`Project Report:`
Deflate Compression, LZ77, and Context-aware Autocomplete using Tries and KMP Algorithm

Harshil Solanki
Data Structures and Algorithms

September 2024

Contents

Chapter 1

Introduction

Data compression and efficient text handling are crucial areas in computer science. This project focuses on three major algorithms and data structures under the domain of data structures and algorithms:

1. Context-aware Autocomplete using Tries and KMP Algorithm.
2. Lempel-Ziv'77 (LZ77) Compression and Decompression.
3. Deflate Compression Algorithm (Combining LZ77 and Huffman Coding).

These techniques serve as a foundation for applications in text editors, file compression utilities, and lossless data compression standards like ZIP and PNG. In this report, we will detail the theory, implementation, and analysis of each algorithm, showcasing how they contribute to efficient storage and retrieval.

Chapter 2

Context-aware Autocomplete Using Tries and KMP Algorithm

2.1 Problem Statement

Implement an autocomplete feature that suggests words based on user input and context. The system should use a Trie to store words and the KMP algorithm to match patterns in the text history for context-aware suggestions.

2.2 Algorithm Explanation

2.2.1 Trie Data Structure

A Trie is used for efficiently storing and searching strings based on prefixes. It consists of nodes where each node represents a character in the alphabet. We use the Trie to find words that match the given prefix quickly.

2.2.2 KMP Pattern Matching Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is used for pattern matching within the text history. It pre-processes the pattern to create an LPS (Longest Prefix Suffix) array, allowing for efficient searching.

2.3 Code Implementation

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>
#include <algorithm>

// TrieNode structure to represent each node in the Trie
class TrieNode {
```

```
public:
    std::unordered_map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() : isEndOfWord(false) {}
};

// Trie class with insertion, searching, and autocomplete
    capabilities
class Trie {
private:
    TrieNode* root;

    // Helper function to collect all words with a given prefix
    void collectWords(TrieNode* node, std::string prefix, std::
        vector<std::string>& results) {
        if (node->isEndOfWord) {
            results.push_back(prefix);
        }
        for (auto& child : node->children) {
            collectWords(child.second, prefix + child.first,
                results);
        }
    }

public:
    Trie() {
        root = new TrieNode();
    }

    // Insert a word into the Trie
    void insert(const std::string& word) {
        TrieNode* current = root;
        for (char ch : word) {
            if (current->children.find(ch) == current->children.end
                ()) {
                current->children[ch] = new TrieNode();
            }
            current = current->children[ch];
        }
        current->isEndOfWord = true;
    }

    // Return words in the Trie with the given prefix
    std::vector<std::string> autocomplete(const std::string& prefix
        ) {
        TrieNode* current = root;
        std::vector<std::string> results;

        for (char ch : prefix) {
            if (current->children.find(ch) == current->children.end
                ()) {
                return results;  // No words with the given prefix
            }
            current = current->children[ch];
        }
        collectWords(current, prefix, results);
        return results;
    }
};
```

```
// Function to compute the Longest Prefix Suffix (LPS) array for
    KMP algorithm
std::vector<int> computeLPSArray(const std::string& pattern) {
    int length = 0;
    int i = 1;
    std::vector<int> lps(pattern.length(), 0);

    while (i < pattern.length()) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}

// KMP search algorithm for finding a pattern in a given text
bool KMPSearch(const std::string& pattern, const std::string& text)
     {
    std::vector<int> lps = computeLPSArray(pattern);
    int i = 0, j = 0;  // i - index for text, j - index for pattern

    while (i < text.length()) {
        if (pattern[j] == text[i]) {
            j++;
            i++;
        }

        if (j == pattern.length()) {
            return true;  // Pattern found
        } else if (i < text.length() && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return false;
}

// Class to implement the context-aware autocomplete with Trie and
    KMP
class ContextAwareAutocomplete {
private:
    Trie trie;                    // Trie for word storage
    std::string textHistory;      // Text history to track user
        inputs

public:
    // Insert a word into the Trie and text history
    void insertWord(const std::string& word) {
        trie.insert(word);
```

```
        textHistory += (textHistory.empty() ? "" : " ") + word;  //
            Add to history with space
    }

    // Get autocomplete suggestions based on the current prefix and
        context
    std::vector<std::string> getSuggestions(const std::string&
        prefix) {
        std::vector<std::string> words = trie.autocomplete(prefix);
        std::vector<std::string> relevantSuggestions;

        // Use KMP to filter words that are contextually relevant
            based on text history
        for (const std::string& word : words) {
            if (KMPSearch(word, textHistory)) {
                relevantSuggestions.push_back(word);
            }
        }

        // Sort by relevance (words in text history appear first)
        std::sort(relevantSuggestions.begin(), relevantSuggestions.
            end());
        return relevantSuggestions;
    }
};

// Main function to test the autocomplete program
int main() {
    ContextAwareAutocomplete autoComplete;

    // Insert sample words
    autoComplete.insertWord("hello");
    autoComplete.insertWord("help");
    autoComplete.insertWord("helicopter");
    autoComplete.insertWord("healthy");
    autoComplete.insertWord("heap");

    // Simulate user typing
    std::string userPrefix;
    std::cout << "Enter a prefix for autocomplete suggestions: ";
    std::cin >> userPrefix;

    // Get and display suggestions
    std::vector<std::string> suggestions = autoComplete.
        getSuggestions(userPrefix);
    std::cout << "Suggestions: ";
    for (const std::string& suggestion : suggestions) {
        std::cout << suggestion << " ";
    }

    return 0;
}
```

2.4 Complexity Analysis

- **Trie Insertion/Autocomplete Search:** $O(m)$, where m is the length of the word.
- **KMP Search:** $O(n+m)$, where n is the length of the text and m is the length of the pattern.

Chapter 3

Lempel-Ziv'77 (LZ77) Compression and Decompression

3.1 Problem Statement

Implement the Lempel-Ziv'77 (LZ77) compression and decompression algorithms for lossless data compression by identifying repeated substrings in the text.

3.2 Algorithm Explanation

3.2.1 LZ77 Compression

LZ77 encodes the text using back-referencing by identifying repeated patterns within a sliding window. The output is a sequence of tuples (offset, length, next character), where:

- **Offset:** Distance back to the start of the matching substring.
- **Length:** Length of the matching substring.
- **Next Character:** First character following the matching substring.

3.2.2 LZ77 Decompression

LZ77 decompression reverses the process by using the (offset, length, next character) tuples to reconstruct the original text.

3.3 Code Implementation

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

```cpp
// Structure to store LZ77 encoding tuples
struct LZ77Tuple {
    int offset;  // Distance to the start of the match
    int length;  // Length of the match
    char nextChar;  // Next character in the look-ahead buffer

    LZ77Tuple(int off, int len, char next) : offset(off), length(
        len), nextChar(next) {}
};

// LZ77 Compression Class
class LZ77Compressor {
private:
    int windowSize;  // Size of the sliding window

public:
    LZ77Compressor(int winSize = 20) : windowSize(winSize) {}

    // Compress the input string using LZ77
    std::vector<LZ77Tuple> compress(const std::string& input) {
        std::vector<LZ77Tuple> encodedData;
        int inputSize = input.size();
        int currentIndex = 0;

        while (currentIndex < inputSize) {
            int matchLength = 0;
            int matchOffset = 0;

            // Define the boundaries of the sliding window and look
                -ahead buffer
            int searchStart = std::max(0, currentIndex - windowSize
                );

            // Search for the longest match in the sliding window
            for (int i = searchStart; i < currentIndex; ++i) {
                int length = 0;
                while (length < inputSize - currentIndex && input[i
                     + length] == input[currentIndex + length]) {
                    ++length;
                }
                // Update the longest match
                if (length > matchLength) {
                    matchLength = length;
                    matchOffset = currentIndex - i;
                }
            }

            // Next character after the match
            char nextChar = currentIndex + matchLength < inputSize
                ? input[currentIndex + matchLength] : '\0';

            // Encode the tuple and advance the index
            encodedData.push_back(LZ77Tuple(matchOffset,
                matchLength, nextChar));
            currentIndex += matchLength + 1;
        }
        return encodedData;
    }

    // Decompress a sequence of LZ77 tuples
    std::string decompress(const std::vector<LZ77Tuple>&
        encodedData) {
```

```
        std::string decompressed;

        for (const LZ77Tuple& tuple : encodedData) {
            if (tuple.offset == 0 && tuple.length == 0) {
                // No match found, just add the next character
                decompressed += tuple.nextChar;
            } else {
                // Replicate the match from the decompressed string
                int start = decompressed.size() - tuple.offset;
                for (int i = 0; i < tuple.length; ++i) {
                    decompressed += decompressed[start + i];
                }
                // Append the next character
                if (tuple.nextChar != '\0') {
                    decompressed += tuple.nextChar;
                }
            }
        }
        return decompressed;
    }
};

// Helper function to print the encoded LZ77 tuples
void printEncodedData(const std::vector<LZ77Tuple>& encodedData) {
    std::cout << "Compressed Data (LZ77 Tuples):\n";
    for (const LZ77Tuple& tuple : encodedData) {
        std::cout << "(" << tuple.offset << ", " << tuple.length <<
             ", '" << tuple.nextChar << "') ";
    }
    std::cout << "\n";
}

int main() {
    std::string inputText;
    std::cout << "Enter text to compress: ";
    std::getline(std::cin, inputText);

    // Create a LZ77Compressor object with a default window size
    LZ77Compressor compressor;

    // Compress the input text
    std::vector<LZ77Tuple> compressedData = compressor.compress(
        inputText);
    printEncodedData(compressedData);

    // Decompress the compressed data
    std::string decompressedText = compressor.decompress(
        compressedData);
    std::cout << "Decompressed Text: " << decompressedText << "\n";

    return 0;
}
```

3.4 Complexity Analysis

- **Compression:** $O(n^2)$ in the worst case due to repeated pattern search.
- **Decompression:** $O(n)$, where n is the length of the compressed data.

Chapter 4

Deflate Compression Using LZ77 and Huffman Coding

4.1 Problem Statement

Implement the Deflate compression algorithm, which combines LZ77 for dictionary compression and Huffman Coding for entropy coding to achieve efficient lossless data compression.

4.2 Algorithm Explanation

4.2.1 LZ77 Phase

The first phase uses LZ77 to identify repeated patterns in the text, storing them as tuples.

4.2.2 Huffman Coding Phase

The Huffman Coding phase builds a frequency table of the symbols generated by LZ77, constructs a Huffman Tree, and assigns binary codes to each symbol. This minimizes the space used by frequently occurring symbols.

4.3 Code Implementation

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <string>
#include <algorithm>

// ---------------------- LZ77 Compression Implementation
    ----------------------

// Structure to store LZ77 encoding tuples
struct LZ77Tuple {
```

```
    int offset;
    int length;
    char nextChar;

    LZ77Tuple(int off, int len, char next) : offset(off), length(
        len), nextChar(next) {}
};

// LZ77 Compression Class
class LZ77Compressor {
private:
    int windowSize;

public:
    LZ77Compressor(int winSize = 20) : windowSize(winSize) {}

    std::vector<LZ77Tuple> compress(const std::string& input) {
        std::vector<LZ77Tuple> encodedData;
        int inputSize = input.size();
        int currentIndex = 0;

        while (currentIndex < inputSize) {
            int matchLength = 0;
            int matchOffset = 0;

            int searchStart = std::max(0, currentIndex - windowSize
                );
            for (int i = searchStart; i < currentIndex; ++i) {
                int length = 0;
                while (length < inputSize - currentIndex && input[i
                     + length] == input[currentIndex + length]) {
                    ++length;
                }
                if (length > matchLength) {
                    matchLength = length;
                    matchOffset = currentIndex - i;
                }
            }
            char nextChar = currentIndex + matchLength < inputSize
                ? input[currentIndex + matchLength] : '\0';
            encodedData.push_back(LZ77Tuple(matchOffset,
                matchLength, nextChar));
            currentIndex += matchLength + 1;
        }
        return encodedData;
    }
};

// ---------------------- Huffman Coding Implementation
    ----------------------

// Structure to represent a Huffman Tree Node
struct HuffmanNode {
    char character;
    int frequency;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char ch, int freq) : character(ch), frequency(freq)
        , left(nullptr), right(nullptr) {}
};
```

```
// Comparator for priority queue (min-heap)
struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->frequency > b->frequency;
    }
};

// Class for Huffman Tree
class HuffmanTree {
private:
    HuffmanNode* root;
    std::unordered_map<char, std::string> huffmanCodes;

    // Helper function to build the Huffman codes from the tree
    void buildHuffmanCodes(HuffmanNode* node, std::string code) {
        if (!node) return;
        if (!node->left && !node->right) {
            huffmanCodes[node->character] = code;
        }
        buildHuffmanCodes(node->left, code + "0");
        buildHuffmanCodes(node->right, code + "1");
    }

public:
    // Build the Huffman Tree based on character frequencies
    void buildTree(const std::unordered_map<char, int>&
        frequencyTable) {
        std::priority_queue<HuffmanNode*, std::vector<HuffmanNode
            *>, Compare> minHeap;
        for (auto& entry : frequencyTable) {
            minHeap.push(new HuffmanNode(entry.first, entry.second)
                );
        }

        while (minHeap.size() > 1) {
            HuffmanNode* left = minHeap.top(); minHeap.pop();
            HuffmanNode* right = minHeap.top(); minHeap.pop();
            HuffmanNode* sumNode = new HuffmanNode('\0', left->
                frequency + right->frequency);
            sumNode->left = left;
            sumNode->right = right;
            minHeap.push(sumNode);
        }

        root = minHeap.top();
        buildHuffmanCodes(root, "");
    }

    // Get the Huffman code for a character
    std::string getHuffmanCode(char ch) {
        return huffmanCodes[ch];
    }

    // Return the complete Huffman codes map
    std::unordered_map<char, std::string> getCodes() {
        return huffmanCodes;
    }
};

// ---------------------- Deflate Compression Implementation
    ----------------------
```

```cpp
// Class to combine LZ77 and Huffman Coding for Deflate Compression
class DeflateCompressor {
private:
    LZ77Compressor lz77;
    HuffmanTree huffman;

public:
    DeflateCompressor(int winSize = 20) : lz77(winSize) {}

    std::string compress(const std::string& input) {
        std::vector<LZ77Tuple> lz77Encoded = lz77.compress(input);

        // Create a frequency table for Huffman coding
        std::unordered_map<char, int> frequencyTable;
        for (const auto& tuple : lz77Encoded) {
            frequencyTable[tuple.nextChar]++;
        }

        // Build the Huffman Tree
        huffman.buildTree(frequencyTable);

        // Encode the LZ77 tuples using Huffman codes
        std::string compressed;
        for (const auto& tuple : lz77Encoded) {
            compressed += huffman.getHuffmanCode(tuple.nextChar);
        }

        return compressed;
    }
};

// ---------------------- Main Function to Test Compression
    ----------------------
int main() {
    std::string inputText;
    std::cout << "Enter text to compress: ";
    std::getline(std::cin, inputText);

    // Create Deflate Compressor object
    DeflateCompressor deflate;

    // Compress the input text
    std::string compressedData = deflate.compress(inputText);
    std::cout << "Compressed Data: " << compressedData << "\n";

    return 0;
}
```

4.4 Complexity Analysis

- **LZ77 Compression:** $O(n^2)$ in the worst case.
- **Huffman Coding:** $O(n \log n)$, where n is the number of distinct symbols.

Chapter 5

Conclusion

The three algorithms explored in this project show how efficient data structures and algorithms can be used to solve complex text processing and compression problems. The Trie and KMP algorithm efficiently handle text suggestion problems, while LZ77 and Deflate demonstrate the principles of lossless compression. Future improvements could include optimizing the LZ77 search phase and enhancing the compression ratio through adaptive Huffman Coding.

Chapter 6

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 3rd Edition.

2. Mark Nelson, *The Data Compression Book*, M&T Books, 1991.

3. David A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the I.R.E., 1952.