Design and Analysis of Algorithms

Mini Project Report

Title: String Matching Algorithms - Naive and Rabin-Karp

Group Members: Rushi Bedmutha (BE-A-45)

Sharvay Chandgude (BE-A-46)

Gaurav Aher (BE-A-47)

Abstract

This project involves the implementation and comparison of two string matching algorithms: the Naive string-matching algorithm and the Rabin-Karp algorithm. These algorithms are essential in various applications, such as text search engines, DNA sequencing, and data mining. By understanding and comparing these algorithms, we can evaluate their efficiency and suitability for different types of input data. This report discusses the theoretical background, implementation details, experimental results, and observations on the performance of these algorithms.

Introduction

String matching, also known as pattern matching, is a fundamental problem in computer science. It involves finding all occurrences of a substring (pattern) within a larger string (text). Efficient string matching is crucial in areas such as search engines, text editing, bioinformatics, and information retrieval. This project focuses on two classic algorithms:

1. **Naive String-Matching Algorithm**
2. **Rabin-Karp Algorithm**

Algorithms Overview

1. **Naive String-Matching Algorithm:**
    - The Naive algorithm is straightforward. It checks for the pattern at every possible position in the text.

- Time Complexity: O((n - m + 1) * m) in the worst case, where n is the length of the text and m is the length of the pattern.
- Space Complexity: O(1) as it uses only a few additional variables.
- Characteristics:
    - Easy to implement.
    - Inefficient for large texts and patterns due to its quadratic time complexity.
    - Suitable for small texts or when the pattern is relatively small compared to the text.

2. **Rabin-Karp Algorithm:**
    - The Rabin-Karp algorithm uses hashing to find the pattern in the text. It compares hash values of the pattern with hash values of substrings in the text.
    - Time Complexity: O(n * m) in the worst case, but O(n + m) on average assuming a good hash function.
    - Space Complexity: O(1) for storing the hash values.
    - Characteristics:
        - More efficient than the Naive algorithm for large texts due to the hashing mechanism.
        - The average-case performance is excellent, but the worst-case performance can degrade due to hash collisions.
        - Ideal for applications where the pattern is searched repeatedly in the same text.

Implementation Details

Both algorithms were implemented in C++ and tested with user input for the text and pattern. The program allows the user to choose which algorithm to use for string matching.

Experimental Setup

1. **Environment:**
    - The implementation was carried out in a C++ development environment.

- User inputs were used for both text and pattern to test the algorithms.
- The performance was observed in terms of execution time and correctness of the matching positions.

- **Test Cases:**
  - Several test cases with varying lengths of text and pattern were used.
  - Example Text: "abracadabra"
  - Example Pattern: "abra"
  - Both algorithms were run on the same input to compare their performance.

Results and Observations

1. **Naive String-Matching Algorithm:**
   - **Output:**
     - For the text "abracadabra" and pattern "abra", the positions returned were 0 and 7.
   - **Performance:**
     - The algorithm performed multiple character comparisons for each position in the text.
     - Execution time increased significantly with larger texts and patterns.
     - The simplicity of the algorithm makes it easy to understand and implement.
2. **Rabin-Karp Algorithm:**
   - **Output:**
     - For the text "abracadabra" and pattern "abra", the positions returned were 0 and 7.
   - **Performance:**
     - The algorithm efficiently computed hash values and compared them, resulting in fewer character comparisons.

- The execution time was significantly lower for larger inputs compared to the Naive algorithm.
- Occasional hash collisions were handled correctly, ensuring accurate matching results.

Detailed Analysis

1. **Efficiency:**
    - The Naive algorithm's efficiency is heavily dependent on the size of the input. For very large texts, it becomes impractical due to its quadratic time complexity.
    - The Rabin-Karp algorithm, on the other hand, benefits from the use of hashing. The average-case time complexity is linear, making it more suitable for large-scale text searches.
2. **Scalability:**
    - The Naive algorithm does not scale well with increasing input sizes.
    - The Rabin-Karp algorithm scales much better, especially when dealing with multiple pattern searches in the same text.
3. **Practical Applications:**
    - The Naive algorithm is useful for small-scale applications or when the pattern size is small relative to the text.
    - The Rabin-Karp algorithm is preferred for large texts and when multiple patterns need to be searched, such as in plagiarism detection or searching through large databases.

Conclusion

Both the Naive and Rabin-Karp algorithms serve as foundational techniques in string matching. The Naive algorithm, while simple and easy to implement, lacks efficiency for large texts and patterns. The Rabin-Karp algorithm, with its hashing mechanism, offers better performance and scalability. This project provided a practical understanding of these algorithms, highlighting their strengths and limitations. Future work could involve exploring more advanced string-matching algorithms like Knuth-Morris-Pratt (KMP) or Boyer-Moore, which offer further improvements in efficiency.