МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

ОТЧЕТ
по производственной практике

по теме:
АЛГОРИТМЫ СЖАТИЯ С ПОТЕРЯМИ

Студент:
Группа № 5140901/31501 *Д.С. Кузик*

Руководитель:
доцент *К.К. Семенов*

Санкт-Петербург
2024

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Задача 9.

Выполнить анализ алгоритмов сжатия с потерями KRLE (K-Run Length Encoding), LTC (Lightweight Coding) и метрологического JPEG. Выполнить описание данных алгоритмов. Реализовать данные алгоритмы в пакете Matlab и на языке Python. Выполнить тестирование. Выполнить сравнительный анализ.

2 ОПИСАНИЕ АЛГОРИТМОВ

2.1 K-RLE

В контексте использования технологии беспроводной сенсорной сети (WSN) для мониторинга окружающей среды двумя основными элементарными функциями WSN являются сбор и передача данных. Однако передача/прием данных требует больших затрат энергии. Чтобы снизить энергопотребление, связанное с передачей, используется сжатие данных с помощью локальной обработки информации.

Рассмотрим новый алгоритм сжатия данных, основанный на кодировании длины выполнения (RLE), который называется K-RLE.

Идея, лежащая в основе этого нового алгоритма, заключается в следующем: пусть K - число, если элемент данных d, $d+K$ или $d-K$ встречается n раз подряд во входном потоке, мы заменяем эти n вхождений одной парой nd [1].

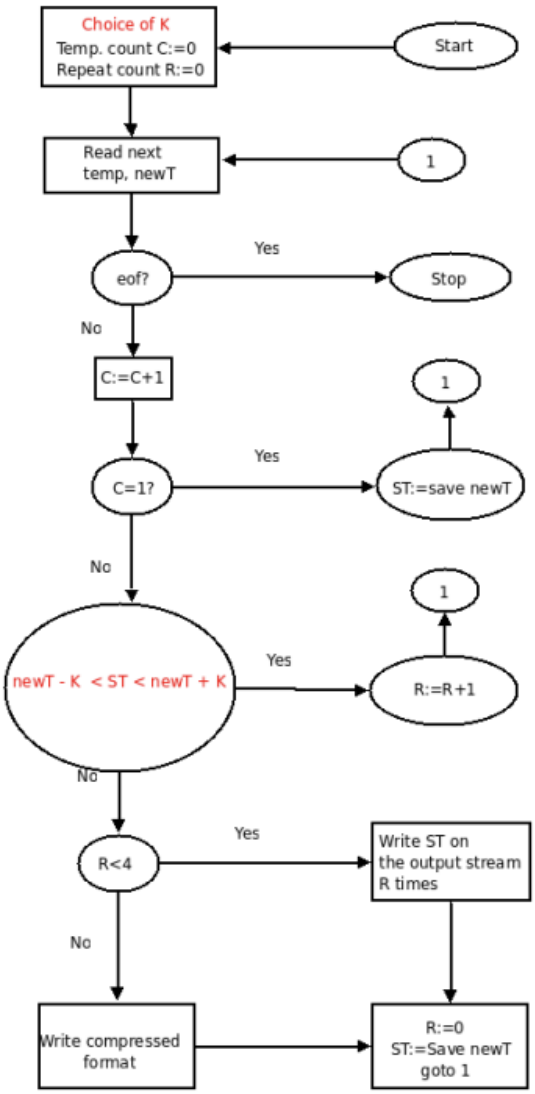

Рисунок 1 — K-RLE Алгоритм

2.2 LTC

Алгоритм LTC (легковесное кодирование), использует линейность во времени для сжатия данных. Наш метод аналогичен кодированию по длине цикла (RLE) в том смысле, что мы пытаемся представить длинную последовательность схожих данных с помощью одного символа. Если при кодировании длины выполнения выполняется поиск строк с повторяющимся символом, мы ищем линейные тренды. Пусть ri = (ti, vi) - точка выборки с допустимой погрешностью e. Исходный набор данных R = r0, r1, ..., rj преобразуется в поток обработанных точек, S = s0, s1, ..., sk, где k ≤ j (обычно k ≤≤ j). Пусть L = l0, l1, ..., lk−1 - набор отрезков, таких, что li - это отрезок прямой, соединяющий две точки si и si+1. Кусочно-непрерывная функция,

определяемая отрезками из L, аппроксимирует R таким образом, что ни одна точка в R не находится на расстоянии более e от ближайшего отрезка прямой в L по вертикали [2].

Алгоритм:

1. Инициализация: Получаем первую точку данных, сохраняем в z. Получаем следующую точку данных (t2, v2), используем ее для инициализации пределов UL (для UL задано значение (t2, v2 + e)) и LL (для LL задано значение (t2, v2 − e)).
2. Вычислите верхнюю линию, которая будет линией, соединяющей z и UL.
3. Вычислите нижнюю линию, которая будет линией, соединяющей z и LL.
4. Получите следующую точку данных. Преобразуйте точку в вертикальный сегмент, используя поле e. Пусть ul - самая высокая точка отрезка. Пусть ll - самая низкая точка отрезка.
5. Если верхняя линия находится ниже ll или нижняя линия находится выше ul, перейдите к пункту 9, в противном случае переходите к следующему шагу
6. Если верхняя строка выше ul, то установите значение UL равным ul.
7. Если нижняя строка ниже ll, то установите значение LL равным ll.
8. Переход ко второму этапу.
9. Снимите заглушку: выведите z в поток выходных данных.
10. Установите точку z на полпути между UL и LL.
11. Установите значение UL равным ul.
12. Установите значение LL равным lll.
13. Переход ко второму этапу

3 РАЗРАБОТАННЫЙ КОД

Исходный код на GitHub:
https://github.com/Denisqu/masters-practice-sem2

3.1 K-RLE

Представим разработанные алгоритмы кодирования и декодирования K-RLE на языке программирования Python:

```python
from typing import List

def k_rle_code(stream: List[int], K: int) -> List[int]:
    # Функция для кодирования потока чисел с использованием модифицированного RLE алгоритма.
    #
    # Алгоритм:
    # 1. Проходим по входному потоку чисел.
    # 2. Если элемент повторяется (учитывая допустимое отклонение K), увеличиваем счетчик повторов.
    # 3. Если элемент не повторяется, вставляем текущие накопленные повторы в результат.
    # 4. В конце вставляем оставшиеся элементы после завершения цикла.
    #
    # Аргументы:
    # stream – входной поток чисел
    # K – допустимое отклонение для повторяющихся чисел
    #
    # Возвращает:
    # Закодированный поток чисел

    result_stream = []
    count = 0
    repeat_count = 0
    ST = 0

    def insert_func():
        if repeat_count < 4:
            result_stream.extend([ST] * (repeat_count + 1))
        else:
            result_stream.extend([repeat_count, ST])

    for i, newT in enumerate(stream):
        count += 1
        if count == 1:
            ST = newT
            continue
        if (newT + K > ST and newT - K < ST) or newT == ST:
            repeat_count += 1
            continue
        insert_func()
        repeat_count = 0
        ST = newT
    insert_func()

    return result_stream
```

```
44
45      def k_rle_decode(stream: List[int], threshold: int) -> List[int]:
46          # Функция для декодирования потока чисел, закодированного модифицированным RLE алгоритмом.
47          #
48          # Алгоритм:
49          # 1. Проходим по входному закодированному потоку чисел.
50          # 2. Если элемент меньше порога и больше 0, считаем его как количество повторов и добавляем соответствующие элементы в результат.
51          # 3. Если элемент не является счетчиком повторов, просто добавляем его в результат.
52          # 4. Переходим к следующему элементу.
53          #
54          # Аргументы:
55          # stream – входной закодированный поток чисел
56          # threshold – порог для определения счетчиков повторов
57          #
58          # Возвращает:
59          # Декодированный поток чисел
60
61          result_stream = []
62          i = 0
63          while i < len(stream):
64              if stream[i] < threshold and stream[i] > 0:
65                  result_stream.extend([stream[i+1]] * (stream[i] + 1))
66                  i += 2
67              else:
68                  current = stream[i]
69                  result_stream.append(current)
70                  i += 1
71          return result_stream
```

Листинг 3.1 — K-RLE реализация на языке Python

3.2 LTC

Представим разработанный алгоритм кодирования LTC на языке программирования Python:

```
1
2       from typing import List, Tuple
3
4       def ltc_code(data_stream: List[Tuple[int, int]], e: int) -> List[Tuple[int, int]]:
5           # Алгоритм:
6           # 1. Инициализация: получение первой точки данных, сохранение её в z. Получение следующей точки (t2, v2),
7           # использование её для инициализации границ UL (верхняя граница) и LL (нижняя граница).
8           # 2. Вычисление highLine как линии, соединяющей z и UL.
9           # 3. Вычисление lowLine как линии, соединяющей z и LL.
10          # 4. Получение следующей точки данных. Преобразование точки в вертикальный сегмент с использованием погрешности e.
11          # Определение ul как верхней точки сегмента и ll как нижней точки сегмента.
12          # 5. Если highLine находится ниже ll или lowLine находится выше ul, переход к шагу 9, иначе продолжение.
13          # 6. Если highLine выше ul, установка UL в ul.
14          # 7. Если lowLine ниже ll, установка LL в ll.
15          # 8. Переход к шагу 2.
16          # 9. Завершение: вывод z в выходной поток данных.
17          # 10. Установка z как точки, находящейся посередине между UL и LL.
18          # 11. Установка UL как ul.
19          # 12. Установка LL как ll.
20          # 13. Переход к шагу 2.
```

```
21
22          def calculate_line(p1: Tuple[float, float], p2: Tuple[int, int])
23           -> Tuple[int, int]:
24              #Вычисляет коэффициенты прямой, проходящей через две точки (p1 и p2)
25              x1, y1 = p1
26              x2, y2 = p2
27              slope = (y2 - y1) / (x2 - x1)
28              intercept = y1 - slope * x1
29              return slope, intercept
30
31          result_stream = []
32          z = data_stream[0]
33          t2, v2 = data_stream[1]
34          UL = (t2, v2 + e)
35          LL = (t2, v2 - e)
36          i = 2
37
38          while i < len(data_stream):
39              highLine = calculate_line(z, UL)
40              lowLine = calculate_line(z, LL)
41              t, v = data_stream[i]
42              ul = (t, v + e)
43              ll = (t, v - e)
44
45              slope_high, intercept_high = highLine
46              slope_low, intercept_low = lowLine
47
48              if (slope_high * t + intercept_high < ll[1])
49              or (slope_low * t + intercept_low > ul[1]):
50                  result_stream.append(z)
51                  z = ((UL[0] + LL[0]) / 2, (UL[1] + LL[1]) / 2)
52                  UL = ul
53                  LL = ll
54              else:
55                  if slope_high * t + intercept_high > ul[1]:
56                      UL = ul
57                  if slope_low * t + intercept_low < ll[1]:
58                      LL = ll
59
60              i += 1
61
62          result_stream.append(z)
63
64          return result_stream
```

Листинг 3.2 — ltc реализация на языке Python

4 ТЕСТИРОВАНИЕ АЛГОРИТМОВ

Для тестирования разработанных алгоритмов сжатия данных были использованы исторические данные о температуре воздуха в Санкт-Петербурге. Алгоритмы были протестированы на предмет эффективности сжатия и потери данных. Данные о температуре были получены с помощью библиотеки meteostat, которая предоставляет доступ к метеорологическим данным. В данном случае были использованы почасовые данные о температуре воздуха в Санкт-Петербурге за период с 1 июня 2018 года по 28 июня 2018 года.

Исходный код для тестирования алгоритмов:

```
from k_rle import k_rle_code, k_rle_decode
from ltc import ltc_code

from datetime import datetime
from meteostat import Hourly, Point
import numpy as np
import matplotlib.pyplot as plt
import time

def create_temperature_stream(multiplier: int, with_timestamps = False):
    start = datetime(2018, 6, 1)
    end = datetime(2018, 6, 28, 23, 59)
    saint_petersburg = Point(59.9387, 30.3256, 10)
    data = Hourly(saint_petersburg, start, end)
    data = data.fetch()
    stream = []
    if not with_timestamps:
        for temp in data['temp']:
            stream.append(int(temp * 1e7))
    else:
        for i, item in enumerate(data['temp']):
            stream.append( (int(data.index.get_level_values("time")[i].timestamp()),
             item) )
    return stream

def test_k_rle():
    multiplier = 1e7
    source_stream = create_temperature_stream(multiplier)
    i_values = []
    compress_rates = []
    data_losses = []
    for i in range(1, 500 + 1, 10):
        coded_stream = k_rle_code(source_stream, multiplier * i / 100)
        decoded_stream = k_rle_decode(coded_stream, 2048)
        compress_rate = int((1 - (len(coded_stream) / len(source_stream))) * 100)
        data_loss = np.mean( np.array(source_stream) != np.array(decoded_stream) )*100

        i_values.append(i / 100)
        compress_rates.append(compress_rate)
        data_losses.append(data_loss)
```

```
41
42          print('--------------------------------')
43          print(f'i = {i / 100}, compress_rate = {compress_rate}%, loss = {data_loss}%')
44          print('--------------------------------')
45
46      plt.figure(figsize=(10, 5))
47
48      plt.subplot(1, 2, 1)
49      plt.plot(i_values, compress_rates)
50      plt.title('Compression Rate vs i')
51      plt.xlabel('i')
52      plt.ylabel('Compression Rate (%)')
53
54      plt.subplot(1, 2, 2)
55      plt.plot(i_values, data_losses)
56      plt.title('Data Loss vs i')
57      plt.xlabel('i')
58      plt.ylabel('Data Loss (%)')
59
60      plt.tight_layout()
61      plt.show()
62
63  def test_ltc():
64      multiplier = 1e7
65      source_stream = create_temperature_stream(multiplier, with_timestamps=True)
66      i_values = []
67      compress_rates = []
68      for i in range(1, 500 + 1, 10):
69          coded_stream = ltc_code(source_stream, i / 100)
70          compress_rate = int((1 - (len(coded_stream) / len(source_stream))) * 100)
71
72          i_values.append(i / 100)
73          compress_rates.append(compress_rate)
74
75          print('--------------------------------')
76          print(f'i = {i / 100}, compress_rate = {compress_rate}%')
77
78      plt.figure(figsize=(5, 5))
79      plt.subplot(1, 2, 1)
80      plt.plot(i_values, compress_rates)
81      plt.title('Compression Rate vs i')
82      plt.xlabel('i')
83      plt.ylabel('Compression Rate (%)')
84      plt.tight_layout()
85      plt.show()
86
87  if __name__ == '__main__':
88      test_k_rle()
89      test_ltc()
```

Листинг 4.1 — Исходный код для тестирования алгоритмов

5 АНАЛИЗ РЕЗУЛЬТАТОВ И ВЫВОДЫ

Предоставим графики, полученные в ходе тестирования алгоритмов в разеделе № 4.

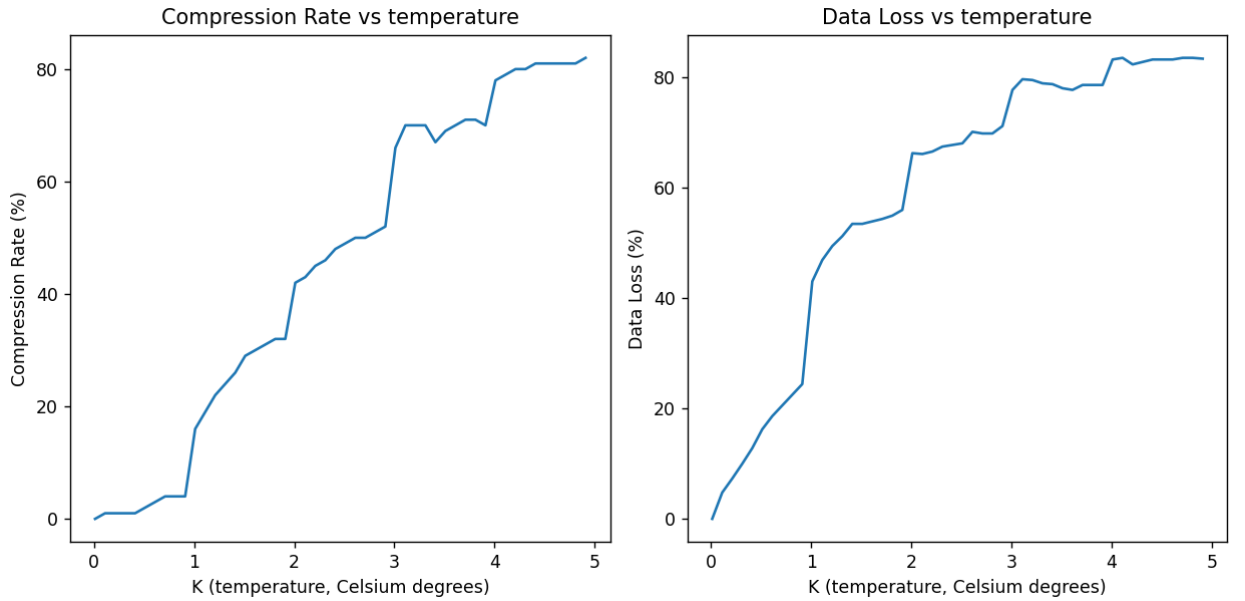

Рисунок 2 — K-RLE результаты тестирования

Алгоритм K-RLE: Результаты показали, что коэффициент сжатия увеличивается с увеличением параметра K, но при этом также возрастает потеря данных. Графики показывают зависимость коэффициента сжатия и потери данных от параметра K.

Несмотря на высокий процент потерь данных, указанный на графике, высокий процент потери данных при сжатии не всегда является проблемой. В некоторых случаях допустимо иметь определенную степень погрешности для достижения значительного уменьшения объема данных, например при сборе метеорологических данных и при оптимизации энергопотребления сети Интернета Вещей.

Алгоритм LTC: Результаты показали, что коэффициент сжатия увеличивается с увеличением параметра ϵ. График показывает зависимость коэффициента сжатия от параметра ϵ.

Графики наглядно демонстрируют компромисс между коэффициентом сжатия и потерей данных при использовании различных параметров для обоих алгоритмов.

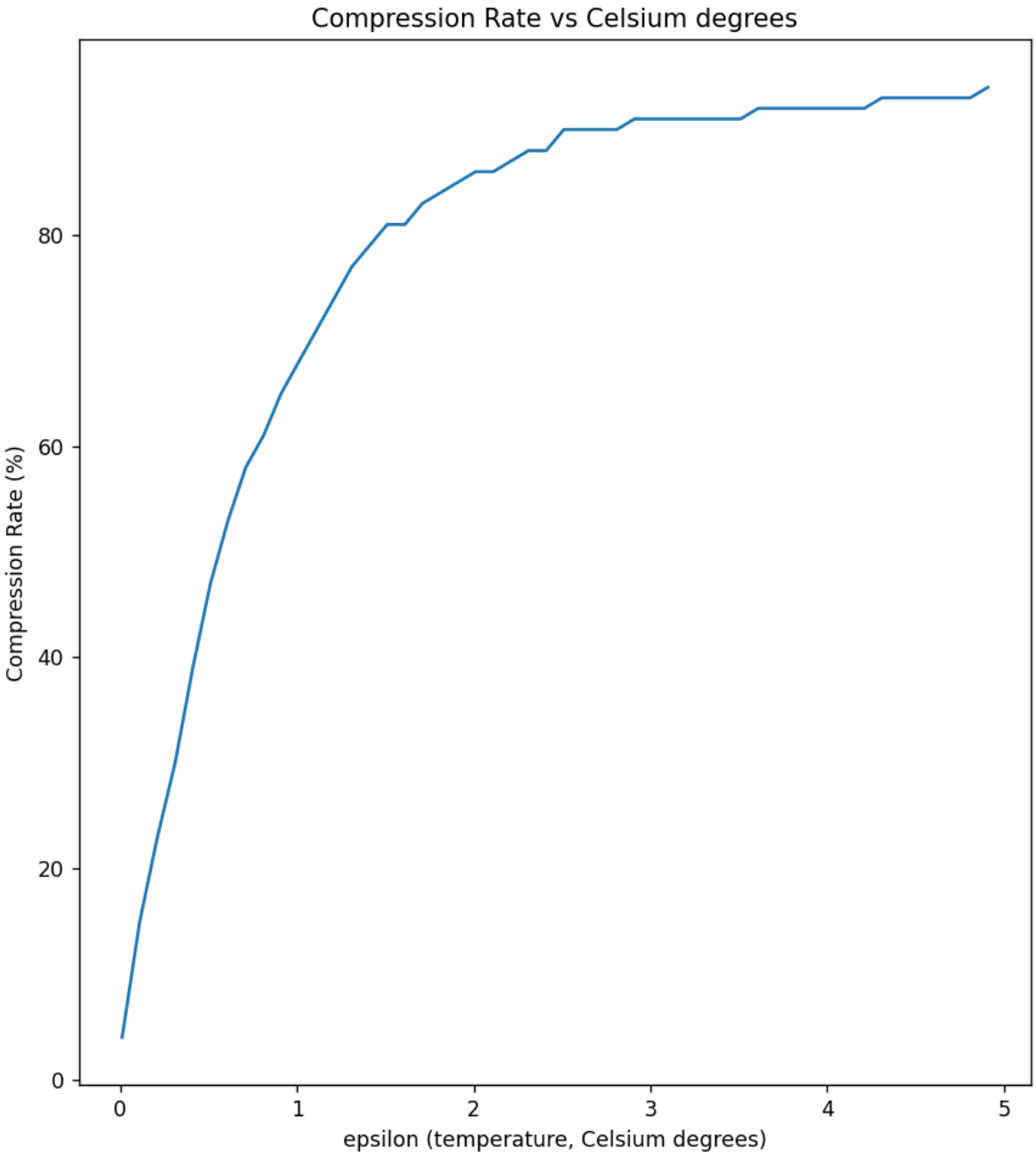

Рисунок 3 — LTC результаты тестрования

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Pamba Capo-Chichi E.*, *Guyennet H.*, *Friedt J.* K-RLE: A new Data Compression Algorithm for Wireless Sensor Network //. — 07/2009. — P. 502–507.
2. *Schoellhammer T.*, *Greenstein B.*, *Osterweil E.*, *Wimbrow M.*, *Estrin D.* Lightweight temporal compression of microclimate datasets [wireless sensor networks] // 29th Annual IEEE International Conference on Local Computer Networks. — 2004. — P. 516–524.