Сравнительный анализ Genetic Engineering Algorithm (GEA) с классическими и современными модификациями генетических алгоритмов для решения задачи о рюкзаке

Выполнила: Комольцева Диана Витальевна МНКДН251

Аннотация

Данный проект посвящен комплексному анализу алгоритма Genetic Engineering Algorithm (GEA), предложенного в оригинальной статье Sohrabi et al. (2023). Основная цель исследования - провести всестороннее сравнение эффективности GEA не только с классическим генетическим алгоритмом (GA), но и с другими популярными модификациями GA.
В работе реализованы и протестированы пять алгоритмов: стандартный GA, оригинальный GEA, GA с турнирной селекцией, адаптивный GA и гибридный GA с локальным поиском. Для оценки производительности алгоритмов использовалась классическая задача о рюкзаке с различными размерами и конфигурациями.
Особое внимание уделено анализу скорости сходимости, качеству получаемых решений, стабильности работы алгоритмов и их вычислительной сложности.

Постановка задачи

Генетические алгоритмы (GA) широко применяются для решения задач комбинаторной оптимизации благодаря своей универсальности и способности находить приближенные решения в сложных дискретных пространствах. Однако классический GA обладает рядом известных ограничений:

- Медленная сходимость при больших пространствах решений;
- Высокая дисперсия результатов;
- Склонность к преждевременной сходимости, особенно в условиях малой популяции или ограниченного числа поколений.
  В оригинальной статье “Genetic Engineering Algorithm (GEA): An Efficient Metaheuristic Algorithm for Solving Combinatorial Optimization Problems” (Sohrabi et al., 2023) представлен новый метаэвристический алгоритм, вдохновленный идеями генной инженерии. Авторы демонстрируют эффективность GEA на нескольких тестовых задачах в сравнении с классическим генетическим алгоритмом.
  Однако статья имеет ограничения, сравнение проводится только с базовой версией GA, и нет сравнения с современными модификациями GA, такими как адаптивные и гибридные алгоритмы. В связи с этим, в текущем исследовании мы ставим следующие цели:

1. Реализация и верификация GEA согласно описанию в оригинальной статье;
2. Расширенное сравнение GEA с различными модификациями GA;
3. Анализ эффективности алгоритмов по множеству критериев: качество решения, сходимость, стабильность.

Анализ литературы

4 Генетический алгоритм и его модификации

Генетические алгоритм это эволюционный алгоритм, имитирующий процесс естественного отбора. Основная цель — поиск приближенного оптимального решения в сложных пространствах.
Генетические алгоритмы, впервые предложенные Холландом в 1975 году, представляют собой эвристические методы оптимизации, основанные на принципах естественного отбора.
Основные шаги алгоритма:

1. **Инициализация популяции** — создание случайного множества возможных решений (хромосом).
2. **Оценка пригодности** — вычисление функции пригодности (fitness) для каждой хромосомы.
3. **Селекция** — выбор родительских особей для размножения, обычно с учетом их пригодности (например, рулеточный или турнирный отбор).
4. **Скрещивание (кроссовер)** — обмен генетической информации между родителями для создания новых потомков.
5. **Мутация** — случайные изменения отдельных генов для поддержания разнообразия.
6. **Замещение популяции** — формирование нового поколения.
   Цикл повторяется до достижения критерия остановки (например, максимальное число поколений или удовлетворительное качество решения).

Метод универсальный, достаточно простотой в реализации, может находить хорошие приближенные решения, но имеет медленную сходимость и риск преждевременной стагнации.
Вычислительная сложность:

$$O(G \cdot N \cdot f)$$

где G — число поколений, N — размер популяции, f — стоимость вычисления функции пригодности.

4.1 Genetic Engineering Algorithm

GEA — это модификация классического GA, вдохновленная процессами генной инженерии. Основная идея — **снизить случайность поиска и направленно улучшать хромосомы** для более эффективного поиска решений.
Особенности GEA:

1. **Инициализация и оценка пригодности** — аналогично классическому GA.

2. **Селекция** — как в GA, но с возможным усилением давления отбора на основе «качества инженерных изменений».
3. **Инженерные операции над хромосомами** — вместо чисто случайного кроссовера и мутации применяются **направленные модификации**, которые усиливают полезные гены и уменьшают влияние вредных.
4. **Дополнительные эвристики** — включают выбор лучших частей хромосом, рекомбинацию только перспективных генов, адаптивное управление мутацией.
5. **Замещение и итерации** — аналогично GA
   Основные преимущества:

- Более быстрый и целенаправленный поиск;
- Улучшенное качество решений по сравнению с классическим GA;
- Снижение вероятности случайных ухудшений хромосом.
  Ограничения:
- Более высокая вычислительная стоимость одного поколения;
- Потенциальное снижение разнообразия при недостаточной адаптивности.
  Вычислительная сложность:
  $O(G \cdot N \cdot (f + h))$,
  где h — затраты на инженерные операции с хромосомами.

4.2 Турнирный отбор

Турнирный отбор — это способ выбора родителей для скрещивания, который усиливает селективное давление и ускоряет сходимость.
Шаги алгоритма:

1. Из текущей популяции случайно выбирается **k особей** (размер турнира).
2. Среди выбранных особей **выбирается лучшая по функции пригодности** для размножения.
3. Процесс повторяется, пока не будет выбран необходимый набор родителей для кроссовера.
   Преимущества:

- Простота реализации;
- Более сильное давление отбора по сравнению с рулеточным методом;
- Устойчивость к шуму в оценке пригодности.
  Ограничения:
- Риск **преждевременной сходимости**;
- Потеря генетического разнообразия при слишком больших турнирах.
  Вычислительная сложность: $O(G \cdot N \cdot f \cdot k)$, где k — размер турнира;

- Miller, B.L., Goldberg, D.E., *Genetic Algorithms, Tournament Selection, and the Effects of Noise*, *Complex Systems*, 1995.

4.3 Адаптивные генетические алгоритмы

Адаптивные GA динамически меняют параметры алгоритма (вероятности кроссовера и мутации) в ходе эволюции для балансировки исследования и эксплуатации.
Шаги алгоритма:

1. Инициализация популяции и оценка пригодности.
2. Сбор статистики о текущем поколении (например, диапазон и среднее значение fitness).
3. **Адаптация параметров**: вероятность мутации и кроссовера корректируется на основе качества и разнообразия популяции.
4. Селекция, скрещивание, мутация с адаптивными параметрами.
5. Замещение и переход к следующему поколению.
   Преимущества:

- Повышенная устойчивость к застреванию в локальных оптимумах;
- Баланс между исследованием новых областей и эксплуатацией найденных решений.
  Ограничения:
- Дополнительные вычислительные затраты на адаптацию параметров;
- Требует корректной настройки функций адаптации.
  Вычислительная сложность: $O(G \cdot N \cdot f + G \cdot C_{adapt})$, где C_{adapt} — вычислительные затраты на адаптацию параметров.

- Srinivas, M., Patnaik, L.M., *Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms*, *IEEE Transactions on Systems, Man, and Cybernetics*, 1994.
- Eiben, A.E., Hinterding, R., Michalewicz, Z., *Parameter Control in Evolutionary Algorithms*, *IEEE Transactions on Evolutionary Computation*, 1999.

4.4 Гибридные генетические алгоритмы (Memetic Algorithms)

Гибридные GA сочетают **глобальный поиск GA** с **локальными методами оптимизации**, что позволяет уточнять решения и повышать их качество.
Шаги алгоритма:

1. Инициализация: Смешанная — 70% случайных решений + 30% жадных.
2. Основной цикл: Оценка → Выбор лучших → Селекция → Кроссовер → Мутация.

3. Ключевая особенность: Случайное применение локального поиска к части потомков для локального улучшения.
4. Локальный поиск: Пробует добавлять или заменять предметы в решении для повышения ценности.
5. Элитизм: Гарантирует сохранение лучших решений в каждом поколении.
6. Остановка: По достижении лимита поколений или при сходимости (отсутствие улучшений).
7. Результат: Возвращает лучшее найденное решение с метриками эффективности.
   Преимущества:

- Высокое качество решений;
- Эффективен для сложных комбинаторных задач;
- Совмещает глобальный и локальный поиск.
  Ограничения:
- Высокая вычислительная стоимость;
- Более сложная реализация.
  Вычислительная сложность: $O(G \cdot N \cdot (f + C_{local}))$, где C_{local} - стоимость локального поиска;

- Moscato, P., *On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms*, 1989.
- Neri, F., Cotta, C., Moscato, P., *Handbook of Memetic Algorithms*, Springer, 2012.

Описание выполненной работы

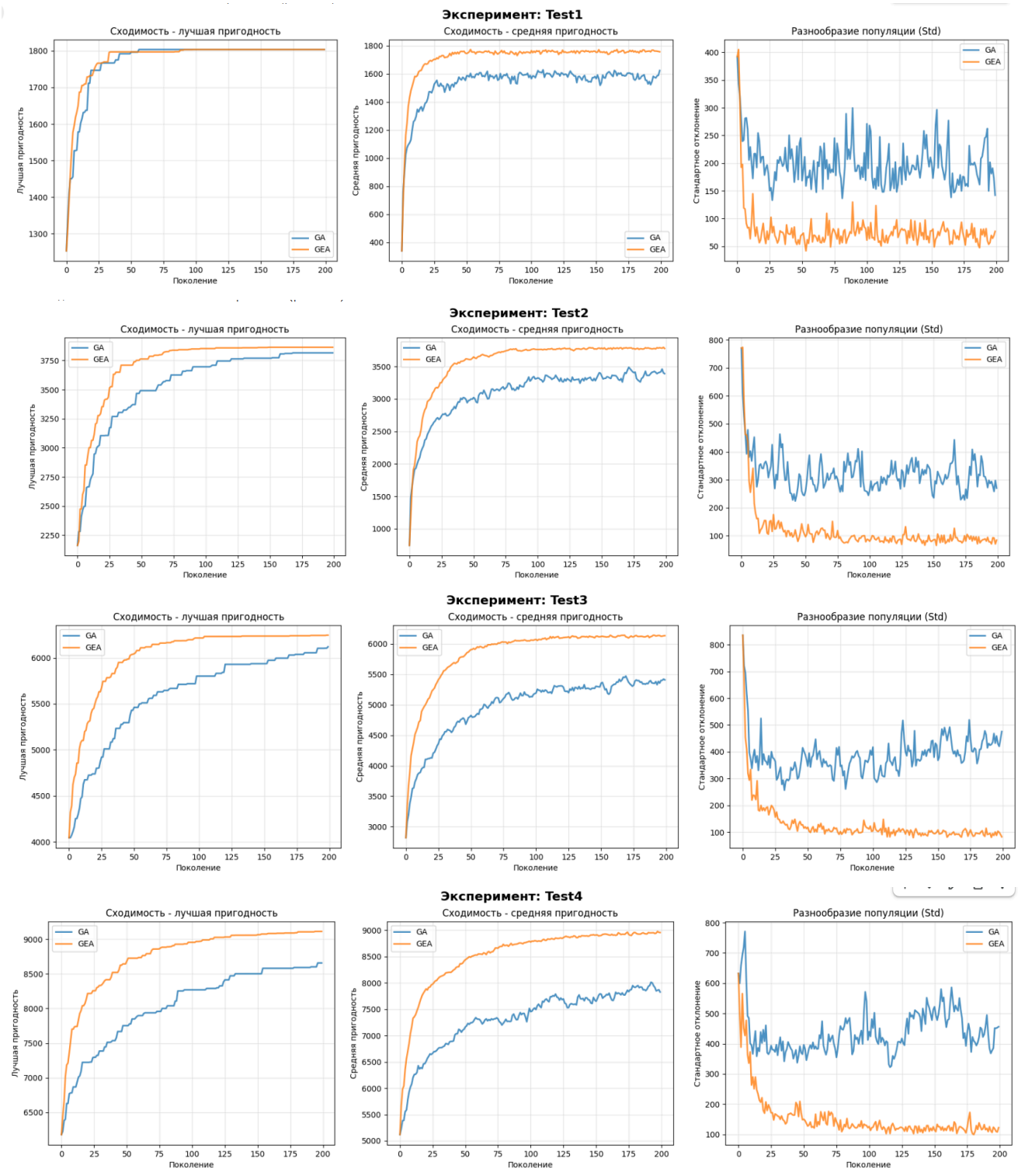
1. Реализация и сравнение GA и GEA

В рамках проекта была выполнена реализация двух основных алгоритмов:

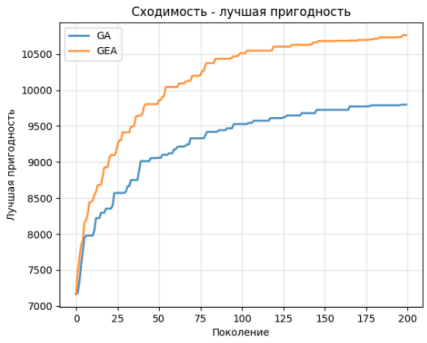
1. **Классического генетического алгоритма (GA)** на основе стандартной схемы Холланда, включающей инициализацию популяции, селекцию, скрещивание, мутацию и замещение.
2. **Genetic Engineering Algorithm (GEA)** согласно описанию в статье Sohrabi et al. (2023), с реализацией направленных операторов изменения хромосом и инженерных эвристик для улучшения поиска.
   После реализации алгоритмы были протестированы на задаче оптимизации рюкзака (Knapsack Problem) с различными размерами предметов и вместимости. Данная задача принадлежит к классу NP-трудных проблем и представляет значительный теоретический и практический интерес. Формальная постановка задачи предполагает наличие множества предметов, каждый из которых характеризуется двумя параметрами: весом и стоимостью. Также задается ограничение по весу — вместимость рюкзака. Цель заключается в выборе подмножества предметов, которое максимизирует суммарную стоимость при условии

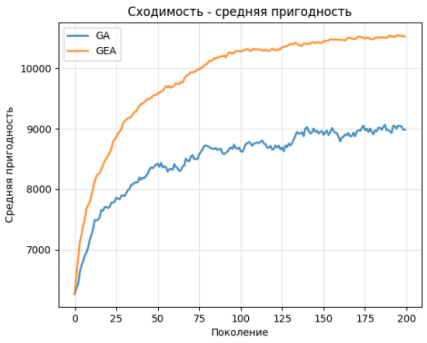
соблюдения ограничения по суммарному весу. Меняя параметры задачи рюкзака (варьируя стоимость предметов, их количество и вместимость) в дальнейшем мы тестируем оба реализованных алгоритма и строим показательные графики.

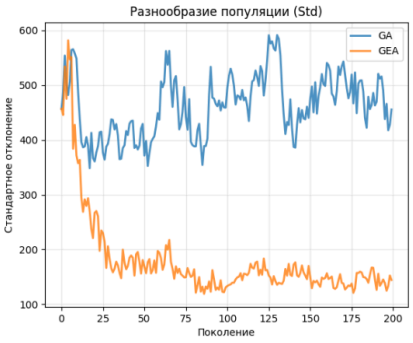
Результаты экспериментов и графики:



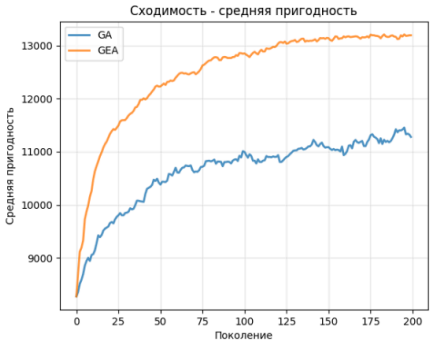
Эксперимент: Test5

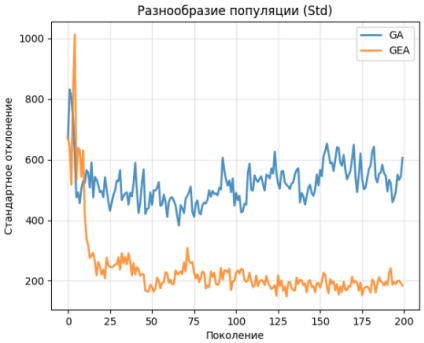


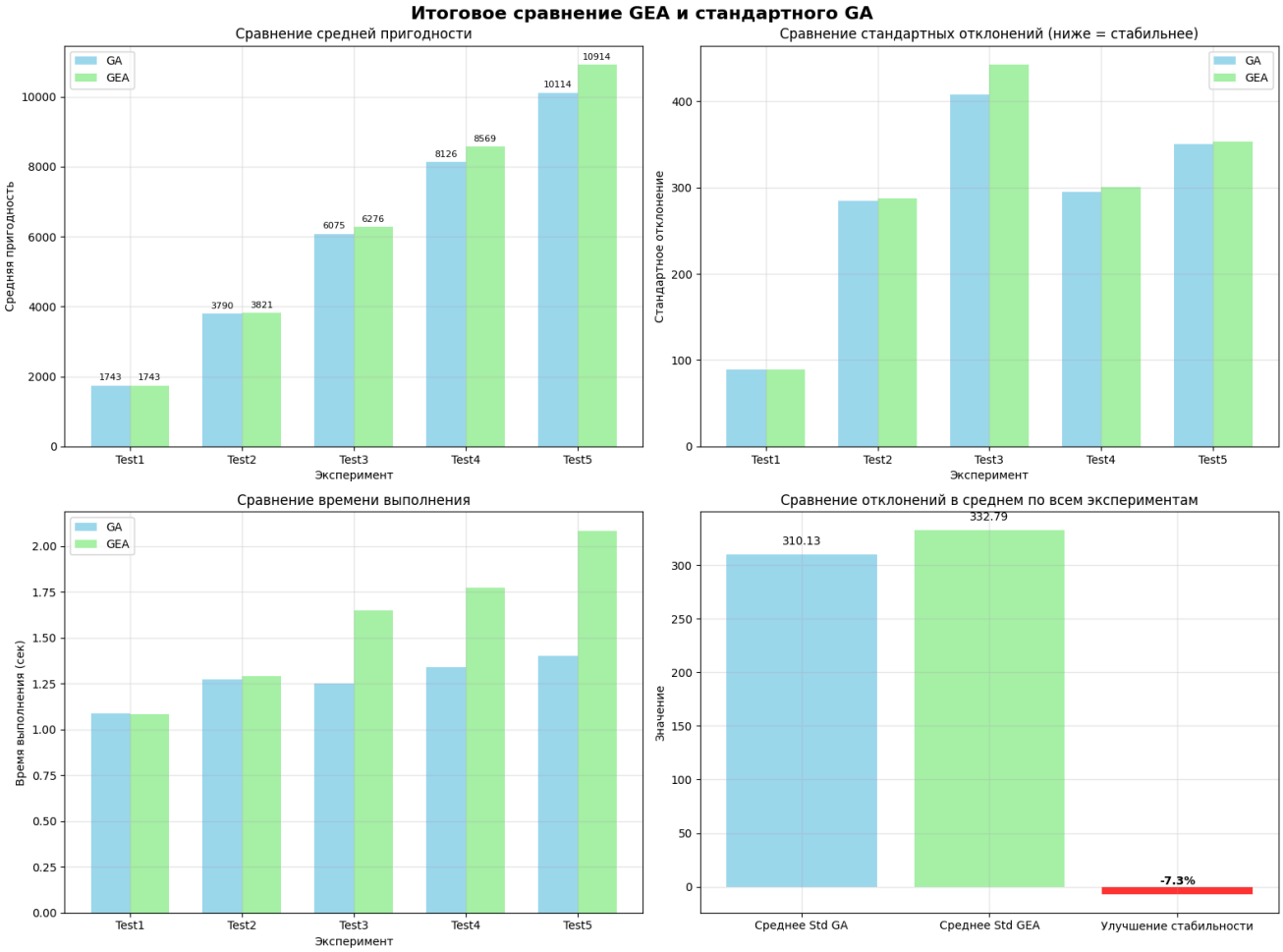


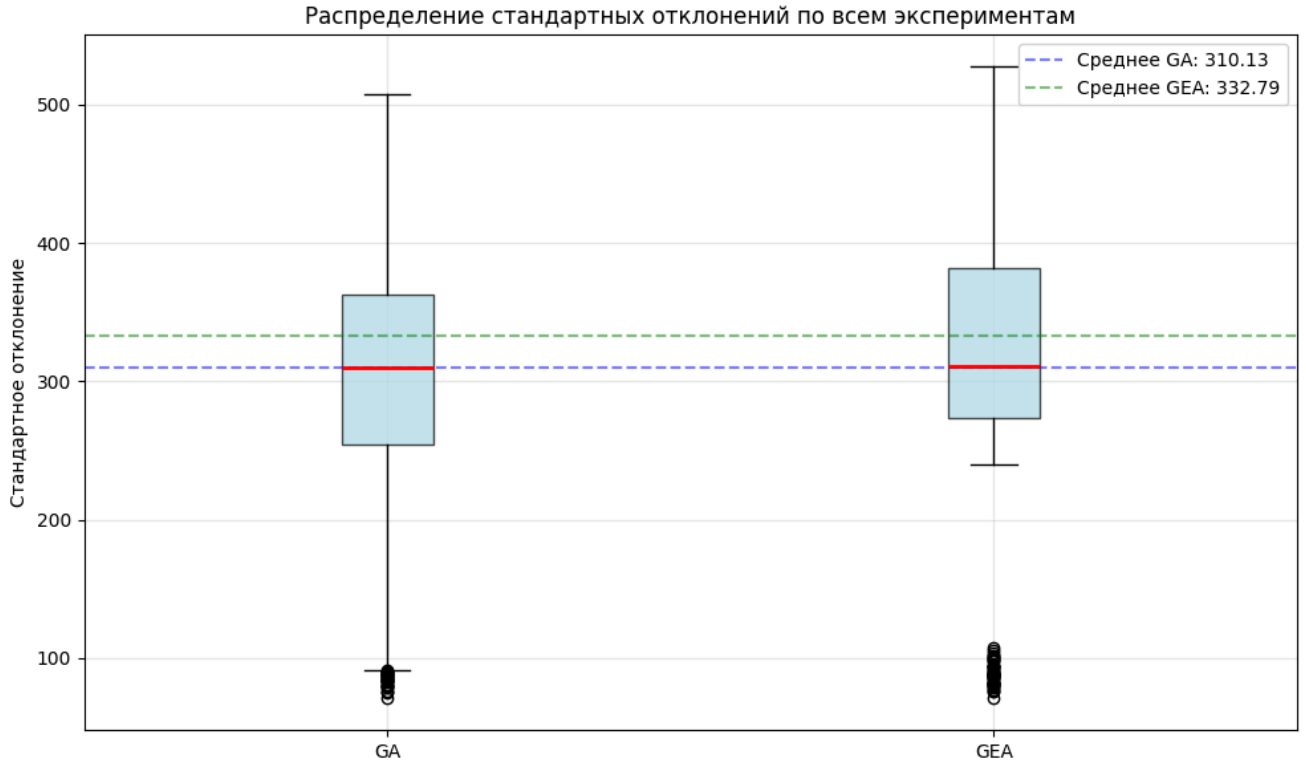


Эксперимент: Test6






Итоговое сравнение GEA и стандартного GA
Сравнение средней пригодности
Средняя пригодность
Эксперимент
GA
GEA
1743
1743
3790
3821
6075
6276
8126
8569
10114
10914
Test1
Test2
Test3
Test4
Test5
Сравнение стандартных отклонений (ниже = стабильнее)
Стандартное отклонение
Сравнение времени выполнения
Время выполнения (сек)
Сравнение отклонений в среднем по всем экспериментам
Значение
310.13
332.79
-7.3%
Среднее Std GA
Среднее Std GEA
Улучшение стабильности


Распределение стандартных отклонений по всем экспериментам
Среднее GA: 310.13
Среднее GEA: 332.79
Стандартное отклонение
GA
GEA

	Кол-во предметов	Вмести-мость	Средний GA	Средний GEA	Std GA	Std GEA	Время GA (с)	Время GEA (с)
Test1	50	100	1743.08	1743.08	88.74	88.74	1.086	1.083
Test2	100	200	3789.77	3821.07	284.18	287.25	1.272	1.292
Test3	150	300	6074.88	6276.33	407.58	442.39	1.253	1.650
Test4	200	400	8125.72	8568.68	295.01	300.56	1.341	1.771
Test5	250	500	10114.35	10913.99	350.38	353.00	1.400	2.083
Test6	300	600	12059.41	13137.81	434.92	524.81	1.481	2.486

Основные наблюдения:

- На простых задачах (Test1 и Test2) **GEA не показывает статистически значимого преимущества** над классическим GA.
- При увеличении сложности задачи (от Test3 до Test6) **GEA демонстрирует постепенное улучшение среднего качества решений**: до 8.9% по сравнению с GA.
- **Время выполнения GEA увеличивается** с ростом сложности задачи, что связано с дополнительными инженерными операциями над хромосомами.
- **Стабильность (стандартное отклонение)** у GEA ниже, чем у GA на более сложных задачах, что говорит о большей вариативности результатов и необходимости аккуратного подбора параметров операторов для каждой решаемой задачи.

2. Реализация модификаций генетического алгоритма и проведение общего анализа

Экспериментальная часть исследования основана на серии сгенерированных тестовых наборов данных, которые моделируют различные сценарии сложности задачи. Были созданы четыре категории тестовых экземпляров, отличающихся количеством предметов: от 50 до 200. Для каждого предмета независимо генерируется вес в интервале от 1 до 20 условных единиц, что соответствует равномерному распределению. Стоимость предмета также определяется случайным образом в диапазоне от 10 до 100 единиц, однако затем корректируется с учетом веса предмета, устанавливая положительную корреляцию между этими параметрами. Такой подход отражает более реалистичные условия, при которых более тяжелые предметы обладают тенденцией к большей стоимости.

Критерий оценки решений, используемый всеми сравниваемыми алгоритмами, представляет собой функцию пригодности, которая вычисляет суммарную стоимость выбранных предметов. В случае соблюдения ограничения по весу значение функции соответствует этой суммарной стоимости. Если же суммарный вес превышает допустимую вместимость, применяется штрафная функция экспоненциального характера. Штраф рассчитывается как произведение константы 100 на экспоненту удвоенного относительного превышения веса. Данная конструкция штрафной функции обеспечивает существенное снижение оценки для

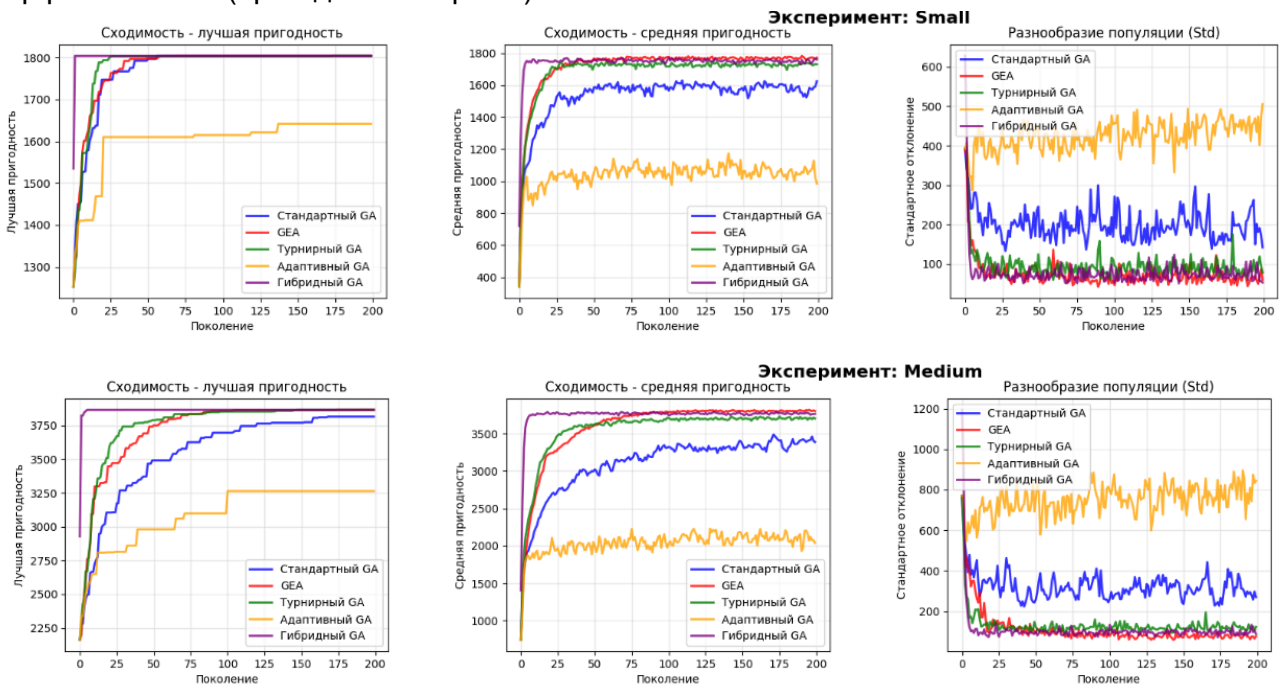
недопустимых решений, особенно при значительных превышениях весового ограничения, что направляет поиск алгоритмов в область допустимых решений.
Данные для тестов:

Тест	Предметы (n_items)	Вместимость (capacity)
Small	50	100
Medium	9.4	200
Large	150	300
X-Large	**200**	**400**

В исследовании были протестированы следующие методы:

1. **Стандартный генетический алгоритм (GA)** с рулеточной селекцией.
2. **Genetic Engineering Algorithm (GEA)** с направленными эвристиками.
3. **GA с турнирной селекцией** — выбор лучшей особи из случайной подгруппы.
4. **Адаптивный GA** — динамическая настройка вероятностей кроссовера и мутации.
5. **Гибридный GA с локальным поиском (Memetic Algorithm)** — комбинирование глобального GA с локальной оптимизацией.
   Метрики сравнения:

- Качество решения (средняя пригодность, fitness)
- Время выполнения (секунды)
- Поколения до сходимости
- Эффективность (пригодность / время)



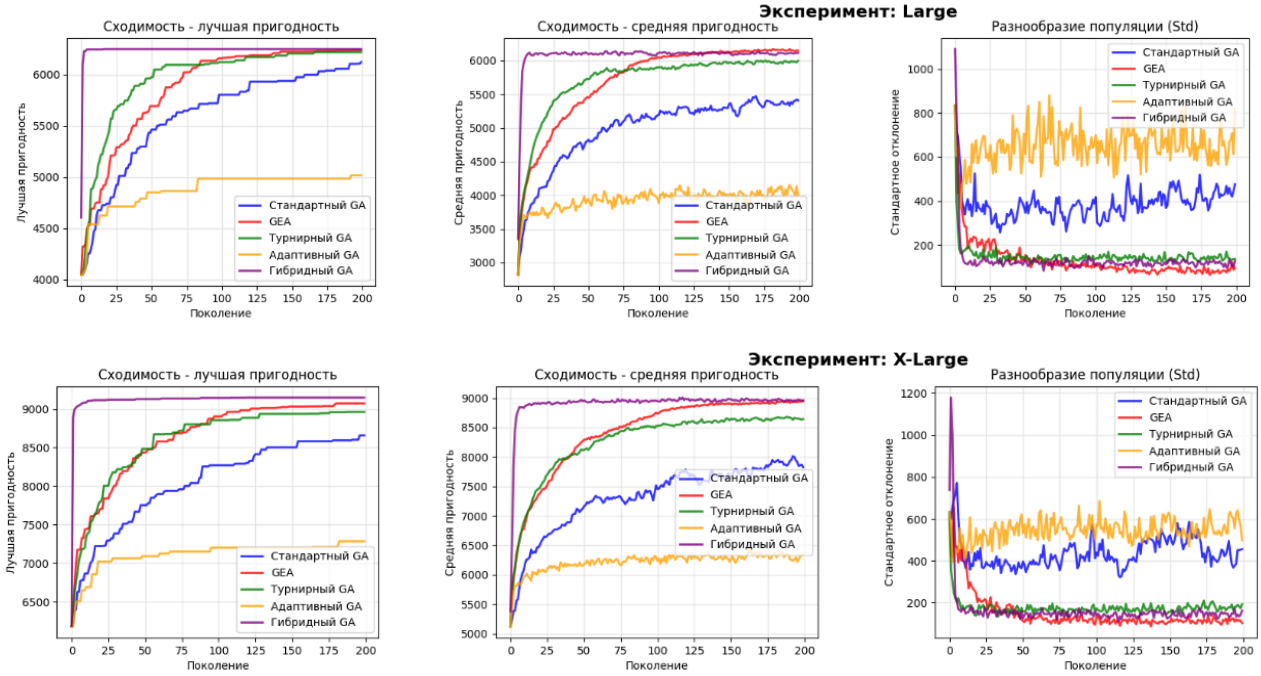


Таблица 1. Качество решения (средняя пригодность)

Эксп.	Размер	Вместимость	Стандартный GA	GEA	Турнирный GA	Адаптивный GA	Гибридный GA
Small	50	100	1754.9	1754.9	1754.9	1599.6	1754.9
Medium	100	200	3794.4	3827.7	3827.9	3233.6	3828.2
Large	150	300	6275.8	6497.8	6468.1	5327.0	6508.3
X-Large	200	400	8154.3	8535.7	8484.4	6864.7	8644.0
Average	125	250	4994.9	5154.0	5133.8	4256.2	5183.9

Таблица 2. Время выполнения (секунды)

Эксп.	Стандартный GA	GEA	Турнирный GA	Адаптивный GA	Гибридный GA
Small	1.149	1.249	0.988	7.016	26.048
Medium	1.145	1.517	1.157	9.002	98.328
Large	1.163	1.777	1.107	10.954	222.726
X-Large	1.487	2.211	1.100	12.640	396.802
Average	1.236	1.688	1.088	9.903	185.976

Таблица 3. Поколения до сходимости

Эксп.	Стандартный GA	GEA	Турнирный GA	Адаптивный GA	Гибридный GA
Small	47	39	36	24	12
Medium	73	86	73	30	18
Large	82	99	80	27	19
X-Large	85	111	86	25	20
Average	72	83	68	27	17

Таблица 4. Процентное улучшение GEA относительно других алгоритмов (%)

Сравнение	Пригодность	Время	Сходимость
GEA vs Стандартный GA	3.2	-36.6	-16.3
GEA vs Турнирный GA	0.4	-55.2	-21.9
GEA vs Адаптивный GA	21.1	82.9	-214.3
GEA vs Гибридный GA	-0.6	99.1	-382.4

Анализ результатов

1. **Качество решений:**
   - Гибридный GA показывает наивысшее качество решений благодаря локальному поиску.

- GEA и Турнирный GA демонстрируют средние значения, улучшая стандартный GA, особенно на сложных задачах.
- Адаптивный GA отстаёт по качеству, так как фокусируется на поддержании разнообразия, а не на агрессивной оптимизации.

2. **Время выполнения:**
   - Наиболее быстрыми оказались Турнирный GA и стандартный GA.
   - GEA требует больше времени, но остаётся приемлемым для средних задач.
   - Адаптивный и особенно гибридный GA значительно медленнее, что связано с дополнительными вычислительными операциями и локальным поиском.
3. **Сходимость:**
   - Гибридный GA и Адаптивный GA сходятся быстрее, но за счёт времени.
   - Стандартный GA и Турнирный GA показывают среднюю скорость сходимости.
   - GEA имеет умеренную сходимость, балансируя качество и время.

В ходе экспериментов GEA показал себя как алгоритм, который эффективно балансирует между качеством решений и временем выполнения. В среднем, GEA улучшает пригодность решений по сравнению со стандартным GA, особенно на задачах средней и высокой сложности. При этом он демонстрирует умеренную скорость сходимости — быстрее стандартного GA и Турнирного GA, но уступает Адаптивному GA и Гибридному GA в аспектах скорости или максимального качества.

Основные наблюдения по другим алгоритмам:

- **Турнирный GA** обеспечивает быструю сходимость и стабильное качество решений. Он показывает хорошее соотношение скорость/качество и подходит для задач, где важна оперативность и простота реализации.
- **Адаптивный GA** фокусируется на сохранении генетического разнообразия, предотвращая преждевременную сходимость. Он особенно полезен для сложных задач с большим количеством локальных оптимумов, где стандартные методы могут застрять.
- **Гибридный GA** достигает наивысшего качества решений за счёт локального поиска, но требует значительно больше времени. Подходит для случаев, когда критично качество результата, а время не является ограничением.
- **Стандартный GA** остаётся простым и стабильным алгоритмом, эффективным для базовых задач, где не требуется сложная адаптация или инъекция новых генов.

Итоговая оценка GEA:
GEA занимает промежуточное место между стандартными и гибридными алгоритмами. Он обеспечивает улучшенное качество по сравнению с базовым GA, остаётся относительно быстрым и демонстрирует хорошую стабильность сходимости. Таким образом, GEA является оптимальным выбором, если требуется баланс между качеством решения, временем выполнения и стабильностью алгоритма.

Приложения

Кодовая реализация тестируемой задачи и алгоритмов:

1. Задача о рюкзаке

```
class KnapsackProblem:

    """Задача о рюкзаке - классическая комбинаторная задача"""

    def __init__(self, n_items: int, capacity: float, seed: int = None):

        self.n_items = n_items

        self.capacity = capacity

        self.seed = seed if seed is not None else np.random.randint(1000)

        np.random.seed(self.seed)

        # Генерируем случайные веса и стоимости

        self.weights = np.random.uniform(1, 20, n_items)

        self.values = np.random.uniform(10, 100, n_items)

        # Делаем стоимость коррелированной с весом

        self.values = self.values * (1 + 0.3 * self.weights /
np.mean(self.weights))

    def evaluate(self, chromosome: np.ndarray) -> float:

        """Оценка решения для задачи о рюкзаке"""

        total_weight = np.dot(chromosome, self.weights)

        total_value = np.dot(chromosome, self.values)

        # Штраф за превышение вместимости

        if total_weight > self.capacity:

            penalty = np.exp(2 * (total_weight - self.capacity) / self.capacity) *
100

            return max(0, total_value - penalty)

        return total_value
```

2. Классический генетический алгоритм

```
class StandardGeneticAlgorithm:
 """Стандартный генетический алгоритм для сравнения с GEA"""
 def __init__(self,
              problem: KnapsackProblem,
              population_size: int = 100,
              max_generations: int = 200,
              crossover_rate: float = 0.8,
              mutation_rate: float = 0.02,
              elite_count: int = 10):
     self.problem = problem
     self.population_size = population_size
     self.max_generations = max_generations
     self.crossover_rate = crossover_rate
     self.mutation_rate = mutation_rate
     self.elite_count = elite_count
     self.chromosome_length = problem.n_items
     self.population = self._initialize_population()
     self.best_solution = None
     self.best_fitness = 0
     self.fitness_history = []
     self.convergence_generation = 0
 def _initialize_population(self) -> List[np.ndarray]:
     """Инициализация случайной популяции"""
```

```
        return [np.random.randint(0, 2, self.chromosome_length)
                for _ in range(self.population_size)]

    def evaluate_population(self) -> List[float]:
        """Оценка пригодности всей популяции"""
        return [self.problem.evaluate(ind) for ind in self.population]

    def select_parents(self, fitness_values: List[float]) -> Tuple[np.ndarray,
np.ndarray]:
        """Рулеточная селекция"""
        total_fitness = sum(fitness_values)
        if total_fitness == 0:
            probabilities = [1/len(fitness_values)] * len(fitness_values)
        else:
            probabilities = [f/total_fitness for f in fitness_values]
        idx1 = np.random.choice(len(self.population), p=probabilities)
        idx2 = np.random.choice(len(self.population), p=probabilities)
        return self.population[idx1].copy(), self.population[idx2].copy()

    def crossover(self, parent1: np.ndarray, parent2: np.ndarray) ->
Tuple[np.ndarray, np.ndarray]:
        """Одноточечный кроссовер"""
        if random.random() > self.crossover_rate:
            return parent1.copy(), parent2.copy()
        point = random.randint(1, self.chromosome_length - 1)
        child1 = np.concatenate([parent1[:point], parent2[point:]])
        child2 = np.concatenate([parent2[:point], parent1[point:]])
        return child1, child2

    def mutate(self, chromosome: np.ndarray) -> np.ndarray:
```

```
        """Мутация особи"""
        mutated = chromosome.copy()
        for i in range(self.chromosome_length):
            if random.random() < self.mutation_rate:
                mutated[i] = 1 - mutated[i]
        return mutated
    def run(self) -> Dict:
        """Запуск стандартного GA"""
        start_time = time.time()
        convergence_threshold = 0.001  # Порог сходимости (0.1%)
        no_improvement_count = 0
        last_best = -np.inf
        for generation in range(self.max_generations):
            # Оценка пригодности
            fitness_values = self.evaluate_population()
            current_best = max(fitness_values)
            # Проверка сходимости
            if generation > 0:
                improvement = (current_best - last_best) / abs(last_best) if
last_best != 0 else 1
                if improvement < convergence_threshold:
                    no_improvement_count += 1
                else:
                    no_improvement_count = 0
                if no_improvement_count >= 10:  # Если 10 поколений без улучшений
```

```
            if self.convergence_generation == 0:
                self.convergence_generation = generation
        last_best = current_best
        # Обновление лучшего решения
        best_idx = np.argmax(fitness_values)
        if fitness_values[best_idx] > self.best_fitness:
            self.best_fitness = fitness_values[best_idx]
            self.best_solution = self.population[best_idx].copy()
        # Сохранение истории
        self.fitness_history.append({
            'best': current_best,
            'average': np.mean(fitness_values),
            'worst': min(fitness_values),
            'std': np.std(fitness_values)
        })
        # Создание нового поколения
        new_population = []
        # Элитизм
        sorted_indices = np.argsort(fitness_values)[::-1]
        for i in range(min(self.elite_count, self.population_size)):
            new_population.append(self.population[sorted_indices[i]].copy())
        # Генерация остальной популяции
        while len(new_population) < self.population_size:
            parent1, parent2 = self.select_parents(fitness_values)
```

```
                child1, child2 = self.crossover(parent1, parent2)
                child1 = self.mutate(child1)
                child2 = self.mutate(child2)
                new_population.append(child1)
                if len(new_population) < self.population_size:
                    new_population.append(child2)
            self.population = new_population[:self.population_size]
        execution_time = time.time() - start_time
        # Если сходимость не была достигнута, считаем что сошлись на последнем поколении
        if self.convergence_generation == 0:
            self.convergence_generation = self.max_generations
        return {
            'best_solution': self.best_solution,
            'best_fitness': self.best_fitness,
            'fitness_history': self.fitness_history,
            'execution_time': execution_time,
            'convergence_generation': self.convergence_generation,
            'total_generations': self.max_generations
        }
```

3. GEA

```
class GeneticEngineeringAlgorithm:
 """Улучшенная реализация GEA по оригинальной статье"""
 def __init__(self,
              problem: KnapsackProblem,
```

```
                 population_size: int = 100,
                 max_generations: int = 200,
                 crossover_rate: float = 0.8,
                 mutation_rate: float = 0.02,
                 elite_ratio: float = 0.3,
                 injection_rate: float = 0.4,
                 threshold: float = 0.6):
        self.problem = problem
        self.population_size = population_size
        self.max_generations = max_generations
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.elite_ratio = elite_ratio
        self.injection_rate = injection_rate
        self.threshold = threshold
        self.chromosome_length = problem.n_items
        self.population = self._initialize_population()
        self.best_solution = None
        self.best_fitness = -np.inf
        self.fitness_history = []
        self.convergence_generation = 0

    def _initialize_population(self) -> List[np.ndarray]:
        return [np.random.randint(0, 2, self.chromosome_length) for _ in
range(self.population_size)]

    def evaluate_population(self) -> List[float]:
```

```
        return [self.problem.evaluate(ind) for ind in self.population]

    def select_parents(self, fitness_values: List[float]) -> Tuple[np.ndarray,
np.ndarray]:

        tournament_size = 3

        parents = []

        for _ in range(2):

            idxs = np.random.choice(self.population_size, tournament_size,
replace=False)

            best_idx = idxs[np.argmax([fitness_values[i] for i in idxs])]

            parents.append(self.population[best_idx].copy())

        return parents[0], parents[1]

    def crossover(self, p1: np.ndarray, p2: np.ndarray) -> Tuple[np.ndarray,
np.ndarray]:

        if random.random() > self.crossover_rate:

            return p1.copy(), p2.copy()

        point = random.randint(1, self.chromosome_length - 1)

        return np.concatenate([p1[:point], p2[point:]]),
np.concatenate([p2[:point], p1[point:]])

    def mutate(self, chromosome: np.ndarray) -> np.ndarray:

        return np.array([gene if random.random() > self.mutation_rate else 1-gene
for gene in chromosome])

    def identify_elite(self, fitness_values: List[float]) -> List[np.ndarray]:

        elite_count = max(1, int(self.elite_ratio * self.population_size))

        elite_idx = np.argsort(fitness_values)[::-1][:elite_count]

        return [self.population[i].copy() for i in elite_idx]

    def analyze_elite(self, elite: List[np.ndarray]) -> Tuple[np.ndarray,
np.ndarray]:
```

```
        elite_array = np.array(elite)

        gene_freq = elite_array.mean(axis=0)

        dominant_genes = (gene_freq >= 0.5).astype(int)

        return dominant_genes, gene_freq

    def inject_genes(self, chromosome: np.ndarray, dominant_genes: np.ndarray,
gene_freq: np.ndarray) -> np.ndarray:

        injected = chromosome.copy()

        for i in range(self.chromosome_length):

            if random.random() < self.injection_rate and gene_freq[i] >=
self.threshold:

                injected[i] = dominant_genes[i]

        return injected

    def run(self) -> Dict:

        start_time = time.time()

        convergence_threshold = 0.001  # Порог сходимости (0.1%)

        no_improvement_count = 0

        last_best = -np.inf

        for gen in range(self.max_generations):

            fitness_values = self.evaluate_population()

            current_best = np.max(fitness_values)

            # Проверка сходимости

            if gen > 0:

                improvement = (current_best - last_best) / abs(last_best) if
last_best != 0 else 1

                if improvement < convergence_threshold:

                    no_improvement_count += 1
```

```
            else:
                no_improvement_count = 0
            if no_improvement_count >= 10:  # Если 10 поколений без улучшений
                if self.convergence_generation == 0:
                    self.convergence_generation = gen
        last_best = current_best
        # Обновление лучшего решения
        best_idx = np.argmax(fitness_values)
        if fitness_values[best_idx] > self.best_fitness:
            self.best_fitness = fitness_values[best_idx]
            self.best_solution = self.population[best_idx].copy()
        self.fitness_history.append({
            'best': current_best,
            'average': np.mean(fitness_values),
            'worst': np.min(fitness_values),
            'std': np.std(fitness_values)
        })
        elite = self.identify_elite(fitness_values)
        dominant_genes, gene_freq = self.analyze_elite(elite)
        new_population = elite.copy()
        while len(new_population) < self.population_size:
            p1, p2 = self.select_parents(fitness_values)
            c1, c2 = self.crossover(p1, p2)
            c1, c2 = self.mutate(c1), self.mutate(c2)
            c1, c2 = self.inject_genes(c1, dominant_genes, gene_freq),
```

```
self.inject_genes(c2, dominant_genes, gene_freq)

                new_population.extend([c1, c2])

            self.population = new_population[:self.population_size]

        exec_time = time.time() - start_time

        # Если сходимость не была достигнута, считаем что сошлись на последнем
поколении

        if self.convergence_generation == 0:

            self.convergence_generation = self.max_generations

        return {

            'best_solution': self.best_solution,

            'best_fitness': self.best_fitness,

            'fitness_history': self.fitness_history,

            'execution_time': exec_time,

            'convergence_generation': self.convergence_generation,

            'total_generations': self.max_generations

        }
```

4. Алгоритм с турнирной секцией

```
class TournamentSelectionGA:

    """Генетический алгоритм с турнирной селекцией - популярная модификация"""

    def __init__(self,

                 problem: KnapsackProblem,

                 population_size: int = 100,

                 max_generations: int = 200,

                 crossover_rate: float = 0.8,

                 mutation_rate: float = 0.02,
```

```
                 tournament_size: int = 3,
                 elite_count: int = 10):
        self.problem = problem
        self.population_size = population_size
        self.max_generations = max_generations
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.tournament_size = tournament_size
        self.elite_count = elite_count
        self.chromosome_length = problem.n_items
        self.population = self._initialize_population()
        self.best_solution = None
        self.best_fitness = 0
        self.fitness_history = []
        self.convergence_generation = 0
    def _initialize_population(self) -> List[np.ndarray]:
        """Инициализация случайной популяции"""
        return [np.random.randint(0, 2, self.chromosome_length)
                for _ in range(self.population_size)]
    def evaluate_population(self) -> List[float]:
        """Оценка пригодности всей популяции"""
        return [self.problem.evaluate(ind) for ind in self.population]
    def tournament_selection(self, fitness_values: List[float]) -> np.ndarray:
        """Турнирная селекция"""
```

```
        participants = np.random.choice(self.population_size, self.tournament_size,
replace=False)

        best_idx = participants[np.argmax([fitness_values[i] for i in
participants])]

        return self.population[best_idx].copy()

    def select_parents(self, fitness_values: List[float]) -> Tuple[np.ndarray,
np.ndarray]:

        """Выбор родителей турнирной селекцией"""

        parent1 = self.tournament_selection(fitness_values)

        parent2 = self.tournament_selection(fitness_values)

        return parent1, parent2

    def crossover(self, parent1: np.ndarray, parent2: np.ndarray) ->
Tuple[np.ndarray, np.ndarray]:

        """Двухточечный кроссовер"""

        if random.random() > self.crossover_rate:

            return parent1.copy(), parent2.copy()

        point1 = random.randint(1, self.chromosome_length - 2)

        point2 = random.randint(point1 + 1, self.chromosome_length - 1)

        child1 = np.concatenate([parent1[:point1], parent2[point1:point2],
parent1[point2:]])

        child2 = np.concatenate([parent2[:point1], parent1[point1:point2],
parent2[point2:]])

        return child1, child2

    def mutate(self, chromosome: np.ndarray) -> np.ndarray:

        """Мутация с переменной вероятностью"""

        mutated = chromosome.copy()

        mutation_prob = self.mutation_rate * (1 + np.random.random() * 0.5)  #
Случайное изменение вероятности
```

```
        for i in range(self.chromosome_length):
            if random.random() < mutation_prob:
                mutated[i] = 1 - mutated[i]
        return mutated
    def run(self) -> Dict:
        """Запуск GA с турнирной селекцией"""
        start_time = time.time()
        convergence_threshold = 0.001
        no_improvement_count = 0
        last_best = -np.inf
        for generation in range(self.max_generations):
            fitness_values = self.evaluate_population()
            current_best = max(fitness_values)
            # Проверка сходимости
            if generation > 0:
                improvement = (current_best - last_best) / abs(last_best) if
last_best != 0 else 1
                if improvement < convergence_threshold:
                    no_improvement_count += 1
                else:
                    no_improvement_count = 0
                if no_improvement_count >= 10:
                    if self.convergence_generation == 0:
                        self.convergence_generation = generation
            last_best = current_best
```

```
            # Обновление лучшего решения
            best_idx = np.argmax(fitness_values)
            if fitness_values[best_idx] > self.best_fitness:
                self.best_fitness = fitness_values[best_idx]
                self.best_solution = self.population[best_idx].copy()
            # Сохранение истории
            self.fitness_history.append({
                'best': current_best,
                'average': np.mean(fitness_values),
                'worst': min(fitness_values),
                'std': np.std(fitness_values)
            })
            # Создание нового поколения
            new_population = []
            # Элитизм
            sorted_indices = np.argsort(fitness_values)[::-1]
            for i in range(min(self.elite_count, self.population_size)):
                new_population.append(self.population[sorted_indices[i]].copy())
            # Генерация остальной популяции
            while len(new_population) < self.population_size:
                parent1, parent2 = self.select_parents(fitness_values)
                child1, child2 = self.crossover(parent1, parent2)
                child1 = self.mutate(child1)
                child2 = self.mutate(child2)
                new_population.append(child1)
```

```
                if len(new_population) < self.population_size:
                    new_population.append(child2)
            self.population = new_population[:self.population_size]
        execution_time = time.time() - start_time
        if self.convergence_generation == 0:
            self.convergence_generation = self.max_generations
        return {
            'best_solution': self.best_solution,
            'best_fitness': self.best_fitness,
            'fitness_history': self.fitness_history,
            'execution_time': execution_time,
            'convergence_generation': self.convergence_generation,
            'total_generations': self.max_generations
        }
```

5. Алгоритм с адаптивными параметрами

```
class AdaptiveGA:
    """Генетический алгоритм с адаптивными параметрами"""
    def __init__(self,
                 problem: KnapsackProblem,
                 population_size: int = 100,
                 max_generations: int = 200,
                 initial_crossover_rate: float = 0.9,
                 initial_mutation_rate: float = 0.05,
                 elite_ratio: float = 0.2):
```

```
        self.problem = problem
        self.population_size = population_size
        self.max_generations = max_generations
        self.crossover_rate = initial_crossover_rate
        self.mutation_rate = initial_mutation_rate
        self.elite_ratio = elite_ratio
        self.chromosome_length = problem.n_items
        self.population = self._initialize_population()
        self.best_solution = None
        self.best_fitness = 0
        self.fitness_history = []
        self.convergence_generation = 0
        self.diversity_history = []
    def _initialize_population(self) -> List[np.ndarray]:
        """Инициализация случайной популяции"""
        return [np.random.randint(0, 2, self.chromosome_length)
                for _ in range(self.population_size)]
    def evaluate_population(self) -> List[float]:
        """Оценка пригодности всей популяции"""
        return [self.problem.evaluate(ind) for ind in self.population]
    def calculate_diversity(self, population: List[np.ndarray]) -> float:
        """Вычисление разнообразия популяции"""
        if len(population) <= 1:
            return 1.0
```

```
        diversity = 0.0

        for i in range(len(population)):

            for j in range(i+1, len(population)):

                diversity += np.sum(population[i] != population[j])

        max_diversity = len(population[0]) * len(population) * (len(population) -
1) / 2

        return diversity / max_diversity if max_diversity > 0 else 0

    def adaptive_parameters(self, diversity: float, generation: int) ->
Tuple[float, float]:

        """Адаптация параметров на основе разнообразия"""

        # Уменьшаем кроссовер и увеличиваем мутацию при низком разнообразии

        target_crossover = self.crossover_rate * (0.5 + 0.5 * diversity)

        target_mutation = self.mutation_rate * (1.5 - 0.5 * diversity)

        # Дополнительная адаптация по поколениям

        progress = generation / self.max_generations

        target_crossover *= (1.0 - 0.5 * progress)  # Уменьшаем кроссовер со
временем

        target_mutation *= (1.0 + 0.5 * progress)   # Увеличиваем мутацию со
временем

        return max(0.3, min(0.95, target_crossover)), max(0.01, min(0.2,
target_mutation))

    def select_parents(self, fitness_values: List[float]) -> Tuple[np.ndarray,
np.ndarray]:

        """Ранжированная селекция"""

        # Ранжируем особи по пригодности

        sorted_indices = np.argsort(fitness_values)[::-1]

        ranks = np.arange(len(fitness_values), 0, -1)

        probabilities = ranks / np.sum(ranks)
```

```
        idx1 = sorted_indices[np.random.choice(len(sorted_indices),
p=probabilities)]

        idx2 = sorted_indices[np.random.choice(len(sorted_indices),
p=probabilities)]

        return self.population[idx1].copy(), self.population[idx2].copy()

    def uniform_crossover(self, parent1: np.ndarray, parent2: np.ndarray,
crossover_rate: float) -> Tuple[np.ndarray, np.ndarray]:

        """Равномерный кроссовер"""

        child1 = parent1.copy()

        child2 = parent2.copy()

        for i in range(self.chromosome_length):

            if random.random() < crossover_rate:

                child1[i], child2[i] = child2[i], child1[i]

        return child1, child2

    def adaptive_mutate(self, chromosome: np.ndarray, mutation_rate: float) ->
np.ndarray:

        """Адаптивная мутация"""

        mutated = chromosome.copy()

        # Вероятность мутации зависит от позиции гена

        for i in range(self.chromosome_length):

            # Гены в середине хромосомы мутируют с большей вероятностью

            position_factor = 1.0 + 0.5 * np.sin(2 * np.pi * i /
self.chromosome_length)

            adjusted_rate = mutation_rate * position_factor

            if random.random() < adjusted_rate:

                mutated[i] = 1 - mutated[i]

        return mutated
```

```
    def run(self) -> Dict:
        """Запуск адаптивного GA"""
        start_time = time.time()
        convergence_threshold = 0.001
        no_improvement_count = 0
        last_best = -np.inf
        for generation in range(self.max_generations):
            fitness_values = self.evaluate_population()
            current_best = max(fitness_values)
            # Проверка сходимости
            if generation > 0:
                improvement = (current_best - last_best) / abs(last_best) if
last_best != 0 else 1
                if improvement < convergence_threshold:
                    no_improvement_count += 1
                else:
                    no_improvement_count = 0
                if no_improvement_count >= 10:
                    if self.convergence_generation == 0:
                        self.convergence_generation = generation
            last_best = current_best
            # Обновление лучшего решения
            best_idx = np.argmax(fitness_values)
            if fitness_values[best_idx] > self.best_fitness:
                self.best_fitness = fitness_values[best_idx]
```

```
                self.best_solution = self.population[best_idx].copy()

            # Вычисление разнообразия и адаптация параметров

            diversity = self.calculate_diversity(self.population)

            self.diversity_history.append(diversity)

            self.crossover_rate, self.mutation_rate =
self.adaptive_parameters(diversity, generation)

            # Сохранение истории

            self.fitness_history.append({

                'best': current_best,

                'average': np.mean(fitness_values),

                'worst': min(fitness_values),

                'std': np.std(fitness_values),

                'diversity': diversity,

                'crossover_rate': self.crossover_rate,

                'mutation_rate': self.mutation_rate

            })

            # Создание нового поколения

            new_population = []

            # Элитизм

            elite_count = max(1, int(self.elite_ratio * self.population_size))

            sorted_indices = np.argsort(fitness_values)[::-1]

            for i in range(min(elite_count, self.population_size)):

                new_population.append(self.population[sorted_indices[i]].copy())

            # Генерация остальной популяции

            while len(new_population) < self.population_size:
```

```
                parent1, parent2 = self.select_parents(fitness_values)
                child1, child2 = self.uniform_crossover(parent1, parent2,
self.crossover_rate)
                child1 = self.adaptive_mutate(child1, self.mutation_rate)
                child2 = self.adaptive_mutate(child2, self.mutation_rate)
                new_population.append(child1)
                if len(new_population) < self.population_size:
                    new_population.append(child2)
            self.population = new_population[:self.population_size]
        execution_time = time.time() - start_time
        if self.convergence_generation == 0:
            self.convergence_generation = self.max_generations
        return {
            'best_solution': self.best_solution,
            'best_fitness': self.best_fitness,
            'fitness_history': self.fitness_history,
            'execution_time': execution_time,
            'convergence_generation': self.convergence_generation,
            'total_generations': self.max_generations,
            'diversity_history': self.diversity_history
        }
```

6. Гибридный генетический алгоритм

```
    class HybridGA:
    """Гибридный генетический алгоритм с локальным поиском"""
```

```python
    def __init__(self,
                 problem: KnapsackProblem,
                 population_size: int = 100,
                 max_generations: int = 200,
                 crossover_rate: float = 0.8,
                 mutation_rate: float = 0.02,
                 local_search_rate: float = 0.3,
                 elite_count: int = 10):
        self.problem = problem
        self.population_size = population_size
        self.max_generations = max_generations
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.local_search_rate = local_search_rate
        self.elite_count = elite_count
        self.chromosome_length = problem.n_items
        self.population = self._initialize_population()
        self.best_solution = None
        self.best_fitness = 0
        self.fitness_history = []
        self.convergence_generation = 0
        self.local_search_applications = 0
    def _initialize_population(self) -> List[np.ndarray]:
        """Инициализация популяции с жадным алгоритмом"""
        population = []
```

```
        # Часть популяции инициализируем случайно
        random_count = int(self.population_size * 0.7)
        for _ in range(random_count):
            population.append(np.random.randint(0, 2, self.chromosome_length))
        # Часть популяции инициализируем жадным алгоритмом
        greedy_count = self.population_size - random_count
        for _ in range(greedy_count):
            solution = self.greedy_initialization()
            population.append(solution)
        return population
    def greedy_initialization(self) -> np.ndarray:
        """Жадная инициализация решения"""
        solution = np.zeros(self.chromosome_length, dtype=int)
        remaining_capacity = self.problem.capacity
        # Сортируем предметы по убыванию отношения стоимость/вес
        value_ratio = self.problem.values / self.problem.weights
        sorted_indices = np.argsort(value_ratio)[::-1]
        for idx in sorted_indices:
            if self.problem.weights[idx] <= remaining_capacity:
                solution[idx] = 1
                remaining_capacity -= self.problem.weights[idx]
        return solution
    def evaluate_population(self) -> List[float]:
        """Оценка пригодности всей популяции"""
```

```
        return [self.problem.evaluate(ind) for ind in self.population]

    def select_parents(self, fitness_values: List[float]) -> Tuple[np.ndarray,
np.ndarray]:

        """Бинарная турнирная селекция"""

        def binary_tournament():

            idx1, idx2 = np.random.choice(self.population_size, 2, replace=False)

            return idx1 if fitness_values[idx1] > fitness_values[idx2] else idx2

        parent1_idx = binary_tournament()

        parent2_idx = binary_tournament()

        return self.population[parent1_idx].copy(),
self.population[parent2_idx].copy()

    def crossover(self, parent1: np.ndarray, parent2: np.ndarray) ->
Tuple[np.ndarray, np.ndarray]:

        """Арифметический кроссовер для бинарных строк"""

        if random.random() > self.crossover_rate:

            return parent1.copy(), parent2.copy()

        # Вероятностный кроссовер

        alpha = random.random()

        child1 = np.zeros(self.chromosome_length, dtype=int)

        child2 = np.zeros(self.chromosome_length, dtype=int)

        for i in range(self.chromosome_length):

            if random.random() < alpha:

                child1[i] = parent1[i]

                child2[i] = parent2[i]

            else:

                child1[i] = parent2[i]
```

```
                child2[i] = parent1[i]

        return child1, child2

    def mutate(self, chromosome: np.ndarray) -> np.ndarray:

        """Инверсная мутация"""

        mutated = chromosome.copy()

        for i in range(self.chromosome_length):

            if random.random() < self.mutation_rate:

                mutated[i] = 1 - mutated[i]

        return mutated

    def local_search(self, chromosome: np.ndarray) -> np.ndarray:

        """Локальный поиск для улучшения решения"""

        improved = chromosome.copy()

        current_fitness = self.problem.evaluate(chromosome)

        # Пытаемся добавить предметы

        for i in range(self.chromosome_length):

            if improved[i] == 0:

                improved[i] = 1

                new_fitness = self.problem.evaluate(improved)

                if new_fitness > current_fitness:

                    current_fitness = new_fitness

                else:

                    improved[i] = 0

        # Пытаемся заменить предметы

        for i in range(self.chromosome_length):

            if improved[i] == 1:
```

```
                for j in range(self.chromosome_length):
                    if improved[j] == 0:
                        # Пробуем заменить i на j
                        temp = improved.copy()
                        temp[i] = 0
                        temp[j] = 1
                        new_fitness = self.problem.evaluate(temp)
                        if new_fitness > current_fitness:
                            improved = temp
                            current_fitness = new_fitness
                            break
        self.local_search_applications += 1
        return improved
    def run(self) -> Dict:
        """Запуск гибридного GA"""
        start_time = time.time()
        convergence_threshold = 0.001
        no_improvement_count = 0
        last_best = -np.inf
        for generation in range(self.max_generations):
            fitness_values = self.evaluate_population()
            current_best = max(fitness_values)
            # Проверка сходимости
            if generation > 0:
```

```
                improvement = (current_best - last_best) / abs(last_best) if
last_best != 0 else 1

                if improvement < convergence_threshold:

                    no_improvement_count += 1

                else:

                    no_improvement_count = 0

                if no_improvement_count >= 10:

                    if self.convergence_generation == 0:

                        self.convergence_generation = generation

            last_best = current_best

            # Обновление лучшего решения

            best_idx = np.argmax(fitness_values)

            if fitness_values[best_idx] > self.best_fitness:

                self.best_fitness = fitness_values[best_idx]

                self.best_solution = self.population[best_idx].copy()

            # Сохранение истории

            self.fitness_history.append({

                'best': current_best,

                'average': np.mean(fitness_values),

                'worst': min(fitness_values),

                'std': np.std(fitness_values)

            })

            # Создание нового поколения

            new_population = []

            # Элитизм
```

```
            sorted_indices = np.argsort(fitness_values)[::-1]
            for i in range(min(self.elite_count, self.population_size)):
                new_population.append(self.population[sorted_indices[i]].copy())
            # Генерация остальной популяции
            while len(new_population) < self.population_size:
                parent1, parent2 = self.select_parents(fitness_values)
                child1, child2 = self.crossover(parent1, parent2)
                child1 = self.mutate(child1)
                child2 = self.mutate(child2)
                # Применяем локальный поиск с заданной вероятностью
                if random.random() < self.local_search_rate:
                    child1 = self.local_search(child1)
                if random.random() < self.local_search_rate:
                    child2 = self.local_search(child2)
                new_population.append(child1)
                if len(new_population) < self.population_size:
                    new_population.append(child2)
            self.population = new_population[:self.population_size]
        execution_time = time.time() - start_time
        if self.convergence_generation == 0:
            self.convergence_generation = self.max_generations
        return {
            'best_solution': self.best_solution,
            'best_fitness': self.best_fitness,
            'fitness_history': self.fitness_history,
```

```
            'execution_time': execution_time,
            'convergence_generation': self.convergence_generation,
            'total_generations': self.max_generations,
            'local_search_applications': self.local_search_applications
        }
```