Введение

Проблема оптимизации является одним из фундаментальных аспектов в различных областях науки и техники. Одной из классических задач оптимизации является 0-1 задача о ранце, которая имеет широкое применение в экономике, производстве, логистике и других областях. Суть этой задачи заключается в том, чтобы уложить как можно большее число ценных вещей в рюкзак при условии, что вместимость рюкзака ограничена.

В последние десятилетия наблюдается стремительное развитие метаэвристических алгоритмов оптимизации, которые позволяют эффективно решать сложные задачи оптимизации в различных областях. Одним из таких алгоритмов является алгоритм оптимизации, основанный на моделировании миграционного поведения бабочек-монархов в природе, Monarch Butterfly Optimization (MBO) [6].

В данной работе рассматривается применение алгоритма MBO для решения 0-1 задачи о ранце. Алгоритм MBO вдохновлен поведением бабочек монархов в процессе их миграции, адаптируясь к окружающей среде и поиску оптимальных путей. Использование этого алгоритма для решения 0-1 задачи о ранце представляет собой инновационный подход, который может привести к улучшению качества решений и повышению производительности.

Целью данной работы является анализ и реализация алгоритма Monarch Butterfly Optimization для решения 0-1 задачи о ранце на языке Python и тестирование созданной реализации. В ходе тестирования была создана база тестовых 0-1 задач о ранце, которая содержала задачи 4 типов: некоррелированных, слабо коррелированных, сильно коррелированных и обратно сильно коррелированных.

Работа состоит из 4 глав и приложения. В главе 1 приводится постановка задачи о ранце, описывается использование динамического программирования, а также рассматриваются различные типы задач о ранце. Далее описывается создание базы тестовых задач и приводятся результаты тестирования программы решения 0-1 задачи о ранце с помощью динамического программирования. В главе 2 представлен алгоритм оптимизации MBO, описывается поведение бабочек-монархов, операторы миграции и регулирования. В главе 3 описана модификация алгоритма MBO для решения 0-1 задачи о ранце и описывается его реализация. В главе 4 анализируются результаты тестирования алгоритма MBO для четырех типов задач.

1 Создание базы тестовых 0-1 задач для ранца

1.1 Постановка 0-1 задачи о ранце

0-1 задача о ранце — это известная NP-полная задача комбинаторной оптимизации. В общем виде задачу можно сформулировать так: требуется выбрать оптимальное подмножество предметов из заданного набора, учитывая их "ценность" и "вес". Основная цель — максимизировать общую ценность выбранных предметов при соблюдении ограничения на суммарный вес, который рассматривается как вместимость ранца [9].

Математически задача формулируется следующим образом: имеется набор из n предметов i с соответствующими ценностями c_i и весами w_i. Задача состоит в том, чтобы заполнить ранец с грузоподъемностью W таким подмножеством предметов, чтобы их суммарная ценность,

$$U = \sum_{i=1}^{n} c_i x_i \tag{1}$$

максимизировалась с учетом ограничения на суммарный вес:

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

В уравнениях 1, 2 x_i — бинарные $\{0, 1\}$ переменные принятия решений, отражающие, попадает ли предмет i в ранец.

В работе рассматриваются только 0-1 задачи о ранце. Далее мы будем кратко обозначать такие задачи ЗР.

1.2 Динамическое программирование

Одним из подходов к решению ЗР является динамическое программирование. Динамическое программирование – это математический метод нахождения оптимальных решений многоэтапных задач [1].

Для решения ЗР мы будем использовать рекуррентные формулы Беллмана для нахождения максимальной стоимости предметов, которые можно уместить в ранец с ограниченной грузоподъемностью W:

$$K[i][W] = max(K(i–1, W), c[i] + K(i–1, W–w[i])),$$

где $K[i][W]$ представляет максимальную стоимость предметов, которые можно уместить в ранце с грузоподъемностью не более W, используя только первые i предметов. c – вектор ценностей предметов, w – вектор весов. В результате определяем оптимальную комбинацию предметов и их ценность $K[n][W]$.

При использовании динамического программирования создается и заполняется таблица, что может потреблять большое количество памяти, особенно при больших входных данных. Это особенно заметно, если ЗР имеет большое число предметов или большую вместимость. Кроме того, время выполнения алгоритма может быть значительным при больших объемах данных. В некоторых случаях это может сделать его неэффективным для практического использования.

Метод динамического программирования позволяет тестировать другие методы решения ЗР.

1.3 Типы задач о ранце

В этом разделе мы определим 4 типа ЗР, которые отражают свойства, влияющие на процесс решения [5].

Некоррелированные, Uncorrelated instances (UI): значения c_j и w_j выбираются случайным образом из $[1, L]$. В таких случаях нет корреляции между ценностью и весом товара.
Слабо коррелированные, Weakly correlated instances (WCI): вес w_j выбирается случайным образом из $[L/10 + 1, L]$, а ценность c_j выбирается из $[w_j - L/10; w_j + L/10]$, где $c_j \geq 1$.
Сильно коррелированные, Strongly correlated instances (SCI): веса w_j распределены в пределах $[1; L]$, а ценность $c_j = w_j + L/10$.
Обратно сильно коррелированные, Inverse strongly correlated instances (ISCI): ценности c_j распределены в пределах $[L/10; L]$, а веса $w_j = c_j - L/10$.

1.4 База тестовых задач о ранце

В этом разделе мы расскажем о структуре задач из базы данных сложных ЗР и объясним выбор вариантов задач с некоррелированными, слабо коррелированными, сильно коррелированными и обратно сильно коррелированными случайными значениями весов и ценностей. База данных задач хранится в текстовом txt-файле. Информация о задачах разделена пустой строкой.

Информация о задаче состоит из 6 строк: размерность задачи n, грузоподъемность W, вектор весов предметов w, вектор ценностей предметов c, вектор оптимального решения x^*, оптимальная ценность C^*.

В базе хранятся, так называемые, сложные задачи, у которых $A = \sum_{i=1}^{n} w_i = 4W$ [4]. В таких условиях ожидаемое количество использованных элементов в оптимальном решении составит примерно $A/2$, и точное решение становится недоступным при больших значениях A.

В оптимальном решении такой задачи будет наблюдаться сильная связь между значением c_i и вероятностью того, что $x_i = 1$. Используя простую эвристику, основанную на этом наблюдении, часто можно быстро получить решения, близкие к оптимальным. Поэтому мы также рассмотрим класс более сложных задач с более однородными распределениями c_i, где c_i равны, а ценность пропорциональна количеству использованных предметов [4].

Размерность задачи n – число из интервала $[l, m]$. В текущем варианте базы присутствуют задачи, где $l = 10$, $m = 100$. Размерности задач выбираются с шагом 10.
Вектор ценностей c – выбирается в зависимости от типа задачи. В текущем варианте базы для UI выбираются независимые случайные числа из интервала $[c_{min}, c_{max}]$, где $c_{min} = 1$, $c_{max} = 100$.
Вектор весов w — выбирается в зависимости от типа задачи. В текущем варианте базы для UI выбираются независимые случайные числа из интервала $[w_{min}, w_{max}]$, где $w_{min} = 1$, $w_{max} = 100$.
Грузоподъемность W — вычисляется по формуле $W = \frac{1}{4}\sum_{i=1}^{n} w_i$.
Вектор оптимального решения x^* — вычисляется с помощью динамического программирования.
Оптимальная ценность C^* – вычислена с помощью динамического программирования.

При условии, что грузоподъемность W близка к $A/2$, решение этой задачи будет состоять преимущественно из предметов с $x_i = 1$. В случае, когда

$W << A/4$, количество допустимых решений будет ограничено, и точное решение может быть относительно легко найдено [4].
Далее мы приведем описание программы создания базы тестовых ЗР для UI, WCI, SCI и ISCI на языке Python.

Программа состоит из следующих пунктов: генерация ЗР, чтение данных из файла, решение ЗР, выполнение и запись результатов.

Чтобы сгенерировать ЗР, мы выполняем функцию, которая создает случайные ЗР с заданным числом предметов (n), весами и ценностями.
c — вектор случайных ценностей. w — вектор случайных весов.

Для UI мы генерируем функцию $generate\_knapsack\_problems\_UI$.
Вектор ценностей и вектор весов выбираются случайным образом из интервала $[1, L]$. В текущем варианте $L = 100$.
Для WCI мы генерируем функцию $generate\_knapsack\_problems\_WCI$.
Вектор весов выбирается из интервала $[L/10 + 1, L]$, а вектор ценностей выбирается из $[w_j - L/10; w_j + L/10]$.
Для SCI мы генерируем функцию $generate\_knapsack\_problems\_SCI$.
Вектор весов выбирается из $[1; L]$, а вектор ценностей вычисляется по формуле $c_j = w_j + L/10$.
Для ISCI мы генерируем функцию $generate\_knapsack\_problems\_ISCI$.
Вектор ценностей выбирается из $[L/10; L]$, а вектор весов вычисляется по формуле $w_j = c_j - L/10$.
Для всех задач в текущем варианте $L = 100$.

Далее мы вычисляем грузоподъемность по формуле $W = \frac{1}{4}\sum_{i=1}^{n} w_i$. В итоге возвращаются грузоподъемность W, веса w и ценности предметов c.

Далее приведем код для генерации случайных задач UI, WCI, SCI и ISCI.

```
def generate_knapsack_problems_UI(num_items):
    w = []
    c = []
    for _ in range(num_items):
        value = np.random.uniform(1, 100)
        weight = np.random.uniform(1, 100)
        w.append(weight)
        c.append(value)
    A = 0
    for j in range(n):
        A += w[j]
```

```
    W = A/4
    return w, c, W


def generate_knapsack_problems_WCI(num_items):
    w = []
    c = []
    L=100
    for _ in range(num_items):
        weight = np.random.uniform(L/10 + 1, L)
        value = np.random.uniform(weight-L/10, weight+L/10)
        w.append(weight)
        c.append(value)
    A = 0
    for j in range(n):
        A += w[j]
    W = A/4
    return w, c, W


def generate_knapsack_problems_SCI(num_items):
    w = []
    c = []
    L = 100
    for _ in range(num_items):
        weight = np.random.uniform(1, L)
        value = weight + L/10
        w.append(weight)
        c.append(value)
    A = 0
    for j in range(num_items):
        A += w[j]
    W = A/4
    return w, c, W


def generate_knapsack_problems_ISCI(num_items):
    w = []
    c = []
    L = 100
    for _ in range(num_items):
        value = np.random.uniform(L/10, L)
        weight = value - L/10
        w.append(weight)
```

```
        c.append(value)
    A = 0
    for j in range(num_items):
        A += w[j]
    W = A/4
    return w, c, W
```

Сгенерированные задачи записываются в текстовый файл *knapsack_problems.txt*. Для каждой задачи записываются размерность, грузоподъемность, веса и ценности. Информация о задачах разделяется пустой строкой.

Функция *Knapsack* решает ЗР, используя динамическое программирование. За основу берется программа из [10] с изменением ввода данных и вывода результатов. Опишем основные этапы выполнения функции *Knapsack*.

Создается двумерный массив K размером $(n + 1) \times (10W + 1)$, где $K[i][w]$ – максимальная стоимость, которую можно получить из первых i предметов, если вместимость ранца равна $w/10$ (так как веса предметов могут быть дробными, умножаем на 10 для получения целочисленного значения).

Заполняем массив K построчно. Если $i = 0$ или $w = 0$, то $K[i][w] = 0$. Если вес i-го предмета меньше или равен текущей вместимости, то мы можем либо добавить i-й предмет, либо не добавлять его. Выбираем максимальную стоимость из этих двух вариантов. Если вес i-го предмета больше текущей вместимости, то мы не можем его добавить и просто копируем значение из предыдущей строки.

После заполнения массива K мы находим комбинацию предметов, которая дает максимальную стоимость. Для этого начинаем с $K[n][W * 10]$ и идем вверх по строкам и влево по столбцам. Если значение $K[i][j]$ отличается от $K[i-1][j]$, то мы добавляем i-й предмет в ранец и переходим на следующую строку и столбец $j - wt[i - 1] * 10$. Если значение $K[i][j]$ равно $K[i - 1][j]$, то i-й предмет не добавляем и переходим только на следующую строку. Полученная комбинация предметов записывается в массив *combination*.

В конце функция возвращает набор выбранных предметов и максимальную стоимость, которую можно получить при заданных входных данных.

```
def knapSack(w, c, n, W):
K = [[0.0 for x in range(int(W*10) + 1)] for x in range(n +
```

```
1)]
    combination = []
    for i in range(n + 1):
        for wt in range(int(W*10) + 1):
            if i == 0 or wt == 0:
                K[i][wt] = 0
            elif w[i-1]*10 <= wt:
                K[i][wt] = max(c[i-1] + K[i-1][wt-int(w[i-1]*10)],
K[i-1][wt])
            else:
                K[i][wt] = K[i-1][wt]
    combination = []
    i = n
    j = int(W*10)
    while i > 0 and j > 0:
        if K[i][j] != K[i-1][j]:
            combination.append(1)
            j -= int(w[i-1]*10)
        else:
            combination.append(0)
        i -= 1
    while len(combination)<n:
        combination.append(0)
    combination.reverse()
    return combination, K[n][int(W*10)]
```

Функция $read\_knapsack\_problem\_from\_file(filename)$ открывает текстовый файл $knapsack\_problems.txt$, извлекает информацию о размерности, грузоподъемности, весах и ценностях предметов, возвращая соответствующие массивы данных. Затем происходит обработка каждой задачи: для каждой вызывается функция $knapSack$, которая решает ЗР с помощью динамического программирования. Результаты, включая размерность, грузоподъемность, веса, ценности, оптимальный набор предметов и максимальную ценность, записываются в новый текстовый файл $solution\_knapsack\_problems.txt$.

Полный текст программы генерации тестовых задачи для UI приведен в приложении 5.1.

1.5 Тестирование программы решения ЗР

Было проведено тестирование созданной программы *knapsack_problem.py* с использованием базы данных, имеющейся на сайте Kaggle [8]. Для тестирования был написан набор функций, который преобразует данные с сайта Kaggle в данные для программы *knapsack_problem.py*.

Полный текст программы преобразования данных с сайта в данные для программы приведен в приложении 5.2.

В результате полученные решения совпадают с решениями, приведенными на сайте Kaggle.

[18 47 20 9 39]	[7 4 17 19 3]	67	[1. 0. 1. 1. 0.]	43.0
[34 46 20 45 3]	[6 6 10 2 14]	92	[1. 0. 1. 0. 1.]	30.0
[28 21 7 48 47]	[3 19 1 15 18]	82	[0. 1. 1. 0. 1.]	38.0
[27 18 11 32 47]	[10 4 12 4 1]	18	[0. 0. 1. 0. 0.]	12.0
[36 41 1 46 40]	[17 6 1 10 8]	42	[1. 0. 1. 0. 0.]	18.0
[42 24 41 28 31]	[16 15 12 19 1]	44	[0. 0. 0. 1. 0.]	19.0
[10 37 24 25 41]	[1 6 19 9 1]	92	[0. 1. 1. 1. 0.]	34.0
[22 30 16 18 37]	[10 16 16 17 12]	63	[1. 0. 1. 1. 0.]	43.0
[40 35 24 28 27]	[19 13 16 3 2]	86	[1. 0. 1. 0. 0.]	35.0
[4 4 21 23 38]	[18 11 18 1 18]	86	[1. 1. 1. 0. 1.]	65.0
[29 28 46 37 15]	[2 3 3 1 5]	58	[0. 1. 0. 0. 1.]	8.0
[26 14 6 34 1]	[18 7 17 3 9]	36	[1. 0. 1. 0. 1.]	44.0
[42 47 37 13 15]	[6 10 16 8 18]	76	[0. 0. 1. 1. 1.]	42.0
[16 24 29 22 5]	[10 6 13 8 8]	83	[1. 0. 1. 1. 1.]	39.0

Рис. 1: База данных с сайта Kaggle

```
5
67.0
18.0 47.0 20.0 9.0 39.0
7.0 4.0 17.0 19.0 3.0
[1, 0, 1, 1, 0]
43.0

5
92.0
34.0 46.0 20.0 45.0 3.0
6.0 6.0 10.0 2.0 14.0
[1, 0, 1, 0, 1]
30.0

5
82.0
28.0 21.0 7.0 48.0 47.0
3.0 19.0 1.0 15.0 18.0
[0, 1, 1, 0, 1]
38.0

5
18.0
27.0 18.0 11.0 32.0 47.0
10.0 4.0 12.0 4.0 1.0
[0, 0, 1, 0, 0]
12.0

5
42.0
36.0 41.0 1.0 46.0 40.0
17.0 6.0 1.0 10.0 8.0
[1, 0, 1, 0, 0]
18.0

5
44.0
42.0 24.0 41.0 28.0 31.0
16.0 15.0 12.0 19.0 1.0
[0, 0, 0, 1, 0]
19.0

5
92.0
10.0 37.0 24.0 25.0 41.0
1.0 6.0 19.0 9.0 1.0
[0, 1, 1, 1, 0]
34.0
```

Рис. 2: Результат работы программы *knapsack_problem.py*

1.6 Применение алгоритма решения задачи о ранце

С помощью написанной программы *knapsack_problem.py* было проведено решение задач из базы данных, имеющейся на сайте Kaggle [9]. База данных хранила в себе задачи 4 типов: UI, WCI, SCI и ISCI.

Для решения был написан набор функций, который преобразует данные с сайта Kaggle в данные для программы *knapsack_problem.py*.

2 Алгоритм Monarch butterfly optimization

В этом разделе мы рассмотрим алгоритм Monarch butterfly optimization (MBO) [3, 6] и реализуем его для решения ЗР на языке Python.

2.1 Описание поведения бабочкек-монархов

Будем считать, что выполнены следующие предположения:

1. Все бабочки-монархи находятся только в Северной Канаде и США (Земле 1) или в Мексике (Земле 2), где они составляют всю популяцию бабочек-монархов.
2. Каждая дочерняя особь бабочки-монарха создается с помощью оператора миграции из бабочки-монарха в Земле 1 или в Земле 2.
3. Чтобы популяция оставалась неизменной, старая бабочка-монарх умирает, как только родится здоровый ребенок. В алгоритмах MBO это можно выполнить, заменяя родительский объект вновь сгенерированным, если он лучше приспособлен по сравнению со своим родительским. С другой стороны, новый сгенерированный объект умирает, если он не демонстрирует лучшей пригодности по отношению к своему родительскому объекту. В этом случае родитель остается живым и нетронутым.
4. Особи бабочки-монарха с лучшими качествами автоматически переходят в следующее поколение, и их не могут изменить никакие операторы. Это может гарантировать, что качество или эффективность популяции бабочек-монархов никогда не будет ухудшаться с увеличением количества поколений.

2.2 Оператор миграции

Бабочки-монархи мигрируют из Земли 1 в Землю 2 в течение апреля и обратно в течение сентября, то есть бабочки-монархи остаются на Земле 1 с апреля по август (5 месяцев) и на Земле 2 с сентября по март (7 месяцев). Бабочки-монархи на Земле 1 и Земле 2 называются Субпопуляцией 1 и Субпопуляцией 2 соответственно.

Введем следующие обозначения: NP – это общая численность популяции бабочек; p – доля (коэффициент) бабочек-монархов на Земле 1. $ceil(x)$ округляет x до ближайшего целого числа, большего или равного x. Следовательно, количество бабочек-монархов на Земле 1 и Земле 2 равно соответственно:

$$NP_1 = ceil(p * NP) \tag{3}$$

$$NP_2 = NP - NP_1, \tag{4}$$

Обозначим через t — номер текущего поколения; r_j — бабочка-монарх, которая случайным образом выбирается из Субпопуляции j (j=1,2); i — номер выбранной бабочки-монарха; x_i — расположение бабочки-монарха с номером i.

$x_{i,k}^{t+1}$ указывает на k-й элемент x_i в поколении $t+1$, который показывает положение бабочки-монарха i. $x_{r_1,k}^{t}$ указывает на k-й элемент x_{r_1}, который показывает вновь созданное положение бабочки-монарха r_1. Процесс миграции можно описать следующим образом:

$$x_{i,k}^{t+1} = x_{r_1,k}^{t} \tag{5}$$

Пусть $peri$ — период миграции (12 месяцев в году); $rand$ — это случайное число, полученное из равномерного распределения.

$$r = rand * peri \tag{6}$$

Когда $r \leq p$, элемент k во вновь созданной последовательности положений бабочек-монархов описывается формулой 5.

Пусть $x_{r_2,k}^{t}$ указывает на k-й элемент x_{r_2}, который показывает новое положение бабочки-монарха r_2. Если $r > p$, то элемент k во вновь созданной бабочке-монархе описывается

$$x_{i,k}^{t+1} = x_{r_2,k}^{t} \tag{7}$$

Таким образом, мы делаем вывод, что метод MBO может сбалансировать направление оператора миграции, регулируя значение коэффициента p.

Если параметр $p > 0.5$, будет выбрано больше элементов из бабочек-монархов на Земле 1. Это указывает на то, что Субпопуляция 1 играет более важную роль в новой бабочке-монархе.

Если параметр $p < 0.5$, будет выбрано больше элементов из бабочек-монархов на Земле 2. Тогда Субпопуляция 2 играет более важную роль в новой бабочке-монархе.

Algorithm 1 Оператор миграции

for $i = 1$ to NP_1 (для всех бабочек-монархов в Субпопуляции 1) **do**
  for $k = 1$ to D (для всех элементов из последовательности положений i-й бабочки-монарха) **do**
    Случайным образом генерируется число $rand$ с помощью равномерного распределения
    $r = rand * peri$
    if $r \leq p$ **then**
      Случайным образом выбирается бабочка-монарх из Субпопуляции 1 (обозначим r_1)
      Получаем k-й элемент x_i^{t+1} как в формуле 5
    else
      Случайным образом выбирается бабочка-монарх из Субпопуляции 2 (обозначим r_2)
      Получаем k-й элемент x_i^{t+1} как в формуле 7
    end if
  end for k
end for i

В статье [6] $p = 5/12$ в соответствии с периодом миграции. Соответственно, оператор миграции может быть представлен в спецификации 1.

2.3 Регулирующий оператор

Положение бабочек-монархов также может быть обновлено с помощью следующего регулирующего оператора.

Введем следующие обозначения: t – это номер текущего поколения; $best$ – бабочка-монарх, которая является лучшей по качествам на Земле 1 и Земле 2; j – номер выбранной бабочки-монарха; x_j – расположение бабочки-монарха с номером k.

$x_{j,k}^{t+1}$ указывает на k-й элемент x_j в поколении $t+1$, который показывает положение бабочки-монарха j. Также $x_{best,k}^{t}$ указывает на k-й элемент x_{best}, который является положением лучшей бабочки-монарха на Земле 1 и Земле 2.

Процесс регулирующего оператора можно просто описать следующим образом. Для всех элементов, показывающих положение бабочки-монарха j, если случайно сгенерированное число $rand$ меньшее или равное p, тогда k можно получить:

$$x_{j,k}^{t+1} = x_{best,k}^{t} \tag{8}$$

Пусть $x_{r_3,k}^{t}$ указывает на k-й элемент x_{r_3}, который случайным образом выбирается из Земли 2. $r_3 \in \{1, 2, ..., NP_2\}$.

Если $rand$ больше, чем p, тогда элемент k положений бабочки может быть получен:

$$x_{j,k}^{t+1} = x_{r_3,k}^{t} \tag{9}$$

Обозначим BAR — скорость регулировки бабочки. dx — это шаг бабочки-монарха j, который может быть получен с помощью полета Леви:

$$dx = Levy\left(x_j^t\right) \tag{10}$$

где $Levy$ — распределение Леви. Функция плотности вероятности распределения Леви:

$$f(x) = \frac{1}{\sqrt{2\Pi x^3}} exp(-\frac{1}{2x}) \tag{11}$$

для $x > 0$

S_{max} — это максимальный шаг, который может пройти особь бабочки-монарха за один раз. t — текущее поколение. ω — весовой коэффициент, который задается формулой:

$$\omega = S_{max}/t^2 \tag{12}$$

Чем больше ω, обозначающее длинный шаг поиска, тем больше влияние dx на $x_{j,k}^{t+1}$ и развитие процесса исследования. Чем меньшее ω, обозначающее короткий шаг поиска, тем меньше влияние dx на $x_{j,k}^{t+1}$ и развитие процесса исследования.

При этом условии, если $r > BAR$, формулу можно дополнить:

$$x_{j,k}^{t+1} = x_{j,k}^{t} + \omega \times (dx_k - 0.5) \tag{13}$$

Соответственно, описание оператора регулирования бабочки можно дать в спецификации 2

Algorithm 2 Регулирующий оператор

for $j = 1$ to NP_2 (для всех бабочек-монархов в Субпопуляции 2) **do**
Рассчитать шаг dx с помощью 10;
Рассчитать весовой коэффициент с помощью 12;
for $k = 1$ to D (ля всех элементов из последовательности положений j-й бабочки-монарха) **do**
Случайным образов генерируется число $rand$ с помощью равномерного распределения
if $r \leq p$ **then**
Получаем k-й элемент x_j^{t+1} как в 8
else
Случайным образом выбирается бабочка-монарх из Субпопуляции 2 (говорим r_2)
Получаем k-й элемент x_j^{t+1} как в 9
if $rand > BAR$ **then**
$x_{j,k}^{t+1} = x_{j,k}^{t+1} + \omega \times (dx_k - 0.5)$;
end if
end if
end for k
end for j

2.4 Спецификация алгоритма MBO

Оптимизируя миграционное поведение особей бабочки-монарха, можно сформировать метод MBO, алгоритм которого дан в спецификации 3.

Сначала оптимизируются все параметры, после чего генерируется начальная популяция и оценивается с помощью функции приспособленности.

Далее положения всех бабочек-монархов обновляются шаг за шагом до тех пор, пока не выполнятся определенные условия. Следует отметить, что для того, чтобы зафиксировать популяцию и уменьшить оценки приспособленности, количество бабочек-монархов, генерируемых оператором миграции и регулирующим оператором, соответствуют NP_1 и NP_2.

Algorithm 3 Алгоритм оптимизации бабочки-монарха

Шаг 1: Инициализация. Установим счётчик поколений $t = 1$. Инициализируем популяцию P из NP особей бабочек-монархов случайно; установим максимальное поколение $MaxGen$, число бабочек-монархов на Земле 1 – NP_1, число бабочек-монархов на Земле 2 – NP_2, максимальный шаг – S_{max}, скорость регулировки бабочки – BAR, период миграции – $peri$, коэффициент миграции – p.

Шаг 2: Оценка приспособленности. Оценить каждую бабочку-монарха в соответствии с ее позицией.

Шаг 3:

while лучшее решение не найдено **or** $t < MaxGen$ **do**

Отсортируйте всех особей бабочек-монархов по степени их приспособленности.

Разделите особей бабочек-монархов на две субпопуляции (Земля 1 и Земля 2);

for $i = 1$ to NP_1 (для всех бабочек-монархов в Субпопуляции 1) **do**

Создайте новую Субпопуляцию 1 в соответствии со спецификацией 1.

end for i

for $j = 1$ to NP_2 (для всех бабочек-монархов в Субпопуляции 2) **do**

Создайте новую Субпопуляция 2 в соответствии с 2.

end for j

Объедините две созданные субпопуляци в единую популяцию;

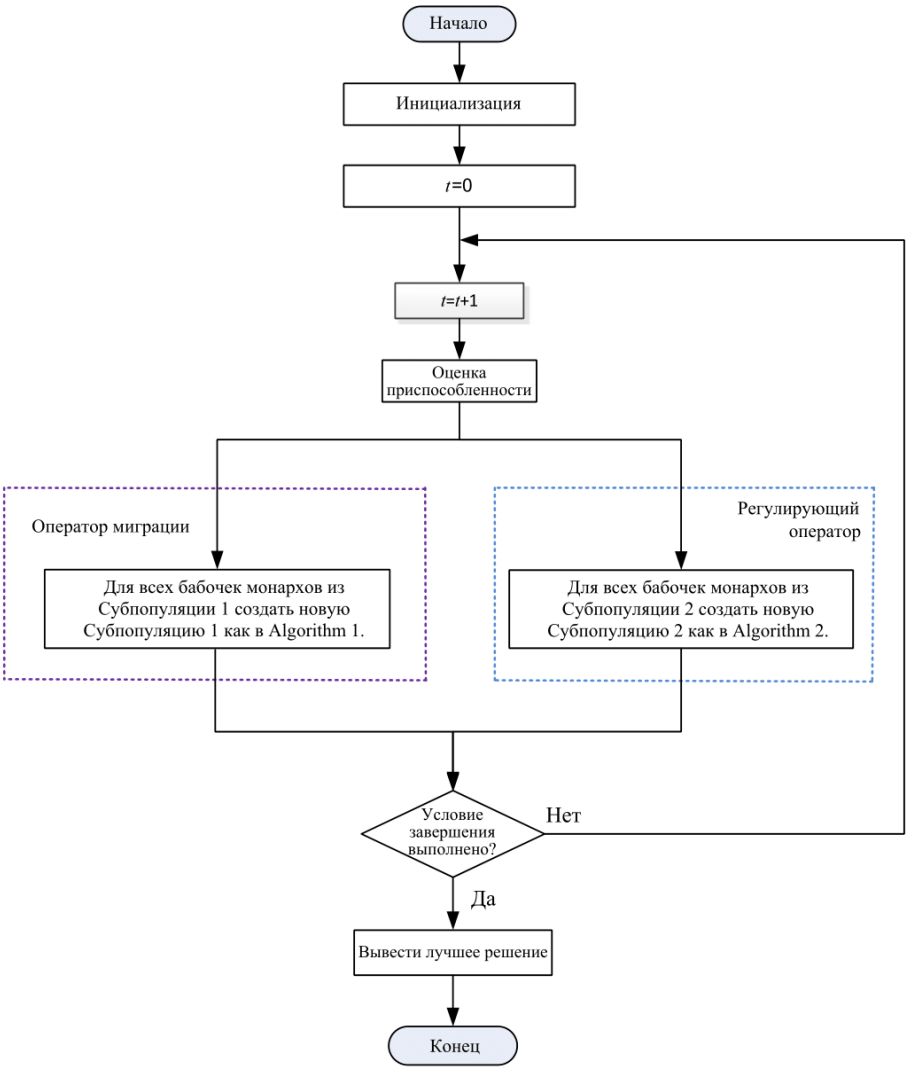
Оцените популяцию в соответствии с недавно обновленными позициями;

$t = t + 1$

Шаг 4:

end while

Шаг 5: Вывести лучшую позицию.



3 Решение ЗР с помощью MBO

3.1 Бинарный алгоритм MBO для ЗР

В этом разделе будет описан бинарный алгоритм MBO (BMBO) для решения ЗР. Во-первых, будет использована гибридная схема кодирования. Во-вторых, будет использоваться новый и эффективный оператор восстановления. В-третьих, разработаны и проверены три вида метода индивидуального распределения двух субпопуляций. В-четвертых, приведено базовое описание BMBO для ЗР. Наконец, мы будем анализировать временную сложность.

3.1.1 Схема кодирования

Исходя из природы ЗР, мы представляем каждого индивида в виде n-битной двоичной строки. Здесь n — размерность решаемой задачи. Но классический алгоритм МВО работает в непрерывном пространстве, то есть оператор миграции и регулирующий оператор бабочки работают непосредственно в n-размерном действительном векторе. Чтобы сохранить механизм эволюции базового МВО, гибридная схема кодирования используется в ВМВО для решения ЗР. Каждая особь бабочки-монарха представлена двумя кортежами $<X, Y>$, где X — пространство поиска, а Y — пространство решений.

Это приводит к взаимно однозначному отображению g из X в Y таких, что

$$g(x) = \begin{cases} 1 & \text{если } sig(x) > 0 \\ 0 & \text{в других случаях} \end{cases} \tag{14}$$

где $sig(x) = 1/(1 + e^{-x})$ — сигмоида.

3.1.2 Оператор восстановления

Обычно при решении ЗР некоторые вновь сгенерированные особи могут нарушать приведенные выше ограничения по суммарному весу. Один из подходов, который обычно используют для работы с ограничениями, — индивидуальный метод оптимизации на основе жадной стратегии.

Основная идея заключается в том, чтобы эффективно упаковывать предметы с учетом их веса и ценности. Если предмет имеет высокую удельную полезность c_i/w_i, то $x_i = 1$ и он будет добавляться в ранец до тех пор, пока общий вес не превысит грузоподъемность ранца W. При этом предметы, которые невозможно упаковать из-за ограничений по весу, будут исключены из выбора и $x_i = 0$.

В статье [3] используется новый и эффективный алгоритм жадной оптимизации. Отметим, что все элементы отсортированы в порядке невозрастания относительно c_i/w_i и начальный индекс сохраняется в массиве $H[1...n]$. Это означает, что
$c_{H[1]}/w_{H[1]} \geq c_{H[2]}/w_{H[2]} \geq ... \geq c_{H[n]}/w_{H[n]}$.

Описание алгоритма жадной оптимизации представлено в спецификации 4.

Algorithm 4 Алгоритм жадной оптимизации

Ввод: $X = \{x_1, x_2, ..., x_n\} \in \{0,1\}^n$, $Wt = \{w_1, w_2, ..., w_n\}$, $C = \{c_1, c_2, ..., c_n\}$, $H[1, ...n]$, W.
Шаг 1: Восстановление
$weight = 0$; $value = 0$; $temp = 0$
for $i = 1$ to n **do**
    $weight = weight + x_i * w_i$
end for i
if $weight > W$ **then**
    for $i = 1$ to n **do**
        $temp = temp + x_{H[i]} * w_{H[i]}$
        if $temp > C$ **then**
            $temp = temp - x_{H[i]} * w_{H[i]}$
            $x_{H[i]} = 0$
        end if
    end for i
    $weight = temp$
end if
Шаг 2: Оптимизация
for $i = 1$ to n **do**
    if $x_{H[i]} = 0$ и $weight + w_{H[i]}$ **then**
        $x_{H[i]} = 1$
    end if
end for i
Шаг 3: Вычисления
for $i = 1$ to n **do**
    $value = value + x_i * c_i$
end for i
Вывод: $X = \{x_1, x_2, ..., x_n\} \in \{0,1\}^n$, $value$

3.1.3 Три вида популяционных стратегий

В алгоритме MBO вся популяция делится на Субпопуляцию 1 (Земля 1) и Субпопуляцию 2 (Земля 2). Более того, из нашего наблюдения после каждой рекомбинации мы видим, что относительно лучшие особи NP_1 относятся к Субпопуляции 1. Кроме того, две субпопуляции объединяются после каждой смены поколения. Чтобы оценить, что индивидуальная

стратегия распределения может влиять на производительность, мы рассмотрим три вида индивидуальных схем распределения в двух подгруппах.

1) BMBO-1: Вся популяция случайным образом делится на Субпопуляцию 1 и Субпопуляцию 2. Кроме того, в течение всего процесса смены поколений вновь генерируемые субпопуляции не рекомбинируются. Есть две популяции, которые ищут оптимальное значение с помощью разных поисковых операций. Кроме того, может происходить небольшой обмен информацией.

2) BMBO-2: После инициализации лучшие отдельные особи NP_1 относятся к Субпопуляции 1, а остальная часть всей популяции принадлежит к субпопуляции 2. Точно так же вновь генерируемые субпопуляции также не рекомбинируются на протяжении всей смены поколений.

3) BMBO-3: Субпопуляции рекомбинируются в определенных поколениях, а не в каждом.

3.1.4 Алгоритм BMBO для задачи о ранце 0-1

Описание алгоритма BMBO представлено в спецификации 5.

3.1.5 Алгоритм BMBO для 0-1 задачи о ранце

В этом подразделе временная сложность BMBO для 0-1 задачи о ранце подлежит оценке. Из алгоритма 5 видно, что время вычислений в основном определяется шагом 1–3. На шаге 1 временная сложность алгоритма быстрой сортировки — $O(n\ log\, n)$. На шаге 2 инициализация NP особей бабочек-монархов имеют временную сложность $O(NP \times n) = O(n^2)$. На шаге 3 оператор миграции тратит время $O(NP_1 \times n) = O(n^2)$, регулирующий оператор — $O(NP_2 \times n) = O(n^2)$, а алгоритм жадной оптимизации тратит время $O(n)$.

Таким образом, BMBO для задачи о ранце 0–1 работает по временной сложности:

$$O(n\ log\, n) + O(n^2) + O(n^2 + O(n^2 + O(n))) = O(n^2)$$

Algorithm 5 Алгоритм оптимизации бабочки-монарха для задачи о ранце 0-1

Шаг 1: Сортировка. Выполнить алгоритм быстрой сортировки для всех элементов в порядке невозрастания c_i/w_i, 1, где индекс элементов хранится в массиве $H[1...n]$.

Шаг 2: Инициализация. Установим счётчик поколений $g = 1$. Установим максимальное поколение $MaxGen$, число бабочек-монархов на Земле 1 – NP_1, число бабочек-монархов на Земле 2 – NP_2, максимальный шаг – S_{max}, скорость регулировки бабочки – BAR, период миграции – $peri$, коэффициент миграции – p, максимальный интервал генерации между рекомбинацией RG. Создадим NP особей бабочек-монархов случайно $\{<X_1, Y_1>, <X_2, Y_2>, ..., <X_{NP}, Y_{NP}>\}$. Рассчитайте пригодность каждой особи $f(Y_i)$, 1.

Шаг 2: Оценка приспособленности. Оценить каждую бабочку-монарха в соответствии с ее позицией.

Шаг 3: Критерий остановки

Разделите особей бабочек-монархов на две субпопуляции
(Земля 1 и Земля 2).
Определите оптимальную особь $<X_{best}, Y_{best}>$
Создайте новую Субпопуляция 1 в соответствии с алгоритмом 1
Создайте новую Субпопуляция 2 в соответствии с алгоритмом 2
Объедините две созданные субпопуляци в единую популяцию
Восстановите и оптимизируйте особей в соответствии с алгоритмом 4
Сохраните лучшие решения
Найдите текущую лучшую позицию $(Y_{best}, f(Y_{best}))$
$g = g + 1$

Шаг 4: Рекомбинация If $g > RG$
Объедините вновь созданные Субпопуляции в одну.

Шаг 5: end while

Шаг 6: Вывести лучшее решение.

3.2 Реализация алгоритма

В этом разделе мы приведем реализацию алгоритма MBO для решения 0-1 задачи о ранце на языке Python.

Этот алгоритм оптимизации использует поведение бабочек для решения задачи о ранце. Он создает популяцию бабочек, которые мигрируют между двумя субпопуляциями, регулируют свои параметры и оптимизируются с использованием жадного алгоритма. Лучшие решения сохраняются и повторяются в течение нескольких поколений до достижения оптимального решения.

Для создания и инициализации популяции бабочек в алгоритме, мы используем функцию $GeneratePopulation(NP, C, Wt)$, принимающую три аргумента: NP - количество бабочек в популяции, C - массив значений ценности элементов и Wt - массив значений их весов. Для каждой бабочки генерируются случайные значения параметров, затем применяется функция *Binary* для кодирования этих значений в бинарный формат. После этого вызывается функция $ValueAndWeights$, чтобы вычислить общий вес и общую ценность для каждой бабочки. Результатом является массив *data*, содержащий информацию о популяции бабочек.

```
def GeneratePopulation(NP , C, Wt ):
d = C.shape[0]
data = np.empty([NP , d*2+2])
for i in range(NP):
    for j in range(d):
        data[i,j] = random.uniform(-3,3)
Binary(data)
ValueAndWeights(data, C, Wt)
return data
```

С помощью функции *Binary* мы применяем гибридную схему кодирования к популяции бабочек, которая преобразует непрерывные значения в двоичные, в зависимости от пороговой функции сигмоиды. Функция *sigmoid* реализует сигмоидную функцию, которая принимает массив x и возвращает значение, вычисленное по формуле 1/(1+exp(-x)). Это значение используется в функции *Binary*, которая принимает массив x и применяет сигмоиду к подмножеству его элементов. Затем каждый элемент этого подмножества преобразуется в 1, если значение сигмоиды больше или равно 0.5, и в 0 в противном случае.

Таким образом, кодируются бинарные значения параметров для каждой бабочки в популяции, что позволяет использовать их для решения ЗР.

```
def sigmoid(x):
    y = 1/(1+np.exp(-x))
    return y

def Binary(x):
    d = int((x.shape[1]-2)/2)
    for i in range(x.shape[0]):
        for j in range(d,d*2):
            if sigmoid(x[i,j-d]) >= 0.5:
                x[i,j] = 1
            else :
                x[i,j] = 0
    return x
```

Функция $ValueAndWeights$ оценивает каждую особь по степени приспособленности, которая определяется ее способностью решать ЗР. Вычисляется общий вес и общая ценность для каждой бабочки в популяции. Она принимает три аргумента: x - массив, представляющий популяцию бабочек, c - массив значений ценности элементов, и w - массив значений их весов. Для каждой бабочки вычисляется суммарная ценность и суммарный вес, основываясь на бинарных значениях, указывающих, какие элементы выбраны. Результаты сохраняются в последних двух столбцах массива x.м

Таким образом, оценка приспособленности позволяет определить, насколько хорошо каждая бабочка справляется с поставленной перед ней задачей, а именно – решением ЗР. В данном случае более высокие значения общего веса и общей ценности соответствуют более приспособленным бабочкам.

```
def ValueAndWeights(x , c , w):
    d = int((x.shape[1]-2)/2)
    NP = x.shape[0]
    for i in range(NP):
        total_v = 0
        total_w = 0
        for j in range(d,d*2):
            total_c = total_c + x[i,j] * c[j-d]
```

```
            total_w = total_w  + x[i,j] * w[j-d]
        w_index = d*2 + 1
        c_index = d*2
        x[i,c_index] = float(total_c)
        x[i,w_index] = float(total_w)
    return x
```

Функция *Update* обновляет популяцию в метаэвристическом алгоритме оптимизации. Сначала применяется гибридная схема кодирования к популяции, затем вычисляется их общий вес и общая ценность на основе массивов c и w. Обновленная популяция возвращается для дальнейшего использования в алгоритме.

```
def Update(x , c ,w):
    x = Binary(x)
    x = ValueAndWeights(x, c, w)
    return x
```

Оператор миграции и регулирующий оператор используются в метаэвристических алгоритмах оптимизации для поддержания разнообразия в популяции и исследования пространства поиска.

С помощью функции *MigrationOperator* мы реализуем оператор миграции, который представляет собой часть метаэвристического алгоритма оптимизации, где популяция разбивается на две субпопуляции, $SP1$ и $SP2$.

Для каждой бабочки из субпопуляции $SP1$ рассчитывается случайное число r в пределах от 0 до $peri$, где $peri$ — это параметр, ограничивающий диапазон миграции. Если случайное число r меньше или равно вероятности миграции p, которая зависит от размеров субпопуляций, то бабочка остается в своей субпопуляции и случайно выбирает другую бабочку $r1$ из той же субпопуляции и заменяет свои элементы элементами выбранной бабочки $r1$. В противном случае бабочка переходит в другую субпопуляцию $SP2$ и случайно выбирает бабочку $r2$ из нее, заменяя свои элементы элементами выбранной бабочки $r2$. После всех миграций субпопуляция $SP1$ обновляется с помощью функции $Update$, которая принимает массив $SP1$, а также массивы значений ценности и весов элементов c и w.

Этот процесс позволяет бабочкам перемещаться между субпопуляциями, что может способствовать исследованию пространства поиска и разнообразию популяции, что в свою очередь может повысить эффективность алгоритма оптимизации.

```
def MigrationOperator(SP1 , SP2 , peri , p , c , w):
    NP1 = SP1.shape[0]
    d = int((SP1.shape[1]-2)/2)
    NP2 = SP2.shape[0]
    p = NP1 / (NP1 + NP2)
    r = 0
    r1 = 0
    r2 = 0
    for i in range(NP1):
        for j in range(d):
            r = np.random.uniform(0 , 1) * peri
            if r <= p:
                r1 = np.random.randint(0,NP1)
                SP1[i,j] =  SP1[r1,j]
            else:
                r2 = np.random.randint(0,NP2)
                SP1[i,j] =  SP2[r2,j]
    SP1 = Update(SP1 , c, w)
    return SP1
```

С помощью функции *AdjustingOperator* мы реализуем регулирующий оператор.

Он применяется для управления изменениями параметров в популяции $SP2$, влияя на их случайное обновление. Сначала определяется размер популяции и количество параметров в индивидууме. Затем вычисляются случайные изменения параметров с использованием экспоненциального распределения Леви. Для каждого индивидуума из популяции решается, будут ли его параметры обновлены до значений лучшего индивидуума $Xbest$ в соответствии с вероятностью p, или же будут случайно изменены в пределах популяции. Если случайное число превышает порог BAR, параметры могут быть дополнительно изменены с использованием параметра *omega*. После этого происходит обновление популяции с помощью функции $Update$, и обновленная популяция возвращается для дальнейшего использования в алгоритме.

Этот оператор помогает исследовать пространство поиска, внося случайные изменения в параметры индивидуумов, что может улучшить эффективность поиска оптимального решения.

```
def AdjustingOperator(SP2 , Xbest , Maxgen , Smax , t , p , BAR
, c , w) :
    NP2 = SP2.shape[0]
```

```
    d   = int((SP2.shape[1] - 2) / 2)
    dx = np.empty(d)
    omega = Smax / (t**2)
    for i in range(d):
        StepSize = math.ceil(np.random.exponential(2 * Maxgen))
        dx[i] = levy.pdf(StepSize)
    for i in range(NP2):
        for j in range(d):
            rand = np.random.uniform(0 , 1)
            if rand <= p:
                SP2[i,j] = Xbest[j]
            else:
                r3 = np.random.randint(0,NP2)
                SP2[i,j] = SP2[r3,j]
                if rand > BAR:
                    SP2[i,j] = SP2[i,j] + omega * (dx[j] - 0.5)
    Update(SP2, c, w)
    return SP2
```

Функция *ArrangeDensity* выполняет упорядочивание элементов по плотности для решения ЗР. Сначала она вычисляет плотность каждого элемента, делением его ценности на вес. Затем создается массив, содержащий пары плотность-индекс элементов. Массив сортируется по возрастанию плотности, чтобы элементы с наибольшей плотностью были в начале. После этого массив индексов элементов переворачивается, чтобы элементы с наибольшей плотностью оказались в начале, что представляет собой желаемый порядок для выбора элементов при заполнении ранца. Функция возвращает упорядоченный массив индексов элементов.

```
def ArrangeDensity(Wt, C):
    dim = Wt.shape[0]
    index = np.arange(Wt.shape[0]).reshape(Wt.shape[0], 1)
    density = C / Wt  # удельная полезность
    H = np.concatenate((density, index), axis=1)
    H = H[np.argsort(H[:, 0])]
    H = np.flip(H, 0)
    H = H[:, 1]
    H = H[::-1]
    result = H.astype(int).reshape(dim, 1)
    return result
```

Функция *Greedy* реализует жадный алгоритм для решения ЗР. В начале функции определяются параметры: X - бинарный вектор,

представляющий решение ЗР, Wt и C - массивы весов и ценностей элементов соответственно, H- массив индексов элементов, отсортированных по плотности, а W - вместимость ранца.

Первый этап алгоритма – восстановление решения. Функция вычисляет общий вес ранца путем скалярного произведения весов элементов и бинарного вектора X. Если общий вес превышает вместимость ранца W, происходит корректировка решения. Для этого функция перебирает элементы в порядке убывания плотности (заданного массивом H) и уменьшает вес ранца, удаляя элементы, начиная с наименее ценных, до тех пор, пока общий вес не станет меньше либо равным вместимости.

Второй этап - оптимизация решения. Функция снова проходит по элементам в порядке убывания плотности и добавляет элементы в ранец, начиная с наиболее ценных и не превышая вместимость. Для этого проверяется, что элемент еще не включен в ранец (его соответствующий бинарный индикатор равен 0) и что добавление элемента не приведет к превышению вместимости. Если оба условия выполнены, элемент добавляется в ранец.

В конце функция возвращает обновленный бинарный вектор X, который представляет оптимальное решение ЗР.

```
def Greedy(X , Wt , C , H , W):
    n = H.shape[0]
    d = int((X.shape[0]-2)/2)
    weight = 0
    value = 0
    temp = 0
    weight = np.dot(X[d:d*2], Wt)
    if (weight>W):
        for i in range(d):
            temp = temp + X[H[i]+d] * Wt[H[i]]
            if (temp > W):
                temp = temp - X[H[i]+d] * Wt[H[i]]
                X[H[i] + d] = 0
                X[H[i]] =  - 3
        weight = temp
    for i in range(d):
        if (X[H[i]+d] == 0) and ((weight + Wt[H[i]]) <= W):
            X[H[i]+d] = 1
            X[H[i]] = 0
            weight = weight + X[H[i]+d] * Wt[H[i]]
```

```
value = np.dot(X[d:d*2], C)
return X
```

Далее мы реализуем алгоритм решения задачи о ранце с помощью алгоритма MBO. Вначале осуществляется чтение данных о ЗР из файла с использованием функции $read\_knapsack\_problem\_from\_file(filename)$, которая принимает имя файла в качестве аргумента и возвращает количество предметов, вместимость ранца, массивы весов и ценностей предметов.

Затем алгоритм решения применяется для каждой ЗР по очереди. Для каждой задачи инициализируются веса и ценности предметов, а также индексы предметов, упорядоченные по плотности. Далее создается начальная популяция бабочек с помощью функции $GeneratePopulation$, которая создает и инициализирует случайными значениями бабочек для заданных весов и ценностей предметов.

Основной цикл алгоритма включает итеративный процесс оптимизации популяции бабочек на протяжении нескольких поколений. На каждой итерации популяция сортируется по степени приспособленности с помощью функции $SortFitness$, которая ранжирует бабочек по их пригодности для решения задачи. Затем популяция разделяется на две субпопуляции, для которых применяются операторы миграции и регулирования с помощью функций $MigrationOperator$ и $AdjustingOperator$ соответственно. После этого каждая бабочка в популяции оптимизируется с помощью жадного алгоритма при помощи функции $Greedy$, который выбирает наиболее подходящие предметы для ранца с учетом их ценности и веса.

После каждой итерации обновляется приспособленность популяции с помощью функции $Update$, которая пересчитывает общую ценность и вес каждой бабочки. В конце выполнения алгоритма лучшие найденные решения для каждой ЗР записываются в файл $solution\_MBO\_knapsack\_problems.txt$.

Полный текст программы алгоритма MBO приведен в приложении 5.3.

4 Тестирование алгоритма MBO

Было проведено тестирование созданной программы *MBO_knapsack.py* с использованием базы данных, имеющейся на сайте Kaggle, которая хранила в себе задачи 4 типов: UI, WCI, SCI и ISCI. Также было проведено тестирование программы с помощью созданной базы данных этих же задач.

Для того, чтобы определить, насколько точное решение, полученное с помощью динамического программирования, отклоняется от приближенного, полученного с помощью алгоритма MBO, вычисляем отклонение от оптимума. Пусть C_{din} – решение, полученное с помощью динамического программирования, C_{MBO} – решение, полученное с помощью алгоритма MBO. Тогда отклонение от оптимума вычисляется по формуле $\frac{C_{din}-C_{MBO}}{C_{din}}$.

С помощью функций *deviation_UI.py*, *deviation_WCI.py*, *deviation_SCI.py*, *deviation_ISCI.py* мы нашли, на сколько процентом оптимальное решение отличается от точного для соответствующих типов задач.

Полный текст программы для SCI приведен в приложении 5.4.

Далее построим графики с помощью функций UI_graph, WCI_graph, SCI_graph и $ISCI\_graph$ зависимости процента отклонения от размерности для задачи каждого типа, где для каждой размерности будут показаны максимальное и среднее отклонения.

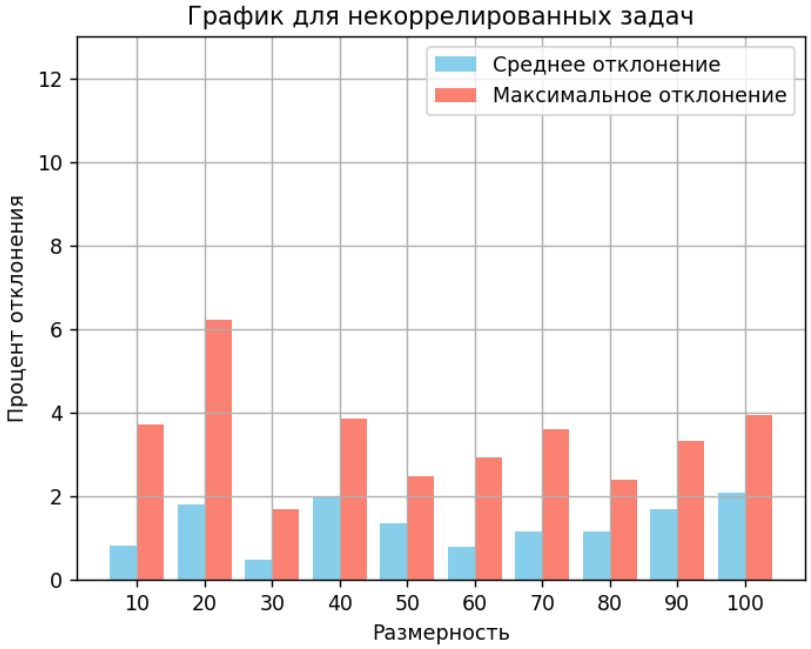


График для некоррелированных задач
Среднее отклонение
Максимальное отклонение
Процент отклонения
0
2
4
6
8
10
12
10
20
30
40
50
60
70
80
90
100
Размерность

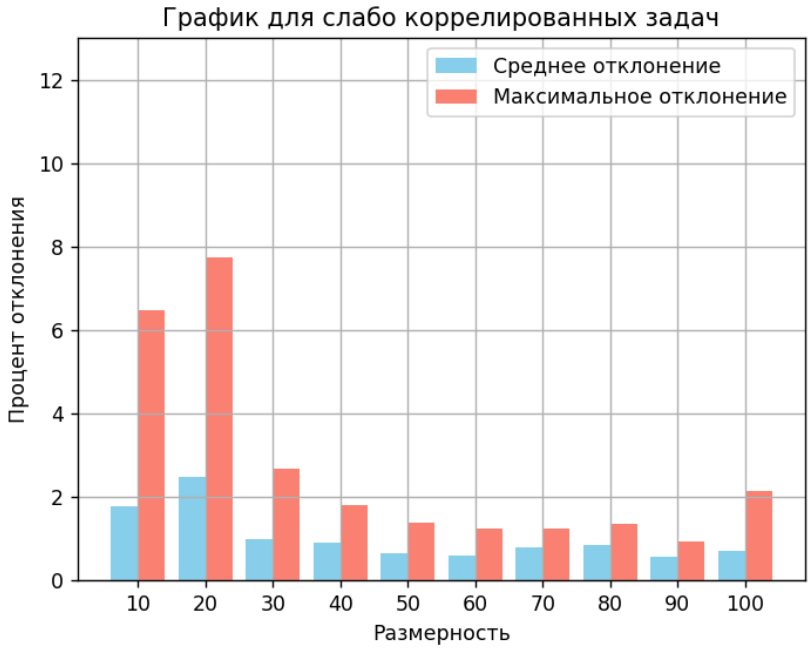
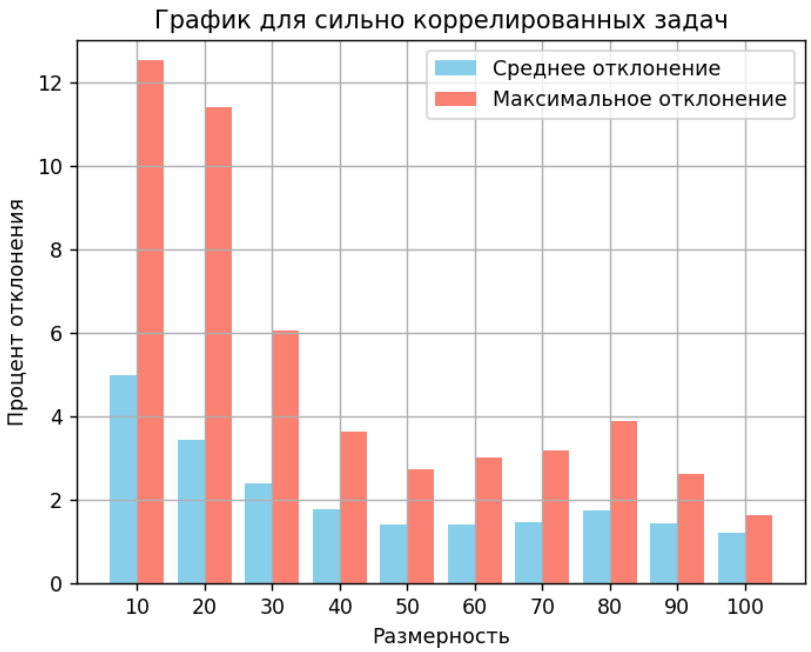
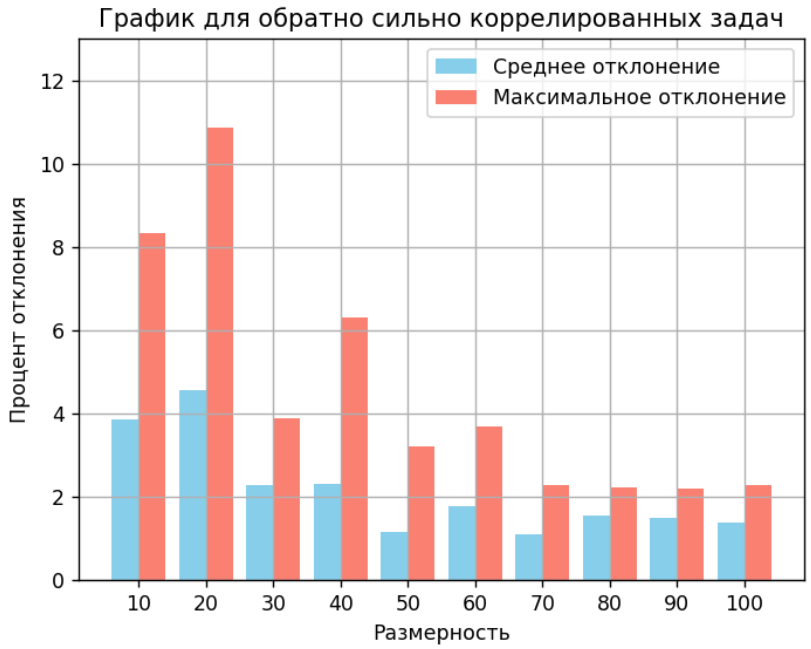


График для слабо коррелированных задач
Среднее отклонение
Максимальное отклонение
Процент отклонения
0
2
4
6
8
10
12
10
20
30
40
50
60
70
80
90
100
Размерность





Полный текст программы для построения графика зависимости процента отклонения от размерности для SCI приведен в приложении 5.5.

Далее построим таблицы для каждого типа задач.

Размерность	Средний процент отклонения	Максимальный процент отклонен
10	0.81	3.72
20	1.79	6.21
30	0.48	1.69
40	1.97	3.86
50	1.36	2.48
60	0.77	2.92
70	1.17	3.59
80	1.15	2.39
90	1.7	3.33
100	2.07	3.95

Рис. 3: Процент отклонения для UI

Размерность	Средний процент отклонения	Максимальный процент отклонен
10	1.79	6.48
20	2.49	7.75
30	0.98	2.68
40	0.89	1.8
50	0.66	1.38
60	0.59	1.23
70	0.8	1.24
80	0.84	1.34
90	0.57	0.92
100	0.7	2.14

Рис. 4: Процент отклонения для WCI

Размерность	Средний процент отклонения	Максимальный процент отклонен
10	4.98	12.51
20	3.43	11.41
30	2.39	6.04
40	1.76	3.64
50	1.4	2.72
60	1.41	3.01
70	1.47	3.19
80	1.74	3.87
90	1.44	2.61
100	1.2	1.63

Рис. 5: Процент отклонения для SCI

Размерность	Средний процент отклонения	Максимальный процент отклонен
10	3.86	8.34
20	4.55	10.87
30	2.28	3.87
40	2.32	6.3
50	1.16	3.21
60	1.78	3.68
70	1.09	2.27
80	1.56	2.23
90	1.49	2.19
100	1.38	2.28

Рис. 6: Процент отклонения для ISCI

Мы получили, что среднее отклонение для UI не превышает 3%, а максимальное отклонение принимается при размерности 20 и оно не превышает 7%. Для WCI среднее отклонение не превышает 3%, а максимальное отклонение принимается при размерности 20 и оно не превышает 8%. Для SCI среднее отклонение не превышает 5%, а максимальное отклонение принимается при размерности 10 и оно не превышает 14%. Для ISCI среднее отклонение не превышает 5%, а максимальное отклонение принимается при размерности 20 и оно не превышает 12%.

В результате мы можем сделать вывод, что среднее отклонение увеличивается с увеличением уровня корреляции между задачами. Для UI и WCI среднее отклонение намного ниже, чем для SCI и ISCI. Это указывает на то, что корреляция между задачами оказывает отрицательное влияние на точность решений.

Кроме того, наблюдается зависимость от размерности. Максимальное отклонение достигает своего максимума при небольших размерностях задач во всех типах задач.

Заключение

В работе была исследована возможность применения нового вида метаэвристического алгоритма, Monarch butterfly optimization, к 0-1 задаче о ранце.

Основные результаты работы заключаются в следующем:

1. Создана база тестовых 0-1 задач о ранце для некоррелированных, слабо коррелированных, сильно коррелированных и обратно сильно

коррелированных величин. Для каждой задачи реализовано решение с использованием метода динамического программирования.

2. Изучен и реализован алгоритм Monarch Butterfly Optimization для решения 0-1 задачи о ранце.

3. Проведено тестирование созданной реализации.

В результате проведённого тестирования было установлено, что максимальное отклонение точного решения, полученного с помощью динамического программирования, от приближенного, вычисленного с использованием алгоритма Monarch Butterfly Optimization, не превышает 13%. Алгоритм продемонстрировал наибольшую эффективность при решении задач с некоррелированными и слабо коррелированными величинами. Однако при сильной и обратно сильной корреляции точность алгоритма снижается, что выражается в более высоких средних и максимальных отклонениях.

Таким образом, цель работы была успешно достигнута.

5 Приложения

5.1 Knapsack problem

```
import random
import time
import math
import numpy as np

def generate_knapsack_problems_UI(num_items):
    w = []
    c = []
    for _ in range(num_items):
        value = np.random.uniform(1, 100)
        weight = np.random.uniform(1, 100)
        w.append(weight)
        c.append(value)
    A = 0
    for j in range(num_items):
        A += w[j]
    W = A/4
    return w, c, W


with open("UI_problems.txt", "w") as file:
    for num_items in range(10, 101, 10):
        for i in range(1, 10):
            weight, value, capacity =
            generate_knapsack_problems_UI(num_items)
            file.write(f"{num_items}\n")
            file.write(f"{capacity}\n")
            line1 = ' '.join(map(str, weight))
            file.write(line1+"\n")
            line2 = ' '.join(map(str, value))
            file.write(line2+"\n")
            file.write("\n")


def knapSack(wt, val, n, W):
    K = [[0.0 for x in range(int(W*10) + 1)] for x in range(n +
1)]
```

```
    combination = []
    for i in range(n + 1):
        for w in range(int(W*10) + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1]*10 <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-int(wt[i-1]*10)],
K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    combination = []
    i = n
    j = int(W*10)
    while i > 0 and j > 0:
        if K[i][j] != K[i-1][j]:
            combination.append(1)
            j -= int(wt[i-1]*10)
        else:
            combination.append(0)
        i -= 1
    while len(combination)<n:
        combination.append(0)
    combination.reverse()
    return combination, K[n][int(W*10)]


def read_knapsack_problem_from_file(filename):
    # Открываем файл на чтение
    with open(filename, 'r') as file:
        # Читаем строки из файла
        lines = file.readlines()
        num_items = []
        capacity = []
        weight = []
        value = []
        # Считываем значения num_items, capacity, weight и
value
        for i in range(0, len(lines), 5):
            num_items.append(int(lines[i].strip()))
            capacity.append(float(lines[i+1].strip()))
            weight.append(list(map(float, lines[i+2].split())))
            value.append(list(map(float, lines[i+3].split())))
        # Возвращаем значения в виде массива
```

```
        return num_items, capacity, weight, value


# Чтение данных из файла
best_combination = []
best_value = []
#knapsack_time = []
num_items, max_W, weight, value =
read_knapsack_problem_from_file('UI_problems.txt')
for i in range(0, len(num_items)):
    #start_time = time.time()
    best_comb, best_v = knapSack(weight[i], value[i], num_items[i],
max_W[i])
    best_combination.append(best_comb)
    best_value.append(best_v)
   # end_time = time.time()
    knapsack_time.append(end_time-start_time)


with open("UI_result.txt", "w") as file:
    for i in range(len(num_items)):
        file.write(f"{num_items[i]}\n")
        file.write(f"{max_W[i]}\n")
        line1 = ' '.join(map(str, weight[i]))
        file.write(line1+"\n")
        line2 = ' '.join(map(str, value[i]))
        file.write(line2+"\n")
        file.write(str(best_combination[i])+"\n")
        file.write(str(best_value[i])+"\n")
       # file.write(str(knapsack_time[i])+"\n")
        file.write("\n")
```

5.2 Преобразование данных с сайта в данные для программы

```
def process_line(line):
    # Разбиваем строку на компоненты
    components = [x.strip() for x in line.split(',')]

    # Извлекаем значения
    values_1 = list(map(int, components[0][1:-1].split()))
    values_2 = list(map(int, components[1][1:-1].split()))
    value_3 = int(components[2])
    values_4 = components[3]
    value_5 = float(components[4])

    # Форматируем вывод
    result = f"{len(values_1)}\n{value_3}\n{' '.join(map(str,
values_1))}\n{' '.join(map(str, values_2))}\n{values_4}\n{value_5}"

    return result

# Имя входного файла
input_file_name = "knapsack_data.txt"
# Имя выходного файла
output_file_name = "knapsack_data_set.txt"

# Читаем строки из входного файла
with open(input_file_name, 'r') as input_file:
    lines = input_file.readlines()

# Обрабатываем строки и записываем результат в выходной файл
with open(output_file_name, 'w') as output_file:
    for line in lines:
        result = process_line(line)
        output_file.write(result + '\n\n')
```

5.3 Monarch butterfly optimization

```
import numpy as np
from scipy.stats import levy
import math
import random
import time

Maxgen = 50 #максимальное поколение
Smax = 1 #максимальный шаг, который моет пройти бабочка за один
раз
t = 1 #номер текущего поколения
p = 5/12 #доля бабочек на Земле 1
BAR = 5/12 #скорость регулировки бабочек
NP = 20 #общая численность популяции
NP1 = 8 #кол-во бабочек на Земле 1
NP2 = 12 #кол-во бабочек на Земле 2
peri = 1.2 #период миграции

#считаем общий вес и общую ценность
def ValueAndWeights(x , c , w): #x имеет размерность x[NP,
d*2+2], d равен размерность v.
    #До d пробегает от 0 до d-1 => исходная популяция.
    d = int((x.shape[1]-2)/2)
    NP = x.shape[0]
    for i in range(NP):
        total_c = 0
        total_w = 0
        for j in range(d,d*2): #от d, тк там заданы 0 и 1
задачи о ранце 0-1
            total_c = total_c + x[i,j] * c[j-d]
            total_w = total_w  + x[i,j] * w[j-d]
        w_index = d*2 + 1
        c_index = d*2
        x[i,c_index] = float(total_c)
        x[i,w_index] = float(total_w)
    return x


def sigmoid(x):
    y = 1/(1+np.exp(-x))
    return y
```

```
def Binary(x): #гибридная схема кодирования
    d = int((x.shape[1]-2)/2)
    for i in range(x.shape[0]):
        for j in range(d,d*2):
            if sigmoid(x[i,j-d]) >= 0.5:
                x[i,j] = 1
            else :
                x[i,j] = 0
    return x

def GeneratePopulation(NP , C, W ): #создаем популяцию
    d = C.shape[0]
    data = np.empty([NP , d*2+2])
    for i in range(NP):
        for j in range(d):
            data[i,j] = random.uniform(-3,3)
    Binary(data)
    ValueAndWeights(data, C, W)
    return data

def SortFitness(x, num_items): #сортировка по приспособленности
    x = x[x[:, num_items*2].argsort()]
    x = np.flip(x,0)
    return x

def Update(x , c ,w): #обновление поколений
    x = Binary(x)
    x = ValueAndWeights(x, c, w)
    return x


def MigrationOperator(SP1 , SP2 , peri , p , c , w): #оператор
Миграции
    NP1 = SP1.shape[0]
    d = int((SP1.shape[1]-2)/2)
    NP2 = SP2.shape[0]
    p = NP1 / (NP1 + NP2)
    r = 0
    r1 = 0
    r2 = 0
    for i in range(NP1):
        for j in range(d):
```

```
            r = np.random.uniform(0 , 1) * peri
            if r <= p:
                r1 = np.random.randint(0,NP1)
                SP1[i,j] =  SP1[r1,j]
            else:
                r2 = np.random.randint(0,NP2)
                SP1[i,j] =  SP2[r2,j]
    SP1 = Update(SP1 , c, w)
    return SP1

def AdjustingOperator(SP2 , Xbest , Maxgen , Smax , t , p , BAR
, c , w) : #регулирующий оператор
    NP2 = SP2.shape[0]
    d   = int((SP2.shape[1] - 2) / 2)
    dx = np.empty(d)
    omega = Smax / (t**2)
    for i in range(d):
        StepSize = math.ceil(np.random.exponential(2 * Maxgen))
        dx[i] = levy.pdf(StepSize)
    for i in range(NP2):
        for j in range(d):
            rand = np.random.uniform(0 , 1)
            if rand <= p:
                SP2[i,j] = Xbest[j]
            else:
                r3 = np.random.randint(0,NP2)
                SP2[i,j] = SP2[r3,j]
                if rand > BAR:
                    SP2[i,j] = SP2[i,j] + omega * (dx[j] - 0.5)
    Update(SP2, c, w)
    return SP2


#упорядочим по плотности
def ArrangeDensity(W, C):
    dim = W.shape[0]
    index = np.arange(W.shape[0]).reshape(W.shape[0], 1)
    density = C / W  # удельная полезность
    H = np.concatenate((density, index), axis=1)
    H = H[np.argsort(H[:, 0])]
    H = np.flip(H, 0)
    H = H[:, 1]
    H = H[::-1]
```

```
    result = H.astype(int).reshape(dim, 1)
    return result

def Greedy(X , Wt , C , H , W):
    #восстановление
    n = H.shape[0] #Массив H - это индексы упорядоченных
элементов по их вместимости.
    d = int((X.shape[0]-2)/2)
    weight = 0
    value = 0
    temp = 0
    #считаем вес
    weight = np.dot(X[d:d*2], Wt)
    if (weight>W): #проверяем, не превышает ли вес вместимость
рюкзака
        for i in range(d): #для каждого элемента в области
исследования
            temp = temp + X[H[i]+d] * Wt[H[i]] #рассматриваем
следующий по емкости элемент в массиве H

#добавляем в переменную временного веса
            if (temp > W): #если выбранный элемент превышает
вместимость рюкзака
                temp = temp - X[H[i]+d] * Wt[H[i]] #возвращаем
предыдущий вес
                X[H[i] + d] = 0  #обновление двоичного значение
                X[H[i]] =  - 3 #обновляем реальное значение
        weight = temp #текущий вес рюкзака

#оптимизация
    for i in range(d):# для всех элементов X[H]
        if (X[H[i]+d] == 0) and ((weight + Wt[H[i]]) <= W):
            X[H[i]+d] = 1 #обновление двоичного значение
            X[H[i]] = 0 #обновляем реальное значение
            weight = weight + X[H[i]+d] * Wt[H[i]]
#вычисление
    value = np.dot(X[d:d*2], C)
    return X


def read_knapsack_problem_from_file(filename):
    # Открываем файл на чтение
    with open(filename, 'r') as file:
```

```
        # Читаем строки из файла
        lines = file.readlines()
        num_items = []
        capacity = []
        weight = []
        value = []

        for i in range(0, len(lines), 5):# Считываем значения
num_items, capacity, weight и value
            num_items.append(int(lines[i].strip()))
            capacity.append(float(lines[i+1].strip()))
            weight.append(np.array(list(map(float,
            lines[i+2].split()))))
            value.append(np.array(list(map(float,
            lines[i+3].split()))))
        # Возвращаем значения в виде массива
        return num_items, capacity, weight, value


num_items, capacity, weight, value =
read_knapsack_problem_from_file('knapsack_problems.txt')
solution = []
#knapsack_time = []
for i in range(len(num_items)):
#создаем вектор весов и ценности
   # start_time = time.time()
    w = weight[i].reshape(num_items[i], 1)
    c = value[i].reshape(num_items[i], 1)
    H = ArrangeDensity(c, w)
    #print(H)
    generate = GeneratePopulation(NP , c, w)
    #print(generate)
#начало алгоритма
    t=1
    while t <= Maxgen:
        generate = SortFitness(generate, num_items[i])
#сортировка каждую бабочку по степени приспособленности
    #разделим бабочек на 2 субпопуляции
        SP1 = generate[:NP1,:]
        SP2 = generate[NP1:,:]
        best = generate[0,:] #лучшая бабочка монарх на земле 1
и земле 2
        SP1 = MigrationOperator(SP1, SP1, peri, p , c , w)
```

```
#новая субпопуляция 1
        SP2 = AdjustingOperator(SP2, best, Maxgen, Smax, t, p,
BAR , c , w) #новая субпопуляция 2
        generate = np.concatenate((SP1, SP2)) #объединение 2-х
созданных субпопуляций
    #оптимизируем особей в соответствии с жадным алгоритмом
        for j in range(generate.shape[0]):
            generate[j] = Greedy(generate[j] , w, c, H ,
capacity[i])
        Update(generate , c,w) #оценка новой популяции
        generate = SortFitness(generate, num_items[i])
        best_solution = generate[0]
        t+=1
        pass

    solution.append(best_solution)
    #end_time = time.time()
    #knapsack_time.append(end_time-start_time)



with open("solution_MBO_knapsack_problems.txt", "w") as file:
    for i in range(len(num_items)):
        file.write(f"{num_items[i]}\n")
        file.write(f"{capacity[i]}\n")
        line1 = ' '.join(map(str, weight[i]))
        file.write(line1+"\n")
        line2 = ' '.join(map(str, value[i]))
        file.write(line2+"\n")
        file.write(str(solution[i
        [len(value[i]):len(value[i])*2]).replace("\n", ""))
        file.write("\n")
        file.write(str(solution[i][len(value[i])*2])+"\n")
       # file.write(str(knapsack_time[i])+"\n")
        file.write("\n")
```

5.4 Отклонение точного решение от приближенного для SCI

```
import random
import time
import math
import numpy as np

with open('SCI_result.txt', 'r') as f1:
    lines1 = f1.readlines()

# Открываем второй файл с задачами о ранце
with open('solution_MBO_SCI.txt', 'r') as f2:
    lines2 = f2.readlines()

# Создаем новый файл для хранения информации
with open('deviation_SCI.txt', 'w') as f_new:
    # Проходим по строкам из первого файла
    for i in range(0, len(lines1), 8):
        f_new.write(lines1[i])
        x1=float(lines1[i+5].strip())
        x2=float(lines2[i+5].strip())
        f_new.write(str(round((x1-x2)/x1*100, 2))+'%'+'\n\n')
```

5.5 Построение графиков зависимости процента отклонения от размерности для SCI

```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict

# Словарь для хранения суммарного процента отклонения и
максимального процента отклонения для каждой размерности
dimension_deviation = defaultdict(lambda: {'total': 0, 'max':
0, 'count': 0})

with open('deviation_SCI.txt', 'r') as file:
    lines = file.readlines()
    for i in range(0, len(lines), 3):
        dimension = int(lines[i].rstrip('\n'))
        percentage = float(lines[i + 1].rstrip('%\n'))
        dimension_deviation[dimension]['total'] += percentage
        dimension_deviation[dimension]['max'] =
        max(dimension_deviation[dimension]['max'], percentage)
        dimension_deviation[dimension]['count'] += 1

# Вычисление среднего отклонения для каждой размерности
dimension_avg_deviation = [dimension_deviation[dim]['total'] /
dimension_deviation[dim]['count'] for dim in
sorted(dimension_deviation.keys())]
# Вычисление максимального отклонения для каждой размерности
dimension_max_deviation = [dimension_deviation[dim]['max'] for
dim in sorted(dimension_deviation.keys())]

# Создание списка размерностей
dimensions = sorted(dimension_deviation.keys())

# Создание списка индексов для позиционирования столбцов
index = np.arange(len(dimensions))
# Ширина каждого столбца
width = 0.4

# Создание столбчатой диаграммы для среднего отклонения
plt.bar(index, dimension_avg_deviation, width=width, color='skyblue',
label='Среднее отклонение')
# Создание столбчатой диаграммы для максимального отклонения
```

```
plt.bar(index + width, dimension_max_deviation, width=width,
color='salmon', label='Максимальное отклонение')

# Настройка осей и заголовка
plt.xlabel('Размерность')
plt.ylabel('Процент отклонения')
plt.title('График для сильно коррелированных задач')
plt.xticks(index + width / 2, dimensions)

# Добавление легенды
plt.legend()

# Отображение графика
plt.grid(True)
plt.show()
```

Список литературы

[1] А. А. Тюхтина, Методы дискретной оптимизации, часть 2, Учебнометодическое пособие, Нижний Новгород: Нижегородский госуниверситет, 2015.

[2] С. А. Чернышев, Основы программирования на Python, Издательство Юрайт, 2023.

[3] Y. Feng, G.-G. Yang, C. Wu, M. Lu, XJ. Zhao, Solving 0–1 knapsack problem by a novel binary monarch butterfly, Neural Comput Applic, 28, 1619–1634, 2017.

[4] M. Ohlsson, C. Peterson, B. Söderberg, Neural Networks for Optimization Problems with Inequality Constraints - the Knapsack Problem. Neural Comput Applic, 5(2), 331-339, 1993.

[5] D. Pisinger, Where are the hard knapsack problems?, Computers and Operations Research, 2271–2284, 2005. Inequality Constraints - the Knapsack Problem. Neural Comput Applic, 5(2), 331-339, 1993.

[6] G.-G. Wang, S. Deb, Z. Cui, Monarch butterfly optimization. Neural Comput Applic, 31, 1995-2014, 2015.

[7] https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D1%80%D1%8E%D0%BA%D0%B7%D0%B0%D0%BA%D0%B5 — Задача о ранце (30.03.2024)

[8] https://www.kaggle.com/code/komalnichat/knapsack-dataset-daa-project/input —- база данных задач о ранце (10.12.2023)

[9] https://www.kaggle.com/datasets/binhthanhdang/dkpdataset – Data set for discounted binary knapsack problem (10.04.2024)

[10] https://www.geeksforgeeks.org/python-program-for-dynamicprogrammin — программа на Python для решения 0-1 задачи о ранце с помощью динамического программирования (30.03.2024)