**Министерство образования Российской Федерации**

**МОСКВОСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. Н.Э.БАУМАНА**

Факультет: Информатика и системы управления (ИУ)  
Кафедра: Информационная безопасность (ИУ8)

**МЕТОДЫ ОПТИМИЗАЦИИ**

**Лабораторная работа №4 на тему:**   
«Решение задачи многокритериальной оптимизации»

Вариант – 1

**Преподаватель:**   
Коннова Н.С.

**Студент:**   
Александров А. Н.

**Группа:**   
ИУ8-34

Москва, 2020

**Цель работы:** изучить постановку задачи многокритериальной оптимизации (МКО); овладеть навыками решения задач МКО с помощью различных методов, осуществить сравнительный анализ результатов, полученных при помощи различных методов.

**Постановка задачи:** выбрать лучшую из альтернатив решения предложенной задачи по варианту из таблицы с точки зрения указанных критериев следующими методами:

1. Заменой критериев ограничениями;
2. Формирование и сужение множества Парето;
3. Методом взвешивания и объединения критериев;
4. Методом анализа иерархий;

**Задача МКО в соответствии с вариантом:**

Задача, альтернативы:   
Покупка автомобиля:

1. Suzuki;
2. Mitsubishi;
3. Honda;
4. Toyota.

Критерии:

1. Стоимость;
2. Расходы на обслуживание;
3. Расход бензина;
4. Комфорт.

Описание предпочтений:

**Стоимость**: Suzuki существенно дороже всех, Honda немного дороже Mitsubishi, Toyota существенно дешевле всех.

**Расходы на обслуживание**: Mitsubishi дороже всех, Toyota и Suzuki примерно равны, Honda дешевле всех.

**Расход бензина:** самый высокий у Suzuki, немного меньше у Honda, существенно меньше у Mitsubishi, самый низкий – у Toyota.

**Комфорт:** самая комфортная – Toyota, чуть менее – Mitsubishi, существенно хуже – Honda, самая некомфортная – Suzuki.

**Ход решения:**

Для решения задачи МКО была написана программа на языке **Python** (см. Приложение А):

В файле ***multicriteria.py*** реализован класс ***Multicriteria,*** в котором описана логика решения задачи всеми четырьмя способами.

Поля класса ***Multicriteria:***

Поле **task\_name\_:** хранит название задачи.

Поле **alternative\_names\_:** хранит названия альтернатив выбора.

Поле **criteria\_names\_:** хранит названия критериев.

Поле **criteria\_weight\_:** хранит вектор весов критериев.

Поле **json\_matrix\_:** хранит матрицу альтернатив, полученную из json -файла.

Поле **alternative\_matrix\_:** хранит матрицу оценок альтернатив.

Поле **normalized\_matrix\_:** хранит нормализированную матрицу оценок альтернатив.

Методы класса ***Multicriteria:***

Метод **NormingVector(self, vector):** нормирует копию переданного вектора и возвращает её.

Метод **NormalizeMatrix(self, matrix):** нормализует копию переданной матрицы и возвращает её.

Метод **OutMatrix(self, matrix):** выводит матрицу альтернатив на экран.

Метод **OutWeight(self):** выводит вектор весов критериев на экран.

Метод **MainCriteriaMethod(self):** производит решение задачи методом главного критерия.

Метод **ParetoMethod(self):** производит решение задачи методом сужения множества Парето.

Метод **NormalizeByColumns(self, current\_matrix):** нормирует столбцы копии переданной марицы и возвращает её.

Метод **CriteriaEvaluation(self):** составляет матрицу экспертной оценки критериев.

Метод **WeighAndCombineMethod(self):** производит решение задачи методом взвешивания и объединения критериев.

Метод **PairCompareMatrix(self, fill\_list):** заполняет матрицу попарных сравнений.

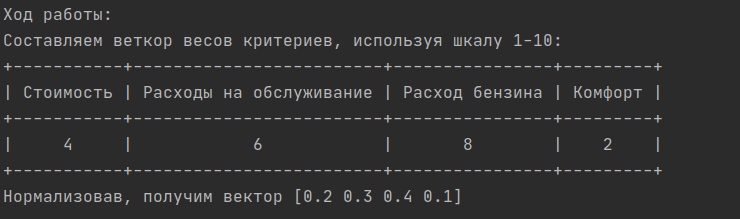
Метод **PairCompareTable(self, names, main\_matrix, sum\_col):** заполняет таблицу попарных сравнений для вывода на экран.

Метод **ConsensusDivision(self, main\_matrix, normalize\_sum\_col):** находит отношение согласованности.

Метод **HierarchiesAnalysisMethod(self):** производит решение задачи методом анализа иерархий.

**Выполнение программы и решение задачи:**

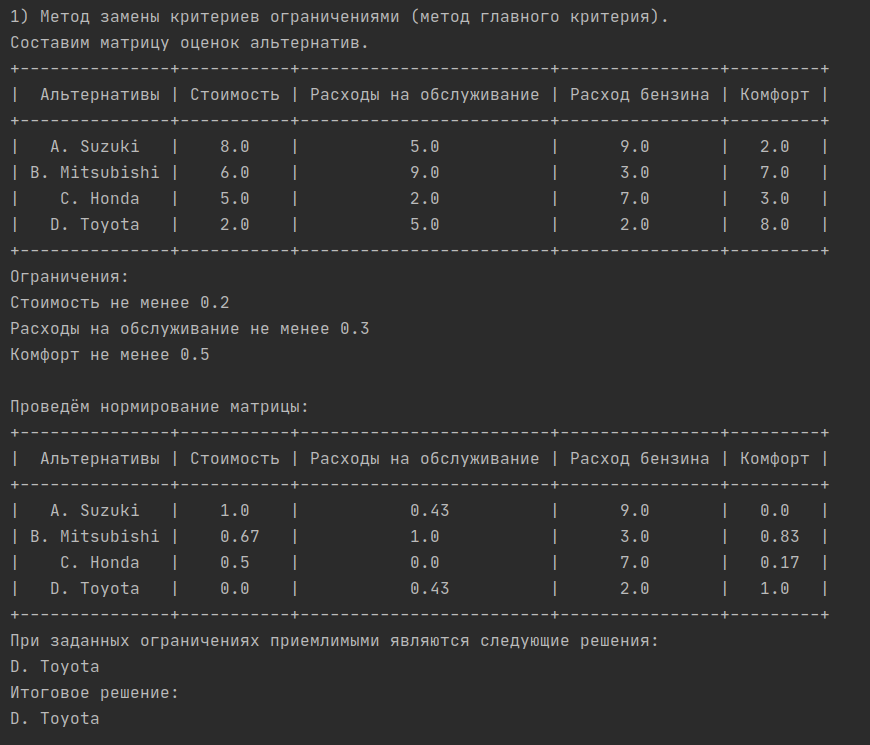
1. Составим вектор весов критериев.

Рисунок 1 Вектор весов критериев.

1. **Метод главного критерия (замена критериев ограничениями):**

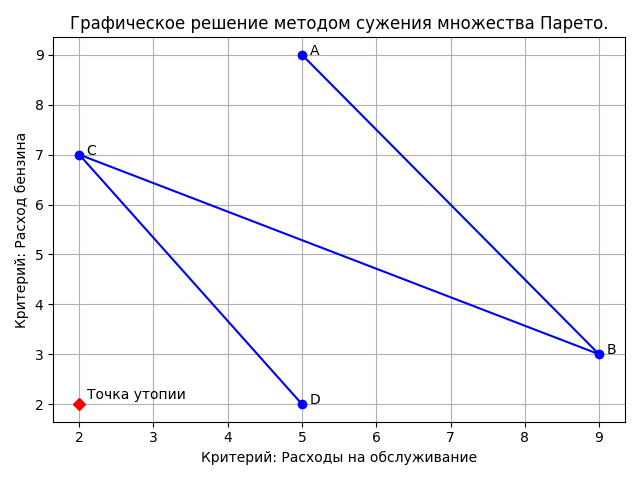
Главным критерием выберем расход бензина .

При заданных условиях приемлимым и единственным является решение: альтернатива *А . Toyota.*

Рисунок 2 Метод главного критерия

3) **Формирование и сужение множества Парето:**

Выберем два критерия: . С помощью библиотеки **Matplotlib** cформируем графически множество Парето:

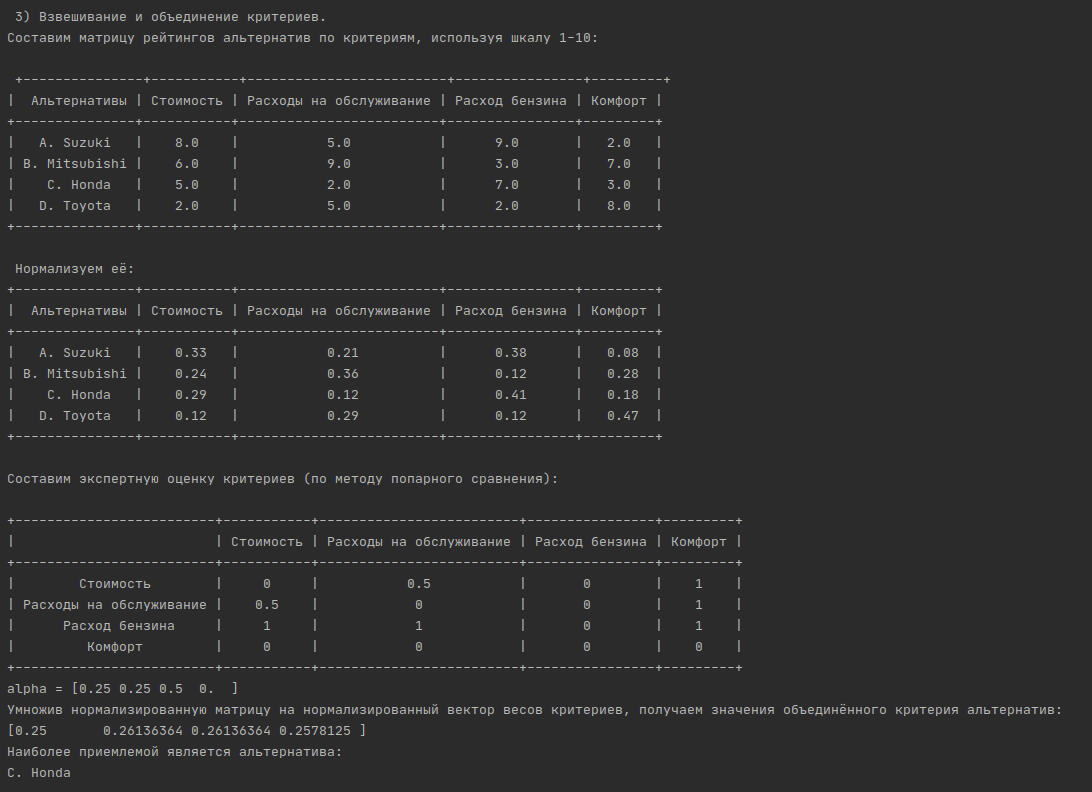
Рисунок 3 Метод сужения множества Парето

Выберем оптимальную альтернативу из множества Парето по минимуму Евклидова расстояния до точки утопии.

Минимальное у альтернативы *A. Toyota*, а значит она оптимальна при заданных критериях.

1. **Взвешивание и объединение критериев (линейная свертка):**

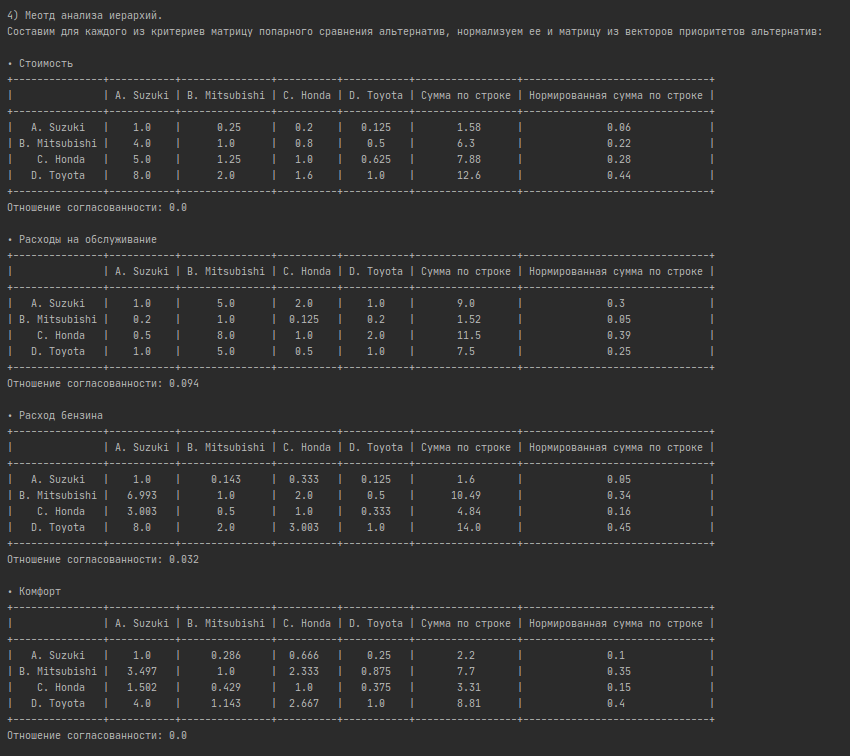
Перемножим полученные нами раннее нормализированные матрицу рейтингов альтернатив и вектор весов критериев. Получим значения объединенного критерия для всех альтернатив. В этом векторе ищем максимальное значение, оно и покажет нам альтернативу:

Рисунок 4 Метод взвешивания и объединения критериев

Здесь лучшей альтернативой стала *B. Honda.*

1. **Метод анализа иерархий.**

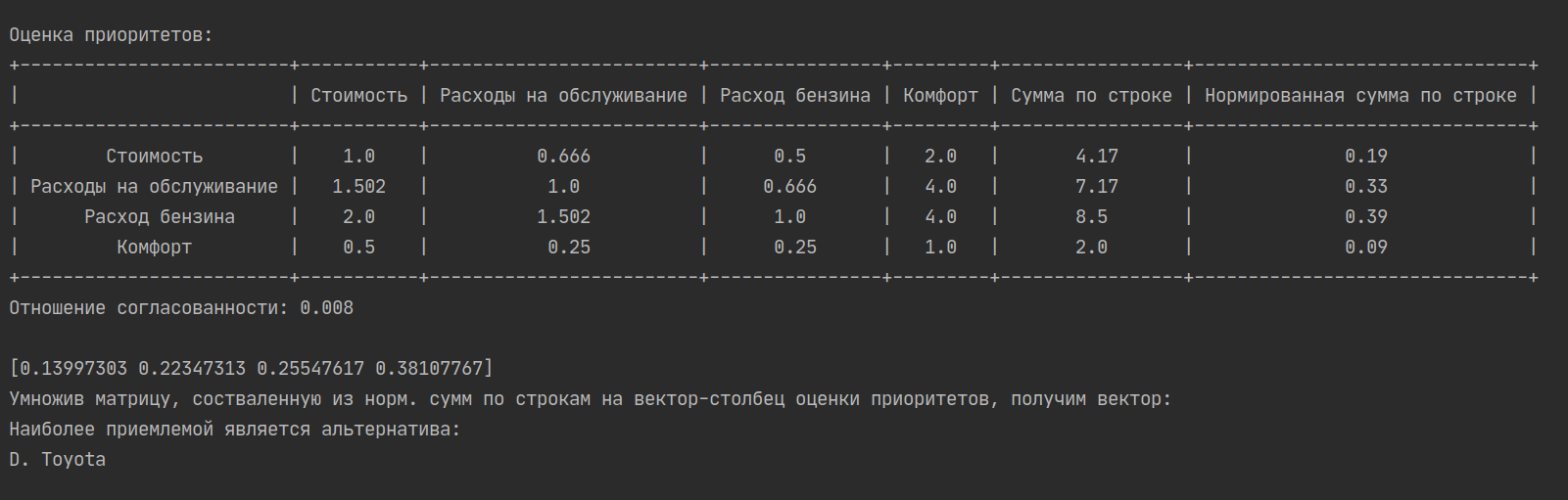
Для каждого критерия составим и нормализуем матрицу попарных альтернатив:

Рисунок 5 Матрицы попарного сравнения альтернатив для каждого критерия

Видим, что для всех матриц отношение согласованности не превосходит 0.1.

Оценка приоритетов критериев:

Умножив матрицу, составленную из нормализированных сумм по строкам на вектор-столбец оценки приоритетов, получим вектор. Оценив вектор, получаем оптимальный вариант *D. Toyota.*

Рисунок 6 Оценка приоритетов критериев

**Вывод:**

В ходе работы была изучена постановка задачи многокритериальной оптимизации, были получены навыки решения задач МКО с помощью различных методов.   
По результатам эксперимента можно увидеть, что не все способы решения дали одинаковое решение. Так результатом метода главного критерия, метода множества Парето, метода анализа иерархий является Toyota, а в методе взвешивания и объединения критериев наилучшая альтернатива – Honda.

В данной работе были рассмотрены методы решения задач, которые могут быть использованы при выборе различных альтернатив по ряду критериев, что может помочь в самых различных областях и ситуациях: выбор автомобиля, материала для строительства, места отдыха и так далее.

**Приложение А:**

*Код программы:*

***1) input\_data.json***

{ "task\_name": "Покупка автомобиля", "alternative\_names": ["A. Suzuki", "B. Mitsubishi", "C. Honda", "D. Toyota"], "criteria\_names": ["Стоимость", "Расходы на обслуживание", "Расход бензина", "Комфорт"], "criterias\_weight": [4, 6, 8, 2], "criterias\_direction": ["min", "min" ,"min" , "max"], "alternative\_matrix": [ [8, 5, 9, 2], [6, 9, 3, 7], [5, 2, 7, 3], [2, 5, 2, 8] ]}

***2) main.py***

# Copyright 2020 Alexey Alexandrov <sks2311211@yandex.ru>*"""**Лабораторная работа № 5**Решение многокритериальной оптимизации.**Цель работы: Изучить постановку задачи МКО;**овдладеть навыками решения задач МКО с помощью различных методов,**осуществить сравнительный анализ резульатов, полученных при помощи разных методов.**Вариант 1.**"""*from multicriteria import \*if \_\_name\_\_ == '\_\_main\_\_': print("Ход работы:") multcrit = Multicriteria("input\_data.json") print(multcrit.OutWeight()) multcrit.MainCriteriaMethod() multcrit.ParetoMethod() multcrit.WeighAndCombineMethod() multcrit.HierarchiesAnalysisMethod()

***2) multicriteria.py***

# Copyright 2020 Alexey Alexandrov <sks2311211@yandex.ru>import jsonimport numpy as npfrom prettytable import PrettyTableimport matplotlib.pyplot as pltfrom scipy.spatial import distanceCRITERIA\_COUNT = 4MAIN\_CRITERIA\_INDEX = 2 # Главный критерий -- расход бензина.MAIN\_CRITERIA\_LIMITS = [0.2, 0.3, 0, 0.5] # Минимально допустимые доли критериев.MAX\_WEIGHT = 10SECOND\_CRITERIA\_INDEX = 1THIRD\_CRITERIA\_INDEX = 2COST\_FILL\_LIST = [0.25, 0.2, 0.125, 0.8, 0.5, 0.625]EXPENSES\_FILL\_LIST = [5, 2, 1, 0.125, 0.2, 2]CONSUMPTION\_FILL\_LIST = [0.143, 0.333, 0.125, 2, 0.5, 0.333]COMFORT\_FILL\_LIST = [0.286, 0.666, 0.25, 2.333, 0.875, 0.375]CRITERIA\_FILL\_LIST = [0.666, 0.5, 2, 0.666, 4, 4]class Multicriteria: *"""Базовый класс задачи многокритериальной оптимизации"""*def \_\_init\_\_(self, path\_to\_file): # Парсим JSON-файл с входными данными with open(path\_to\_file, "r") as read\_file: json\_data = json.load(read\_file) # Задача. self.task\_name\_ = json\_data["task\_name"] # Альтернативы. self.alternative\_names\_ = list(json\_data["alternative\_names"]) # Критерии. self.criteria\_names\_ = list(json\_data["criteria\_names"]) # Вектор весов критериев. self.criteria\_weight\_ = np.array(json\_data["criterias\_weight"]) # Нормализованный вектор весов критериев. self.normalized\_weight\_ = self.NormingVector(self.criteria\_weight\_) # Матрица А оценок для альтернатив. self.json\_matrix\_ = np.array(json\_data["alternative\_matrix"], dtype=np.float64) self.alternative\_matrix\_ = self.AlternativeMatrix(np.array(json\_data["alternative\_matrix"], dtype=np.float64), list(json\_data["criterias\_direction"])) # Нормализованная матрица. self.normalized\_matrix\_ = self.NormalizeMatrix(self.alternative\_matrix\_) def NormingVector(self, vector): *"""Нормирует вектор."""*normalize\_weight = vector.copy() weight\_sum = np.sum(normalize\_weight) normalize\_weight = normalize\_weight / weight\_sum return normalize\_weight def AlternativeMatrix(self, alternative\_matrix, criterias\_direction): *"""Своидит все криитерии к максимизации."""*for j in range(len(criterias\_direction)): if criterias\_direction[j] == "min": for i in range(alternative\_matrix.shape[0]): alternative\_matrix[i][j] = MAX\_WEIGHT - alternative\_matrix[i][j] + 1 return alternative\_matrix def NormalizeMatrix(self, matrix): *"""Нормализует матрицу."""*normalized\_matrix = matrix.copy() minimums = normalized\_matrix.min(axis=0) maximums = normalized\_matrix.max(axis=0) for j in range(normalized\_matrix.shape[1]): if j != MAIN\_CRITERIA\_INDEX: for i in range(matrix.shape[0]): normalized\_matrix[i][j] = (normalized\_matrix[i][j] - minimums[j]) / ( maximums[j] - minimums[j]) return normalized\_matrix def OutMatrix(self, matrix): *"""Выводит матрицу альтернатив."""*table = PrettyTable() table.field\_names = ["Альтернативы"] + self.criteria\_names\_ for i in range(len(self.alternative\_names\_)): new\_row = [self.alternative\_names\_[i]] for j in range(len(self.criteria\_names\_)): new\_row.append(round(matrix[i][j], 2)) table.add\_row(new\_row) return table def OutWeight(self): *"""Выводит вектор весов критериев."""*out = "Составляем веткор весов критериев, используя шкалу 1-10:\n" table = PrettyTable() table.field\_names = self.criteria\_names\_ table.add\_row(self.criteria\_weight\_) out += table.\_\_str\_\_() out += "\nНормализовав, получим вектор " + self.normalized\_weight\_.\_\_str\_\_() return out def MainCriteriaMethod(self): *"""Решение методом главного критерия."""*print("\n1) Метод замены критериев ограничениями (метод главного критерия).\n" "Составим матрицу оценок альтернатив.") print(self.OutMatrix(self.json\_matrix\_)) matrix = self.normalized\_matrix\_.copy() maximums = matrix.max(axis=0) print("Ограничения:") for j in range(len(self.criteria\_names\_)): if j != MAIN\_CRITERIA\_INDEX: print(f"{self.criteria\_names\_[j]} не менее {MAIN\_CRITERIA\_LIMITS[j] \* maximums[j]}") print(f"\nПроведём нормирование матрицы:\n{self.OutMatrix(self.NormalizeMatrix(self.json\_matrix\_))}") constraints = [] for j in range(len(self.criteria\_names\_)): if j == MAIN\_CRITERIA\_INDEX: constraints.append(None) else: constraints.append(MAIN\_CRITERIA\_LIMITS[j] \* maximums[j]) acceptable\_rows = [] for i in range(len(self.alternative\_names\_)): row = matrix[i] if (row < MAIN\_CRITERIA\_LIMITS).any(): continue acceptable\_rows.append(i) if len(acceptable\_rows): print("При заданных ограничениях приемлимыми являются следующие решения:") for i in acceptable\_rows: print(self.alternative\_names\_[i]) max\_alternative = None for i in acceptable\_rows: curr = self.normalized\_matrix\_[i][MAIN\_CRITERIA\_INDEX] if max\_alternative is None or self.normalized\_matrix\_[max\_alternative][MAIN\_CRITERIA\_INDEX] < curr: max\_alternative = i print("Итоговое решение:") print(self.alternative\_names\_[max\_alternative]) else: print("При заданных ограничениях не нашлось приемлимых решений.") def ParetoMethod(self): *"""Решение формированием и сужением множества Парето."""*print(f"\n 2) Формирование и сужение множества Парето. \n" f"Выберем в качестве критериев для данного метода {self.criteria\_names\_[SECOND\_CRITERIA\_INDEX]} и " f"{self.criteria\_names\_[THIRD\_CRITERIA\_INDEX]}.\n" f"{self.criteria\_names\_[SECOND\_CRITERIA\_INDEX]} - по оси X, " f"{self.criteria\_names\_[THIRD\_CRITERIA\_INDEX]} - по оси Y.\n" f"Сформируем множество Парето графическим методом. (см. график)") plt.title("Графическое решение методом сужения множества Парето.") plt.xlabel(f"Критерий: {self.criteria\_names\_[SECOND\_CRITERIA\_INDEX]}") plt.ylabel(f"Критерий: {self.criteria\_names\_[THIRD\_CRITERIA\_INDEX]}") xValues = self.json\_matrix\_[:, SECOND\_CRITERIA\_INDEX] yValues = self.json\_matrix\_[:, THIRD\_CRITERIA\_INDEX] plt.grid() plt.plot(xValues, yValues, "b") euclid\_length = [] for i in range(len(self.json\_matrix\_[:, SECOND\_CRITERIA\_INDEX])): x\_i = self.json\_matrix\_[i, SECOND\_CRITERIA\_INDEX] y\_i = self.json\_matrix\_[i, THIRD\_CRITERIA\_INDEX] plt.plot(x\_i, y\_i, "bo") plt.text(x\_i + 0.1, y\_i, self.alternative\_names\_[i][0]) euclid\_distance = distance.euclidean((x\_i, y\_i), (xValues.min(), yValues.min())) euclid\_length.append(euclid\_distance) plt.plot(xValues.min(), yValues.min(), "rD") plt.text(xValues.min() + 0.1, yValues.min() + 0.1, "Точка утопии") plt.show() plt.savefig("pareto.png") min\_index = min(enumerate(euclid\_length), key=lambda x: x[1])[0] print(f"Исходя из графика можно сказать, что Евклидово расстояние до " f"точки минимально для варианта:\n{self.alternative\_names\_[min\_index]}") def NormalizeByColumns(self, current\_matrix): *"""Нормализует колонки в матрице."""*matrix = current\_matrix.copy() for i in range(len(self.criteria\_names\_)): col\_sum = np.sum(matrix[i]) matrix[i] = matrix[i] / col\_sum return matrix def CriteriaEvaluation(self, y12, y13, y14, y23, y24, y34): *"""Составляет """*table = PrettyTable() table.field\_names = [""] + self.criteria\_names\_ table.add\_row([self.criteria\_names\_[0]] + [0, y12, y13, y14]) table.add\_row([self.criteria\_names\_[1]] + [1 - y12, 0, y23, y24]) table.add\_row([self.criteria\_names\_[2]] + [1 - y13, 1 - y23, 0, y34]) table.add\_row([self.criteria\_names\_[3]] + [1 - y14, 1 - y24, 1 - y34, 0]) return table def WeighAndCombineMethod(self): *"""Решение методом взвешивания и объединения критериев."""*rating\_matrix = self.NormalizeByColumns(self.alternative\_matrix\_) rm = self.NormalizeByColumns(self.json\_matrix\_) print("\n 3) Взвешивание и объединение критериев. \n" f"Составим матрицу рейтингов альтернатив по критериям, используя шкалу 1-10: \n\n " f"{self.OutMatrix(self.json\_matrix\_)} \n\n Нормализуем её: \n" f"{self.OutMatrix(rm)}\n") print("Составим экспертную оценку критериев (по методу попарного сравнения):\n") y12 = 0.5 y13 = 0 y14 = 1 y23 = 0 y24 = 1 y34 = 1 print(self.CriteriaEvaluation(y12, y13, y14, y23, y24, y34)) weight\_vector = np.array([y12 + y13 + y14, y12 + y14, y14 + y24 + y34, 0]) weight\_vector = self.NormingVector(weight\_vector) print(f"alpha = {weight\_vector}") weight\_vector.transpose() combine\_criteria = rating\_matrix.dot(weight\_vector) print(f"Умножив нормализированную матрицу на нормализированный вектор весов критериев, " f"получаем значения объединённого критерия альтернатив:\n{combine\_criteria}") max\_index = None for i in range(len(combine\_criteria) - 1, 0, -1): if max\_index is None or combine\_criteria[i] > combine\_criteria[max\_index]: max\_index = i print(f"Наиболее приемлемой является альтернатива:\n{self.alternative\_names\_[max\_index]}") def PairCompareMatrix(self, fill\_list): *"""Заполянет матрицу попарных сравнений."""*k = 0 pc\_matrix = np.ones((CRITERIA\_COUNT, CRITERIA\_COUNT)) # Заполняем верхний треугольник. for i in range(CRITERIA\_COUNT): for j in range(CRITERIA\_COUNT): if i < j: pc\_matrix[i][j] = round(fill\_list[k], 3) k += 1 k = 0 # Заполняем нижний треугольник. for i in range(CRITERIA\_COUNT): for j in range(CRITERIA\_COUNT): if i < j: pc\_matrix[j][i] = round(1 / fill\_list[k], 3) k += 1 return pc\_matrix def PairCompareTable(self, names, main\_matrix, sum\_col, normalize\_sum\_col): *"""Составляет таблицу с матрицей попарных сравнений"""*table = PrettyTable() table.field\_names = [""] + names + ["Сумма по строке", "Нормированная сумма по строке"] for i in range(len(self.alternative\_names\_)): row = [names[i]] + list(main\_matrix[i]) row.append(round(sum\_col[i], 2)) row.append(round(normalize\_sum\_col[i], 2)) table.add\_row(row) return table def ConsensusDivision(self, main\_matrix, normalize\_sum\_col): *"""Находит отношение согласованности."""*columns\_sum = np.sum(main\_matrix, axis=0) mult\_col = columns\_sum \* normalize\_sum\_col return (np.sum(mult\_col) - CRITERIA\_COUNT) / (CRITERIA\_COUNT - 1) def HierarchiesAnalysisMethod(self): *"""Решение методом анализа иерархий."""*print("\n4) Меотд анализа иерархий.\nСоставим для каждого из критериев матрицу попарного сравнения альтернатив," " нормализуем ее и матрицу из векторов приоритетов альтернатив:\n") fill\_lists = [COST\_FILL\_LIST, EXPENSES\_FILL\_LIST, CONSUMPTION\_FILL\_LIST, COMFORT\_FILL\_LIST] hierarchies\_matrix = None for i in range(len(self.criteria\_names\_)): print(f"• {self.criteria\_names\_[i]}") main\_matrix = self.PairCompareMatrix(fill\_lists[i]) sum\_col = np.sum(main\_matrix, axis=1) normalize\_sum\_col = self.NormingVector(sum\_col) print(self.PairCompareTable(self.alternative\_names\_, main\_matrix, sum\_col, normalize\_sum\_col)) print(f"Отношение согласованности: {round(self.ConsensusDivision(main\_matrix, normalize\_sum\_col), 3)}\n") if hierarchies\_matrix is None: hierarchies\_matrix = normalize\_sum\_col.transpose() else: hierarchies\_matrix = np.c\_[hierarchies\_matrix, normalize\_sum\_col.transpose()] print("Оценка приоритетов:") criteria\_matrix = self.PairCompareMatrix(CRITERIA\_FILL\_LIST) sum\_col = np.sum(criteria\_matrix, axis=1) normalize\_sum\_col = self.NormingVector(sum\_col) print(self.PairCompareTable(self.criteria\_names\_, criteria\_matrix, sum\_col, normalize\_sum\_col)) print(f"Отношение согласованности: {round(self.ConsensusDivision(criteria\_matrix, normalize\_sum\_col), 3)}\n") normalize\_sum\_col.transpose() resulted\_vec = hierarchies\_matrix.dot(normalize\_sum\_col.transpose()) print(resulted\_vec) print("Умножив матрицу, состваленную из норм. сумм по строкам на вектор-столбец оценки приоритетов, " "получим вектор:") max\_index = np.argmax(resulted\_vec) print(f"Наиболее приемлемой является альтернатива:\n{self.alternative\_names\_[max\_index]}")