

Решение задачи “Статистика текста”

1. Импорт библиотек

```
import re
import zipfile
from collections import Counter
```

Для решения задачи используются стандартные библиотеки Python:

- `re` для работы с регулярными выражениями при извлечении слов
- `zipfile` для создания архива с результатами
- `Counter` из `collections` для подсчета частот длин слов при расчете моды

2. Извлечение слов из текста

```
def find_words(content):
    return re.findall(r'[a-zA-Za-яА-ЯёЁ]+', content)
```

Функция `find_words` извлекает все слова из текста по правилам задачи. Регулярное выражение `[a-zA-Za-яА-ЯёЁ]+` находит непрерывные последовательности буквенных символов (латинских и кириллических, включая ё), игнорируя все остальные символы как разделители.

3. Чтение файла и извлечение слов

```
with open('text.txt', 'r', encoding='utf-8') as file:
    content = file.read()
```

```
words = find_words(content)
```

Считываем содержимое входного файла `text.txt` в кодировке UTF-8 и передаем его в функцию извлечения слов. Результат сохраняется в список `words`.

4. Сохранение извлеченных слов

```
with open('words.txt', 'w', encoding='utf-8') as file:
    for word in words:
        file.write(word + '\n')
```

Записываем каждое извлеченное слово в файл `words.txt`, каждое слово на новой строке. Файл сохраняется в кодировке UTF-8 для корректного отображения русских и английских букв.

5. Расчет статистических метрик

```
def count_stats(lengths):
    count = len(lengths)
    if count == 0:
        return 0.0, 0, 0

    total = sum(lengths)
    avg = total / count

    sorted_lengths = sorted(lengths)
    position = (count - 1) // 2
    middle = sorted_lengths[position]

    freq = Counter(lengths)
    max_count = max(freq.values())
    candidates = [l for l, c in freq.items() if c == max_count]
    common = min(candidates)

    return avg, middle, common

word_lengths = [len(w) for w in words]
average, middle, common = count_stats(word_lengths)
```

Функция `count_stats` рассчитывает три статистические метрики:

- **Среднее арифметическое:** сумма длин всех слов деленная на их количество
- **Медиана:** значение в середине отсортированного списка длин (нижняя медиана при четном числе элементов)
- **Мода:** наиболее частая длина слова, при равенстве частот выбирается наименьшее значение

Сначала создаем список длин всех слов, затем передаем его в функцию расчета статистики.

6. Сохранение результатов

```
with open('result.txt', 'w', encoding='utf-8') as file:
    file.write(f"{average:.6f}\n")
    file.write(f"{middle}\n")
    file.write(f"{common}\n")
```

Записываем результаты в файл `result.txt` в требуемом формате:

- Первая строка: среднее арифметическое с точностью 6 знаков после запятой
- Вторая строка: медиана (целое число)

- Третья строка: мода (целое число)

7. Создание архива

```
with zipfile.ZipFile('task1.zip', 'w') as archive:  
    archive.write('words.txt')  
    archive.write('result.txt')
```

Создаем ZIP-архив task1.zip и добавляем в него оба результирующих файла: words.txt и result.txt.

Решение задачи “Чистильщик данных”

1. Импорт необходимых библиотек

```
import zipfile
import re
import os
```

Для решения задачи используются стандартные библиотеки Python:

- zipfile для работы с ZIP-архивами
- re для проверки формата чисел с помощью регулярных выражений
- os для удаления временных файлов

2. Подготовка инструментов для проверки данных

```
number_pattern = re.compile(r'^-?(?:0|[1-9]\d*)(?:\.\d+)?$', re.ASCII)
```

```
valid_rows = []
column_sums = [0.0, 0.0, 0.0]
row_count = 0
```

Компилируем регулярное выражение для строгой проверки формата вещественных чисел. Инициализируем списки для хранения корректных строк и сумм по столбцам, а также счетчик строк.

3. Поиск и обработка CSV-файлов в архиве

```
with zipfile.ZipFile('data.zip', 'r') as archive:
    csv_list = [
        name for name in archive.namelist()
        if '/' not in name and '\\' not in name and name.lower().endswith('.csv')
    ]

    for csv_name in csv_list:
        with archive.open(csv_name) as file:
            content = file.read().decode('utf-8')
            lines = content.splitlines()
            for row in lines:
                cleaned_row = row.rstrip('\r')
                if not cleaned_row:
                    continue

                if re.search(r'[ \t]', cleaned_row):
                    continue
```

```

values = cleaned_row.split(',')
if len(values) != 3:
    continue

is_valid = True
for value in values:
    if not number_pattern.fullmatch(value):
        is_valid = False
        break
if not is_valid:
    continue

valid_rows.append(cleaned_row)
numbers = [float(v) for v in values]
for i in range(3):
    column_sums[i] += numbers[i]
row_count += 1

```

Открываем входной архив и находим все CSV-файлы в корневой директории. Для каждого файла:

- Считываем содержимое и разбиваем на строки
- Проверяем каждую строку на соответствие правилам:
- Отсутствие пробелов и табуляций
- Ровно три значения, разделенных запятыми
- Каждое значение соответствует формату вещественного числа
- Сохраняем корректные строки и обновляем суммы по столбцам

4. Вычисление средних значений

```

if row_count == 0:
    averages = [0.0, 0.0, 0.0]
else:
    averages = [
        column_sums[0] / row_count,
        column_sums[1] / row_count,
        column_sums[2] / row_count
    ]

```

Вычисляем среднее арифметическое для каждого столбца по всем корректным строкам. Если корректных строк нет, используем нулевые значения.

5. Сохранение результатов в файлы

```
with open('valid_lines.txt', 'w', encoding='utf-8') as output_file:
    for row in valid_rows:
        output_file.write(row + '\n')

with open('result.txt', 'w', encoding='utf-8') as result_file:
    result_file.write(f"{averages[0]:.6f} {averages[1]:.6f} {averages[2]:.6f}")
```

Записываем все корректные строки в файл `valid_lines.txt` в исходном формате. В файл `result.txt` записываем средние значения для каждого столбца с точностью 6 знаков после запятой.

6. Создание итогового архива

```
with zipfile.ZipFile('task2.zip', 'w', zipfile.ZIP_DEFLATED) as final_archive:
    final_archive.write('valid_lines.txt')
    final_archive.write('result.txt')

os.remove('valid_lines.txt')
os.remove('result.txt')
```

Создаем итоговый архив `task2.zip` и добавляем в него оба результирующих файла. После создания архива удаляем временные файлы.

Решение задачи “Детектор аномалий во временном ряде”

1. Импорт библиотек

```
import math
import zipfile
```

Для решения задачи используются стандартные библиотеки Python:

- `math` для математических вычислений (квадратный корень)
- `zipfile` для создания архива с результатами

2. Чтение данных из файла

```
with open('series.txt', 'r', encoding='utf-8') as file:
    data = [float(line.strip()) for line in file.readlines()]
```

Считываем все значения из файла `series.txt`, преобразуем их в числа с плавающей точкой и сохраняем в список `data`.

3. Вычисление статистик для окон

```
def calculate_window_stats(values, center_index):
    window = values[center_index-5:center_index+6]
    window_size = len(window)

    mean_value = sum(window) / window_size

    squared_diffs = [(x - mean_value) ** 2 for x in window]
    variance = sum(squared_diffs) / window_size
    std_dev = math.sqrt(variance)

    return mean_value, std_dev
```

Функция `calculate_window_stats` вычисляет среднее значение и стандартное отклонение для окна из 11 точек вокруг указанного индекса. Используется популяционная формула для стандартного отклонения.

4. Анализ аномалий и формирование результатов

```
N = len(data)
checks_content = []
anomaly_indices = []

for i in range(5, N-5):
```

```

mu, sigma = calculate_window_stats(data, i)

is_anomaly = 0
if sigma > 1e-9 and abs(data[i] - mu) > 2 * sigma:
    is_anomaly = 1
    anomaly_indices.append(str(i))

checks_content.append(f"{i} {mu:.8f} {sigma:.8f} {is_anomaly}")

```

Для каждого индекса в диапазоне [5, N-6]:

- Вычисляем среднее и стандартное отклонение для окна
- Проверяем условия аномалии: ненулевое стандартное отклонение и отклонение больше 2σ
- Формируем строки для файла checks.txt и собираем индексы аномалий

5. Запись результатов в файлы

```

with open('checks.txt', 'w', encoding='utf-8') as file:
    file.write('\n'.join(checks_content))

with open('result.txt', 'w', encoding='utf-8') as file:
    file.write('\n'.join(anomaly_indices))

```

Сохраняем результаты в два файла:

- checks.txt содержит все вычисленные значения для каждого индекса
- result.txt содержит только индексы обнаруженных аномалий

6. Создание архива

```

with zipfile.ZipFile('task3.zip', 'w') as archive:
    archive.write('checks.txt')
    archive.write('result.txt')

```

Создаем ZIP-архив task3.zip и добавляем в него оба результирующих файла. Архив автоматически закрывается после завершения операции.

7. Сохранение результатов

```

with open('similarities.txt', 'w') as file:
    for user_id, value in similarities.items():
        file.write(f"{user_id} {value:.8f}\n")

with open('result.txt', 'w') as file:

```

```
file.write(f"{recommended}\n")

with zipfile.ZipFile('task4.zip', 'w') as archive:
    archive.write('similarities.txt')
    archive.write('result.txt')
```

Сохраняем результаты напрямую в файлы текущей директории:

- `similarities.txt` содержит пары “ID пользователя — схожесть” с точностью 8 знаков после запятой
- `result.txt` содержит индекс рекомендуемого товара
- Оба файла добавляются в корень архива `task4.zip`

Решение задачи “Анализ n-грамм”

1. Импорт необходимых библиотек

```
import os
import re
import zipfile
from collections import Counter
import tempfile
```

Для решения задачи используются стандартные библиотеки Python:

- `os` для работы с файловой системой
- `re` для очистки текста с помощью регулярных выражений
- `zipfile` для работы с архивами
- `Counter` для подсчета частот n-грамм
- `tempfile` для временного хранения распакованных файлов

2. Чтение и сортировка файлов из архива

```
with tempfile.TemporaryDirectory() as temp_folder:
    with zipfile.ZipFile('corpus.zip', 'r') as archive:
        archive.extractall(temp_folder)

    text_files = []
    for root, _, files in os.walk(temp_folder):
        for name in files:
            if name.lower().endswith('.txt'):
                path = os.path.join(root, name)
                text_files.append(path)

    text_files.sort(key=lambda x: os.path.basename(x).lower())
```

Распаковываем архив во временную папку, находим все текстовые файлы (независимо от регистра расширения) и сортируем их в алфавитном порядке имен файлов, игнорируя регистр.

3. Объединение содержимого файлов

```
full_text = ""
for file_path in text_files:
    with open(file_path, 'r', encoding='utf-8') as f:
        full_text += f.read()
```

Читаем содержимое каждого файла в отсортированном порядке и объединяем в одну строку. Используем кодировку UTF-8 для корректной работы с русскими и английскими символами.

4. Очистка текста

```
clean_text = re.sub(r'^a-zA-Za-яА-ЯёЁ', '', full_text)
```

Удаляем все символы, кроме букв: латинских (a-z, A-Z), кириллических (а-я, А-Я) и символов ё, Ё. Результат - непрерывная строка только из букв без пробелов и знаков препинания.

5. Подсчет униграмм и биграмм

```
single_chars = list(clean_text)
char_pairs = [clean_text[i:i+2] for i in range(len(clean_text) - 1)]
```

```
single_counts = Counter(single_chars)
pair_counts = Counter(char_pairs)
```

Создаем списки униграмм (отдельных символов) и биграмм (пар соседних символов). Используем Counter для подсчета частоты каждой n-граммы в очищенном тексте.

6. Сортировка и выбор топ-10

```
top_single = sorted(single_counts.items(), key=lambda x: (-x[1], x[0]))[:10]
top_pairs = sorted(pair_counts.items(), key=lambda x: (-x[1], x[0]))[:10]
```

Сортируем n-граммы по двум критериям:

1. По убыванию частоты (основной критерий)
2. По возрастанию ASCII-кода символов (вторичный критерий при равных частотах)

Выбираем первые 10 элементов из каждого отсортированного списка.

7. Сохранение результатов

```
with open('cleaned.txt', 'w', encoding='utf-8') as f:
    f.write(clean_text)
```

```
with open('result.txt', 'w', encoding='utf-8') as f:
    for gram, count in top_single:
        f.write(f"{gram}\t{count}\n")
    for gram, count in top_pairs:
        f.write(f"{gram}\t{count}\n")
```

Записываем очищенный текст в файл `cleaned.txt` (в одну строку без пробелов).
Записываем результаты подсчета в файл `result.txt`: сначала 10 униграмм, затем 10 биграмм, каждая запись в формате “n-граммачастота”.

8. Создание итогового архива

```
with zipfile.ZipFile('task5.zip', 'w', zipfile.ZIP_DEFLATED) as final_archive:  
    final_archive.write('cleaned.txt')  
    final_archive.write('result.txt')
```

```
os.remove('cleaned.txt')  
os.remove('result.txt')
```

Создаем ZIP-архив `task5.zip` с двумя результирующими файлами и удаляем временные файлы после упаковки.

Решение задачи “Обработка изображений фильтрами”

1. Импорт библиотек

```
import numpy as np
from PIL import Image
import math
import zipfile
import os
```

Для решения задачи используются следующие библиотеки:

- numpy для эффективных числовых вычислений с массивами изображений
- PIL (Pillow) для загрузки, обработки и сохранения изображений
- math для математических вычислений при создании гауссова ядра
- zipfile и os для создания итогового архива

2. Создание гауссова фильтра

```
def create_gaussian_filter(filter_size=5, sigma_value=1.0):
    filter_matrix = np.zeros((filter_size, filter_size))
    center_point = filter_size // 2

    for row in range(filter_size):
        for col in range(filter_size):
            dx = row - center_point
            dy = col - center_point
            filter_matrix[row, col] = (1 / (2 * math.pi * sigma_value**2)) * math.exp(-(dx**2 + dy**2) / (2 * sigma_value**2))

    normalized_filter = filter_matrix / np.sum(filter_matrix)
    return normalized_filter
```

Эта функция создает гауссово ядро фильтра:

- Генерирует матрицу 5×5 со значениями гауссова распределения
- Использует математическую формулу гауссова распределения
- Нормализует ядро так, чтобы сумма всех значений равнялась точно 1.0
- Обеспечивает правильное взвешивание для эффекта размытия

3. Свертка изображения с обработкой границ

```
def convolve_image(input_image, filter_kernel):
    image_height, image_width = input_image.shape
    kernel_size = filter_kernel.shape[0]
    padding_size = kernel_size // 2

    padded_image = np.pad(input_image, padding_size, mode='reflect')
    output_image = np.zeros_like(input_image, dtype=np.float32)

    for y in range(image_height):
        for x in range(image_width):
            region = padded_image[y:y+kernel_size, x:x+kernel_size]
            output_image[y, x] = np.sum(region * filter_kernel)

    return output_image
```

Эта функция применяет свертку к изображению: - Обработывает пиксели на границах с использованием отражения - Извлекает области изображения, соответствующие размеру ядра - Вычисляет взвешенную сумму с использованием гауссова ядра - Возвращает размытое изображение с точностью с плавающей точкой

4. Функция сохранения ядра

```
def save_filter_kernel(kernel_matrix, output_file):
    with open(output_file, 'w') as file_handle:
        for row_index in range(kernel_matrix.shape[0]):
            row_values = [f"{value:.6f}" for value in kernel_matrix[row_index]]
            file_handle.write(" ".join(row_values) + "\n")
```

Сохраняет гауссово ядро в текстовый файл:

- Форматирует каждое значение ровно с 6 знаками после запятой
- Записывает 5 строк с 5 значениями в каждой, разделенными пробелами
- Создает требуемый файл kernel.txt в правильном формате

5. Основной конвейер обработки изображений

```
def execute_image_processing():
    source_image = Image.open('image.png').convert('L')
    image_data = np.array(source_image, dtype=np.float32)
```

```

gaussian_filter = create_gaussian_filter(5, 1.0)
save_filter_kernel(gaussian_filter, 'kernel.txt')

blurred_data = convolve_image(image_data, gaussian_filter)
blurred_image = Image.fromarray(np.clip(blurred_data, 0, 255).astype(np.uint
8))
blurred_image.save('blurred.png')

sharpening_factor = 1.5
detail_mask = image_data - blurred_data
sharpened_data = image_data + sharpening_factor * detail_mask

sharpened_image = Image.fromarray(np.clip(sharpened_data, 0, 255).astype(np.
uint8))
sharpened_image.save('sharpened.png')

inverted_data = 255 - sharpened_data
final_image = Image.fromarray(np.clip(inverted_data, 0, 255).astype(np.uint8
))
final_image.save('result.png')

archive_files = ['kernel.txt', 'blurred.png', 'sharpened.png', 'result.png']
with zipfile.ZipFile('task6.zip', 'w') as archive:
    for file_name in archive_files:
        if os.path.exists(file_name):
            archive.write(file_name)

return final_image

```

Полный конвейер обработки:

- Загружает полутоновое входное изображение
- Вычисляет и сохраняет гауссово ядро
- Применяет гауссово размытие и сохраняет промежуточный результат
- Применяет нерезкое маскирование с коэффициентом $\alpha=1.5$ для повышения четкости
- Выполняет инверсию изображения ($255 - \text{значение}$ после повышения четкости)
- Ограничивает все значения пикселей диапазоном $[0, 255]$ и конвертирует в 8-битный формат
- Создает итоговый ZIP-архив со всеми требуемыми файлами

6. Выполнение

```
final_result = execute_image_processing()
```

Запускает весь конвейер обработки изображений и создает итоговый результат вместе со всеми необходимыми файлами вывода.

Решение задачи “Матричное представление свёртки”

1. Импорт необходимых библиотек

```
import numpy as np
import os
import zipfile
```

Для решения задачи используются библиотеки:

- numpy для работы с матрицами и векторами
- os для проверки существования файлов
- zipfile для создания архива с результатами

2. Построение матрицы свёртки

```
def create_convolution_matrix(filter_weights, step_size, image_dimension=8):
    filter_height, filter_width = filter_weights.shape
    output_height = (image_dimension - filter_height) // step_size + 1
    output_width = (image_dimension - filter_width) // step_size + 1

    total_output_pixels = output_height * output_width
    total_input_pixels = image_dimension * image_dimension

    convolution_matrix = np.zeros((total_output_pixels, total_input_pixels))

    for output_row in range(output_height):
        for output_col in range(output_width):
            matrix_row = output_row * output_width + output_col
            start_row = output_row * step_size
            start_col = output_col * step_size

            for filter_row in range(filter_height):
                for filter_col in range(filter_width):
                    input_row = start_row + filter_row
                    input_col = start_col + filter_col
                    matrix_col = input_row * image_dimension + input_col
                    convolution_matrix[matrix_row, matrix_col] = filter_weights[
filter_row, filter_col]

    return convolution_matrix
```

Функция `create_convolution_matrix` строит матрицу свёртки по заданным параметрам:

- Вычисляет размеры выходного изображения по формуле

- Создаёт нулевую матрицу нужного размера
- Для каждой позиции выходного изображения определяет соответствующее окно во входном изображении
- Размещает веса ядра в соответствующих позициях матрицы

3. Чтение ядер из файлов

```
first_kernel = np.loadtxt('kernel1.txt')
second_kernel = np.loadtxt('kernel2.txt')
third_kernel = np.loadtxt('kernel3.txt')
```

Считываем три ядра свёртки из файлов. Каждое ядро хранится в отдельном файле в текстовом формате.

4. Построение матриц для каждого ядра

```
matrix_for_first = create_convolution_matrix(first_kernel, 1)
matrix_for_second = create_convolution_matrix(second_kernel, 2)
matrix_for_third = create_convolution_matrix(third_kernel, 2)
```

Строим три матрицы свёртки:

- Для первого ядра (2×2) с шагом 1
- Для второго ядра (3×3) с шагом 2
- Для третьего ядра (4×4) с шагом 2

5. Сохранение результатов в файлы

```
def write_matrix_to_file(data_matrix, output_filename):
    with open(output_filename, 'w', encoding='utf-8') as output_file:
        for matrix_row in data_matrix:
            formatted_line = ' '.join([f"{value:.3f}" for value in matrix_row])
            output_file.write(formatted_line + '\n')

write_matrix_to_file(matrix_for_first, 'result1.txt')
write_matrix_to_file(matrix_for_second, 'result2.txt')
write_matrix_to_file(matrix_for_third, 'result3.txt')
```

Функция `write_matrix_to_file` сохраняет матрицу в файл:

- Каждая строка матрицы записывается в отдельную строку файла
- Числа форматируются с точностью до 3 знаков после запятой
- Файл сохраняется в кодировке UTF-8

6. Создание архива с результатами

```
archive = zipfile.ZipFile('task7.zip', 'w', zipfile.ZIP_DEFLATED)
archive.write('result1.txt')
archive.write('result2.txt')
archive.write('result3.txt')
archive.close()
```

Создаём ZIP-архив task7.zip и добавляем в него все три файла с результатами. Архив использует сжатие для уменьшения размера файлов.

7. Основной код выполнения

```
if not (os.path.exists('kernel1.txt') and os.path.exists('kernel2.txt') and os.p
ath.exists('kernel3.txt')):
    kernel_one = np.array([[1.0, 0.5], [0.0, 0.0]])
    kernel_two = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]])
    kernel_three = np.array([[0.01, 0.02, 0.03, 0.04], [0.05, 0.06, 0.07, 0.08],
                             [0.09, 0.10, 0.11, 0.12], [0.13, 0.14, 0.15, 0.16]])
    np.savetxt('kernel1.txt', kernel_one, fmt='%.3f')
    np.savetxt('kernel2.txt', kernel_two, fmt='%.3f')
    np.savetxt('kernel3.txt', kernel_three, fmt='%.3f')

first_kernel = np.loadtxt('kernel1.txt')
second_kernel = np.loadtxt('kernel2.txt')
third_kernel = np.loadtxt('kernel3.txt')

matrix_for_first = create_convolution_matrix(first_kernel, 1)
matrix_for_second = create_convolution_matrix(second_kernel, 2)
matrix_for_third = create_convolution_matrix(third_kernel, 2)

write_matrix_to_file(matrix_for_first, 'result1.txt')
write_matrix_to_file(matrix_for_second, 'result2.txt')
write_matrix_to_file(matrix_for_third, 'result3.txt')

archive = zipfile.ZipFile('task7.zip', 'w', zipfile.ZIP_DEFLATED)
archive.write('result1.txt')
archive.write('result2.txt')
archive.write('result3.txt')
archive.close()
```

Основной код:

- Проверяет наличие файлов с ядрами, при их отсутствии создаёт тестовые ядра
- Считывает ядра из файлов
- Строит матрицы свёртки для каждого ядра
- Сохраняет результаты в файлы
- Создаёт архив с результатами

Решение задачи “Оптимальная политика в решётке”

1. Чтение сетки из файла

```
import os
import zipfile

def load_grid(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        return [list(line.strip()) for line in f]
```

Функция `load_grid` считывает сетку из файла `grid.txt`, сохраняя каждую строку как список символов. Это позволяет легко обращаться к отдельным клеткам по координатам.

2. Поиск координат ключевых объектов

```
def find_objects(grid):
    objects = {'S': [], 'T': [], 'X': []}
    for i, row in enumerate(grid):
        for j, cell in enumerate(row):
            if cell in objects:
                objects[cell].append((i, j))
    return objects
```

Функция `find_objects` проходит по всей сетке и сохраняет координаты стартовой позиции S, сокровища T и ловушек X в соответствующие списки словаря.

3. Движение по сетке

```
def next_position(row, col, direction, height, width):
    new_row, new_col = row, col

    if direction == 'up':
        new_row = max(0, row - 1)
    elif direction == 'down':
        new_row = min(height - 1, row + 1)
    elif direction == 'left':
        new_col = max(0, col - 1)
    elif direction == 'right':
        new_col = min(width - 1, col + 1)

    return new_row, new_col
```

Функция `next_position` вычисляет новую позицию агента при движении в заданном направлении. Если движение выходит за границы сетки, агент остается на текущей позиции.

4. Расчет награды за переход

```
def get_reward(cell):
    if cell == 'T':
        return 10.0
    elif cell == 'X':
        return -5.0
    else:
        return -0.1
```

Функция `get_reward` определяет награду за переход в указанную клетку: +10 за сокровище, -5 за ловушку, -0.1 за все остальные переходы.

5. Алгоритм Value Iteration

```
def value_iteration(grid, gamma=0.9, steps=100):
    height = len(grid)
    width = len(grid[0])
    directions = ['up', 'down', 'left', 'right']
    values = [[0.0] * width for _ in range(height)]
    first_step_values = None

    for step in range(steps):
        new_values = [[0.0] * width for _ in range(height)]
        for i in range(height):
            for j in range(width):
                if grid[i][j] in ['T', 'X']:
                    new_values[i][j] = 0.0
                    continue

                best_value = -float('inf')
                for direction in directions:
                    ni, nj = next_position(i, j, direction, height, width)
                    reward = get_reward(grid[ni][nj])
                    value = reward + gamma * values[ni][nj]
                    if value > best_value:
                        best_value = value
                new_values[i][j] = best_value

        if step == 0:
            first_step_values = [row[:] for row in new_values]

        values = new_values

    return values, first_step_values
```

Алгоритм Value Iteration выполняет 100 итераций для вычисления оптимальной ценности каждого состояния. На каждой итерации:

- Для терминальных состояний (T, X) значение остается равным 0
- Для остальных состояний вычисляется максимальная ценность по всем возможным действиям
- После первой итерации сохраняются значения для файла v1.txt

6. Генерация выходных файлов

```
grid = load_grid('grid.txt')
objects = find_objects(grid)

with open('map.txt', 'w', encoding='utf-8') as f:
    for obj_type in ['S', 'T', 'X']:
        for pos in objects[obj_type]:
            f.write(f"{obj_type} {pos[0]} {pos[1]}\n")

final_values, first_values = value_iteration(grid)

with open('v1.txt', 'w', encoding='utf-8') as f:
    for row in first_values:
        f.write(' '.join(f"{x:.1f}" for x in row) + '\n')

start_pos = objects['S'][0]
start_value = final_values[start_pos[0]][start_pos[1]]

with open('result.txt', 'w', encoding='utf-8') as f:
    f.write(f"{start_value:.6f}")
```

Программа выполняет следующие шаги:

1. Считывает сетку из файла grid.txt
2. Находит координаты всех ключевых объектов
3. Записывает координаты в файл map.txt в требуемом формате
4. Запускает алгоритм Value Iteration для вычисления ценностей состояний
5. Сохраняет значения после первой итерации в файл v1.txt с точностью до 1 знака
6. Находит значение стартовой позиции после 100 итераций и записывает его в result.txt с точностью до 6 знаков

7. Создание архива с результатами

```
with zipfile.ZipFile('task8.zip', 'w') as archive:
    archive.write('map.txt')
```

```
archive.write('v1.txt')  
archive.write('result.txt')
```

На завершающем этапе программа создает ZIP-архив task8.zip и добавляет в него все три сгенерированных файла.