Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики
«Кафедра вычислительной математики и программирования»

Курсовая работа по предмету "Дискретный анализ"

Студент: Шипилов К. Ю.
Преподаватель: Сорокин С. А.
Группа: М8О-303Б-22
Дата: 02.01.2024
Оценка:
Подпись:

Москва 2024г.

Оглавление

Описание задачи

Реализовать алгоритм LZW. Начальный словарь включает малые латинские буквы и специальный символ конца файла (EOF).

На ввод подаются команды двух типов:

- compress <text>: Сжимает текст, состоящий из малых латинских букв, и возвращает последовательность кодов.
- decompress <codes>: Восстанавливает текст по последовательности кодов, полученных в результате сжатия.

Теоретические основы

Алгоритм LZW (Lempel-Ziv-Welch) – это универсальный алгоритм сжатия данных без потерь на основе динамического словаря, в котором содержатся сегменты текста закодированные индексами. Важным свойством словаря является то, что каждый префикс слова из словаря находится в словаре. Это позволяет эффективно реализовать словарь используя trie.

Изначально словарь содержит все символы входного алфавита и пополняется в процессе сжатия или восстановления данных.

При сжатии обрабатывается считанный сегмент текста *w* и следующий за ним символ *a*. Изначально считанный сегмент *w* инициализируется первым символом текста.

- Если *wa* не содержится в словаре, то *w* кодируется значением из словаря, в словарь добавляется код для *wa*, а дальше продолжается обработка для сегмента, состоящего из символа *a* и символа, следующего за ним.
- Если *wa* содержится в словаре, то происходит обработка сегмента *wa* и символа, следующего за ним.

Алгоритм восстановления текста симметричен сжатию.

- Считывается код *c*.
- Записывается сегмент *w*, который соответствует индексу *c* в словаре.
- В словарь добавляется слово *wa*, где *a* – первая буква следующего сегмента.

Сжатие и восстановление может быть реализовано за линейное время, относительно размера входных данных.

Описание алгоритма

При сжатии в качестве словаря используется *std::map*. Это увеличивает асимптотическую сложность алгоритма до O(N * logN), но такой сложности оказалось достаточно для выполнения работы, а также использование std::map вместо std::unordered_map позволило избежать коллизий, которые могли возникнуть при работе с хеш-таблицей.

Словарь заполняется малыми латинскими символами и символом EOF. Для хранения закодированной последовательности сегментов используется *std::vector*. Текущий обрабатываемый сегмент инициализируется пустой строкой и выполняется цикл, в котором перебираются все символы текста. Если в словаре есть код для сегмента с добавленным к нему следующим символом, то к сегменту добавляется этот символ и происходит обработка следующего символа. Если в словаре нет кода для сегмента с добавленным к нему следующим символом, то сегмент кодируется значением из словаря, в словарь добавляется код для текущего сегмента с добавленным к нему следующим символом и происходит обработка нового сегмента, который состоит из следующего символа.

После завершения цикла, если текущий сегмент не пустой, то он кодируется значением из словаря, и в конец закодированного текста добавляется символ EOF.

При восстановлении текста в качестве словаря используется std::vector. Сначала в словарь добавляются все символы алфавита. После этого выполняется обработка кодов в цикле. Если текущий код соответствует символу EOF, то цикл прерывается. Для всех кодов, кроме первого к слову, соответствующему последнему коду, добавленному в словарь, дописывается первая буква слова, соответствующего текущему коду. К восстановленному тексту добавляется

слово соответствующее текущему коду. В конец словаря добавляется новое слово, которое состоит из слова, закодированного текущим кодом, и неизвестной последней буквы. Неизвестная буква добавится на следующем шаге алгоритма.

Исходный код

lzw.cpp

```
#include <iostream>
#include <map>
#include <vector>

std::vector<size_t> compress(const std::string& text) {
  std::map<std::string, size_t> dictionary;
  size_t next_code = 0;
  for (char c = 'a'; c <= 'z'; ++c) {
    dictionary[std::string(1, c)] = next_code++;
  }
  dictionary["EOF"] = next_code++;

  std::vector<size_t> codes;
  std::string substring;
  for (char c : text) {
    if (dictionary.find(substring + c) == dictionary.end()) {
      codes.push_back(dictionary[substring]);
      dictionary[substring + c] = next_code++;
      substring = std::string(1, c);
    } else {
      substring += c;
    }
  }
  if (!substring.empty()) {
    codes.push_back(dictionary[substring]);
  }
  codes.push_back(dictionary["EOF"]);
  return codes;
}

std::string decompress(const std::vector<size_t>& codes) {
```

```
  std::vector<std::string> dictionary;
  for (char c = 'a'; c <= 'z'; ++c) {
    dictionary.push_back(std::string(1, c));
  }
  dictionary.push_back("EOF");

  std::string text;
  for (size_t code : codes) {
    if (dictionary[code] == "EOF") {
      break;
    }
    if (dictionary.size() != 27) {
      dictionary.back() += dictionary[code].front();
    }
    text += dictionary[code];
    dictionary.push_back(dictionary[code]);
  }
  return text;
}

int main() {
  std::string action;
  std::cin >> action;
  if (action == "compress") {
    std::string text;
    std::cin >> text;
    std::vector<size_t> codes = compress(text);
    for (size_t code : codes) {
      std::cout << code << " ";
    }
    std::cout << std::endl;
  } else if (action == "decompress") {
    std::vector<size_t> codes;
    size_t code;
    while (std::cin >> code) {
      codes.push_back(code);
    }
    std::string text = decompress(codes);
    std::cout << text << std::endl;
  }
```

```
}
----------------------------------------------------------
```

Результаты

Test 1

Input	Output
compress abracadabra	0 1 17 0 2 0 3 27 29 26

Test 2

Input	Output
decompress 0 1 17 0 2 0 3 27 29 26	abracadabra

Test 3

Input	Output
compress azazax	0 25 27 0 23 26

Test 4

Input	Output
decompress 0 25 27 0 23 26	azazax

Для проверки эффективности сжатия использовался сгенерированный набор тестов.

- Первый тест содержит 1000 случайных символов

- Второй тест содержит 10000 случайных символов
- Третий тест содержит текст из первого теста, повторенный 20 раз (20000 символов)
- Четвертый тест содержит текст из второго теста, повторенный 10 раз (100000 символов)

Размер исходных данных вычислялся как количество символов, умноженное на 5, так как для кодирования алфавита из 27 символов необходимо 5 бит.

Размер сжатых данных вычислялся как количество кодов, умноженное на двоичный логарифм максимального кода в словаре.

Коэффициент сжатия K вычислялся как отношение размера сжатых данных к размеру исходных, умноженное на 100%.

А эффективность сжатия рассчитывалась по формуле:

Эффективность = 100% - K

Тест	Количество символов	Количество кодов	Количество бит для одного кода	Эффективность сжатия	Время выполнения
1	1000	600	10	-20%	2.0 ms ± 0.4 ms
2	10000	4315	13	-12.19%	8.4 ms ± 0.5 ms
3	20000	5018	13	34.77%	12.5 ms ± 0.5 ms
4	100000	27138	15	18.59%	68.1 ms ± 1.2 ms

Вывод

Алгоритм LZW показывает высокую эффективность для больших текстов с повторяющимися паттернами. Главное преимущество алгоритма — это простота и компактность реализации, а также то, что время выполнения линейно зависит от длины текста. Для более оптимальной работы с большими данными можно улучшить управление памятью, например, используя собственную реализацию словаря, основанную на trie, чтобы эффективнее хранить строки.