   
направление  $\leftarrow$  единичный\_вектор\_от( $j$  к  $i$ )  
смещение $[i] \leftarrow$  смещение $[i]$  + направление \* сила  
смещение $[j] \leftarrow$  смещение $[j]$  - направление \* сила

3. Вычисление притягивающих сил:

для каждого ребра  $(i, j)$  в графе:  
расстояние  $\leftarrow$  расстояние\_между( $i, j$ )  
сила  $\leftarrow$  расстояние $^2 /$  константа\_k  
направление  $\leftarrow$  единичный\_вектор\_от( $i$  к  $j$ )  
смещение $[i] \leftarrow$  смещение $[i]$  - направление \* сила  
смещение $[j] \leftarrow$  смещение $[j]$  + направление \* сила

4. Обновление координат вершин:

для каждой вершины  $i$ :  
модуль\_смещения  $\leftarrow$  длина\_вектора(смещение $[i]$ )  
если модуль\_смещения  $> 0$ :  
ограниченное\_смещение  $\leftarrow$  смещение $[i] * \min(\text{модуль\_смещения}, \text{температура}) /$   
модуль\_смещения  
новая\_позиция  $\leftarrow$  позиция $[i] +$  ограниченное\_смещение  
позиция $[i] \leftarrow$  обрезать\_в\_границах(новая\_позиция, ширина, высота)

5. Охлаждение системы:

шаг\_охлаждения  $\leftarrow$  начальная\_температура / количество\_шагов  
температура  $\leftarrow$  температура - шаг\_охлаждения

2. Динамическое программирование.

Метод динамического программирования предлагается использовать для решения задачи о рюкзаке 0-1 при  $D = 2$ . Двухмерность означает, что каждый предмет обладает стоимостью и требует затрат только одного ресурса (масса).

Он отличается от ГА тем, что гарантирует нахождение оптимального глобального решения за счёт построения таблицы всех возможных промежуточных состояний. Это его ключевое преимущество перед эвристическими алгоритмами, но оно достигается за счет высокой вычислительной сложности по памяти и времени, что делает его непрактичным для больших входных данных.

Пусть  $A(k, s)$  - максимальная суммарная стоимость предметов, которые можно уложить в рюкзак вместимости  $s$ , если можно использовать только первые  $k$  предметов:  $\{n_1, n_2, \dots, n_k\}$ . Назовем это набором допустимых предметов для  $A(k, s)$ .

$$A(k, 0) = 0$$

$$A(0, s) = 0$$

Найдем  $A(k, s)$ . Возможны 2 варианта.

Если предмет  $k$  не попал в рюкзак. Тогда  $A(k, s)$  равно максимальной стоимости рюкзака с такой же вместимостью и набором допустимых предметов  $\{n_1, n_2, \dots, n_{k-1}\}$ , то есть  $A(k, s) = A(k - 1, s)$ .

Если  $k$  попал в рюкзак. Тогда  $A(k, s)$  равно максимальной стоимости рюкзака, где вес  $s$  уменьшаем на вес  $k$ -ого предмета и набор допустимых предметов  $\{n_1, n_2, \dots, n_{k-1}\}$  плюс стоимость  $k$ , то есть  $A(k - 1, s - r_{k1}) + p_k$ .

$$A(k, s) = \begin{cases} A(k - 1, s), & b_k = 0 \\ A(k - 1, s - r_{k1}) + p_k, & b_k = 1 \end{cases}$$

То есть

$$A(k, s) = \max(A(k - 1, s), A(k - 1, s - r_{k1}) + p_k)$$

Стоимость искомого набора равна  $A(N, C_1)$ , так как нужно найти максимальную стоимость рюкзака, где все предметы допустимы и вместимость рюкзака  $C_1$ .

Псевдокод алгоритма приведен ниже.

Входные данные:

$n$ : количество предметов

$W$ : вместимость рюкзака

$\text{вес}[1..n]$ : массив весов предметов

$\text{ценность}[1..n]$ : массив ценностей предметов

Выходные данные:

$X[1..n]$ : бинарный вектор выбранных предметов (1 – выбран, 0 – не выбран)

Шаги:

1. Инициализация таблицы:

Создать таблицу  $D[0..n][0..W]$ , где  $D[i][w]$  – максимальная ценность для первых  $i$  предметов и веса  $w$ .

Для всех  $w$  от 0 до  $W$ :

$D[0][w] \leftarrow 0$

Для всех  $i$  от 0 до  $n$ :

$D[i][0] \leftarrow 0$

2. Заполнение таблицы:

Для  $i$  от 1 до  $n$ :

Для  $w$  от 1 до  $W$ :

Если  $\text{вес}[i] \leq w$ :

$\text{ценность\_с\_предметом} \leftarrow \text{ценность}[i] + D[i-1][w - \text{вес}[i]]$

$\text{ценность\_без\_предмета} \leftarrow D[i-1][w]$

$D[i][w] \leftarrow \max(\text{ценность\_с\_предметом}, \text{ценность\_без\_предмета})$

Иначе:

$D[i][w] \leftarrow D[i-1][w]$

3. Восстановление решения:

$X[1..n] \leftarrow$  массив нулей размером  $n$

$w \leftarrow W$

Для  $i$  от  $n$  вниз до 1:

Если  $D[i][w] \neq D[i-1][w]$ :

$X[i] \leftarrow 1$

$w \leftarrow w - \text{вес}[i]$

4. Возврат результата:

вернуть X

### 3. Полный перебор.

Метод полного перебора предлагается использовать для решения симметричной задачи коммивояжера для небольшого ( $\leq 12$ ) количества вершин.

Метод генерирует все перестановки вершин в графе и для каждой из них вычисляет длину цикла, проходящего по ним в выбранном порядке, выбирая перестановку с минимальной длиной пути.

Ключевое отличие от ГА и метода ветвей и границ заключается в отсутствии какой-либо оптимизации поиска – перебираются абсолютно все варианты. Это делает его крайне неэффективным для задач с большим количеством вершин из-за факториальной вычислительной сложности  $O(N!)$ .

Псевдокод алгоритма приведен ниже.

**Входные данные:**

n: количество городов (включая стартовый)  
стартовый\_город: начальная точка (обычно 0)  
d[i][j]: матрица расстояний между городами i и j

**Выходные данные:**

лучший\_маршрут: массив порядка посещения городов

**Шаги:**

1. Инициализация:

```
города_для_перебора ← [1, 2, ..., n-1]
лучший_маршрут ← [0] + города_для_перебора
текущая_длина ← сумма расстояний маршрута [0→1→2→...→n-1→0]
```

2. Перебор всех перестановок:

```
для каждой перестановки order в permutations(города_для_перебора):
    текущий_маршрут ← [0] + order
    длина ← d[0][order[0]]
    для i от 0 до len(order)-1:
        длина += d[order[i]][order[i+1]]
    длина += d[order[-1]][0]
    если длина < текущая_длина:
        лучший_маршрут ← текущий_маршрут
        текущая_длина ← длина
```

3. Возврат результата:

```
вернуть лучший_маршрут
```

4. Метод ветвей и границ. [9]

Метод ветвей и границ предлагается использовать для решения симметричной задачи коммивояжера при количестве вершин, не превышающем 20. Этот метод является улучшением полного перебора: он также перебирает возможные маршруты, но использует эвристическую оценку (нижнюю границу стоимости) для отсечения заведомо невыгодных путей, что значительно сокращает пространство поиска. Основная идея заключается в

следующем: если частичный маршрут не может привести к лучшему результату, чем уже найденный, он отбрасывается (принцип отсечения ветвей).

Псевдокод алгоритма приведен ниже.

Входные данные:

n: количество городов (включая стартовый)  
стартовый\_город: начальная точка (обычно 0)  
d[i][j]: матрица расстояний между городами i и j

Выходные данные:

лучший\_маршрут: массив порядка посещения городов

Шаги:

1. Инициализация:

лучший\_маршрут  $\leftarrow$  nil  
лучшая\_стоимость  $\leftarrow \infty$   
 $\min_d[i] \leftarrow$  минимальное расстояние от города i до любого другого  
(предвычисление)

2. Определение рекурсивной функции dfs(current, count, cost, маршрут, посещенные):

Если  $count == n$ :  
    total\_cost  $\leftarrow$  cost +  $d[current][0]$   
    если total\_cost < лучшая\_стоимость:  
        обновить лучший\_маршрут и лучшую\_стоимость  
    вернуть  
    Вычислить нижнюю\_границу:  
        lb  $\leftarrow$  cost +  $\min_d[current] + \sum(\min_d[i] \text{ для непосещенных } i)$   
    Если lb  $\geq$  лучшая\_стоимость:  
        прервать ветку (отсечь)  
    Иначе:  
        для каждого непосещенного города i:  
             отметить i как посещенный  
             вызвать dfs(i, count+1, cost +  $d[current][i]$ , маршрут + [i], посещенные)  
             снять отметку посещения города i

3. Вызов dfs(0, 1, 0, [0], [False]\*n, где посещенные[0] = True)

4. Возврат результата:

вернуть лучший\_маршрут

5. Simple Genetic Algorithm (SGA). [10]

SGA представляет собой базовую версию ГА. Процесс начинается с формирования начальной популяции решений, после чего каждый кандидат оценивается по заранее определенной функции приспособленности. При отсутствии удовлетворяющего решения происходит переход к эволюционным операциям: отбору элиты, скрещиванию и мутации, что позволяет постепенно улучшать качество решений.

Псевдокод алгоритма приведен ниже.

Входные данные:

популяция: текущая популяция решений  
размер\_популяции: целевой размер новой популяции  
размер\_элиты: количество сохраняемых лучших решений  
размер\_пула\_родителей: диапазон выбора родителей (от 0 до N-1)  
функция\_скрещивания: оператор создания потомков из двух родителей

**функция\_мутации:** оператор модификации потомка

**Выходные данные:**

Обновленная популяция решений

**Шаги:**

1. **Оценка поколения:**  
отсортировать популяцию по ухудшению приспособленности (Fitness)
2. **Инициализация новой популяции:**  
новая\_популяция  $\leftarrow$  первые размер\_элиты элементов из популяция
3. **Генерация новых решений:**  
пока длина(новая\_популяция) < размер\_популяции:  
    индекс\_родителя1  $\leftarrow$  случайное\_число(от 0 до размер\_пула\_родителей - 1)  
    индекс\_родителя2  $\leftarrow$  случайное\_число(от 0 до размер\_пула\_родителей - 1)  
    если индекс\_родителя1 == индекс\_родителя2:  
        пропустить итерацию (т.е. продолжить цикл)  
        родитель1  $\leftarrow$  текущая\_популяция [индекс\_родителя1]  
        родитель2  $\leftarrow$  текущая\_популяция [индекс\_родителя2]  
        потомки  $\leftarrow$  функция\_скрещивания(родитель1, родитель2)  
        для каждого потомок в потомки:  
            модифицированный\_потомок  $\leftarrow$  функция\_мутации(потомок)  
            добавить модифицированный\_потомок в новая\_популяция  
        если длина(новая\_популяция)  $\geq$  размер\_популяции:  
            прервать цикл
4. **Обновление популяции:**  
популяция  $\leftarrow$  новая\_популяция

## 6. Steady State Genetic Algorithm (SSGA). [11]

«Steady State» означает отсутствие поколений. Отличается от SGA тем, что вместо добавления решений-потомков в популяцию следующего поколения заменяет два старых решения на два лучших решения из группы двух родителей и двух их потомков, сохраняя размер популяции и сохраняя большее разнообразие.

Псевдокод алгоритма приведен ниже.

**Входные данные:**

популяция: текущая популяция решений

размер\_популяции: общее количество решений

функция\_скрещивания: оператор создания потомков

функция\_мутации: оператор модификации решений

турнирный\_отбор: метод выбора родителей

**Выходные данные:**

Обновленная популяция решений

**Шаги:**

1. **Оценка поколения:**  
отсортировать популяцию по убыванию приспособленности
2. **Цикл обновления:**  
пока не будет произведена замена (или определенное количество итераций):  
    индекс\_родителя1  $\leftarrow$  турнирный\_отбор(популяция)  
    индекс\_родителя2  $\leftarrow$  турнирный\_отбор(популяция)  
    если индекс\_родителя1 == индекс\_родителя2:  
        продолжить цикл

```

родитель1 ← популяция [индекс_родителя1]
родитель2 ← популяция [индекс_родителя2]

потомки ← функция_скрещивания(родитель1, родитель2)

для каждого i в диапазоне длины(потомки):
    модифицированный_потомок ← функция_мутации(потомки[i])
    // Замена наихудших особей или случайных особей в популяции
    // В данном псевдокоде: замена двух наихудших особей
    позиция_замены ← размер_популяции - i - 1
    популяция [позиция_замены] ← модифицированный_потомок
    замена_произведена ← истина // Устанавливаем флаг, если замена
    произведена

```

3. Возврат результата:  
вернуть популяция

## 7. Non-dominated Sorting Genetic Algorithm 2 (NSGA2). [12]

NSGA2 является одним из самых известных и эффективных алгоритмов для решения многокритериальных оптимизационных задач. Это улучшенный многоцелевой эволюционный алгоритм, основанный на не-доминирующей сортировке. В отличие от предыдущих алгоритмов, он обладает меньшей вычислительной сложностью  $O(MN^2)$  (где  $M$  – количество целей,  $N$  – размер популяции), использует элитизм (сохраняет лучшие решения) и эффективно поддерживает разнообразие. Эти улучшения делают NSGA2 более эффективным по сравнению с такими алгоритмами, как PAES и SPEA, особенно в задачах с несколькими противоречивыми целями.

Псевдокод алгоритма NSGA2 приведен ниже.

**Входные данные:**

начальная\_популяция: текущая популяция решений (может быть пустой)  
размер\_популяции: количество решений в каждом поколении  
лимит\_поколений: максимальное число поколений

**Выходные данные:**

Последняя популяция

**Шаги:**

1. Инициализация:

если размер(начальная\_популяция) < размер\_популяции:  
 начальная\_популяция ← случайная\_популяция(размер\_популяции)  
 номер\_поколения ← 0

2. Основной цикл:

пока номер\_поколения < лимит\_поколений:  
 номер\_поколения ← номер\_поколения + 1

потомки ← создать\_потомков(начальная\_популяция) // с помощью кроссовера и мутации

объединенная\_популяция ← начальная\_популяция + потомки

фронты ← быстрая\_не\_доминирующая\_сортировка(объединенная\_популяция)

новая\_популяция ← []

```

для каждого фронт в фронты:
    вычислить_расстояние_разреженности(фронт) // Crowding Distance
    если длина(новая_популяция) + длина(фронт) > размер_популяции:
        отсортировать фронт по убыванию расстояния разреженности
        оставшиеся ← размер_популяции – длина(новая_популяция)
        добавить первые оставшиеся особи из фронт в новая_популяция
        прервать цикл
    иначе:
        добавить все особи из фронт в новая_популяция
начальная_популяция ← новая_популяция

```

**3. Возврат результата:**  
вернуть начальная\_популяция

В основе метода NSGA2 лежит алгоритм не-доминирующей сортировки. Рассмотрим его псевдокод.

**Входные данные:**

популяция: список особей (индивидуумов) с векторными решениями

**Выходные данные:**

фронты: список списков особей, отсортированных по уровням не-доминирования

**Шаги:**

**1. Инициализация:**

```

фронты ← пустой список
размер ← количество особей в популяции
количество_доминирующих ← массив размером размер, заполненный нулями
кого_доминирует ← массив размером размер, каждый элемент – пустой список

```

**2. Подсчет доминирования:**

```

для каждой i от 0 до размер – 1:
    для каждой j от 0 до размер – 1:
        если i == j:
            продолжить
        если решение(i) доминирует решение(j):
            добавить j в кого_доминирует[i]
        иначе если решение(j) доминирует решение(i):
            увеличить количество_доминирующих[i] на 1

```

**3. Формирование первого фронта:**

```

первый_фронт_индексы ← пустой список
для каждой i от 0 до размер – 1:
    если количество_доминирующих[i] == 0:
        установить ранг особи i как 0
        добавить i в первый_фронт_индексы

```

**4. Основной цикл построения фронтов:**

```

текущий_ранг ← 0
пока первый_фронт_индексы не пуст:
    текущий_фронт_особы ← пустой список особей
    следующий_фронт_индексы ← пустой список индексов

```

```

для каждого индекса i в первый_фронт_индексы:
    установить ранг особи i как текущий_ранг
    добавить особь i в текущий_фронт_особы

```

```

для каждого j в кого_доминирует[i]:
    уменьшить количество_доминирующих[j] на 1
    если количество_доминирующих[j] == 0:

```

```

    добавить j в следующий_фронт_индексы

    добавить текущий_фронт_особи в список фронтов
    текущий_ранг ← текущий_ранг + 1
    первый_фронт_индексы ← следующий_фронт_индексы

5. Возврат результата:
    вернуть фронты

5. Возврат результата
    вернуть фронты

```

При этом важно дать определение доминации одного решения над другим. Пусть  $a = (a_1, a_2, \dots, a_m)$  и  $b = (b_1, b_2, \dots, b_m)$  – векторы значений целевых функций для решений  $a$  и  $b$  соответственно, при этом задачи считаются задачами минимизации. Тогда говорят, что решение  $a$  доминирует решение  $b$  (обозначается как  $a \prec b$ ), если выполняются два условия:

$$\begin{cases} \forall i \in \{1, \dots, m\} a_i \leq b_i, \\ \exists j \in \{1, \dots, m\} a_j < b_j. \end{cases} \quad (19)$$

Иными словами,  $a$  не хуже  $b$  по всем критериям и строго лучше хотя бы по одному из них.

## 8. Strength Pareto Evolutionary Algorithm 2 (SPEA2). [13]

SPEA2 является улучшенной версией оригинального алгоритма SPEA и предназначен для эффективного решения многокритериальных оптимизационных задач. Одной из ключевых особенностей SPEA2 является использование внешнего архива для сохранения лучших недоминированных решений, а также детальная оценка приспособленности с учетом как доминирования, так и плотности расположения решений.

Псевдокод алгоритма приведен ниже.

**Входные данные:**

начальная\_популяция: исходный набор решений  
размер\_популяции: количество решений в каждом поколении  
размер\_архива: число элитных решений, хранящихся между поколениями  
лимит\_поколений: максимальное число итераций  
k\_для\_плотности: параметр для оценки плотности (k-Nearest Neighbors)  
оператор\_скрещивания: функция скрещивания двух решений  
оператор\_мутации: функция мутации одного решения  
оператор\_турнира: бинарный турнирный отбор

**Выходные данные:**

Архив элитных решений

**Шаги:**

1. Инициализация:  
начальная\_популяция ← сгенерировать\_случайно()  
архив ← пусто
2. Основной цикл:  
для поколения от 1 до лимит\_поколений:

```

объединенная_популяция ← начальная_популяция + архив

для каждого решения i в объединенной_популяции:
    сила[i] ← число решений, которые доминируют i

для каждого решения i в объединенной_популяции:
    сырья_приспособленность[i] ← сумма сила[j] по всем j, доминирующими i

для каждого решения i в объединенной_популяции:
    расстояние_до_k_соседа ← computeKthDist(объединенная_популяция, i,
k_для_плотности)
    плотность[i] ← 1 / (расстояние_до_k_соседа + 2) // +2, чтобы избежать
деления на ноль

для каждого решения i в объединенной_популяции:
    итоговая_приспособленность[i] ← сырья_приспособленность[i] + плотность[i]

архив_претендентов ← выбрать все решения из объединенной_популяции с
сырой_приспособленностью < 1 (недоминированные решения)
архив ← очистить

если размер(архив_претендентов) <= размер_архива:
    добавить все решения из архив_претендентов в архив
иначе: // Размер архива_претендентов > размер_архива
    отсортировать архив_претендентов по возрастанию плотности
    добавить первые размер_архива решений из архив_претендентов в архив

потомки ← []
пока длина(потомки) < размер_популяции:
    родитель1 ← оператор_турнира(архив)
    родитель2 ← оператор_турнира(архив)

    дети ← оператор_скрещивания(родитель1, родитель2)

    для каждого ребенок в дети:
        мутант ← оператор_мутации(ребенок)
        добавить мутант в потомки
        если длина(потомки) >= размер_популяции:
            прервать

    начальная_популяция ← потомки

```

3. Возврат результата:  
вернуть архив

### **1.3. Анализ результатов решения тестовых задач с помощью выбранных методов**

1. Задача о расположении вершин графа на плоскости.

Для работы ГА необходимо определить генетические операторы скрещивания и мутации. Для решения задачи о расположении вершин графа на плоскости в этом разделе для всех ГА будут использоваться операторы равномерного скрещивания с коэффициентом 0.5 [14] и равномерной мутации. Подробнее они описаны в [подразделе 2.1](#).

Так же, для всех ГА была взята одинаковая величина размера популяции в 500 особей. Основываясь на предварительных тестах, для SGA были выбраны размеры элиты и пула

родителей, равные 10% и 50% популяции соответственно. Аналогично, для SPEA2 размер архива был выбран равным размеру популяции, а параметр  $k$  был выбран по следующей формуле [13]:

$$k = \lfloor \sqrt{p + a} \rfloor, \quad (20)$$

где  $p$  – размер популяции,  $a$  – размер архива

Поскольку укладка графа состоит из точек с вещественными координатами, для решения этой задачи подходит вещественное кодирование генома. Было предложено несколько моделей вещественного кодирования [15]. Для применения ГА к этой задаче воспользуемся одним из предложенных методов [16], адаптированным под конкретную формулировку.

Укладка графа представлена хромосомой – последовательностью точек в двухмерном пространстве. Длина этой последовательности равна количеству вершин укладываемого графа. Каждая точка описывается двумя генами, принимающими вещественные значения на отрезке  $[0, 1]$ . Все гены инициализируются случайными значениями в допустимом диапазоне.

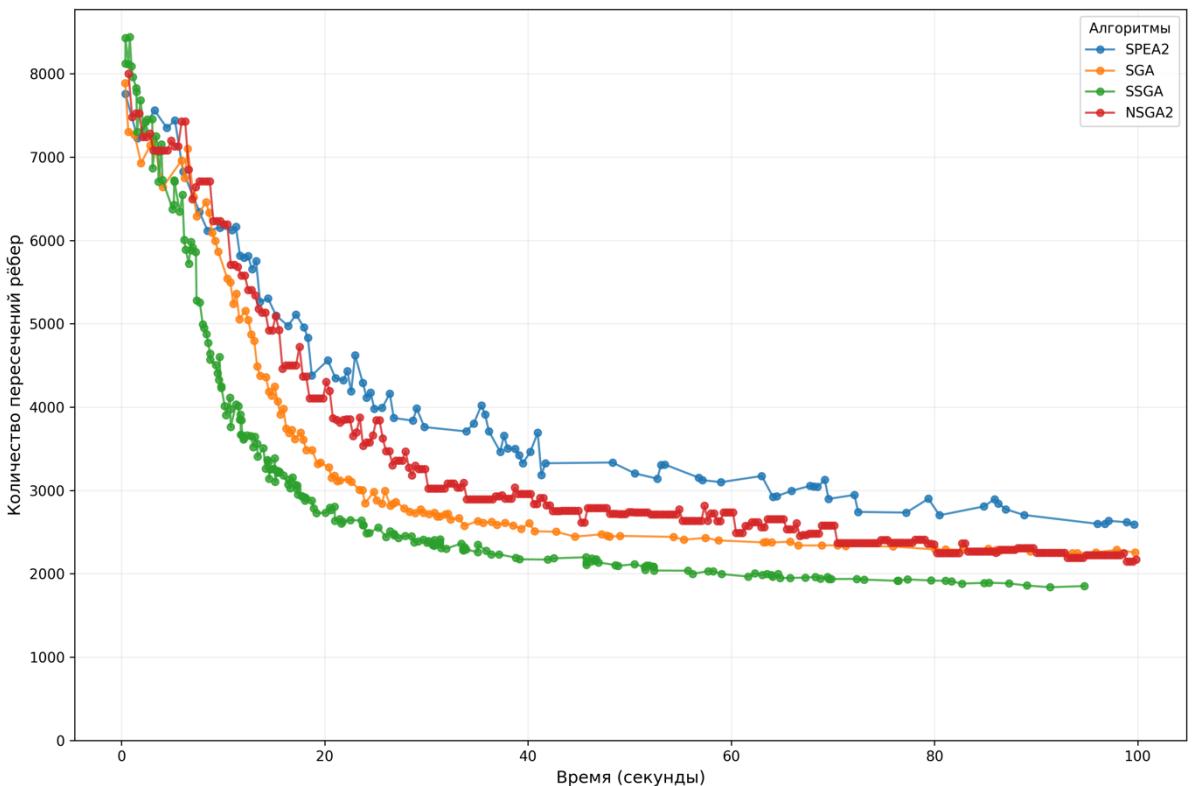


Рисунок 4 – Зависимость количества пересечений ребер от времени для ГА (ПР100)

Поскольку используемые ГА совершают разное количество вычислений на каждом своем шаге, при их сравнении между собой имеет смысл строить зависимость достигнутого результата от затраченного алгоритмом на выполнение времени, а не от количества выполненных шагов.

Для всех четырех описанных ГА показана зависимость метрики количества пересечений ребер в графе для лучшего решения алгоритма от времени (Рисунок 4). Где «лучшее решение» означает решение с наименьшим значением функции приспособленности  $F_{fit}$  в популяции. Как видно из графика, эти функции не являются монотонно убывающими, потому что оптимизация функции приспособленности необязательно приводит к оптимизации каждой отдельной метрики. Представлен график для процесса укладки произвольного графа со 100 вершинами (Рисунок 4).

Аналогичный график построен для планарного графа с 200 вершинами (Рисунок 5). Можно заметить, что для планарных графов алгоритмам, ожидаемо, удается снизить количество пересечений в большее число раз относительно изначальных значений.

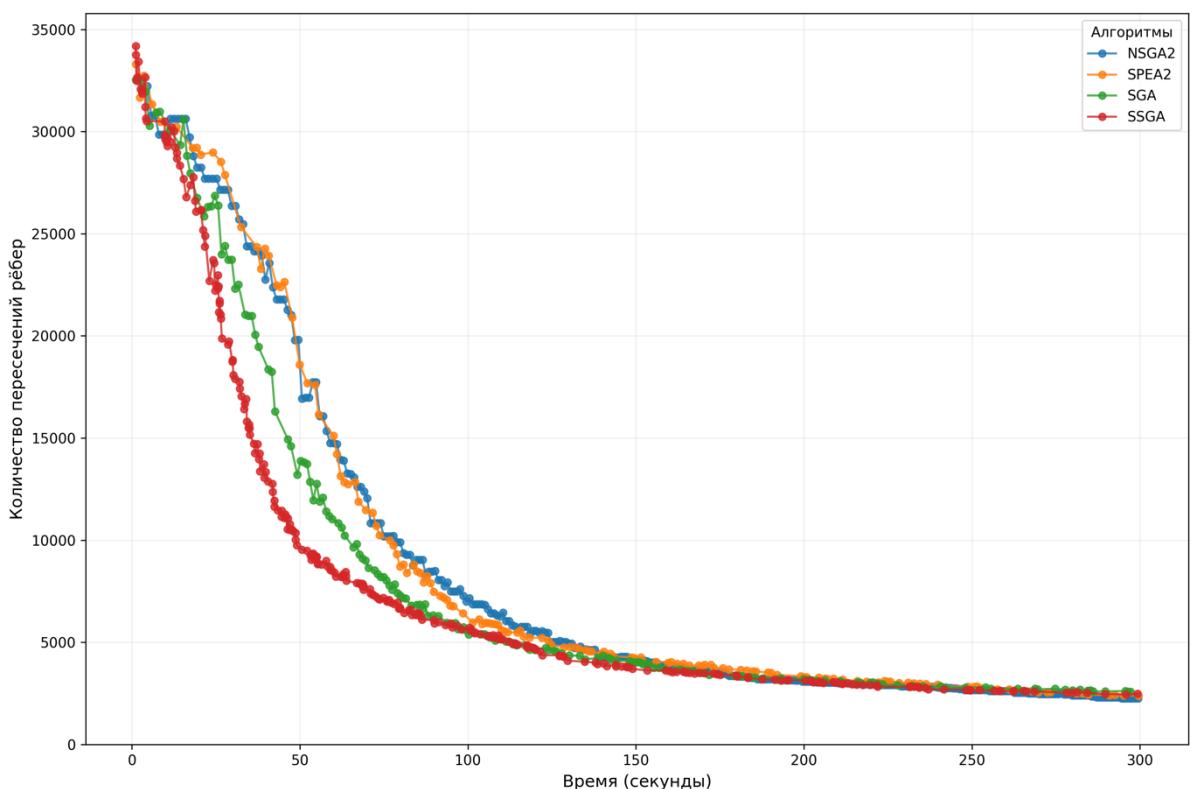


Рисунок 5 – Зависимость количества пересечений ребер от времени для ГА (ПЛ200)

Так же получены графики для планарных и произвольных графов с 25, 50, 100 и 200 вершинами. Они представлены в [приложении](#).

Показан результат укладки планарного графа с 50 вершинами с помощью SSGA на протяжении 115000 шагов (Рисунок 6). Слева – лучшее решение на 1 шаге алгоритма с 1353 пересечениями ребер, справа – лучшее решение, достигнутое алгоритмом с 87 пересечениями ребер.

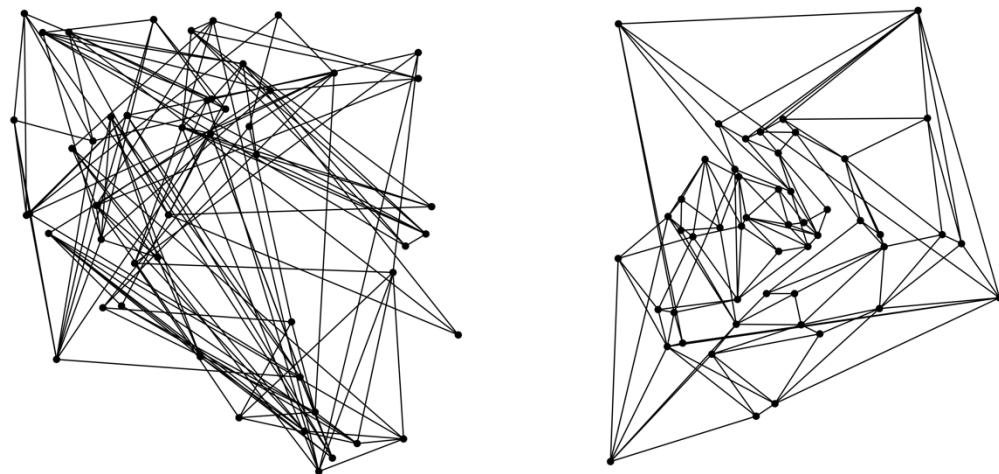


Рисунок 6 – Укладка ПЛ50 с помощью SSGA

Поскольку время выполнения алгоритма зависит от внешних факторов, для обеспечения воспроизводимости результатов количество шагов алгоритма было использовано в качестве устойчивой метрики вместо времени. Значения этого параметра для алгоритмов и графов разного размера были подобраны эмпирически на основе анализа графиков, которые показывают, что за выбранное количество шагов укладка графа, как правило, стабилизируется (Рисунок 4, Рисунок 5, Рисунок 23, Рисунок 24, Рисунок 25, Рисунок 26, Рисунок 27, Рисунок 28). Таблица 1 содержит полученные значения для графов с 25, 50, 100 и 200 вершинами.

Таблица 1 – Количество шагов алгоритмов для размеров графов

Алгоритм	25	50	100	200
SGA	600	500	450	370
SSGA	125000	115000	100000	85000
NSGA2	300	360	380	330
SPEA2	150	220	320	310

По таблице видно, что комплексные алгоритмы NSGA2 и SPEA2 более эффективно используют свои шаги, добиваясь сопоставимых результатов за меньшее их количество. С другой стороны, каждый шаг этих алгоритмов выполняется дольше, чем у более простых SGA и SSGA. Так же можно заметить, что количество шагов каждого алгоритма, требуемых для того, чтобы укладка графа стабилизировалась практически не зависит от размера графа.

Графики позволяют сравнить работу алгоритмов в частных случаях. Для более качественного сравнения были вычислены усредненные значения.

Таблица 2 – Среднее количество пересечений ребер (ГА)

Алгоритм	ПЛ25	ПР25	ПЛ50	ПР50	ПЛ100	ПР100	ПЛ200	ПР200
SGA	1.02	81.66	29.74	431.44	366.50	2139.67	2539.22	10571.22
SSGA	0.68	79.42	26.96	388.78	304.02	2021.22	2181.33	10277.78
NSGA2	0.64	85.60	31.56	441.78	288.20	2280.44	2105.44	10326.22
SPEA2	0.84	97.52	46.96	450.56	385.00	2509.89	2433.22	10996.11

Таблица 2 содержит средние значения метрики количества пересечений ребер в графе для лучшего решения алгоритмов. Значения приведены для планарных и произвольных графов с 25, 50, 100 и 200 вершинами.

Можно заметить, что с ростом количества вершин в 2 раза, полученные значения количества пересечений растут примерно на порядок. Более того, совпадает порядок величин для произвольных графов и планарных с вдвое большим количеством вершин.

Таблица 3 содержит доли случаев, когда планарный граф удавалось полностью «распутать» – получить планарную укладку с 0 пересечений ребер.

Значения в обеих таблицах получены с помощью усреднения по 50 тестам для каждого алгоритма и типа графов.

Таблица 3 – Доля распутанных решений (ГА)

Алгоритм	ПЛ25	ПЛ50	ПЛ100	ПЛ200
SGA	90%	34%	0%	0%
SSGA	94%	38%	0%	0%
NSGA2	94%	24%	0%	0%
SPEA2	92%	2%	0%	0%

Кроме ГА, для решения задачи укладки графов был использован ФР. Основываясь на предварительных тестах, для него были выбраны значения параметров количества шагов и начальной температуры равные 2000 и 0.005 соответственно. Параметр  $k$  был выбран отдельно для каждого типа графов так, чтобы в результате работы алгоритма полученная укладка занимала большую часть доступной области, но не упиралась в стенки. Таблица 4 содержит выбранные значения.

Таблица 4 – Параметр  $k$  алгоритма ФР для графов

ПЛ25	ПР25	ПЛ50	ПР50	ПЛ100	ПР100	ПЛ200	ПР200
0.7	1.0	0.6	0.95	0.5	0.9	0.4	0.85

Тесты показали, что конкретные значения параметров алгоритма не оказывают большого влияния на количество пересечений ребер в получаемой укладке, если вершины графа не достигают границ заданной области и алгоритм успевает прийти к равновесному состоянию, на что обычно требуется порядка 1000 шагов.

Представлены результаты укладки графов с 50 вершинами с помощью ФР: слева – произвольного, справа – планарного (Рисунок 7).

Таблица 5 содержит средние значения метрики количества пересечений ребер в графе для укладок, полученных с помощью ФР с вышеописанными параметрами. Значения приведены для планарных и произвольных графов с 25, 50, 100 и 200 вершинами.

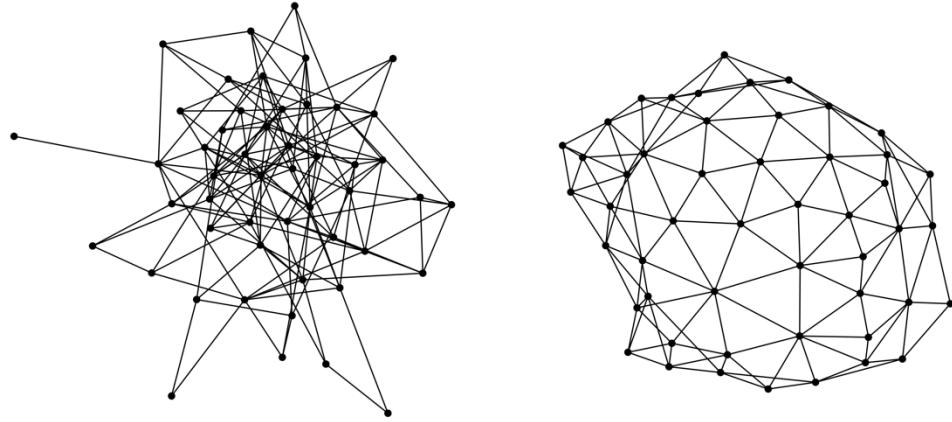


Рисунок 7 – Укладки графов с 50 вершинами с помощью ФР

Таблица 5 – Среднее количество пересечений ребер (ФР)

Алгоритм	ПЛ25	ПР25	ПЛ50	ПР50	ПЛ100	ПР100	ПЛ200	ПР200
ФР	17.04	170.72	50.56	649.68	142.78	2569.40	357.46	10239.40

Эффективность данного алгоритма заметно возрастает с увеличением размера графа.

Хотя на графах размером 25 и 50 вершин он уступает ГА по качеству результатов, на 100 и 200 вершинах их результаты для произвольных графов становятся сопоставимыми. При этом для планарных графов на больших размерах алгоритм демонстрирует значительное преимущество. Примечательно, что подобно ГА, количество шагов, требуемых ФР, практически не зависит от размера графа. Однако выполняется он существенно быстрее, особенно на больших графах.

Как видно из таблицы, количество пересечений в планарных графах растет линейно с увеличением числа вершин (удвоение вершин приводит примерно к двухкратному-трехкратному росту пересечений). Это объясняется следующим: алгоритм успешно минимизирует пересечения внутри графа, однако на краях укладки возникает дисбаланс сил, притягивающий вершины внутрь и создающий дополнительные пересечения (Рисунок 7). Поскольку размер краевой зоны пропорционален общему размеру графа, рост пересечений также носит линейный характер.

Таблица 6 – Доля распутанных решений (ФР)

Алгоритм	ПЛ25	ПЛ50	ПЛ100	ПЛ200
ФР	0.7%	0.0%	0.0%	0.0%

Таблица 6 содержит доли случаев, когда планарный граф удавалось полностью «распутать» – достичь планарной укладки без пересечений ребер.

Значения в обеих таблицах получены с помощью усреднения по 1000 тестам для каждого типа графов.

Сопоставив значения в таблицах, можно сделать вывод о том, что среди ГА SSGA и NSGA2 лучше всего подходят для решения задачи размещения вершин графа на плоскости, не забывая о том, что из них только NSGA2 является многоцелевым. ФР показывает многообещающие результаты, превосходящие ГА, на больших планарных графах.

## 2. Набор задач ZDT.

Для задач многокритериальной оптимизации из этого набора заранее известны фронты Парето, поэтому на них удобно проверять корректность используемого оптимизационного алгоритма. В данной работе тесты проводились на задачах набора в пространстве размерности 30. У всех ГА была одинаковая величина популяции в 100 особей.

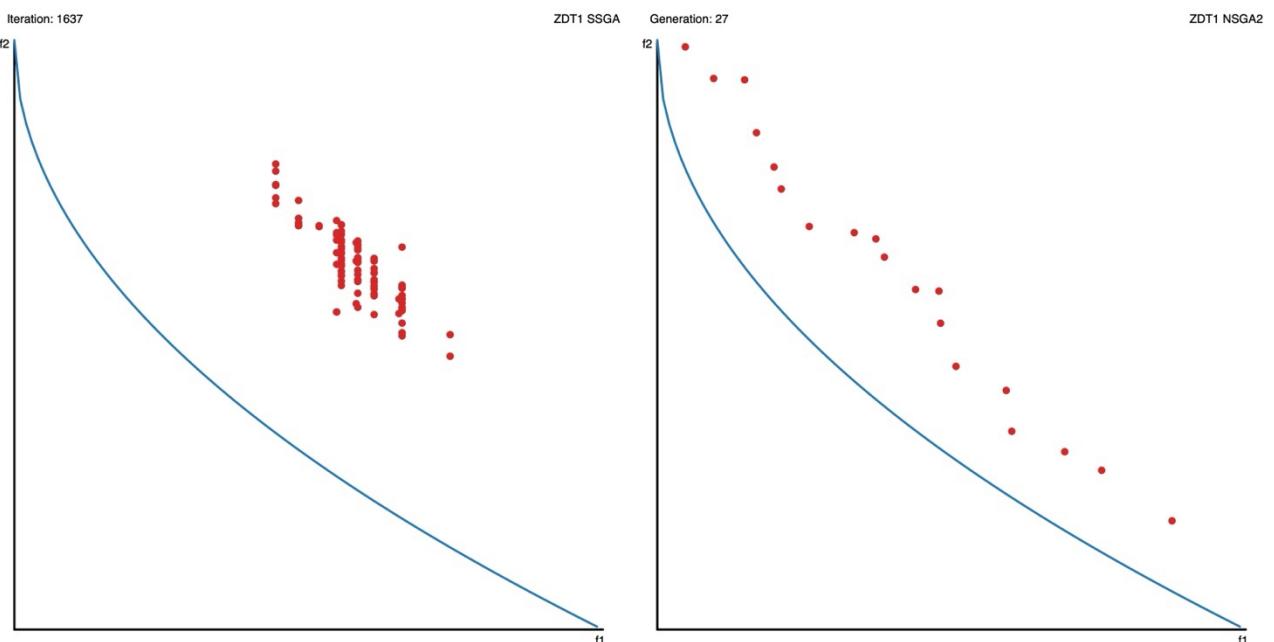


Рисунок 8 – Приближение к фронту Парето

Показаны промежуточные результаты приближения к фронту Парето в задаче ZDT1 для алгоритмов SSGA (слева) и NSGA2 (справа) (Рисунок 8). По графикам видна разница между многоцелевым алгоритмом (NSGA2) и одноцелевым (SSGA). Многоцелевой алгоритм оптимизирует значение каждой метрики отдельно и решения в популяции распределяются равномерно вдоль фронта. Одноцелевой алгоритм, в свою очередь, оптимизирует только общую функцию приспособленности  $F_{fit}$ , вследствие чего решения распределяются «облаком», движущимся к точке оптимума  $F_{fit}$ .

Представлен устоявшийся фронт Парето для задачи ZDT3, построенный с помощью NSGA2 (Рисунок 9). По графику видно, что все решения очень близки к истинному фронту и равномерно распределены вдоль него.

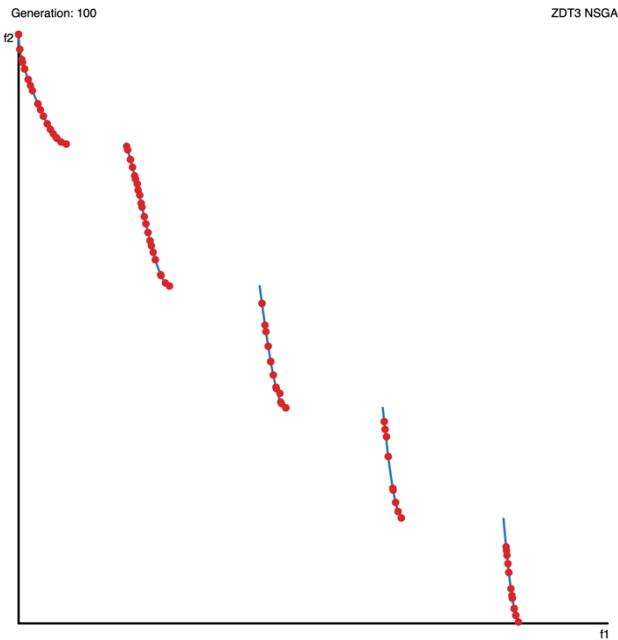


Рисунок 9 – Устоявшийся фронт Парето

Так же получены графики устоявшихся фронтов для задач ZDT1, ZDT2, ZDT3, ZDT4, ZDT6, построенных с помощью алгоритмов SGA, SSGA, NSGA2, SPEA2. Они представлены в [приложении](#).

### 3. Задача о рюкзаке 0-1.

Поскольку эта задача имеет только одну цель оптимизации, на ней тестировались только одноцелевые ГА: SGA и SSGA. Для двухмерного случая они сравнивались с методом динамического программирования.

В ГА хромосома представлялась последовательностью генов, длина которой была равна количеству предметов. Каждый ген представлял собой логический бит, два возможных состояния которого кодировали решение о том, брать соответствующий предмет или нет.

Был использован оператор равномерного скрещивания с коэффициентом 0.5 [14]. Оператор мутации представлял из себя изменение значений 1% случайно выбранных генов.

Показан график зависимости значения функции приспособленности от времени, затраченного алгоритмом SGA на его достижение (Рисунок 10). Алгоритм применялся к двухмерной задаче с 10000 предметов. Численность популяции составляла 1000 особей. Предварительно было получено оптимальное решение методом динамического программирования и значения на графике нормированы по нему. Поиск оптимума традиционным методом занял 7.11 секунд, а SGA достигает 95% от него за ~40 секунд работы.

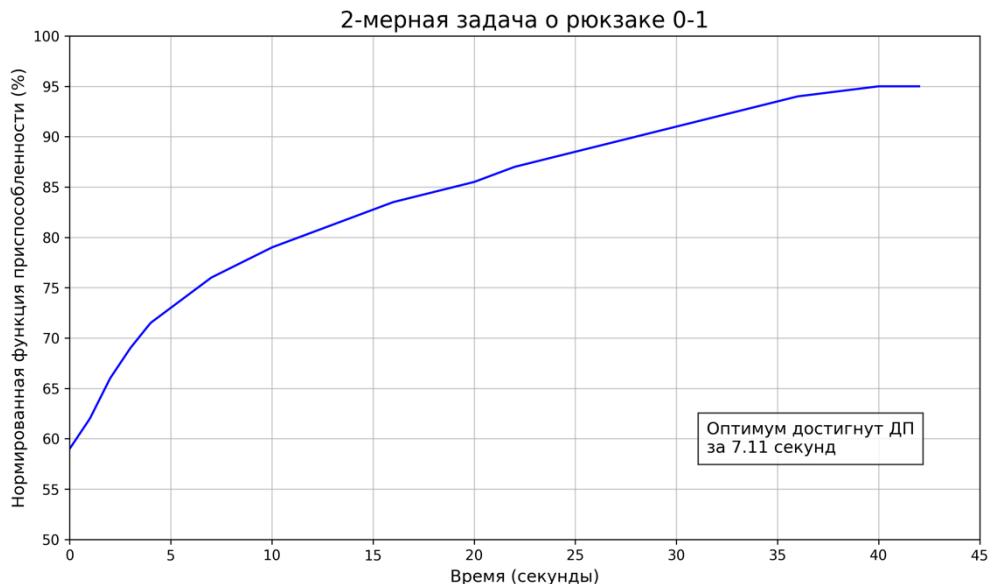


Рисунок 10 – Решение 2-мерной задачи о рюкзаке 0-1

Показаны графики зависимости ценности рюкзака от количества шагов, совершенных SGA (Рисунок 11, Рисунок 12). В обоих случаях численность популяции составляла 1000 особей, а задача состояла в выборе из 10000 предметов. Отличие между этими двумя графиками заключается в том, что в первом случае задача была 10-мерной (Рисунок 11), а во втором – 100-мерной (Рисунок 12).

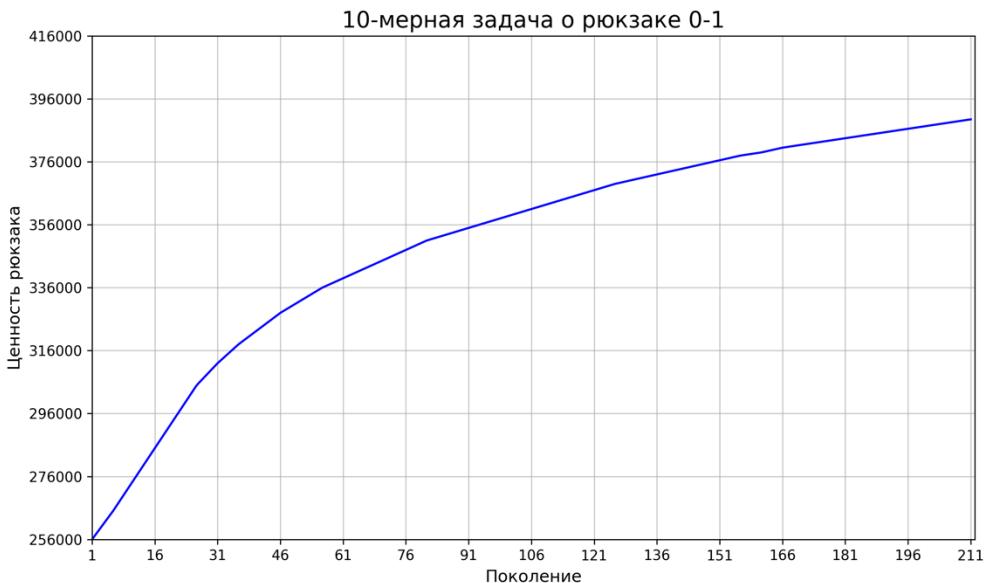


Рисунок 11 – Решение 10-мерной задачи о рюкзаке 0-1

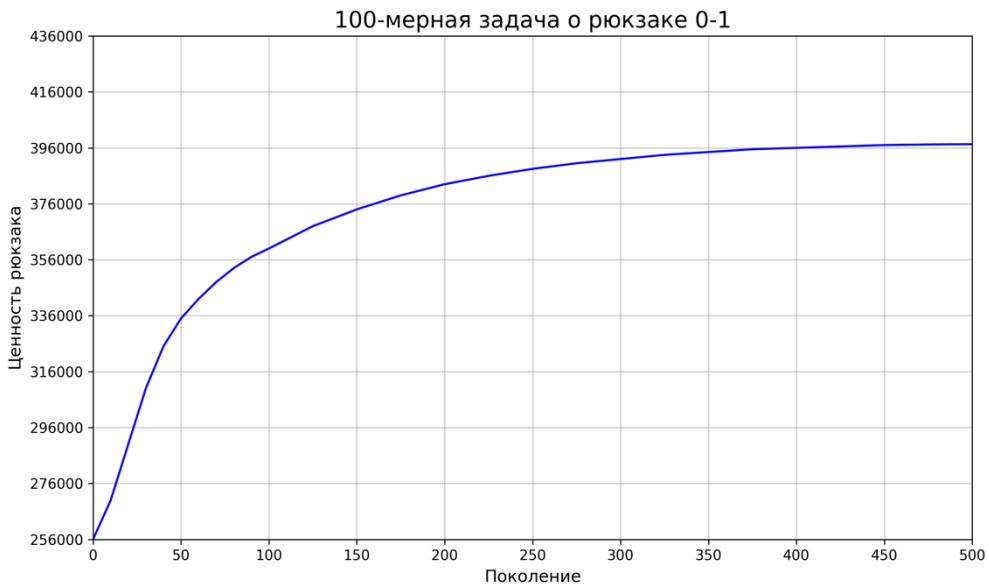


Рисунок 12 – Решение 100-мерной задачи о рюкзаке 0-1

Судя по результатам, для двухмерного случая очень сложно оправдать использование SGA. Результат вполне сопоставим по качеству и довольно быстро достигает 95% эталона, но время работы в несколько раз больше. Однако для многомерных случаев SGA оказывается очень полезным. По графикам видно, что качество результата, к которому сходится алгоритм, практически не зависит от размерности задачи. При этом количество требуемых на это поколений тоже практически не растет вместе с размерностью задачи.

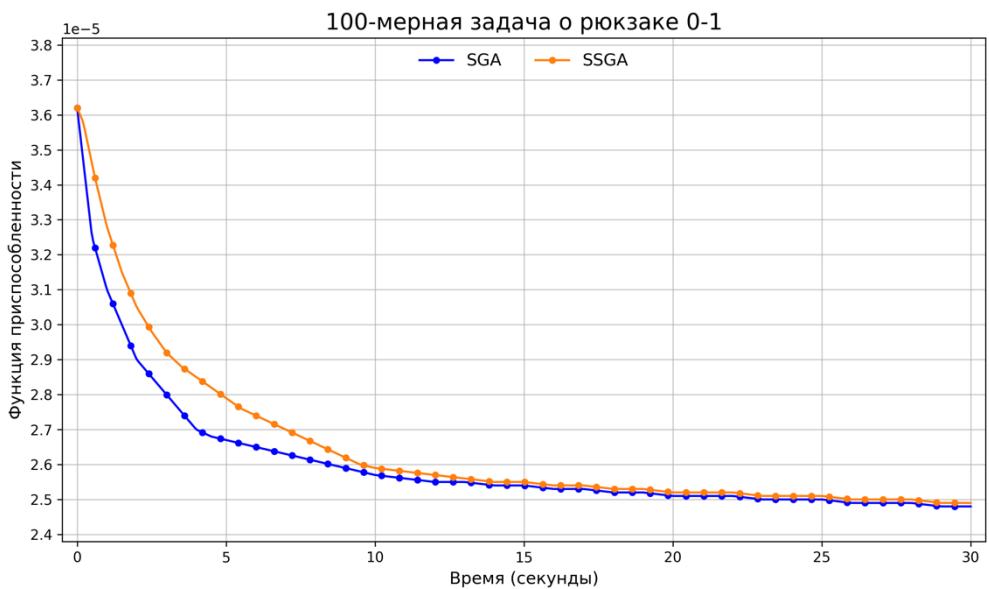


Рисунок 13 – Сравнение SGA и SSGA на 100-мерной задаче о рюкзаке 0-1

Представлен график, иллюстрирующий сходимость алгоритмов SGA и SSGA на 100-мерной задаче о рюкзаке 0-1 (Рисунок 13).

#### 4. Симметрическая задача о коммивояжере.

Эта задача, так же, как и задача о рюкзаке, имеет только одну цель оптимизации, поэтому на ней тестировались только однотелевые ГА: SGA и SSGA. Для небольшого количества вершин они сравнивались с методом полного перебора и методом ветвей и границ.

В ГА хромосома представлялась последовательностью генов, длина которой была равна количеству вершин. Каждый ген представлял собой число – номер города на соответствующей позиции в пути.

Был использован оператор равномерного скрещивания с коэффициентом 0.5 [14]. Оператор мутации представлял из себя перестановку очередности двух случайных вершин.

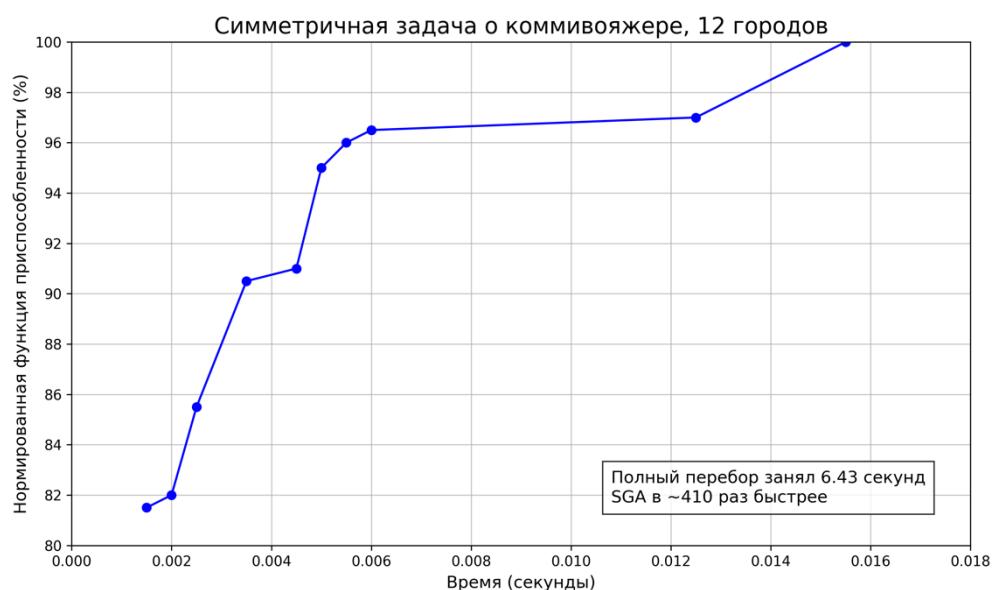


Рисунок 14 – Решение симметрической задачи о коммивояжере с 12 городами

Показан график зависимости значения функции приспособленности от времени, затраченного алгоритмом SGA на его достижение (Рисунок 14). Алгоритм применялся к задаче с 12 городами. Численность популяции составляла 500 особей. Предварительно было получено оптимальное решение методом полного перебора и значения на графике нормированы по нему. Поиск оптимума перебором занял 6.43 секунд, а SGA достигает его в несколько сот раз быстрее. Для задач, с количеством городов больше 12, алгоритм полного перебора практически не применим.

Представлен схожий график, но для сравнения SGA с методом ветвей и границ на задаче с 20 городами (Рисунок 15). ГА оказался на порядок быстрее. Причем, чем больше городов в задаче, тем эффективнее ГА относительно традиционных методов.

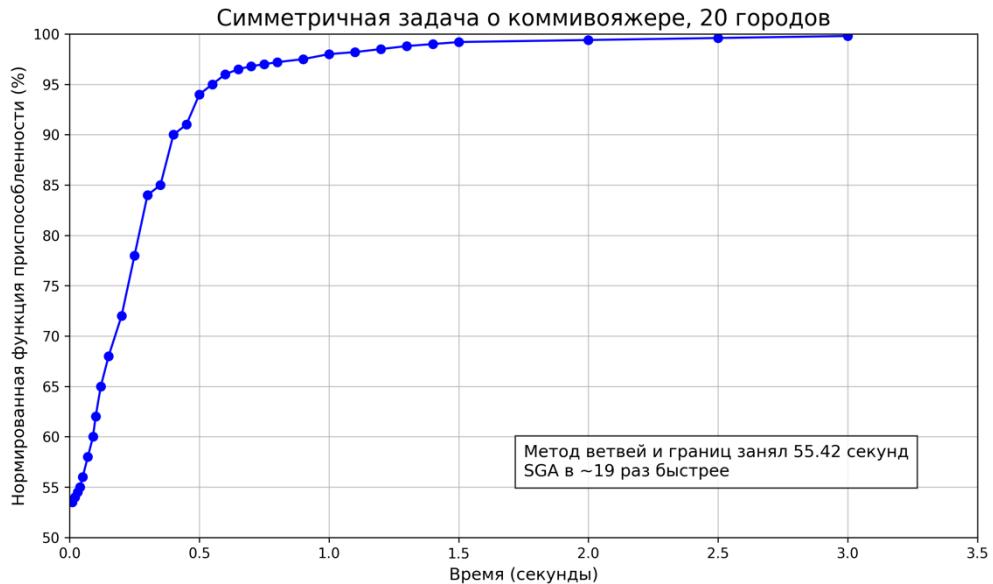


Рисунок 15 – Решение симметричной задачи о коммивояжере с 20 городами

С помощью ГА можно эффективно находить решения для задач с большим количеством городов, что представляется невозможным при использовании методов полного перебора и ветвей и границ.

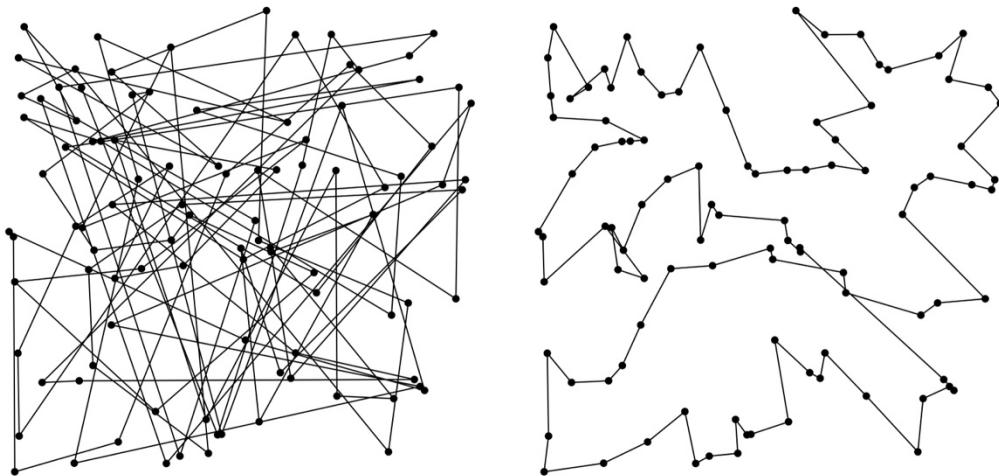


Рисунок 16 – Пример решения симметричной задачи о коммивояжере с 100 городами

Показан результат поиска минимального гамильтонова цикла в задаче со 100 городами с помощью SSGA (Рисунок 16). Слева – случайное начальное решение с длиной пути 4459, справа – лучшее установившееся решение, достигнутое алгоритмом за 500 секунд ( $\sim 1.5 \cdot 10^7$  шагов) с длиной 880.

Процесс схождения алгоритма при решении предыдущей задачи изображен отдельно (Рисунок 17). Строки в изображении соответствуют каждой лучшей перестановке вершин, полученной алгоритмом в процессе выполнения.

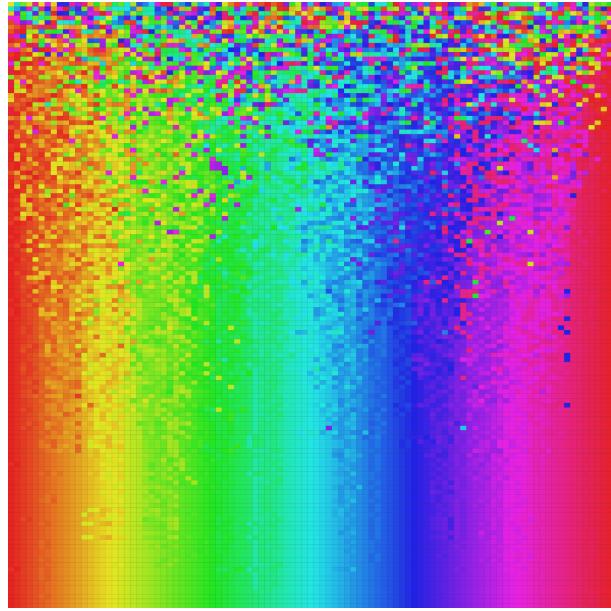


Рисунок 17 – Процесс схождения алгоритма

По графику видно, как решение стабилизуется со временем. Так же можно заметить, что некоторые участки в пути устанавливаются раньше других из-за локальных особенностей взаимного расположения вершин.



Рисунок 18 – Сравнение SGA и SSGA для задачи с 100 городами

Представлен график, иллюстрирующий сходимость алгоритмов SGA и SSGA на симметричной задаче о коммивояжере со 100 вершинами (Рисунок 18).

## Выводы

1. Сформулированы оптимизационные задачи.
2. Описаны методы решения тестовых задач.
3. Проанализированы результаты решения тестовых задач с помощью выбранных методов.

## **Цели и задачи на УИР**

Основной задачей данной работы является разработка системы, реализующей ГА и применение ее к расположению вершин графа на плоскости.

Для решения данной задачи необходимо решить следующие подзадачи.

1. Формулировка оптимизационных задач.
2. Описание методов решения тестовых задач.
3. Анализ результатов решения тестовых задач с помощью выбранных методов.
4. Разработка генетических операторов.
5. Разработка гибридных методов.
6. Разработка требований к создаваемой системе.
7. Проектирование системы на основе требований.
8. Программная реализация системы и алгоритмов.

## **Раздел 2. Разработка новых алгоритмов, адаптированных к поставленной задаче**

В данном разделе работы представлена разработка новых алгоритмов, адаптированных к задаче размещения вершин графа на плоскости с учетом нескольких критериев оптимизации. Она включает в себя разработку новых генетических операторов и гибридных методов, эффективность которых впоследствии будет оценена.

### **2.1. Разработка генетических операторов**

#### **1. Равномерное скрещивание. [14]**

Это оператор рекомбинации, обменивающий генетический материал между родительскими хромосомами на уровне отдельных генов. В отличие от блочных методов, равномерное скрещивание обрабатывает каждый ген независимо.

Оператор генерирует двух потомков ( $C_1, C_2$ ) из двух родительских хромосом ( $P_1, P_2$ ). Хромосомы представлены как последовательности  $N$  генов:  $P_1 = \{(x_{1,1}, y_{1,1}), \dots, (x_{1,N}, y_{1,N})\}$  и  $P_2 = \{(x_{2,1}, y_{2,1}), \dots, (x_{2,N}, y_{2,N})\}$ . Для каждой  $i$ -ой позиции гена (координаты вершины) определяется, произойдет ли обмен, на основе заданного порога вероятности  $p_{swap}$ .

Для каждого  $i \in \{1, \dots, N\}$ , генерируется случайное число  $r \in [0,1]$ . Если  $r < p_{swap}$ , гены обмениваются между родителями:

$$\begin{aligned}C_{1,i} &= (x_{2,i}, y_{2,i}) \\C_{2,i} &= (x_{1,i}, y_{1,i})\end{aligned}$$

В противном случае ( $r \geq p_{swap}$ ), гены наследуются от своих «родных» родителей:

$$C_{1,i} = (x_{1,i}, y_{1,i})$$

$$C_{2,i} = (x_{2,i}, y_{2,i})$$

Этот механизм обеспечивает высокое смешивание генетического материала, что способствует широкому исследованию пространства поиска и снижает вероятность преждевременной сходимости. Степень перемешивания регулируется параметром  $p_{swap}$ .

## 2. Равномерная мутация (Р).

Это оператор мутации, который изменяет значение случайно выбранного гена в хромосоме, заменяя его новым значением, сгенерированным равномерно в пределах допустимого диапазона. Этот механизм способствует широкому исследованию пространства поиска и поддержанию генетического разнообразия.

Оператор применяется к одной особи, представленной хромосомой  $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$ , где  $N$  – количество вершин графа, а координаты  $(x_i, y_i)$  ограничены областью  $[0, W] \times [0, H]$ .

Случайно выбирается индекс вершины для мутации  $i_{mut}$ . Новые значения для координат  $X$  и  $Y$  выбранной вершины  $i_{mut}$  генерируются из равномерного распределения:

$$\begin{aligned} x'_{i_{mut}} &\sim U(0, W) \\ y'_{i_{mut}} &\sim U(0, H) \end{aligned}$$

Где  $U(a, b)$  обозначает равномерное распределение на интервале  $[a, b]$ . Мутированная хромосома  $M$  формируется путем замены старых координат вершины  $i_{mut}$  на новые, сохраняя остальные гены без изменений:

$$\begin{aligned} M_j &= S_j, \text{ для } j \neq i_{mut} \\ M_{i_{mut}} &= (x'_{i_{mut}}, y'_{i_{mut}}) \end{aligned}$$

Этот тип мутации эффективно вводит радикальные изменения в хромосому, позволяя алгоритму исследовать удаленные области пространства решений.

## 3. Нормальная мутация (Н).

Это оператор мутации, который изменяет значение случайно выбранного гена в хромосоме путем добавления к нему случайного смещения, взятого из нормального (гауссова) распределения. Этот подход позволяет производить «тонкие» настройки существующих решений, способствуя локальному поиску вокруг текущих оптимумов.

Оператор применяется к отдельной особи, представленной хромосомой  $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$ , где координаты  $(x_i, y_i)$  лежат в диапазоне  $[0, W] \times [0, H]$ .

Случайно выбирается индекс вершины для мутации  $i_{mut}$ . К ее текущим координатам добавляются смещения  $\Delta x$  и  $\Delta y$ , сгенерированные из нормального распределения:

$$\begin{aligned} \Delta x &\sim N(0, \sigma_x^2) \\ \Delta y &\sim N(0, \sigma_y^2) \end{aligned}$$

где стандартные отклонения  $\sigma_x$  и  $\sigma_y$  зависят от размеров плоскости и коэффициента масштабирования  $k$ :  $\sigma_x = W \cdot k$  и  $\sigma_y = H \cdot k$ . Новые координаты вершины  $i_{mut}$  определяются как:

$$x'_{i_{mut}} = \text{clamp}(x_{i_{mut}} + \Delta x, 0, W)$$

$$y'_{i_{mut}} = \text{clamp}(y_{i_{mut}} + \Delta y, 0, H)$$

Этот тип мутации обеспечивает сфокусированный поиск вблизи текущего решения, что может быть особенно эффективно на более поздних этапах эволюции, когда требуется доводка.

#### 4. Зеркальная мутация (3). [16]

Это оператор мутации, который изменяет случайно выбранный ген (координаты вершины) путем его отражения относительно центральной оси по одной из координат ( $X$  или  $Y$ ). Этот оператор вводит дискретные, но предсказуемые изменения, способствуя исследованию симметричных или противоположных областей пространства решений.

Случайно выбирается индекс вершины для мутации:  $i_{mut}$ . Затем, с равной вероятностью (0.5), выбирается ось для отражения ( $X$  или  $Y$ ).

Если выбрана ось  $X$ :

$$x'_{i_{mut}} = W - x_{i_{mut}}$$

$$y'_{i_{mut}} = y_{i_{mut}}$$

Если выбрана ось  $Y$ :

$$x'_{i_{mut}} = x_{i_{mut}}$$

$$y'_{i_{mut}} = H - y_{i_{mut}}$$

Этот механизм позволяет алгоритму исследовать области пространства поиска, которые могут быть симметричными относительно текущего решения, что особенно полезно в задачах с потенциальной симметрией оптимальных укладок.

#### 5. Процентная мутация (П). [16]

Это оператор мутации, который модифицирует значение случайно выбранного гена путем умножения его на случайный коэффициент, близкий к единице. Этот метод позволяет тонко настраивать значения генов, смешая их в определенном диапазоне относительно текущего значения, что эффективно для локального поиска.

Случайно выбирается индекс вершины для мутации  $i_{mut}$ . Затем, с равной вероятностью (0.5), выбирается координата ( $X$  или  $Y$ ) для изменения.

Если выбрана ось  $X$ :

$$x'_{i_{mut}} = \text{clamp}(x_{i_{mut}} \cdot \alpha, 0, W)$$

$$y'_{i_{mut}} = y_{i_{mut}}$$

Если выбрана ось  $Y$ :

$$x'_{i_{mut}} = x_{i_{mut}}$$
$$y'_{i_{mut}} = \text{clamp}(y_{i_{mut}} \cdot \alpha, 0, H)$$

где  $\alpha$  — случайный коэффициент, сгенерированный равномерно в диапазоне [0.8,1.2), т.е.  $\alpha \sim U(0.8,1.2)$ .

Этот оператор способствует исследованию окрестностей текущих решений, позволяя алгоритму «доводить» точность положения вершин.

#### 6. Фиксированная равномерная мутация (ФР).

Это специализированный оператор мутации, который фокусируется на разрешении пересечений ребер графа. В отличие от других операторов, выбирающих ген для мутации случайным образом из всех возможных, этот оператор модифицирует только те вершины, которые инцидентны пересекающимся ребрам («запутанные» вершины). Модификация происходит путем замены координат выбранной запутанной вершины на новые значения, равномерно сгенерированные в пределах допустимого диапазона. Этот подход позволяет алгоритму концентрировать усилия на наиболее проблемных участках графа, не нарушая уже оптимизированные части укладки.

Сначала определяются «запутанные» вершины  $V_{tangled}$ , то есть вершины, инцидентные ребрам, которые пересекаются. Если таких вершин нет, мутация не производится. В противном случае, случайным образом выбирается индекс  $i_{mut}$  из множества  $V_{tangled}$ . Новые значения для координат  $X$  и  $Y$  выбранной вершины  $i_{mut}$  генерируются из равномерного распределения так же, как при обычной равномерной мутации.

Этот оператор является целенаправленным методом мутации, который позволяет алгоритму эффективно «распутывать» граф, фокусируя вычислительные ресурсы на наиболее проблемных областях, тем самым ускоряя сходимость к укладкам с минимальным количеством пересечений.

#### 7. Фиксированная нормальная мутация (ФН).

Это специализированный оператор мутации, который, как и фиксированная равномерная мутация, целенаправленно модифицирует только вершины, инцидентные пересекающимся ребрам. Однако, в отличие от равномерной мутации, изменение координат производится путем добавления случайного смещения, взятого из нормального распределения. Это позволяет осуществлять «тонкие» настройки положений «запутанных» вершин, способствуя более локальному итеративному разрешению пересечений.

Оператор применяется к особи  $S$ . Сначала определяются «запутанные» вершины  $V_{tangled}$ . Если  $V_{tangled}$  пусто, мутация не производится. В противном случае, случайным

образом выбирается индекс  $i_{mut}$  из множества  $V_{tangled}$ . К ее текущим координатам добавляются смещения  $\Delta x$  и  $\Delta y$ , сгенерированные из нормального распределения, точно так же как при обычной нормальной мутации.

Этот оператор эффективно доводит укладку графа, фокусируясь на устраниении оставшихся пересечений путем небольших корректировок.

#### 8. Фиксированная процентная мутация (ФП).

Это специализированный оператор мутации, который, как и другие «фиксированные» мутации, целенаправленно модифицирует только вершины, инцидентные пересекающимся ребрам ( $V_{tangled}$ ). Если  $V_{tangled}$  пусто, мутация не производится. В противном случае, случайным образом выбирается индекс  $i_{mut}$  из  $V_{tangled}$ . Координаты  $X$  или  $Y$  (выбирается случайно с равной вероятностью) этой вершины изменяются путем умножения на случайный коэффициент  $\alpha \sim U(0.8, 1.2)$ , аналогично обычной процентной мутации.

Этот оператор обеспечивает локальную подстройку положений «запутанных» вершин, способствуя эффективному разрешению пересечений.

#### 9. Адаптивная нормальная мутация (АН).

Это гибридный оператор мутации, который динамически изменяет свое поведение в зависимости от текущего состояния графа, а именно, от наличия пересечений ребер. Его цель – сначала распутать граф, а затем перейти к оптимизации других метрик, сохраняя при этом целостность графа без пересечений.

Сначала проверяется наличие «запутанных» вершин  $V_{tangled}$ .

Если  $V_{tangled}$  не пусто: применяется фиксированная нормальная мутация с коэффициентом  $k$ . Это позволяет сфокусироваться на разрешении существующих пересечений путем небольших, целенаправленных сдвигов координат.

Если  $V_{tangled}$  пусто: проводится попытка нормальной мутации с коэффициентом  $k$  для случайно выбранной вершины. Если в результате такой мутации появляются новые пересечения, изменение отменяется, и совершается новая попытка. Этот процесс повторяется до тех пор, пока мутация не будет произведена без появления пересечений, или пока не будет достигнуто максимальное количество попыток  $T_{max}$ . Если все попытки исчерпаны и не удалось найти мутацию без пересечений, хромосома остается неизменной.

Такой адаптивный подход позволяет ГА эффективно справляться с задачей минимизации пересечений на ранних этапах, а затем плавно переходить к улучшению других характеристик укладки графа, гарантируя, что полученные решения остаются без пересечений.

#### 10. Консервативная нормальная мутация (КН).

Это оператор мутации, который смещает случайно выбранный ген (координаты вершины) с использованием нормального распределения, но при этом гарантирует, что количество пересечений ребер не увеличится.

Случайно выбирается индекс  $i_{mut}$  для мутации. К текущим координатам вершины  $i_{mut}$  добавляются смещения  $\Delta x$  и  $\Delta y$ , сгенерированные из нормального распределения, как при обычной нормальной мутации. Если полученное изменение приводит к увеличению количества пересечений по сравнению с исходным состоянием, то мутация отменяется, и координаты вершины  $i_{mut}$  восстанавливаются.

Этот оператор позволяет производить локальные уточнения укладки графа, при этом активно поддерживая достигнутый прогресс в минимизации пересечений, что делает его ценным инструментом для доводки решений в задачах с жесткими ограничениями по пересечениям.

## 2.2. Разработка гибридных методов

### 1. Мутация по вектору натяжения (ВН).

Это специализированный оператор мутации, который использует принципы силовых алгоритмов укладки графов, в частности, ФР, для смещения случайно выбранной вершины. Во многом он похож на «Tension Vector Mutation (TVM)», описанный авторами [16]. Цель данной мутации — улучшить общую эстетику укладки, перемещая вершины под действием притягивающих и отталкивающих сил, моделирующих структуру графа.

Случайным образом выбирается вершина, чьи координаты будут модифицированы. Для этой вершины вычисляется вектор смещения  $\vec{p}_i'$  по формуле (18), который является суперпозицией притягивающих и отталкивающих сил, действующих на нее, как это описано в ФР.

Единственное отличие в том, что  $T$  – текущая температура, не меняется со временем, а определяется через параметр  $\epsilon$  и размеры области размещения:

$$T = \min(W, H) \cdot \epsilon \quad (21)$$

Этот оператор позволяет локально оптимизировать положение вершин, используя «физическую» модель графа, что способствует получению более сбалансированных и равномерных укладок.

### 2. Фиксированная мутация по вектору натяжения (ФВН).

Это специализированный оператор мутации, который применяет принципы мутации по вектору натяжения, но целенаправленно только к «запутанным» вершинам ( $V_{tangled}$ ). Он позволяет локально оптимизировать положения проблемных вершин, устранивая пересечения ребер с использованием физической модели графа.

### 3. SSGA-FR.

Это гибридный алгоритм укладки графов, сочетающий SSGA с итеративным силовым методом ФР. Похожий метод «GA+TV» был описан авторами [16]. Основная идея заключается в использовании SSGA для начального этапа разрешения пересечений, а затем передаче наилучшего решения для последующей оптимизации с помощью ФР, что позволяет ФР избежать попадания в локальные оптимумы.

На первом этапе SSGA применяется к задаче укладки графа в течение заданного числа итераций. По завершении этапа SSGA, лучшая найденная особь (укладка графа) передается в качестве начального состояния для ФР. На втором этапе ФР алгоритм итеративно оптимизирует укладку, используя модель отталкивающих и притягивающих сил, как уже было описано.

Предполагается, что такой последовательный подход позволит эффективно сочетать глобальный поисковый потенциал SSGA с локальной оптимизационной способностью ФР, потенциально обеспечивая более высокое качество результирующих укладок графов.

### 4. FR-NSGA2.

Это гибридный алгоритм укладки графов, который последовательно объединяет алгоритм ФР с многоцелевым NSGA2. Основная идея заключается в использовании быстрой способности ФР к первоначальному разрешению большей части пересечений ребер, после чего NSGA2 дорабатывает решение, справляясь с оставшимися пересечениями и оптимизируя множество метрик.

На первом этапе ФР быстро уменьшает количество пересечений. Полученное решение затем используется для инициализации популяции NSGA2: одна особь представляет собой напрямую результат ФР, а остальные  $N - 1$  особей генерируются путем применения мутаций к этому решению. Затем NSGA2 проводит заданное количество итераций, используя эту начальную популяцию.

Гипотеза заключается в том, что ФР, будучи быстрым методом, может быть неэффективен для тонкой подстройки решений под сложные метрики и подвержен проблеме «заворачивания» краев графа, что приводит к появлению новых пересечений. Комбинация с NSGA2 призвана преодолеть эти недостатки, позволяя ФР быстро устранить основные пересечения, а NSGA2 – доработать укладку, разрешить оставшиеся краевые пересечения и оптимизировать множество целевых функций.

### 5. FR-SSGA-NSGA2.

Это трехфазный гибридный алгоритм укладки графов, последовательно объединяющий алгоритм ФР, SSGA и NSGA2. Гипотеза, лежащая в основе этой комбинации, состоит в использовании преимуществ каждого метода на разных этапах оптимизации.

На первом этапе ФР быстро устраниет большинство пересечений ребер. Несмотря на свою скорость, ФР склонен к образованию краевых пересечений и не оптимизирует заданные метрики. Полученное решение затем служит отправной точкой для второго этапа.

На втором этапе SSGA, показавший высокую эффективность в распутывании графов, продолжает процесс минимизации пересечений, включая краевые. Популяция для SSGA инициализируется на основе результата ФР.

На заключительном этапе NSGA2 принимает текущую популяцию от SSGA и переходит к многоцелевой оптимизации. В отличие от одноцелевого SSGA, NSGA2 позволяет получить множество решений, распределенных вдоль фронтов Парето, что обеспечивает компромисс между различными метриками после того, как проблема пересечений будет в значительной степени решена.

## **Выводы**

1. Предложен набор генетических операторов.
2. Разработаны гибридные алгоритмы укладки графов.
3. Выдвинуты гипотезы относительно эффективности гибридных алгоритмов.

## **Раздел 3. Проектирование системы, реализующей разработанные алгоритмы**

В данном разделе представлены требования к разрабатываемой программной системе, а также ее архитектура, предназначенная для реализации предложенных ГА и операторов.

### **3.1. Разработка требований к создаваемой системе**

При разработке системы, реализующей ГА, были сформулированы следующие функциональные требования.

- Генерация графов (произвольных и планарных).
- Реализация разработанных генетических операторов и гибридных методов.
- Оценка функции приспособленности (количество пересечений ребер, равномерность расстояния между вершинами и углов между смежными ребрами).
- Сохранение результатов (лучшие решения, прогресс оптимизации).

Кроме того, были сформулированы нефункциональные требования.

- Производительность. Система должна обеспечивать эффективное решение для графов с 200 вершинами за приемлемое время (не более 10 минут).
- Модульность и расширяемость. Система должна обеспечивать легкое добавление новых ГА, операторов и типов задач без существенных изменений в существующей кодовой базе.

- Гибкость конфигурации. Пользователи должны иметь возможность тонкой настройки параметров каждого ГА (размер популяции, количество итераций/время выполнения, генетические операторы и т.д.) через удобный интерфейс.
- Удобство использования и отладки. Система должна предоставлять средства для мониторинга прогресса алгоритмов и логирования промежуточных и итоговых решений.
- Наличие примеров. Для демонстрации функциональности и облегчения освоения, система должна включать реализации нескольких классических оптимизационных задач.

### **3.2. Проектирование системы на основе требований**

На основе сформулированных требований, система была спроектирована как публичная модульная библиотека. Такая архитектура позволяет гибко конфигурировать и переиспользовать компоненты для решения разнообразных оптимизационных задач с помощью ГА.

Основные компоненты системы организованы в следующие модули.

- `algos`. Модуль, содержащий реализации различных генетических алгоритмов (SGA, NSGA2, SPEA2, SSGA) и общий интерфейс GA.
- `problems`. модуль для определения различных оптимизационных задач;
- `graphplane`. Специализированный подмодуль для задачи укладки графов на плоскости, включающий определения генетических операторов, специфичных для этой задачи.
- `knapsack`, `tsp`, `zdt`. Подмодули других задач.

Далее приведено проектирование основных типов данных.

- `Graph`. Структура, представляющая абстрактное описание графа. Структура включает следующие поля.
  - `NumVertices`. Целое число (`int`), представляющее общее количество вершин в графе.
  - `NumEdges`. Целое число (`int`), представляющее общее количество ребер в графе.
  - `Edges`. Срез (`[]Edge`) структур `Edge`, каждая из которых описывает одно ребро графа.
- `Edge`. Вспомогательная структура, определяющая ребро через индексы двух соединяемых вершин.
  - `From`. Индекс первой вершины (`int`).
  - `To`. Индекс второй вершины (`int`).
- `Solution`. Общий интерфейс для любого решения оптимизационной задачи. Это позволяет алгоритмам работать с различными типами задач унифицированным образом. Интерфейс определяет следующие методы.

- Objectives(). Метод, который возвращает срез значений целевых функций ([]float64). Для однокритериальных задач обычно возвращает срез с одним элементом.
  - Fitness(). Метод, который возвращает единственное значение приспособленности решения (float64).
- GraphPlaneSolution. Конкретная реализация интерфейса Solution для задачи укладки графа. Эта структура представляет собой «особь» в генетическом алгоритме и хранит текущее размещение вершин графа на плоскости.
  - Graph. Указатель на структуру Graph, описывающую топологию графа. Использование указателя (\*Graph) подразумевает, что множество вершин и ребер остается неизменным для данной задачи, тогда как их положение на плоскости (VertPositions) меняется.
  - Width, Height. Размеры плоскости (float64), на которой располагаются вершины графа, ограничивающие область мутации и скрещивания.
  - VertPositions. Срез ([]VertexPos) структур VertexPos, хранящий координаты всех вершин на плоскости, которые изменяются генетическими операторами.
- VertexPos. Структура, хранящая координаты вершины на плоскости.
  - X. Координата по оси X (float64).
  - Y. Координата по оси Y (float64).
- CrossoverFunc. Представляет функциональный тип оператора скрещивания. Принимает две родительские особи (Solution) и возвращает срез новых потомков ([]Solution).
- MutationFunc. Представляет функциональный тип оператора мутации. Принимает одну особь (Solution) и возвращает новую (мутированную) особь (Solution).

Разработанная система реализована на языке программирования Go, что обеспечивает высокую производительность и кроссплатформенность.

- Система имеет следующие требования к программному и аппаратному окружению.
- Язык программирования. Go (версия 1.20 и выше).
  - Операционная система. Библиотека является кроссплатформенной и может быть скомпилирована и запущена на любых операционных системах, поддерживающих Go (Linux, Windows, MacOS).
  - Зависимости. Для работы системы используются следующие внешние и стандартные библиотеки.
    - [github.com/fogleman/delaunay](https://github.com/fogleman/delaunay). Используется для триангуляции Делоне при генерации планарных графов.

- [gonum.org/v1/gonum](https://github.com/gonum/gonum). Библиотека для численных вычислений, используется для вычисления метрик и функции приспособленности.
- Стандартные библиотеки Go (`encoding/json`, `fmt`, `log`, `math`, `math/rand`, `os`, `time`). Используются для работы с файлами, выводом, математикой, случайными числами и временем.
- Среда выполнения. Для запуска скомпилированного исполняемого файла не требуется установка Go Runtime; приложение является самодостаточным бинарным файлом. Для сборки проекта требуется установленный Go SDK.

Далее приведена оценка необходимого объема памяти. Объем, потребляемый системой, в основном зависит от двух факторов: размера графа (количества вершин и ребер) и размера популяции генетического алгоритма.

Расчет размера одной особи (`GraphPlaneSolution`).

- `VertPositions`. Срез структур `VertexPos`, где каждая структура состоит из двух `float64` координат ( $X, Y$ ). Каждая координата занимает 8 байт. Таким образом, для графа с  $V$  вершинами этот срез будет потреблять примерно  $V \cdot 2 \cdot 8 = 16V$  байт.
- `Edges` (в структуре `Graph`). Срез структур `Edge`, каждая содержит два `int` (`From`, `To`). На 64-битных системах `int` занимает 8 байт, следовательно, для графа с  $E$  ребрами (например,  $E = 3V$ ) это  $3 \cdot V \cdot 2 \cdot 8 = 48V$  байт. Поскольку `GraphPlaneSolution` содержит указатель на `Graph`, а не его копию, данные `Edges` не дублируются для каждой особи, но один раз занимают память.
- Прочие поля. Занимают незначительный объем памяти по сравнению с `VertPositions`.

Расчет общего объема памяти.

- Общая потребляемая память будет примерно равна сумме произведения количества особей в популяции на размер одной из них, размера одной структуры `Graph` и накладных расходов Go Runtime.
- Для графа с  $V = 200$  вершинами,  $E = 3V$  ребрами и популяции в 500 особей расчет будет следующим: размер одной особи составит  $200 \cdot 16 = 3200$  байт, а размер графа –  $200 \cdot 3 \cdot 16 = 9600$  байт. Следовательно, без учета накладных расходов, общий объем составит примерно  $3200 \cdot 500 + 9600 = 1\,609\,600$  байт (~1.5 мегабайта).

## Выводы

1. Сформулированы и обоснованы детальные функциональные и нефункциональные требования к создаваемой системе.
2. Спроектирована система в формате публичной модульной библиотеки на языке Go.

3. Разработана модульная архитектура, основанная на четких интерфейсах и функциональных типах для генетических операторов.
4. Детально описаны ключевые структуры данных, используемые для представления графов и их размещений.
5. Определены требования к программному окружению и проведена оценка объемов оперативной памяти.

## **Раздел 4. Программная реализация системы и достигнутые показатели**

В данном разделе работы представлено описание программной реализации модулей реализованной системы и продемонстрированы показатели, достигнутые системе в процессе работы.

### **4.1. Программная реализация системы и алгоритмов**

Программная реализация разработанной системы и алгоритмов осуществлена в соответствии с принципами модульного проектирования, что обеспечивает высокую гибкость, расширяемость и универсальность. Весь исходный код реализован в формате публичной библиотеки, написанной на языке Go, что способствует открытости и облегчает интеграцию в сторонние проекты.

Процесс кодирования системы включал следующие ключевые этапы.

- Реализация базовых генетических алгоритмов. Ядро системы включает имплементации четырех фундаментальных ГА: SGA, SSGA, NSGA2 и SPEA2. Код данных алгоритмов и быстрой не-доминирующей сортировки, используемой в NSGA2, доступен в [приложениях](#).
- Имплементация генетических операторов. Все разработанные генетические операторы, описанные в разделе 2, реализованы как функции, удовлетворяющие разработанному интерфейсу. Это позволяет динамически передавать их в ГА, обеспечивая высокую степень кастомизации.
- Разработка интерфейсов для задач и решений. Для достижения универсальности определены абстрактные интерфейсы для представления оптимизационных задач и их решений. Это позволяет подключать к реализованным ГА любые новые задачи, соответствующие этим интерфейсам.
- Реализация типовых оптимизационных задач и классических методов. Для демонстрации функциональности в состав библиотеки включены реализации нескольких задач.
  - Задача об укладке графов на плоскости. Реализован специализированный объект задачи, адаптированный под специфику укладок графов.
  - Задача о рюкзаке 0-1. Реализована задача и метод ее решения с использованием динамического программирования (см. [приложение](#)).

- Задача о коммивояжере. Реализована симметричная задача о коммивояжере с использованием полного перебора и метода ветвей и границ (см. [приложения](#)).
- Набор задач ZDT. Реализованы тестовые функции ZDT, часто используемые для оценки многоцелевых алгоритмов.
- Модули генерации графов. Для тестирования и демонстрации реализованы алгоритмы генерации произвольных и планарных графов (см. [приложение](#)).
- Имплементация метода Фрюхтермана-Рейнгольда (ФР). Метод силового расположения графов ФР также реализован в библиотеке, что позволяет использовать его как самостоятельно, так и в рамках гибридных алгоритмов (см. [приложение](#)).
- Модуль логирования. Разработан унифицированный модуль для протоколирования хода выполнения алгоритмов, позволяющий отслеживать промежуточные и итоговые решения, а также ключевые метрики прогресса.
- Система конфигурирования параметров. Параметры каждого ГА, такие как размер популяции, критерии останова (количество шагов или время выполнения), операторы скрещивания и мутации, а также специфические параметры для каждого оператора, вынесены в отдельные структуры, обеспечивая удобство их настройки.

Все эти компоненты интегрированы в единую библиотеку, предоставляющую гибкий и мощный инструментарий для исследования и применения генетических алгоритмов в задачах оптимизации.

#### **4.2. Достигнутые показатели**

Данные получены в результате экспериментального исследования, проведенного на разработанной системе. Все значения в таблицах этого подраздела являются усредненными по 50 тестам.

Таблица 7 – Среднее количество пересечений ребер для разных коэффициентов скрещивания

0.20	0.25	0.30	0.35	0.40	0.45	0.50
324.06	319.67	312.20	308.20	304.11	307.67	320.97

Таблица 7 содержит средние значения количества пересечений ребер в планарном графе с 100 вершинами после  $10^5$  шагов работы SSGA с размером популяции 500 и оператором мутации Р при использовании разных коэффициентов для оператора скрещивания.

По результатам в таблице можно сделать вывод, что этот коэффициент не оказывает значительного влияния на результат. Тем не менее, наиболее оптимальным коэффициентом при решении данной задачи является 0.4.

Таблица 8 содержит средние значения количества пересечений ребер в планарном графе с 100 вершинами после  $10^5$  шагов работы SSGA с размером популяции 500 и

равномерным оператором скрещивания с коэффициентом 0.4 при использовании разных операторов мутации.

Таблица 8 – Среднее количество пересечений ребер для разных операторов мутации

P	H	З	П	ФР	ФН	ФП	АН	КН	ВН	ФВН
306.23	178.31	849.15	315.77	281.08	199.69	335.62	246.54	154.85	623.08	1094.31

Для всех операторов, основанных на нормальной мутации, коэффициент масштабирования  $k$  был выбран равным 0.1. Для АН параметр максимального количества попыток  $T_{max}$  был выбран равным 3. Для операторов, основанных на мутации по вектору натяжения параметр  $\epsilon$  был выбран равным 0.01.

По результатам в таблице можно сделать вывод, что оператор КН подходит для решения данной задачи более остальных. Его программный код представлен в [приложении](#).

Гипотезы о гибридных операторах мутации ВН и ФВН не подтвердились. З показал плохие результаты, что говорит об асимметричности задачи. Операторы, основанные на нормальной мутации, в среднем, показывают лучшие результаты, чем все остальные.

Таблица 9 – Среднее количество пересечений ребер (новые алгоритмы)

Алгоритм	ПЛ25	ПР25	ПЛ50	ПР50	ПЛ100	ПР100	ПЛ200	ПР200
SSGA-FR	17.87	159.89	53.97	711.33	137.78	2542.56	370.78	10222.56
FR-NSGA2	0.00	90.33	0.00	379.33	0.00	1621.22	43.56	6575.22
FR-SSGA-NSGA2	3.84	82.11	31.87	355.78	75.67	1582.78	861.00	7095.89

Таблица 10 – Среднее значение функции приспособленности (новые алгоритмы)

Алгоритм	ПЛ25	ПР25	ПЛ50	ПР50	ПЛ100	ПР100	ПЛ200	ПР200
SSGA-FR	38.05	338.56	133.51	1590.31	426.89	6650.75	1445.88	29965.28
FR-NSGA2	0.92	182.98	0.94	762.52	1.78	3311.41	167.35	16600.15
FR-SSGA-NSGA2	8.43	166.93	62.79	722.68	174.83	3267.47	2861.22	17244.89

Таблица 9 и Таблица 10 содержат средние значения метрики количества пересечений ребер в графе и функции приспособленности для лучшего решения новых алгоритмов соответственно. Значения приведены для планарных и произвольных графов с 25, 50, 100 и 200 вершинами. У всех ГА размер популяции составлял 500 особей, использовался равномерный оператор скрещивания с коэффициентом 0.4 и оператор мутации КН.

В алгоритме SSGA-FR этап SSGA работал 100000 шагов, а для ФР были выбраны значения параметров количества шагов и начальной температуры равные 2000 и 0.005 соответственно. Параметр  $k$  был подобран так же, как и при тестировании в разделе 1.

В алгоритме FR-NSGA2 параметры ФР были такими же, а этап NSGA2 работал 350 шагов.

В алгоритме FR-SSGA-NSGA2 параметры ФР были такими же, этап SSGA работал 70000 шагов, а этап NSGA2 – 250 шагов.

Таблица 11 – Доля распутанных решений (новые алгоритмы)

Алгоритм	ПЛ25	ПЛ50	ПЛ100	ПЛ200
SSGA-FR	4%	0%	0%	0%
FR-NSGA2	100%	100%	100%	44%
FR-SSGA-NSGA2	74%	26%	0%	0%

Таблица 11 содержит доли случаев, когда планарный граф удавалось полностью «распутать» – получить планарную укладку с 0 пересечений ребер.

Анализ полученных результатов позволяет сделать следующие выводы относительно производительности гибридных алгоритмов.

- **SSGA-FR** не продемонстрировал значительных преимуществ по сравнению с автономным ФР. Это указывает на то, что качество конечной укладки, достигаемое ФР, мало зависит от начального решения, предоставляемого ГА.
- **FR-SSGA-NSGA2** показал неоднозначные результаты. На графах с 25 и 50 вершинами его эффективность была сопоставима с классическими ГА и ФР. Однако, на больших произвольных графах этот подход в среднем приводил к меньшему количеству пересечений ребер. В случае больших планарных графов, FR-SSGA-NSGA2 достигал меньшего числа пересечений по сравнению с классическими ГА, но уступал алгоритму ФР.
- **FR-NSGA2** продемонстрировал наиболее выдающиеся результаты, значительно превзойдя все остальные подходы как на планарных, так и на произвольных графах всех исследованных размеров. Для произвольных графов его производительность была незначительно ниже, чем у FR-SSGA-NSGA2, при этом выявленная разница может быть компенсирована небольшим увеличением числа шагов на этапе NSGA2. Особенно примечательно, что FR-NSGA2 является единственным алгоритмом, способным стабильно получать планарные укладки для больших графов. В то время как классические алгоритмы не справлялись с планарной укладкой графов со 100 и 200 вершинами, FR-NSGA2 достигал 100% планарных укладок для графов со 100 вершинами и 44.35% для графов с 200 вершинами, что превосходит лучшие результаты классических методов даже для графов с 50 вершинами.

На графике сравнивается среднее количество пересечений ребер, достигнутое различными алгоритмами (SGA, SSGA, NSGA2, SPEA2, ФР, SSGA-FR, FR-NSGA2, FR-SSGA-NSGA2) для планарных и произвольных графов с количеством вершин 25, 50, 100 и 200 (Рисунок 19). Ось Y представляет количество пересечений, при этом меньшие значения соответствуют лучшим результатам.

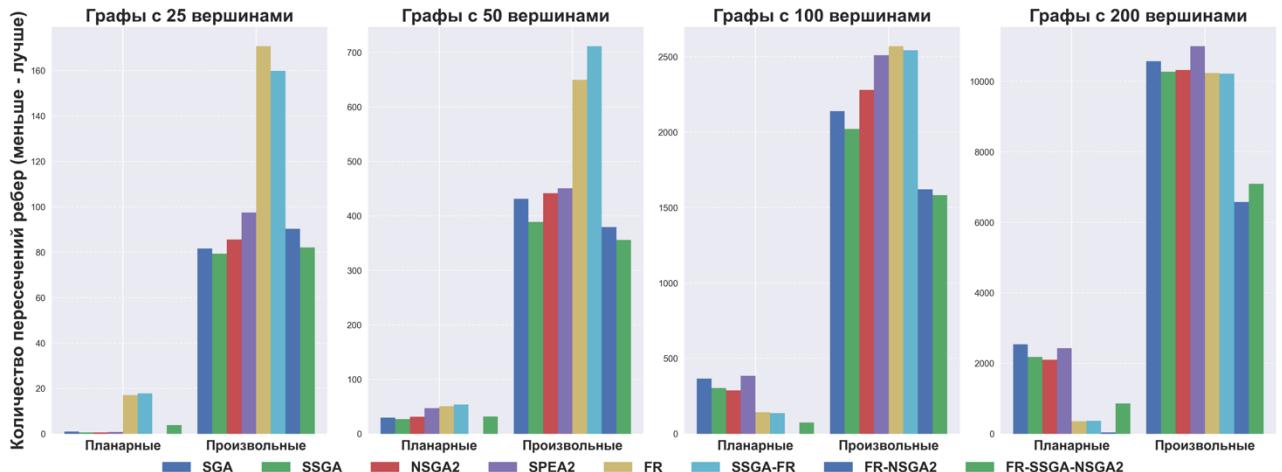


Рисунок 19 – Среднее количество пересечений ребер (все алгоритмы)

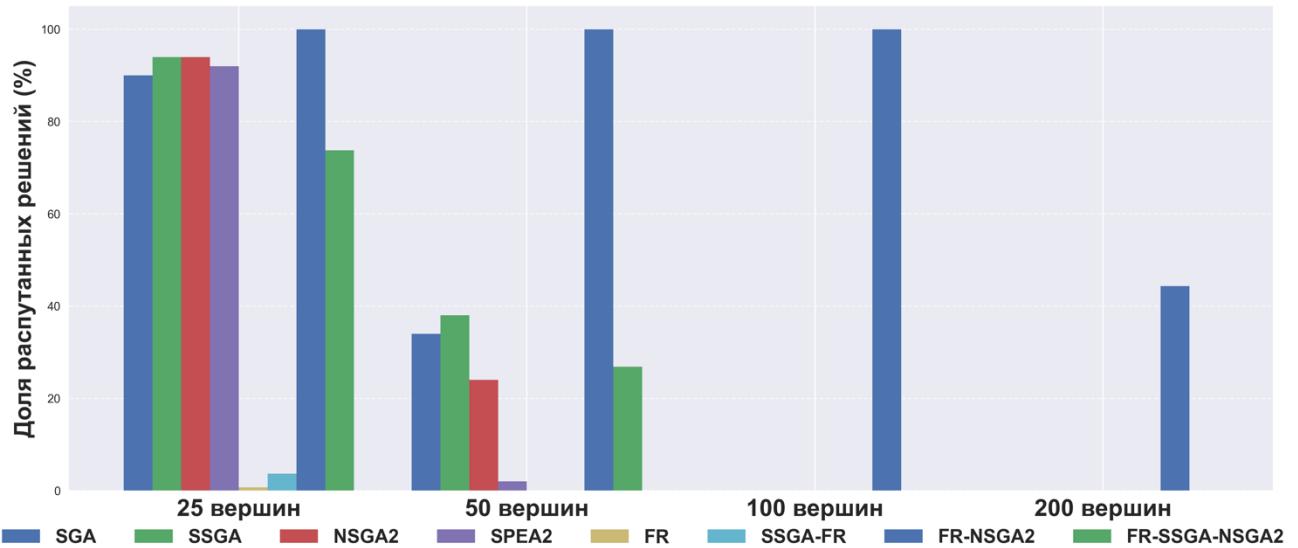


Рисунок 20 – Доля распутанных решений (все алгоритмы)

Отображена доля успешно «распутанных» (т.е. с нулевым количеством пересечений ребер) решений в процентах для различных алгоритмов (SGA, SSGA, NSGA2, SPEA2, ФР, SSGA-FR, FR-NSGA2, FR-SSGA-NSGA2) при работе с планарными графами, содержащими 25, 50, 100 и 200 вершин (Рисунок 20). Ось Y показывает процент распутанных решений, где более высокие значения указывают на лучшую производительность.

Показаны этапы процесса укладки планарного графа с 200 вершинами с помощью FR-NSGA2 (Рисунок 21). Слева – случайная начальная укладка, в центре – после завершения этапа ФР, справа – в конце работы алгоритма.

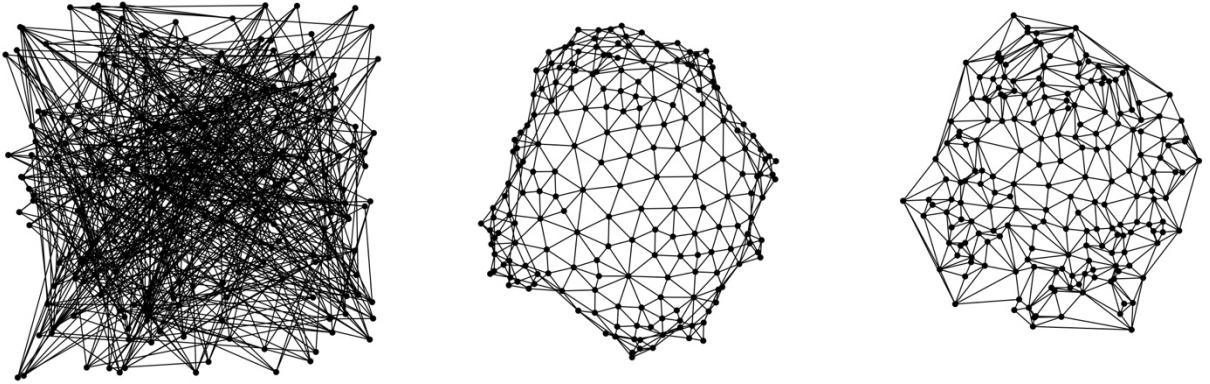


Рисунок 21 – Этапы процесса укладки ПЛ200 с помощью FR-NSGA2

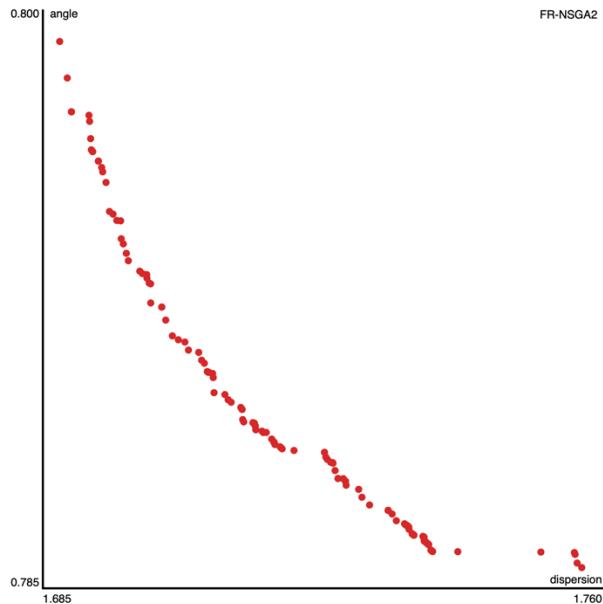


Рисунок 22 – Проекция фронта Парето

Изображена проекция фронта Парето на плоскость, где ось  $X$  соответствует метрике равномерности расположения вершин, а  $Y$  – метрике равномерности углов между смежными ребрами (Рисунок 22). Данный фронт получен алгоритмом FR-NSGA2 при решении ранее описанной задачи (Рисунок 21).

Приведенные результаты и их графическое представление убедительно демонстрируют, что гибридные алгоритмы, в частности FR-NSGA2, значительно повышают эффективность укладки графов, позволяя достигать высококачественных решений, включая планарные укладки для графов большой размерности, что ранее было недостижимо классическими методами.

## Выводы

1. Описаны этапы программной реализации разработанной модульной библиотеки, охватывающие кодирование ГА, специализированных генетических операторов и универсальных интерфейсов.

2. Представлены результаты эмпирических исследований, демонстрирующие влияние параметров генетических операторов (коэффициент скрещивания) и выбор типа мутации на эффективность алгоритма SSGA в минимизации пересечений ребер для планарных графов. Выявлен оптимальный коэффициент скрещивания и определен наиболее эффективный оператор мутации для данной задачи.
3. Выполнена сравнительная оценка производительности новых гибридных алгоритмов (SSGA-FR, FR-NSGA2, FR-SSGA-NSGA2) на различных типах и размерах графов. Это позволило проанализировать их способность к сокращению числа пересечений и достижению планарных укладок, что является ключевым для дальнейшей оптимизации.
4. Полученные данные подтверждают потенциал гибридных подходов в решении комплексных задач укладки графов, демонстрируя, как последовательное применение различных алгоритмов может способствовать преодолению их индивидуальных ограничений и повышению общего качества решений.

## Заключение

В настоящей работе приведены итоги работы, посвященной разработке и анализу генетических алгоритмов для решения многокритериальной задачи размещения вершин графа на плоскости. Данная проблема имеет высокую практическую значимость в областях визуализации данных, проектирования сетей и биоинформатики, где читаемость графовой структуры является ключевым фактором.

В ходе работы был выполнен анализ существующих подходов, выявлены их сильные и слабые стороны, что позволило четко определить цели и задачи исследования. Основной акцент был сделан на создании гибридных методов, способных эффективно находить компромиссные решения по нескольким критериям одновременно.

Для достижения поставленной цели были последовательно решены следующие задачи.

1. Сформулированы оптимационные задачи.
2. Описаны методы решения тестовых задач.
3. Проанализированы результаты решения тестовых задач с помощью выбранных методов.
4. Разработаны новые генетические операторы.
5. Разработаны методы, использующие комбинацию нескольких алгоритмов.
6. Сформулированы требования к создаваемой системе.
7. Спроектирована и описана система, реализующая новые алгоритмы, удовлетворяющая разработанным требованиям.

8. Описаны этапы кодирования, приведены примеры фрагментов кода для ключевых модулей.
9. Представлено сравнение эффективности разработанных методов с существующими подходами. Выявлены зависимости качества решений от параметров алгоритма.

Ключевым практическим результатом работы является созданная программная система, реализованная в виде гибкой и расширяемой библиотеки. Она не только является инструментом для решения поставленной задачи укладки графов, но и служит платформой для дальнейших исследований в области эволюционных вычислений. Экспериментально подтверждено, что предложенные гибридные алгоритмы превосходят базовые подходы по качеству итоговых решений.

Дальнейшее развитие данной работы видится в нескольких перспективных направлениях. Во-первых, это обобщение разработанных алгоритмов для решения более широкого класса задач, включая укладку графов в трехмерном пространстве. Во-вторых, проведение дополнительного, более глубокого анализа влияния параметров ГА на конечный результат с целью разработки методик их автоматической настройки. Наконец, в-третьих, значительный потенциал для улучшения заключается в оптимизации производительности разработанной системы за счет внедрения современных подходов к параллельным и распределенным вычислениям.

## **Список литературы**

1. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. Москва: Физматлит, 2022. 368с с.
2. Цой Ю.Р., Спицын В.Г. Исследование генетического алгоритма с динамически изменяемым размером популяции // Труды Международной научно-технической конференции “Интеллектуальные системы (IEEE AIS’05)”. 2005. С. 241-246.
3. Thomas H.C., Charles E.L., Ronald L.R., Clifford S. Section 33.2: Determining whether any pair of segments intersects // В кн.: Introduction to Algorithms. — Second Edition. MIT Press and McGraw-Hill, 1990. С. 934—947.
4. Чумаченко А.А., Шадричева М.С. Триангуляция Делоне // АЛЛЕЯ НАУКИ, Т. 1, 2017. С. 413-415.
5. Eckart Z., Kalyanmoy D., Lothar T. Comparison of multiobjective evolutionary algorithms: empirical results. // Evolutionary Computation. 2000. Т. 8(2). С. 173–195.
6. Silvano M., Paolo T. Knapsack problems. Wiley, 1990. 306 с.
7. Ed G.G., Punnen A.P. The traveling salesman problem and its variations, 2002.
8. Fruchterman T.M.J., Reingold E.M. Graph drawing by force-directed placement // Software: Practice and Experience. November 1991. Т. 21(11). С. 1129-1164.
9. Land A.H., Doig A.G. An automatic method for solving discrete programming problems. // В кн.: 50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. С. 105-132.
10. Vose M.D. The simple genetic algorithm: foundations and theory. MIT press, 1999.
11. Agapie A., Wright A.H. Theoretical analysis of steady state genetic algorithms // Applications of mathematics. 2014. Т. 59. С. 509-525.
12. Kalyanmoy D., Samir A., Amrit P., Meyarivan T. International conference on parallel problem solving from nature // A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. Berlin. 2000. С. 849-858.
13. Zitzler E., Laumanns M., Thiele L. SPEA2: Improving the strength Pareto evolutionary algorithm // TIK Report. May 2001. Т. 103.
14. Syswerda G. Proc. 3rd Intl Conference on Genetic Algorithms 1989 // Uniform Crossover in Genetic Algorithms. 1989. Т. 3.
15. Goldberg D.E. Genetic Algorithms in Search, Optimization, and Machine Learning. 1989.
16. Vrajitoru D., El-Gamil B..R. Genetic algorithms for graph layouts with geometric constraints // Computational Intelligence., 2006. С. 66-71.

## Приложения

### Приложение 1. Программный код алгоритмов генерации графов

```
func NewRandomGraph(numVertices, numEdges int) *Graph {
    // Проверка корректности количества ребер:
    // - не может быть отрицательным
    // - не может превышать максимальное количество ребер
    // в неориентированном графе без петель: n(n-1)/2
    if numEdges < 0 || numEdges > numVertices*(numVertices-1)/2 {
        panic("edgeNum is out of valid range") // аварийное завершение программы при ошибке
    }

    // Создаем пустой список ребер нужной емкости
    edges := make([]Edge, 0, numEdges)

    // Множество для отслеживания уже добавленных ребер и предотвращения дубликатов
    edgeSet := make(map[string]bool)

    // Пока не будет достигнуто нужное количество ребер
    for len(edges) < numEdges {
        u := rand.IntN(numVertices) // случайная вершина u
        v := rand.IntN(numVertices) // случайная вершина v

        if u == v {
            continue // пропускаем петли (ребро из вершины в саму себя)
        }

        // Упорядочиваем вершины так, чтобы (u, v) и (v, u) считались одинаковыми ребрами
        small, large := u, v
        if u > v {
            small, large = v, u
        }

        // Создаем строковый ключ для текущего ребра (например, "2-5")
        key := fmt.Sprintf("%d-%d", small, large)

        if edgeSet[key] {
            continue // если такое ребро уже есть, пропускаем (не допускаем кратных ребер)
        }

        // Добавляем ребро в множество и список
        edgeSet[key] = true
        edges = append(edges, Edge{From: small, To: large})
    }

    // Возвращаем сгенерированный график
    return &Graph{
        NumVertices: numVertices,
        NumEdges: numEdges,
        Edges: edges,
    }
}
```

```

func NewRandomPlanarGraph(numVertices int) *Graph {
    // 1) Генерация случайных точек внутри единичного квадрата [0, 1] × [0, 1]
    pts := make([]delaunay.Point, numVertices)
    for i := range pts {
        // Случайные координаты X и Y
        pts[i] = delaunay.Point{X: rand.Float64(), Y: rand.Float64()}
    }

    // 2) Вычисление триангуляции Делоне по сгенерированным точкам.
    // Это даст максимальный планарный граф
    // (все возможные ребра, не пересекающие друг друга).
    tri, err := delaunay.Triangulate(pts)
    if err != nil {
        panic(err) // В случае ошибки завершить выполнение с сообщением
    }

    // 3) Извлечение уникальных неориентированных ребер из триангуляции
    edgeMap := make(map[[2]int]struct{}) // Используем map для удаления дубликатов
    for ti := 0; ti < len(tri.Triangles); ti += 3 {
        // Каждая тройка индексов образует один треугольник
        for k := 0; k < 3; k++ {
            // Пары вершин треугольника
            a, b := tri.Triangles[ti+k], tri.Triangles[ti+(k+1)%3]
            if a > b {
                a, b = b, a // Упорядочиваем вершины, чтобы избежать дублирования
            }
            edgeMap[[2]int{a, b}] = struct{}{} // Добавляем неориентированное ребро
        }
    }

    // 4) Преобразование карты ребер в срез (список ребер)
    edges := make([]Edge, 0, len(edgeMap))
    for e := range edgeMap {
        edges = append(edges, Edge{From: e[0], To: e[1]}) // Создаем структуру ребра
    }

    // 5) Построение и возврат структуры графа
    return &Graph{
        NumVertices: numVertices, // Общее количество вершин
        NumEdges: len(edges), // Общее количество ребер
        Edges: edges, // Список ребер
    }
}

```

## Приложение 2. Программный код алгоритма Фрюхтермана-Рейнгольда

```

func (s *ForceDirectedSolver) Iterate() {
    n := s.Graph.NumVertices

    // Вектор смещений для каждого узла (инициализируем нулями).
    // Он будет накапливать результирующую силу, действующую на каждый узел.
    disp := make([]VertexPos, n)

    // Вычисляем силы отталкивания между каждой парой узлов (repulsive force).
    for i := range n {
        for j := i + 1; j < n; j++ {
            // Разность координат между узлами i и j
            dx := s.VertPositions[i].X - s.VertPositions[j].X
            dy := s.VertPositions[i].Y - s.VertPositions[j].Y

            // Расстояние между узлами
            // Добавляем небольшой эпсилон, чтобы избежать деления на ноль
            d := math.Hypot(dx, dy) + 1e-9

            // Сила отталкивания: пропорциональна  $k^2 / d^2$ 
            force := (s.k * s.k) / (d * d)

            // Изменяем смещения для узлов i и j в противоположных направлениях
            disp[i].X += dx * force
            disp[i].Y += dy * force
            disp[j].X -= dx * force
            disp[j].Y -= dy * force
        }
    }

    // Вычисляем силы притяжения вдоль ребер графа (attractive force).
    for u, neigh := range s.adj {
        for _, v := range neigh {
            // Разность координат между связанными узлами u и v
            dx := s.VertPositions[u].X - s.VertPositions[v].X
            dy := s.VertPositions[u].Y - s.VertPositions[v].Y

            // Расстояние между ними
            d := math.Hypot(dx, dy) + 1e-9

            // Сила притяжения: пропорциональна  $d^2 / k$ 
            force := (d * d) / s.k

            // Нормализуем вектор направления
            dxNorm := dx / d
            dyNorm := dy / d

            // Изменяем смещения в направлении притяжения
            disp[u].X -= dxNorm * force
            disp[u].Y -= dyNorm * force
            disp[v].X += dxNorm * force
            disp[v].Y += dyNorm * force
        }
    }

    // Обновляем позиции узлов на основе рассчитанных смещений
    for i := range n {
        dx := disp[i].X
        dy := disp[i].Y

        // Модуль результирующего смещения
        dispMag := math.Hypot(dx, dy)

        if dispMag > 0 {
            // Ограничиваем длину смещения текущей температурой (темпер. = шаг движения)
            scale := math.Min(dispMag, s.temp) / dispMag
            dx *= scale
            dy *= scale
        }

        // Обновляем координаты узла, при этом ограничиваем их размерами области
        s.VertPositions[i].X = clamp(s.VertPositions[i].X+dx, 0, s.Width)
        s.VertPositions[i].Y = clamp(s.VertPositions[i].Y+dy, 0, s.Height)
    }

    // Понижаем температуру (охлаждение), чтобы уменьшать движения со временем
    s.temp -= s.coolingStep
}

```

### Приложение 3. Программный код метода динамического программирования для решения задачи о рюкзаке 0-1

```
// AlgorithmicSolution реализует решение задачи о рюкзаке методом динамического программирования
func (p *KnapsackProblem) AlgorithmicSolution() problems.AlgorithmicSolution {
    // Инициализация базовых параметров
    n := p.Params.ItemsNum           // Количество предметов
    capacity := p.Params.Constraints[0] // Вместимость рюкзака

    // Подготовка массивов весов и ценностей предметов
    weights := make([]int, n) // Веса предметов
    values := make([]int, n) // Ценности предметов
    for i := range n {
        values[i] = p.Items[i].Value      // Извлекаем ценность предмета
        weights[i] = p.Items[i].Resources[0] // Извлекаем вес предмета (первый ресурс)
    }

    // Инициализация таблицы динамического программирования
    // dp[i][w] – максимальная ценность для первых i предметов и веса w
    dp := make([][]int, n+1)
    for i := range dp {
        dp[i] = make([]int, capacity+1) // +1 для работы с нулевыми индексами
    }

    // Заполнение таблицы DP
    for i := 1; i <= n; i++ { // Перебор предметов
        for w := 1; w <= capacity; w++ { // Перебор весов
            currentWeight := weights[i-1] // Вес текущего предмета (индекс i-1 из-за смещения)

            // Если предмет помещается в текущий рюкзак
            if currentWeight <= w {
                include := values[i-1] + dp[i-1][w-currentWeight] // Ценность с текущим предметом
                exclude := dp[i-1][w]                                // Ценность без текущего предмета
                dp[i][w] = max(include, exclude)                      // Выбираем лучший вариант
            } else {
                dp[i][w] = dp[i-1][w] // Предмет не помещается – берем предыдущий максимум
            }
        }
    }

    // Восстановление решения
    selectedBits := make([]bool, n) // Массив выбранных предметов
    for i, w := n, capacity; i > 0; i-- {
        // Если ценность изменилась относительно предыдущего шага
        if dp[i][w] != dp[i-1][w] {
            selectedBits[i-1] = true // Помечаем предмет как выбранный
            w -= weights[i-1]        // Уменьшаем оставшийся доступный вес
        }
    }

    // Возвращаем результат в формате AlgorithmicSolution
    return problems.AlgorithmicSolution{
        Solution: &KnapsackSolution{
            problemParams: p.Params,
            items:         p.Items,
            Bits:          selectedBits, // Бинарный вектор выбранных предметов
        },
    }
}
```

## Приложение 4. Программный код метода ветвей и границ

```

// Метод ветвей и границ
func (p *TSPProblem) AlgorithmicSolution() problems.AlgorithmicSolution {
    startTime := time.Now()
    n := p.Params.CitiesNum

    log.Printf(`Начало решения задачи коммивояжера методом ветвей и границ для %d городов`, n)

    bestCost := math.Inf(1) // Лучшая (наименьшая) стоимость маршрута на данный момент
    var bestPath []int // Лучший путь (включая стартовый город)

    // Предварительный расчет: для каждого города находим минимальное расстояние до любого другого
    // города
    minEdge := make([]float64, n)
    for i := 0; i < n; i++ {
        min := math.Inf(1)
        for j := 0; j < n; j++ {
            if i != j {
                d := p.Cities[i].Distance(p.Cities[j])
                if d < min {
                    min = d
                }
            }
        }
        minEdge[i] = min
    }

    var nodesVisited int64 // Счетчик посещенных узлов (для статистики)

    // Рекурсивная функция поиска с отсечением ветвей
    var dfs func(current, count int, currCost float64, path []int, visited []bool)
    dfs = func(current, count int, currCost float64, path []int, visited []bool) {
        nodesVisited++

        // Каждые 10 миллионов шагов выводим прогресс
        if nodesVisited%10_000_000 == 0 {
            log.Printf(`Посещено узлов: %d, текущая лучшая стоимость: %.5f`, nodesVisited, bestCost)
        }

        // Если все города посещены, добавляем обратный путь в начальный город
        if count == n {
            totalCost := currCost + p.Cities[current].Distance(p.Cities[0])
            if totalCost < bestCost {
                bestCost = totalCost
                bestPath = make([]int, len(path))
                copy(bestPath, path)
                log.Printf(`Найден новый лучший путь со стоимостью: %.5f после %d узлов`, bestCost,
nodesVisited)
            }
            return
        }

        // Оценка нижней границы стоимости для текущего частичного маршрута
        lb := currCost + minEdge[current]
        for i := range n {
            if !visited[i] {
                lb += minEdge[i]
            }
        }

        // Если нижняя граница не лучше текущего наилучшего результата – отсечение ветви
        if lb >= bestCost {
            return
        }

        // Продолжаем построение маршрута, пробуем все непосещенные города
        for i := 1; i < n; i++ {
            if !visited[i] {
                visited[i] = true
                dfs(i, count+1, currCost+p.Cities[current].Distance(p.Cities[i]), append(path, i),
visited)
                visited[i] = false
            }
        }
    }
}

```

```

// Инициализация: стартуем из города 0
visited := make([]bool, n)
visited[0] = true
dfs(0, 1, 0.0, []int{0}, visited)

elapsed := time.Since(startTime)
log.Printf(`«Рекурсивный обход завершен. Всего узлов: %d, Лучшая стоимость: %.5f, Время
выполнения: %v», nodesVisited, bestCost, elapsed)

// Формирование решения: исключаем стартовый город из VisitingOrder
visitingOrder := bestPath[1:]
bestSolution := TSPSolution{
    problemParams: p.Params,
    cities:         p.Cities,
    VisitingOrder: visitingOrder,
    CachedFitness: 1.0 / bestCost,
}

return problems.AlgorithmicSolution{
    Solution: &bestSolution,
    TimeTook: elapsed,
}
}

```

## Приложение 5. Программный код метода полного перебора для решения симметричной задачи о коммивояжере

```
// AlgorithmicSolution реализует полный перебор для задачи коммивояжера (TSP)
func (p *TSPProblem) AlgorithmicSolution() problems.AlgorithmicSolution {
    // 1. Инициализация списка городов (исключаем начальный город 0)
    cities := make([]int, p.Params.CitiesNum-1)
    for i := range p.Params.CitiesNum - 1 {
        cities[i] = i + 1 // Создаем список [1, 2, ..., CitiesNum-1]
    }

    // 2. Инициализация начального решения (прямой порядок городов)
    bestSolution := TSPSolution{
        problemParams: p.Params,
        cities:         p.Cities,
        VisitingOrder: cities, // Начальный маршрут: 0 → 1 → 2 → ... → N → 0
    }

    // 3. Полный перебор всех возможных перестановок
    for _, order := range permutations(cities) {
        // Создаем кандидата с текущей перестановкой
        solution := TSPSolution{
            problemParams: p.Params,
            cities:         p.Cities,
            VisitingOrder: order,
        }

        // 4. Проверка на улучшение решения
        if solution.Fitness() > bestSolution.Fitness() {
            bestSolution = solution // Обновляем лучшее решение
        }
    }

    // 5. Возвращаем оптимальное решение
    return problems.AlgorithmicSolution{
        Solution: &bestSolution,
    }
}
```

## Приложение 6. Программный код SGA

```
// Evolve выполняет один шаг эволюции для Simple GA (SGA)
func (alg *Algorithm) Evolve() {
    // 1. Оценка текущего поколения (сортировка по приспособленности)
    alg.evaluateGeneration()

    // 2. Создание новой популяции
    newPopulation := make([]problems.Solution, 0, alg.params.PopulationSize)

    // 3. Применение элитизма: сохранение лучших решений
    newPopulation = append(newPopulation, alg.population[:alg.eliteSize]...)
    // alg.eliteSize – количество сохраняемых элитных особей
    // alg.population должен быть отсортирован в порядке ухудшения Fitness()

    // 4. Генерация оставшейся части популяции
    for len(newPopulation) < alg.params.PopulationSize {
        // 4.1. Случайный выбор двух разных родителей из пула
        p1Ind := rand.IntN(alg.matingPoolSize) // Индекс первого родителя
        p2Ind := rand.IntN(alg.matingPoolSize) // Индекс второго родителя
        if p1Ind == p2Ind {
            continue // Исключаем скрещивание особи с самой собой
        }

        // 4.2. Получение родителей из популяции
        parent1 := alg.population[p1Ind]
        parent2 := alg.population[p2Ind]

        // 4.3. Скрещивание для получения потомков
        children := alg.params.CrossoverFunc(parent1, parent2)
        // CrossoverFunc реализует специфичную для задачи логику скрещивания

        // 4.4. Обработка и добавление потомков
        for _, child := range children {
            // Применение мутации к потомку
            child = alg.params.MutationFunc(child)

            // Добавление в новую популяцию с контролем размера
            newPopulation = append(newPopulation, child)
            if len(newPopulation) >= alg.params.PopulationSize {
                break // Прерывание при достижении целевого размера
            }
        }
    }

    // 5. Замена старой популяции новой
    alg.population = newPopulation
}
```

## Приложение 7. Программный код SSGA

```
// Evolve выполняет один шаг эволюции для Steady-State GA (SSGA)
func (alg *Algorithm) Evolve() {
    // 1. Оценка текущего поколения (сортировка по приспособленности)
    alg.evaluateGeneration()

    // 2. Флаг успешной замены особей
    replaced := false

    // 3. Цикл гарантирует замену минимум одной особи
    for !replaced {
        // 3.1. Турнирный отбор родителей
        p1Ind := alg.tournamentSelect() // Индекс первого родителя
        p2Ind := alg.tournamentSelect() // Индекс второго родителя

        // 3.2. Проверка на разных родителей
        if p1Ind == p2Ind {
            continue // Пропуск инбридинга
        }

        // 3.3. Получение родителей
        parent1 := alg.population[p1Ind] // Лучший из турнира 1
        parent2 := alg.population[p2Ind] // Лучший из турнира 2

        // 3.4. Генерация потомков
        children := alg.params.CrossoverFunc(parent1, parent2)

        // 3.5. Обработка потомков
        for i := range children {
            // Применение мутации
            children[i] = alg.params.MutationFunc(children[i])

            // Замена худших особей в конце популяции
            // PopulationSize-i-1: последовательно заменяем с конца
            alg.population[alg.params.PopulationSize-i-1] = children[i]
        }
    }

    // 4. Подтверждение замены
    replaced = true // Выход из цикла
}
```

## Приложение 8. Программный код NSGA2 и быстрой не-доминирующей сортировки

```
func (alg *Algorithm) Run() {
    // Если популяция еще не инициализирована полностью, создаем начальную популяцию.
    if len(alg.population) < alg.params.PopulationSize {
        alg.initPopulation()
    }

    // Основной цикл: повторяется, пока не достигнут лимит по поколениям.
    for alg.generation < alg.GenerationLimit {
        alg.generation++ // Увеличиваем счетчик поколения.

        // Генерируем потомков (offspring) с помощью селекции, скрещивания и мутации.
        offspring := alg.makeOffspring()

        // Объединяем родительскую и дочернюю популяции.
        combined := append(alg.population, offspring...)

        // Применяем быструю не-доминирующую сортировку (fast non-dominated sorting).
        // Разбиваем объединенную популяцию на фронты (fronts) по уровню доминирования.
        fronts := fastNonDominatedSort(combined)

        // Создаем новую популяцию следующего поколения.
        nextPopulation := make([]Individual, 0, alg.params.PopulationSize)

        // Проходим по каждому фронту в порядке приоритета.
        for _, front := range fronts {
            // Вычисляем расстояние разреженности (crowding distance) для фронта.
            // Это помогает сохранить разнообразие решений внутри фронта.
            computeCrowdingDistance(front)

            // Если добавление всего фронта превысит размер популяции:
            if len(nextPopulation)+len(front) > alg.params.PopulationSize {
                // Сортируем фронт по убыванию расстояния разреженности.
                // Чем больше расстояние – тем более "разнообразное" и ценное решение.
                sort.Slice(front, func(i, j int) bool {
                    return front[i].CrowdingDistance > front[j].CrowdingDistance
                })

                // Добавляем столько особей из фронта,
                // сколько нужно до полного заполнения популяции.
                remaining := alg.params.PopulationSize - len(nextPopulation)
                nextPopulation = append(nextPopulation, front[:remaining]...)

                // Прекращаем добавление фронтов, так как популяция заполнена.
                break
            } else {
                // Если фронт помещается целиком – добавляем его целиком.
                nextPopulation = append(nextPopulation, front...)
            }
        }

        // Обновляем текущую популяцию.
        alg.population = nextPopulation
    }
}
```

```

func fastNonDominatedSort(pop []Individual) [][]Individual {
    fronts := [][]Individual{} // Список фронтов (каждый фронт – срез Individuals).
    n := len(pop) // Размер популяции.

    // domCount[i] – количество особей, доминирующих над особью i.
    domCount := make([]int, n)

    // dominatedSet[i] – список индексов особей, которые доминирует особь i.
    dominatedSet := make([][]int, n)

    // Первый проход: сравниваем каждую пару особей в популяции.
    for i := range n {
        dominatedSet[i] = []int{} // Инициализируем пустое множество доминируемых особей.
        for j := range n {
            if i == j {
                continue // Пропускаем сравнение особи с самой собой.
            }
            // Если i доминирует j, добавляем j в список доминируемых i.
            if dominates(pop[i].Solution, pop[j].Solution) {
                dominatedSet[i] = append(dominatedSet[i], j)
            } else if dominates(pop[j].Solution, pop[i].Solution) {
                // Если j доминирует i, увеличиваем счетчик domCount для i.
                domCount[i]++
            }
        }
    }

    currentFront := []int{} // Индексы особей нулевого фронта.

    // Находим особей, которых никто не доминирует (domCount == 0) – они попадают в первый фронт.
    for i := range n {
        if domCount[i] == 0 {
            pop[i].Rank = 0 // Присваиваем ранг 0.
            currentFront = append(currentFront, i)
        }
    }

    curRank := 0 // Текущий ранг фронта.
    for len(currentFront) > 0 {
        var front []Individual // Список особей текущего фронта.
        nextFront := []int{} // Индексы особей следующего фронта.

        // Обрабатываем текущий фронт.
        for _, i := range currentFront {
            pop[i].Rank = curRank // Устанавливаем ранг текущей особи.
            front = append(front, pop[i]) // Добавляем особь в фронт.

            // Уменьшаем domCount для всех особей, доминируемых данной.
            for _, j := range dominatedSet[i] {
                domCount[j]--
                if domCount[j] == 0 {
                    // Если особь j больше никто не доминирует – она попадает в следующий фронт.
                    nextFront = append(nextFront, j)
                }
            }
        }

        fronts = append(fronts, front) // Сохраняем текущий фронт.
        curRank++ // Переходим к следующему рангу.
        currentFront = nextFront // Обрабатываем следующий фронт на следующей итерации.
    }

    return fronts // Возвращаем список фронтов.
}

```

## Приложение 9. Программный код SPEA2

```
func (alg *Algorithm) Run() {
    // Инициализация начальной популяции.
    alg.initPopulation()

    // Архив элитных решений изначально пуст.
    alg.archive = nil

    // Основной эволюционный цикл.
    for alg.generation < alg.GenerationLimit {
        alg.generation++ // Увеличиваем счетчик поколения.

        // Объединяем текущую популяцию и архив элитных решений.
        combined := slices.Concat(alg.population, alg.archive)

        // Вычисляем приспособленность для всех решений:
        // сила доминирования, сырья приспособленность, плотность и итоговая оценка.
        alg.assignFitness(combined)

        // Обновляем архив элитных решений на основе приспособленности.
        alg.updateArchive(combined)

        // Генерируем новое поколение потомков.
        alg.reproduce()
    }
}

// assignFitness вычисляет силу доминирования, сырью приспособленность, плотность и итоговую оценку.
func (alg *Algorithm) assignFitness(all []Individual) {
    // 1) Вычисляем силу (strength) для каждого решения.
    // Сила – это количество других решений, которые доминируются данным решением.
    for i := range all {
        all[i].strength = 0
        for j := range all {
            if dominates(all[i].sol, all[j].sol) {
                all[i].strength++
            }
        }
    }

    // 2) Вычисляем сырью приспособленность (rawFit).
    // Суммируем силы всех решений, которые доминируют текущее.
    for i := range all {
        all[i].rawFit = 0
        for j := range all {
            if dominates(all[j].sol, all[i].sol) {
                all[i].rawFit += all[j].strength
            }
        }
    }

    // 3) Оцениваем плотность (density).
    // Используем расстояние до k-го ближайшего соседа.
    for i := range all {
        all[i].density = computeKthDist(all, i, alg.params.DensityKth)
        // Формула из статьи: плотность = 1 / (расстояние + 2)
        all[i].density = 1.0 / (all[i].density + 2.0)
    }

    // 4) Итоговая приспособленность = сырья + плотность.
    for i := range all {
        all[i].fitness = all[i].rawFit + all[i].density
    }
}

// updateArchive обновляет архив элитных решений с помощью отбора и усечения.
func (alg *Algorithm) updateArchive(combined []Individual) {
    // 1) Отбираем все решения с rawFit < 1 (недоминируемые).
    nd := filter(combined, func(ind Individual) bool {
        return ind.rawFit < 1
    })

    // 2) Если решений слишком мало – заполняем архив доминируемыми.
    if len(nd) < alg.params.ArchiveSize {
        nd = append(nd, selectDominated(combined, alg.params.ArchiveSize-len(nd))...)
    }
}
```

```

    }

    // 3) Если решений слишком много – применяем k-ближайшее усечение (k-NN truncation).
    alg.archive = truncateToSize(nd, alg.params.ArchiveSize)
}

// reproduce создает новое поколение с помощью бинарного турнира, скрещивания и мутации.
func (alg *Algorithm) reproduce() {
    var nextP []Individual

    // Генерация до тех пор, пока не наберем нужное количество решений.
    for len(nextP) < alg.params.PopulationSize {
        // Бинарный турнирный отбор для двух родителей.
        p1 := alg.tournamentSelect()
        p2 := alg.tournamentSelect()

        // Скрещивание двух родителей → потомки.
        children := alg.params.CrossoverFunc(p1.sol, p2.sol)

        // Применяем мутацию к каждому потомку и добавляем его в новое поколение.
        for _, child := range children {
            child = alg.params.MutationFunc(child)
            nextP = append(nextP, Individual{sol: child})
        }
    }

    // Обновляем популяцию новым поколением.
    alg.population = nextP
}

```

## Приложение 10. Графики сравнения ГА для задачи укладки графов

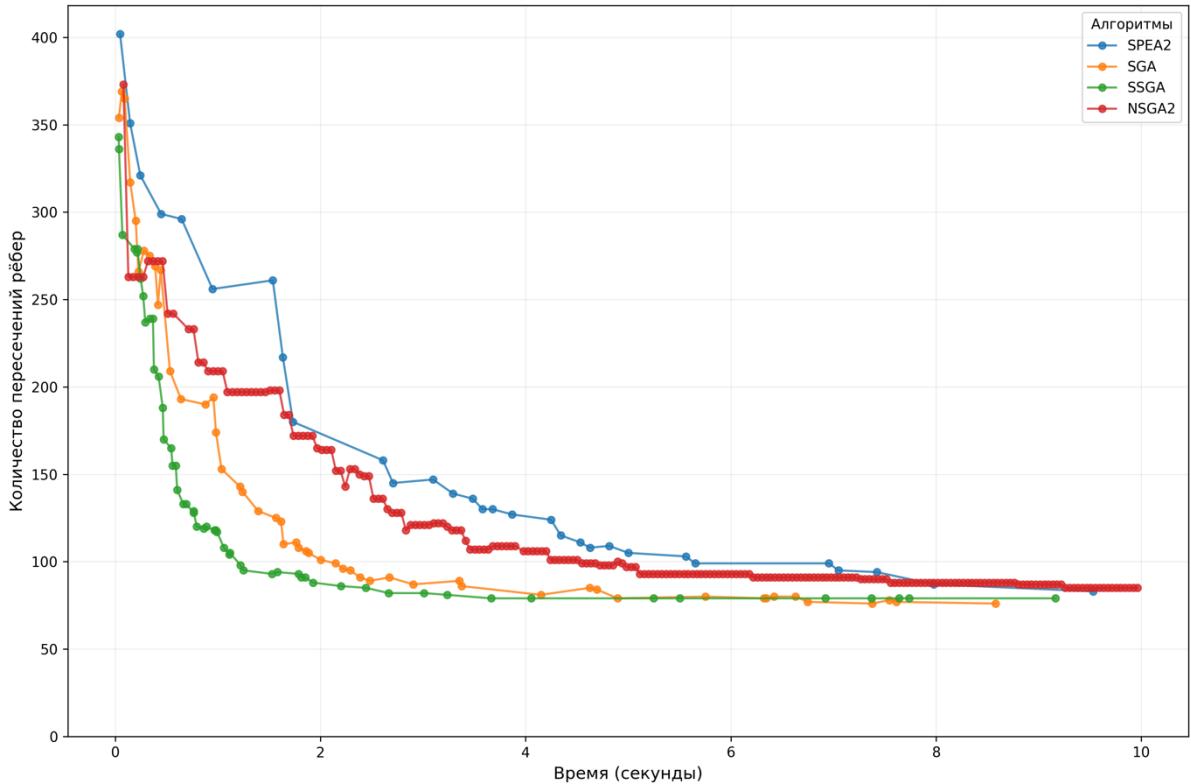


Рисунок 23 – Зависимость количества пересечений ребер от времени для ГА (ПР25)

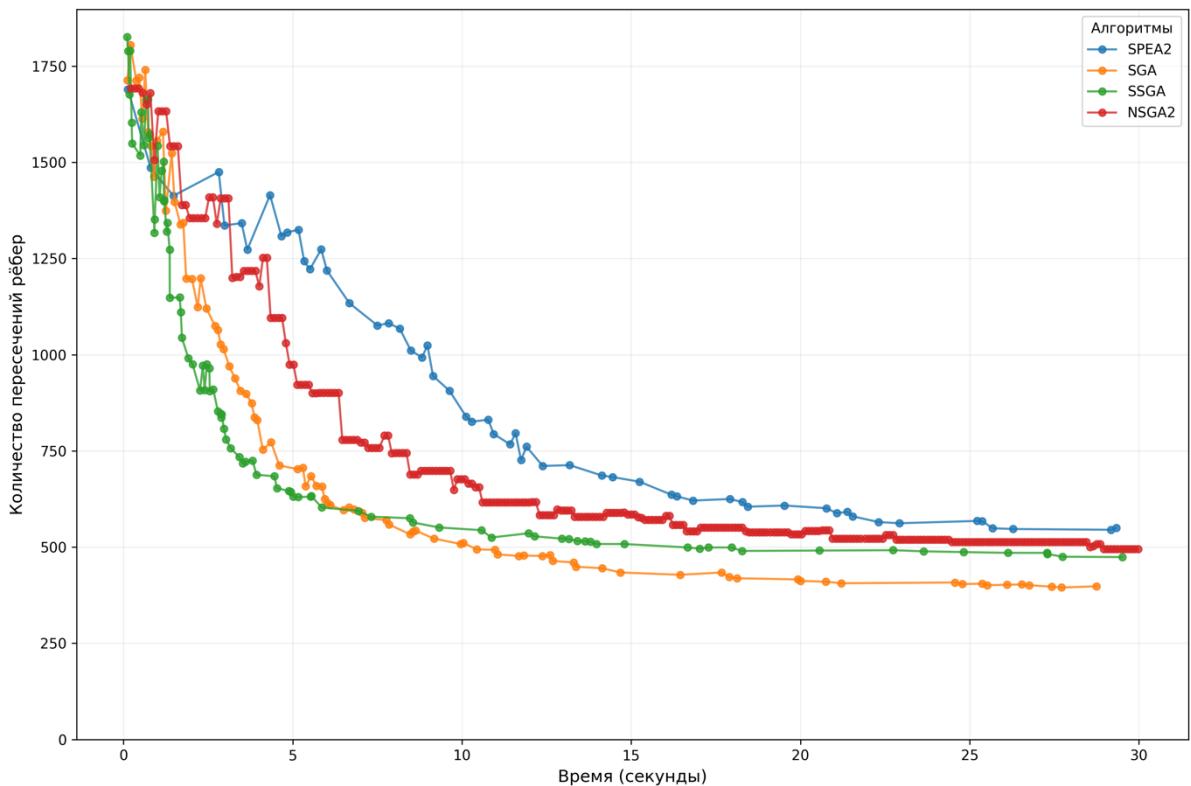


Рисунок 24 – Зависимость количества пересечений ребер от времени для ГА (ПР50)

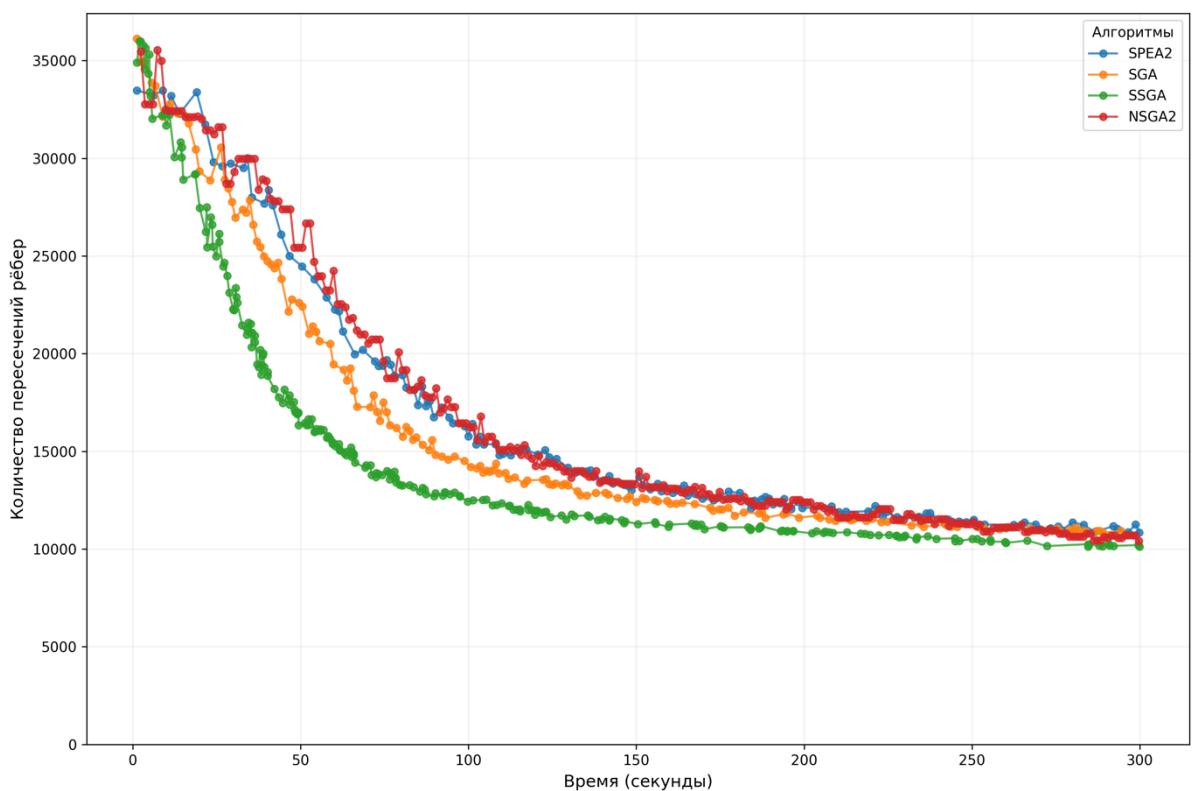


Рисунок 25 – Зависимость количества пересечений ребер от времени для ГА (ПР200)

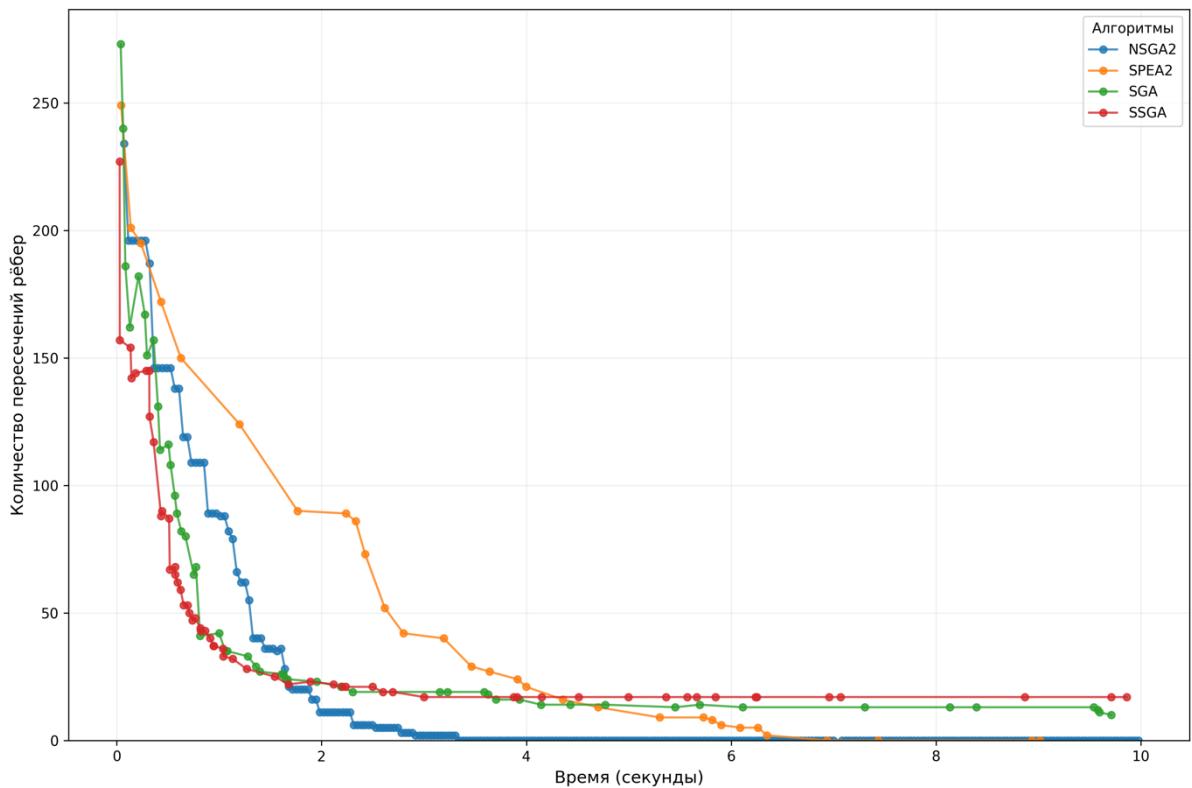


Рисунок 26 – Зависимость количества пересечений ребер от времени для ГА (ПЛ25)

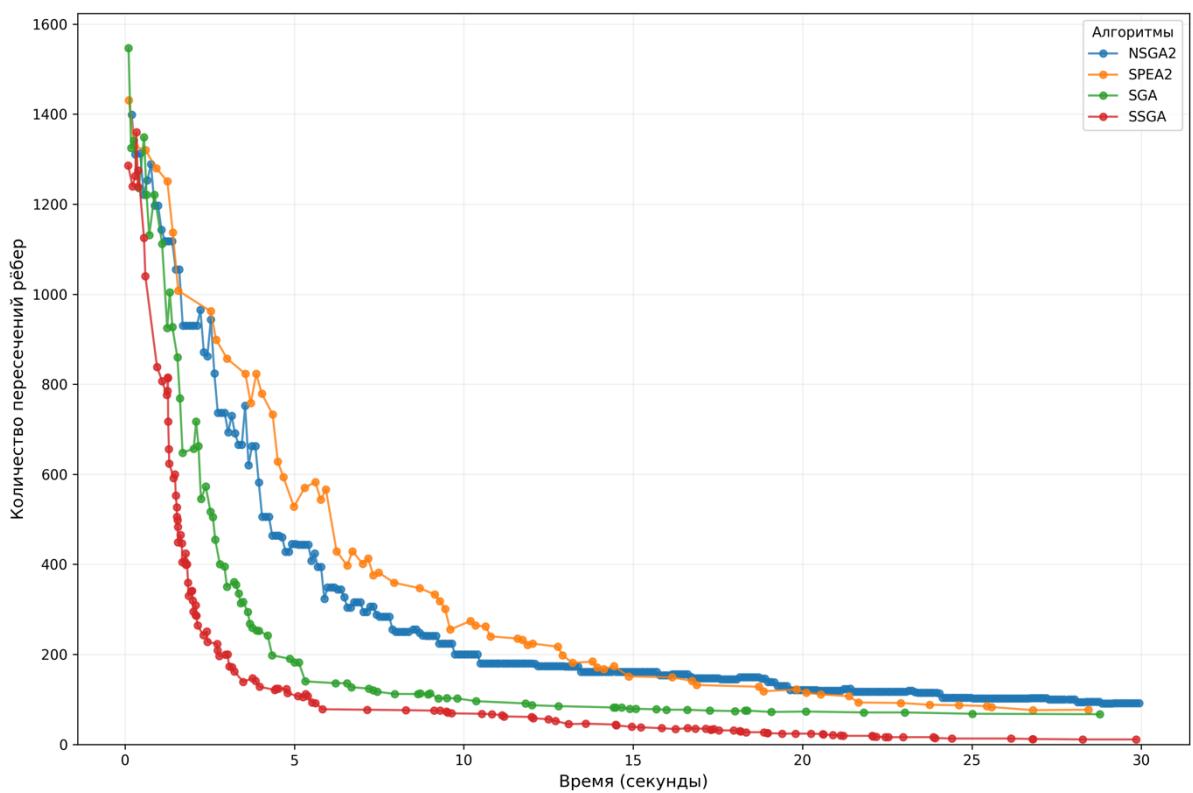


Рисунок 27 – Зависимость количества пересечений ребер от времени для ГА (ПЛ50)

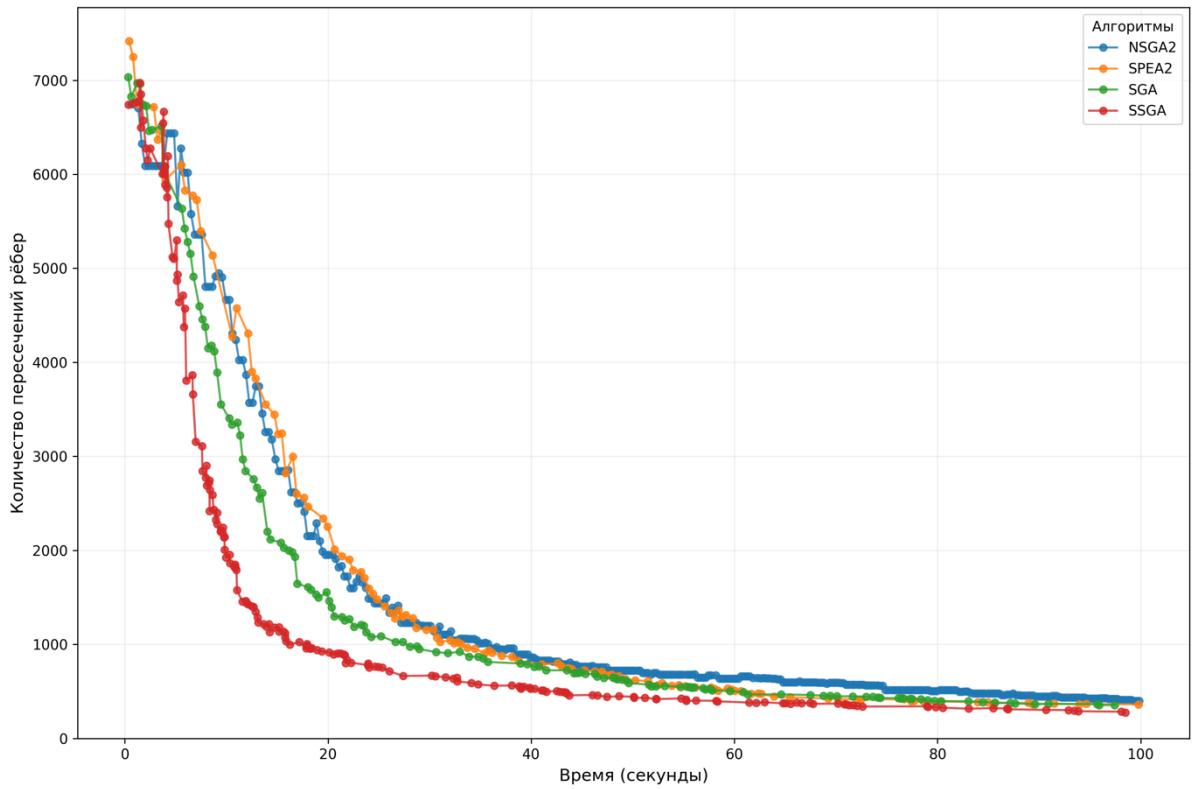


Рисунок 28 – Зависимость количества пересечений ребер от времени для ГА (ПЛ100)

## Приложение 11. Фронты Парето для задач из набора ZDT, построенные ГА

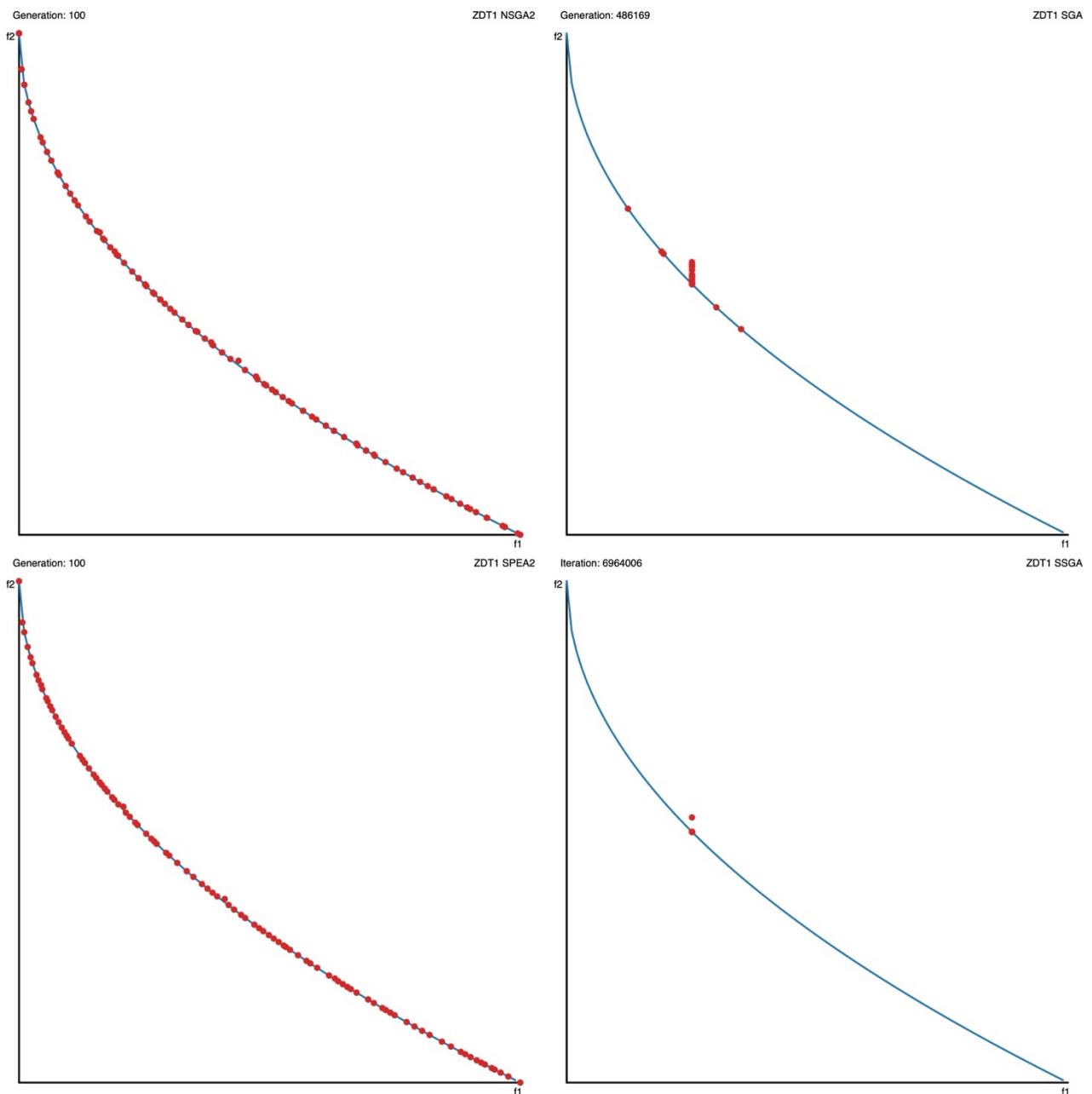


Рисунок 29 – Фронты для ZDT1

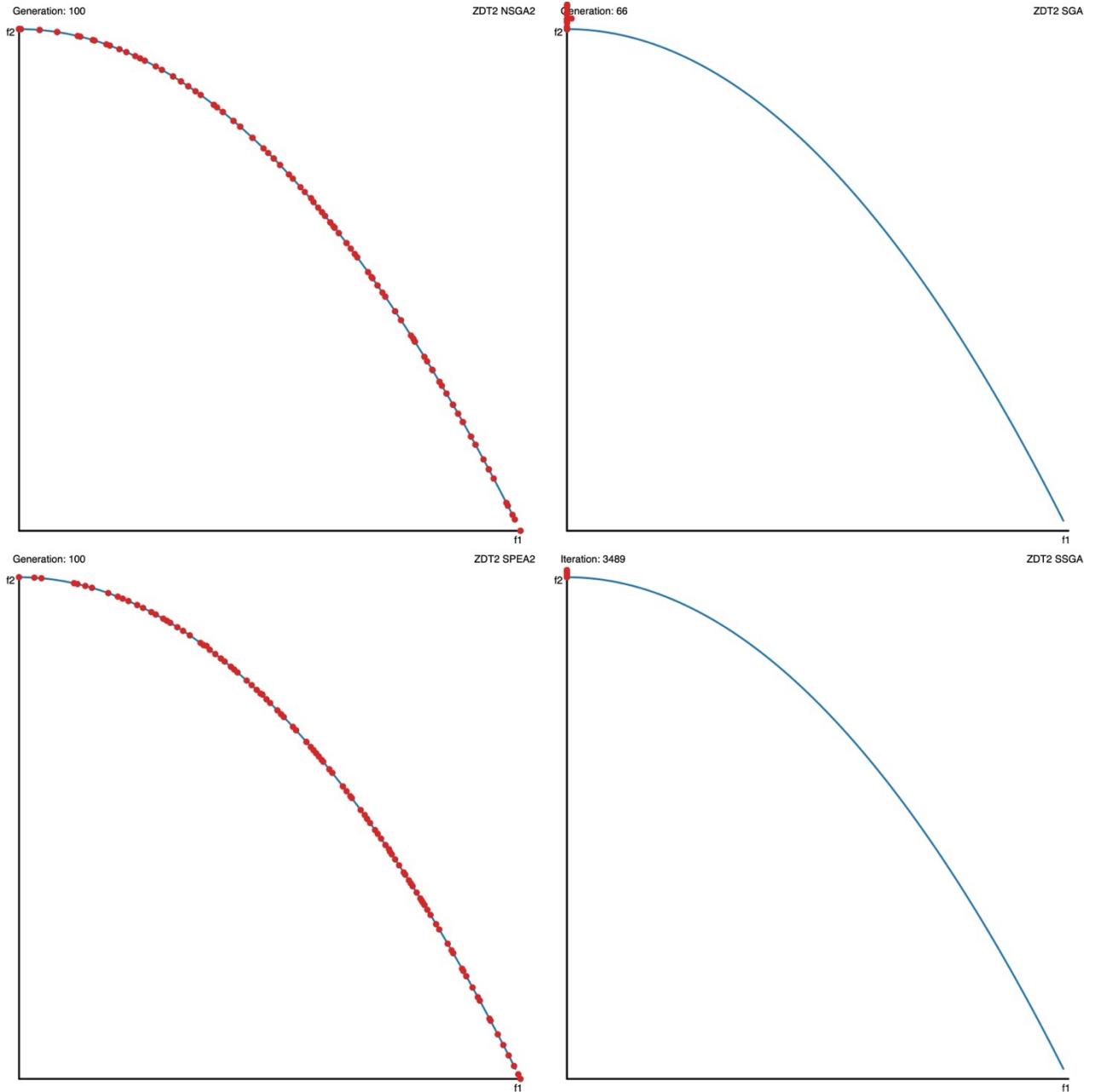


Рисунок 30 – Фронты для ZDT2

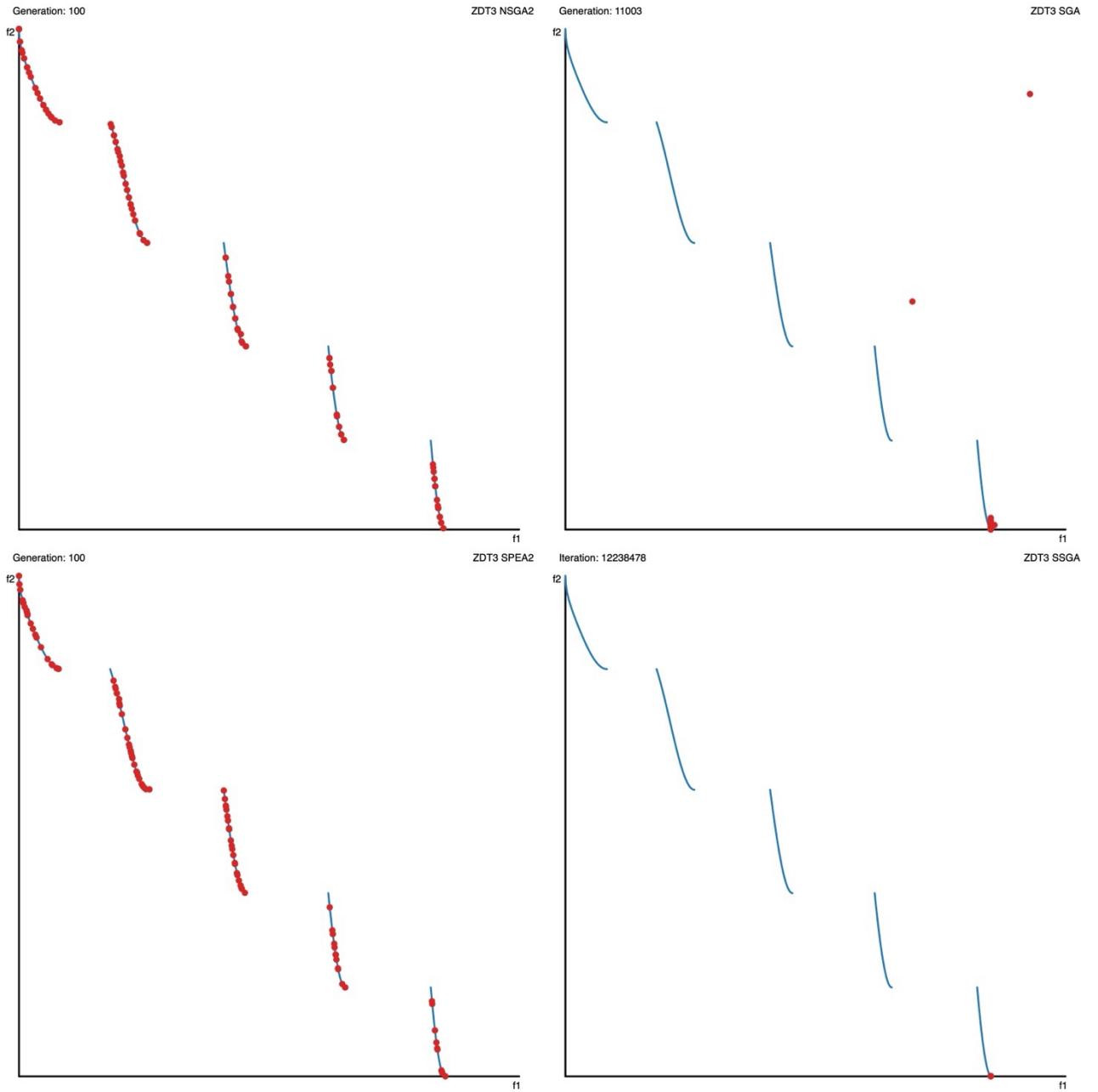


Рисунок 31 – Фронты для ZDT3

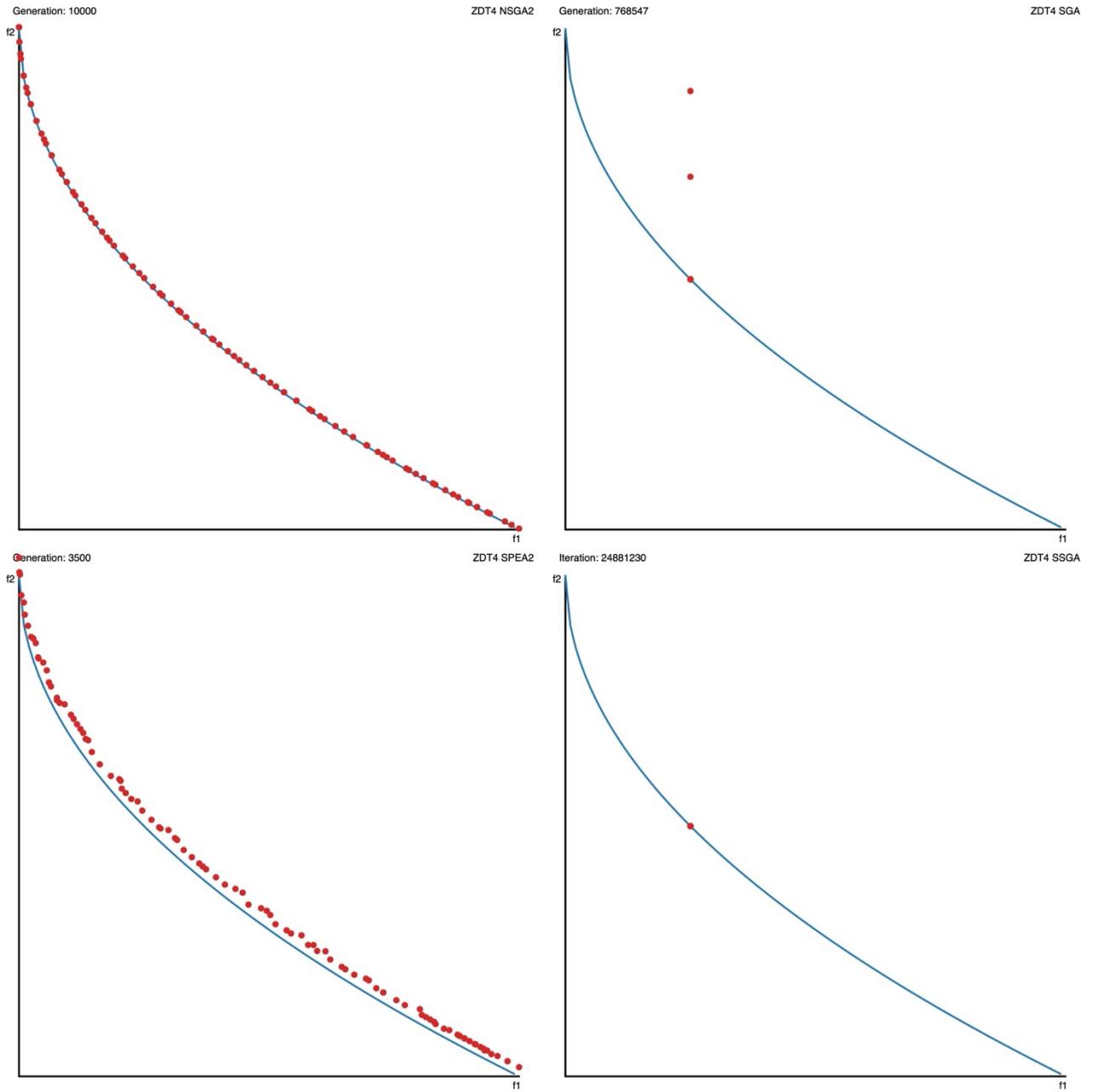


Рисунок 32 – Фронты для ZDT4

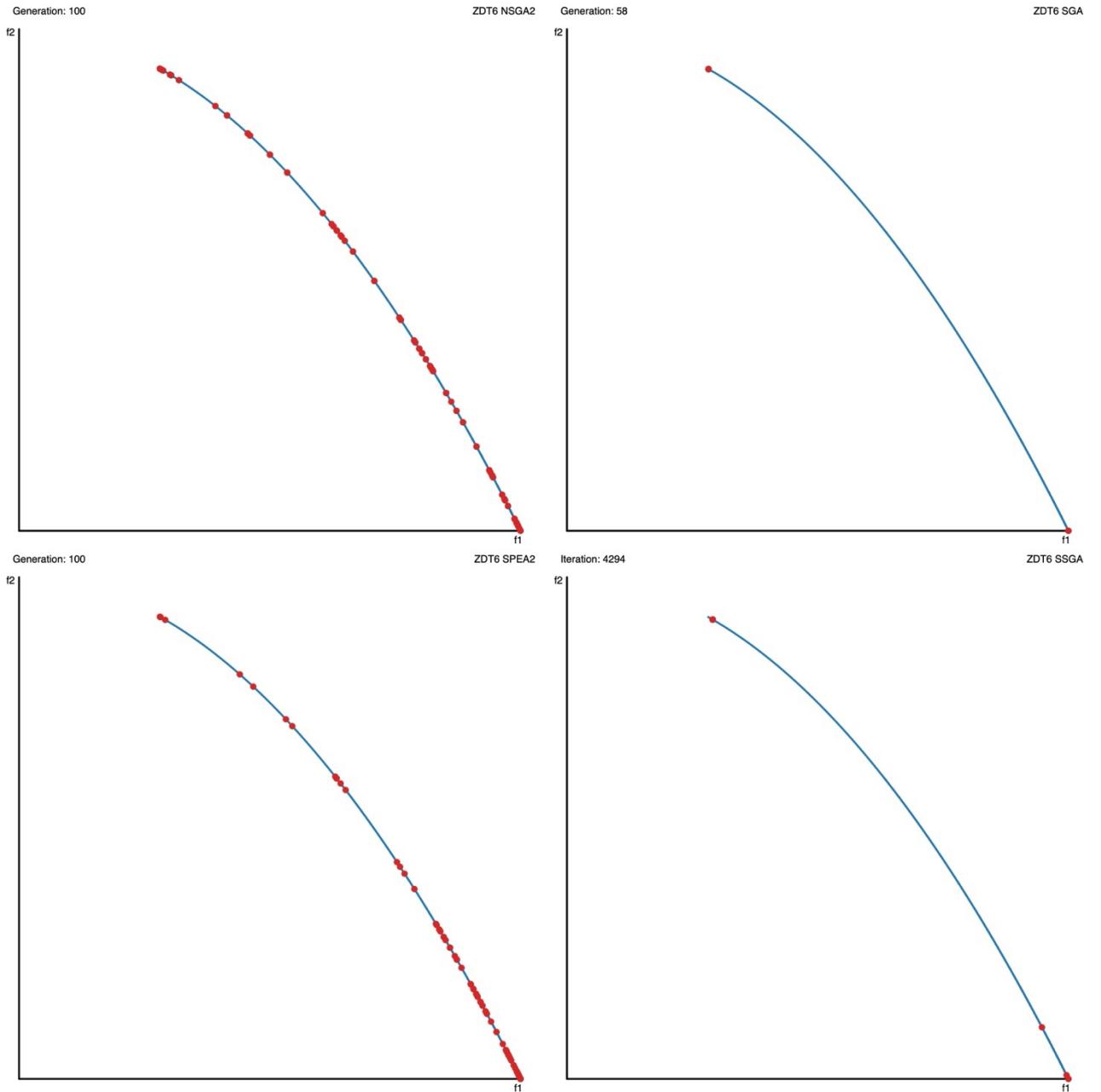


Рисунок 33 – Фронты для ZDT6

## Приложение 12. Программный код оператора консервативной нормальной мутации

```
func ConservativeNorm(k float64) problems.MutationFunc {
    // Возвращает функцию мутации, которая реализует консервативную нормальную мутацию.
    // k – коэффициент масштабирования для стандартного отклонения нормального распределения.
    return func(individual problems.Solution) problems.Solution {
        // Проверяем, что переданная особь является решением задачи укладки графа.
        s, ok := individual.(*graphplane.GraphPlaneSolution)
        if !ok {
            // Если тип не соответствует, вызываем панику,
            // так как это указывает на ошибку в использовании.
            panic("invalid individual")
        }

        // Создаем новую особь (потомка) на основе текущей особи 's'.
        // Это важно, чтобы не изменять исходную особь напрямую,
        // так как мутация должна возвращать новую особь.
        m := &graphplane.GraphPlaneSolution{Graph: s.Graph, Width: s.Width, Height: s.Height}
        // Инициализируем слайс для хранения позиций вершин новой особи.
        m.VertPositions = make([]graphplane.VertexPos, len(s.VertPositions))
        // Копируем позиции вершин из родительской особи в новую.
        copy(m.VertPositions, s.VertPositions)

        // Случайно выбираем индекс вершины, которая будет мутирована.
        // rand.IntN(n) возвращает случайное целое число в диапазоне [0, n).
        i := rand.IntN(len(m.VertPositions))

        // Сохраняем текущее количество пересечений ребер до мутации.
        // Это необходимо для проверки, не ухудшило ли мутация качество решения.
        oldIntersections := m.CountIntersections()
        // Сохраняем исходные координаты выбранной вершины.
        // Это позволит "откатить" мутацию, если она приведет к ухудшению.
        oldX := m.VertPositions[i].X
        oldY := m.VertPositions[i].Y

        // Генерируем случайные смещения для координат
        // X и Y из нормального распределения.
        // rand.NormFloat64() возвращает случайное число из стандартного
        // нормального распределения (среднее 0, ст. отклонение 1).
        // Смещения масштабируются на основе ширины/высоты плоскости и коэффициента 'k',
        // что позволяет контролировать "силу" мутации.
        dx := rand.NormFloat64() * s.Width * k
        dy := rand.NormFloat64() * s.Height * k
        // Применяем смещения к выбранной вершине.
        // clamp ограничивает новые координаты в пределах
        // допустимого диапазона [0, Width] и [0, Height].
        m.VertPositions[i].X = clamp(m.VertPositions[i].X+dx, 0, s.Width)
        m.VertPositions[i].Y = clamp(m.VertPositions[i].Y+dy, 0, s.Height)

        // Проверяем, увеличилось ли количество пересечений после мутации.
        // Если новое количество пересечений больше, чем старое, мутация считается неудачной.
        if oldIntersections < m.CountIntersections() {
            // Откатываем изменения: восстанавливаем исходные координаты вершины.
            m.VertPositions[i].X = oldX
            m.VertPositions[i].Y = oldY
        }

        // Возвращаем мутированную (или не мутированную, если мутация ухудшила результат) особь.
        return m
    }
}
```