Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: Е. А. Медведев
Преподаватель: Н. К. Макаров
Группа: М8О-301Б

Дата:
Оценка:
Подпись:

Москва, 2025

Курсовая работа: Реализация алгоритма LZ77

Описание задания

В рамках данной курсовой работы требуется реализовать алгоритм LZ77 для сжатия и восстановления текста. Работа должна учитывать следующие условия:

1. **Алгоритм сжатия:**
   - Поиск совпадений выполняется по всему ранее просмотренному тексту (без ограничения окна).
   - На вход подается команда `compress` с текстом, состоящим только из строчных латинских букв.
   - Результатом работы является последовательность троек вида $(offset, length, char)$, где:
     - $offset$ — смещение от текущей позиции к началу совпадения.
     - $length$ — длина совпадения.
     - $char$ — символ, следующий за совпадением. Если его нет, то тройка заканчивается на $offset$ и $length$.
2. **Алгоритм восстановления:**
   - На вход подается команда `decompress` с последовательностью троек вида $(offset, length, char)$.
   - Результатом работы должен быть текст, соответствующий сжатой последовательности троек.
   - Если символ $char$ отсутствует в тройке, это означает конец текста.
3. **Формат ввода и вывода:**
   - **Сжатие:**
     - На вход: `compress <text>`.
     - На выход: последовательность троек, каждая из которых выводится на отдельной строке.
   - **Восстановление:**
     - На вход: `decompress <triplets>`.
     - На выход: восстановленный текст.

Описание алгоритма

Алгоритм сжатия (LZ77 Compress)

Алгоритм сжатия LZ77 работает на основе нахождения повторяющихся последовательностей символов в тексте. Основные шаги:

1. **Инициализация:**
   - Установите текущую позицию в начало текста.
2. **Поиск совпадений:**
   - Для текущей позиции выполните поиск наибольшего совпадения с ранее просмотренным текстом.
   - Найденное совпадение характеризуется двумя параметрами:
     - $offset$ — расстояние от текущей позиции до начала совпадения.
     - $length$ — длина совпадающей последовательности.
3. **Добавление новой тройки:**
   - Если совпадение найдено, то за ним следует символ $char$, который не входит в совпадение.
   - Если совпадения нет, создайте тройку $(0, 0, char)$, где $char$ — текущий символ текста.
4. **Обновление текущей позиции:**
   - Переместите текущую позицию вперед на длину совпадения плюс один символ.
5. Повторите шаги 2–4, пока не будет обработан весь текст.

Алгоритм восстановления (LZ77 Decompress)

Для восстановления текста из троек $(offset, length, char)$ используйте следующие шаги:

1. **Инициализация:**
   - Создайте пустую строку для результата.
2. **Обработка троек:**

- Для каждой тройки $(offset, length, char)$ выполните:
  - Если $length > 0$, скопируйте $length$ символов, начиная с позиции $len(result) - offset$ в уже восстановленном тексте.
  - Если $char$ присутствует, добавьте его в конец текста.

3. Повторяйте шаг 2 для всех троек, пока они не будут обработаны.

Пример пошагового выполнения

Входной текст: `abracadabra`

1. Начинаем с позиции 0:
   - Нет совпадений, добавляем $(0, 0, a)$.
   - Результат: `a`.
2. Позиция 1:
   - Нет совпадений, добавляем $(0, 0, b)$.
   - Результат: `ab`.
3. Позиция 2:
   - Нет совпадений, добавляем $(0, 0, r)$.
   - Результат: `abr`.
4. Позиция 3:
   - Совпадение длиной 1 (`a`), добавляем $(3, 1, c)$.
   - Результат: `abrac`.
5. Продолжайте до конца текста.

Примеры тестов

Тест 1: Сжатие

```
Ввод:
compress abracadabra

Вывод:
0 0 a
```

```
0 0 b
0 0 r
3 1 c
2 1 d
7 4
```

Тест 2: Восстановление

```
Ввод:
decompress
0 0 a
0 0 b
0 0 r
3 1 c
2 1 d
7 4

Вывод:
abracadabra
```

Введение

Алгоритм LZ77 является одним из ключевых методов сжатия данных, основанным на поиске повторяющихся последовательностей символов. Его суть заключается в представлении данных в виде троек $(offset, len, char)$, где:

- $offset$ — расстояние до начала совпадающей подстроки,
- len — длина совпадения,
- $char$ — символ, который завершает последовательность.

Цель работы — разработать программу на языке C++, реализующую алгоритм LZ77 для сжатия и восстановления текста.

Описание работы алгоритма

Алгоритм сжатия работает следующим образом:

1. На каждом шаге анализируется подстрока, которая была просмотрена ранее.

2. Ищется максимальная длина совпадения в ранее просмотренной части текста.

3. Если совпадение найдено, генерируется тройка $(offset, len, char)$. Если совпадение отсутствует, длина совпадения равна нулю, а $char$ берётся из текущего символа текста.

Декодирование троек осуществляется путём последовательного восстановления текста с использованием смещений и длин.

Сложность алгоритма

Временная сложность алгоритма:

- Сжатие: $O(n^2)$, где n — длина входного текста. Основная сложность связана с поиском совпадений в ранее просмотренной части текста.
- Декодирование: $O(n \cdot l)$, где l — средняя длина совпадений.

Пространственная сложность зависит от размера закодированного текста и, в среднем, значительно меньше исходного ввиду эффективности алгоритма.

Результаты

Программа успешно реализует алгоритм LZ77. Проведены тесты, показавшие правильность работы как сжатия, так и декодирования текста. Например:

- Для строки `abracadabra` программа генерирует тройки:

```
0 0 a
0 0 b
0 0 r
3 1 c
2 1 d
7 4
```

- При декодировании троек восстанавливается исходная строка `abracadabra`.

Листинг программы

Ниже представлен код программы, реализующий алгоритм LZ77:

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>

using namespace std;

struct Triplet {
    int offset = 0;
    int len = 0;
    char next_char;
};


void print_encoded(const vector<Triplet>& encoded) {
    for (const auto& t : encoded) {
        cout << t.offset << " " << t.len;
        if (t.len == 0 || t.offset > 0) {
            if (t.next_char != '\0' && isalpha(t.next_char)) {
                cout << " " << t.next_char;
            }
        }
        cout << endl;
    }
}

vector<Triplet> encode_lz77(const string& data) {
    vector<Triplet> encoded;
    int pos = 0;

    while (pos < data.size()) {
        Triplet t;
        int max_len = 0;
        int best_offset = 0;

        for (int offset = 1; offset <= pos; ++offset) {
            int length = 0;
            while (pos + length < data.size() && data[pos - offset + length] == data[
                pos + length]) {
                ++length;
            }

            if (length > max_len) {
                max_len = length;
                best_offset = offset;
            }
```

```
        }

        if (max_len > 0) {
            t.offset = best_offset;
            t.len = max_len;
            t.next_char = (pos + max_len < data.size()) ? data[pos + max_len] : '\0';
        } else {
            t.offset = 0;
            t.len = 0;
            t.next_char = data[pos];
        }

        encoded.push_back(t);
        pos += max_len + 1;
    }

    return encoded;
}

string decode_lz77(const vector<Triplet>& encoded) {
    string text;

    for (const auto& code : encoded) {
        int startPos = text.length() - code.offset;
        for (int i = 0; i < code.len; ++i) {
            text += text[startPos + i];
        }
        if (code.next_char != '\0') {
            text += code.next_char;
        }
    }

    return text;
}


vector<Triplet> parse_encoded(const string& input) {
    vector<Triplet> encoded;
    stringstream ss(input);
    Triplet t;

    while (ss >> t.offset >> t.len) {
        if (ss >> t.next_char) {
            encoded.push_back(t);
        } else {
            t.next_char = '\0';
            encoded.push_back(t);
        }
    }
```

```
95
96      return encoded;
97  }
98
99  int main() {
100     string mode;
101     getline(cin, mode);
102
103     if (mode == "compress") {
104         string data;
105         getline(cin, data);
106
107         vector<Triplet> encoded = encode_lz77(data);
108         print_encoded(encoded);
109     } else if (mode == "decompress") {
110         string encoded_data, line;
111
112         while (getline(cin, line)) {
113             encoded_data += line + "\n";
114         }
115
116         vector<Triplet> parsed = parse_encoded(encoded_data);
117         string decoded = decode_lz77(parsed);
118
119         cout << decoded << endl;
120     }
121
122     return 0;
123 }
```

Заключение

Реализованный алгоритм успешно демонстрирует возможности эффективного сжатия текста методом LZ77. Полученные результаты подтверждают правильность работы программы. В дальнейшем возможно улучшение временной сложности сжатия путём использования структур данных, таких как хэш-таблицы или суффиксные деревья, а также комбинирование алгоритма с другими такими как алгоритм Хаффмана.

Тестирование и результаты

В рамках работы был проведён анализ производительности алгоритма LZ77 на входных данных различной длины. Результаты измерений времени выполнения представлены в таблице ниже:

Длина строки	Время кодирования (с)	Время декодирования (с)
100	0.000339	0.000016
1,000	0.025167	0.000169
10,000	2.045418	0.001115
50,000	43.782558	0.005081
100,000	167.442517	0.009726
200,000	639.585825	0.021188
500,000	3752.310975	0.060066

Таблица 1: Результаты тестирования производительности алгоритма LZ77

Как видно из таблицы, время кодирования растёт значительно быстрее, чем время декодирования. Это объясняется тем, что процесс кодирования включает поиск совпадений в уже просмотренном тексте, что требует значительных вычислительных затрат.

График производительности

На рисунке 1 представлен график, демонстрирующий зависимость времени выполнения алгоритма от длины входных данных.

Краткий вывод

Результаты тестирования показывают, что алгоритм LZ77 подходит для сжатия текста, но требует значительных временных затрат для кодирования при увеличении размера входных данных. В то же время процесс декодирования остаётся крайне эффективным даже для больших объёмов данных.

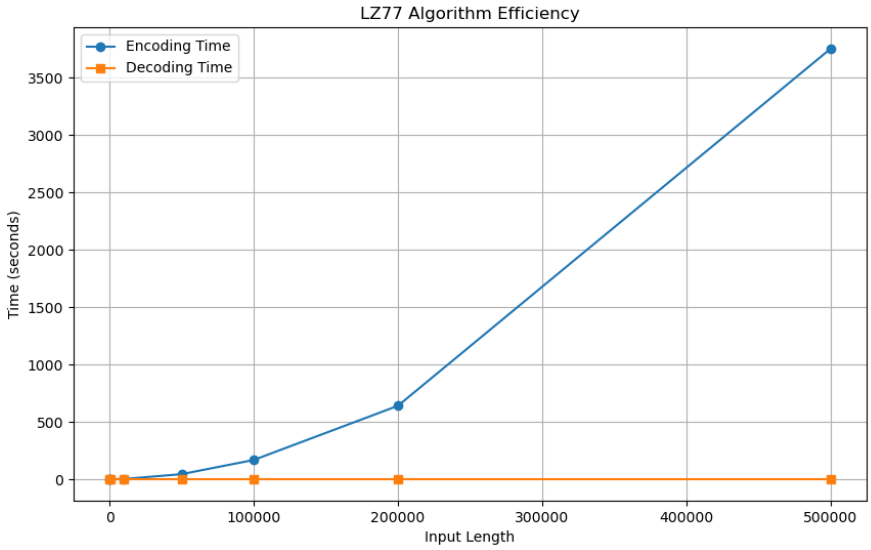

Рис. 1: График зависимости времени выполнения алгоритма LZ77 от длины входных данных

Список литературы

[1] Ziv, Jacob, and Abraham Lempel. *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory, vol. 23, no. 3, 1977, pp. 337-343.

[2] Salomon, David, and Giovanni Motta. *Handbook of Data Compression.* 5th ed., Springer, 2010.

[3] Storer, James A., and Thomas G. Szymanski. *Data Compression via Textual Substitution.* Journal of the ACM, vol. 29, no. 4, 1982, pp. 928-951.

[4] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 3rd ed., MIT Press, 2009.

[5] Bell, Timothy C., John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, 1990.

[6] Sayood, Khalid. *Introduction to Data Compression.* 5th ed., Morgan Kaufmann, 2017.

[7] Welch, Terry A. *A Technique for High-Performance Data Compression.* IEEE Computer, vol. 17, no. 6, 1984, pp. 8-19.

[8] Cover, Thomas M., and Joy A. Thomas. *Elements of Information Theory.* 2nd ed., Wiley-Interscience, 2006.

[9] Manber, Udi. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

Выводы

Преимущества алгоритма LZ77

- **Высокая степень сжатия**: Алгоритм эффективно уменьшает размер данных за счёт использования повторяющихся последовательностей.
- **Простота реализации**: Основной механизм поиска совпадений и кодирования легко реализовать с помощью базовых структур данных.
- **Отсутствие необходимости в словаре**: LZ77 не требует заранее определённого словаря, поскольку использует уже просмотренную часть текста.
- **Гибкость**: Подходит для различных типов данных, включая текстовые и бинарные файлы.

Недостатки алгоритма LZ77

- **Высокая вычислительная сложность**: Поиск совпадений требует значительных временных затрат, особенно при обработке длинных входных строк.
- **Ограничения для больших данных**: Производительность может ухудшаться при увеличении объёма данных из-за необходимости анализа всей ранее просмотренной части текста.
- **Неоптимальное сжатие для коротких строк**: На малых данных алгоритм может быть менее эффективным, так как число повторяющихся последовательностей ограничено.
- **Проблемы с памятью**: При сжатии больших строк увеличивается потребность в памяти для хранения просмотренной части текста.

Возможности улучшения алгоритма

- **Использование скользящего окна**: Ограничение размера области поиска (например, последние N символов) для уменьшения сложности и улучшения производительности.

- **Оптимизация структуры поиска**: Применение хеш-таблиц, деревьев суффиксов или других эффективных структур данных для ускорения поиска совпадений.
- **Параллелизация**: Разделение входных данных на блоки и выполнение сжатия параллельно для увеличения скорости обработки.
- **Адаптивные параметры**: Динамическая настройка параметров, таких как размер окна и минимальная длина совпадения, для достижения баланса между степенью сжатия и скоростью.
- **Сжатие с использованием статистики**: Интеграция с энтропийным кодированием, например, методом Хаффмана или арифметическим кодированием, для дальнейшего уменьшения размера выходных данных.

Заключение

Алгоритм LZ77 демонстрирует высокую эффективность при сжатии данных за счёт поиска повторяющихся последовательностей. Однако его вычислительная сложность остаётся значительным препятствием для обработки больших данных. При внедрении описанных выше методов можно значительно улучшить качество алгоритма и ускорить временную сложность. В результате работы над курсовой работой я изучил материалы, относящиеся к алгоритмам сжатия, выявил различия и особенности алгоритма LZ77, реализовал его на практике и протестировал на длинных строках вплоть до 500000 символов.