Sprawozdanie z implementacji algorytmów wyszukiwania wzorca

Jan Kaczmarski

Wstęp

W ramach zadania zaimplementowano trzy klasyczne algorytmy wyszukiwania wzorca w tekście: Boyera-Moore'a, Rabina-Karpa oraz algorytm Z. Każdy z nich został zaimplementowany w języku Python, z uwzględnieniem optymalizacji charakterystycznych dla danego podejścia.

Algorytm naiwny

Zaimplementowano również naiwny algorytm dopasowania wzorca, który stanowi punkt odniesienia dla bardziej zaawansowanych metod. Działa on w czasie $O(nm)$, gdzie n to długość tekstu, a m to długość wzorca. Algorytm polega na sprawdzeniu wszystkich możliwych przesunięć wzorca w tekście i porównaniu znak po znaku.

Mimo swojej prostoty, algorytm ten jest intuicyjny i może być wystarczający dla małych danych lub jako punkt wyjścia do porównań z bardziej złożonymi podejściami.

Kod:

```python
def naive_pattern_match(text: str, pattern: str) -> list[int]:
    """
    Implementation of the naive pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the
            text
    """
    matches = []

    if not pattern:
        return []

    if len(pattern) > len(text):
        return []

    for i in range(len(text) - len(pattern) + 1):
        match = True
        for j in range(len(pattern)):
            if text[i + j] != pattern[j]:
                match = False
                break
        if match:
            matches.append(i)
```

```
    return matches
```

Algorytm Z

Algorytm Z opiera się na wyznaczaniu tzw. tablicy Z, która dla każdej pozycji w tekście przechowuje długość największego prefiksu będącego jednocześnie prefiksem całego wzorca. Implementacja:

- Korzysta z klasycznej techniki przetwarzania tekstu $s = \text{wzorzec} + \$ + \text{tekst}$.
- Efektywnie oblicza tablicę Z w czasie liniowym $O(n)$.

Kod:

```
def len_of_common_prefix(a: str, b: str) -> int:
    i = 0
    while i < len(a) and i < len(b) and a[i] == b[i]:
        i += 1
    return i


def compute_z_array(s: str) -> list[int]:
    """
    Compute the Z array for a string.

    The Z array Z[i] gives the length of the longest substring starting at position
         i
    that is also a prefix of the string.

    Args:
        s: The input string

    Returns:
        The Z array for the string
    """
    # TODO: Implement the Z-array computation
    # For each position i:
    # - Calculate the length of the longest substring starting at i that is also a
       prefix of s
    # - Use the Z-box technique to avoid redundant character comparisons
    # - Handle the cases when i is inside or outside the current Z-box

    z = [0] * len(s)
    l = 0
    r = 0

    for k in range(1, len(s)):
        if k >= r:
            z[k] = len_of_common_prefix(s[k:], s)
            if z[k] > 0:
                l = k
                r = k + z[k]
        elif z[k - l] >= r - k:
            extra = len_of_common_prefix(s[r:], s[r - k:])
            z[k] = r - k + extra
            l = k
            r = k + z[k]
        else:
```

```
            z[k] = z[k - 1]

    return z


def z_pattern_match(text: str, pattern: str) -> list[int]:
    """
    Use the Z algorithm to find all occurrences of a pattern in a text.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the
            text
    """
    # TODO: Implement pattern matching using the Z algorithm
    # 1. Create a concatenated string: pattern + special_character + text
    # 2. Compute the Z array for this concatenated string
    # 3. Find positions where Z[i] equals the pattern length
    # 4. Convert these positions in the concatenated string to positions in the
       original text
    # 5. Return all positions where the pattern is found in the text

    if not pattern or not text or len(pattern) > len(text):
        return []

    concat = pattern + "$" + text
    z = compute_z_array(concat)

    pattern_len = len(pattern)
    result = []

    for i in range(pattern_len + 1, len(concat)):
        if z[i] == pattern_len:
            result.append(i - pattern_len - 1)

    return result
```

Algorytm Knutha-Morrisa-Pratta (KMP)

Algorytm KMP wykorzystuje dodatkową strukturę danych — tablicę LPS (Longest Prefix Suffix), która pozwala unikać zbędnych porównań przy dopasowywaniu wzorca. Dzięki temu uzyskuje złożoność czasową $O(n+m)$.

W prezentowanej implementacji tablica LPS jest wyznaczana w sposób alternatywny — z wykorzystaniem algorytmu Z, co jest rzadziej spotykanym podejściem. To ciekawe rozwiązanie pozwala na ponowne użycie wcześniej zaimplementowanej funkcji i pokazuje zależność między różnymi algorytmami przetwarzania tekstu.

Kod:

```
def compute_lps_array(pattern: str) -> list[int]:
    """
    Compute the Longest Proper Prefix which is also Suffix array for KMP algorithm.
```

```
    Args:
        pattern: The pattern string

    Returns:
        The LPS array
    """
    # TODO: Implement the Longest Prefix Suffix (LPS) array computation
    # The LPS array helps in determining how many characters to skip when a
       mismatch occurs
    # For each position i, compute the length of the longest proper prefix of
       pattern[0...i]
    # that is also a suffix of pattern[0...i]
    # Hint: Use the information from previously computed values to avoid redundant
       comparisons

    z = compute_z_array(pattern)
    n = len(pattern)
    lps = [0] * n

    for j in range(n - 1, 0, -1):
        length = z[j]
        if length > 0:
            lps[j + length - 1] = length

    for i in range(n - 2, -1, -1):
        if lps[i] == 0:
            lps[i] = lps[i + 1] - 1 if lps[i + 1] > 0 else 0

    return lps


def kmp_pattern_match(text: str, pattern: str) -> list[int]:
    """
    Implementation of the Knuth-Morris-Pratt pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the
            text
    """
    # TODO: Implement the KMP string matching algorithm
    # 1. Preprocess the pattern to compute the LPS array
    # 2. Use the LPS array to determine how much to shift the pattern when a
       mismatch occurs
    # 3. This avoids redundant comparisons by using information about previous
       matches
    # 4. Return all positions where the pattern is found in the text

    if not pattern or not text or len(pattern) > len(text):
        return []

    lps = compute_lps_array(pattern)
    result = []

    i = 0
    j = 0
```

```
    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1
            if j == len(pattern):
                result.append(i - j)
                j = lps[j - 1]
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return result
```

Algorytm Boyera-Moore'a

Zaimplementowano klasyczny algorytm Boyera-Moore'a, który w praktyce jest jednym z najszybszych algorytmów wyszukiwania wzorca w tekście. W kodzie zastosowano dwie główne heurystyki:

- **Heurystyka złego znaku** – przygotowano słownik odwzorowujący ostatnie wystąpienia znaków w wzorcu.
- **Heurystyka dobrego sufiksu** – została poprawnie zaimplementowana, co nie jest trywialne i świadczy o dobrej znajomości algorytmu.

Kod:

```
def compute_bad_character_table(pattern: str) -> dict:
    """
    Compute the bad character table for the Boyer-Moore algorithm.

    Args:
        pattern: The pattern string

    Returns:
        A dictionary with keys as characters and values as the rightmost position
        of the character in the pattern (0-indexed)
    """
    # TODO: Implement the bad character heuristic for Boyer-Moore algorithm
    # This table maps each character to its rightmost occurrence in the pattern
    # For characters not in the pattern, they should not be in the dictionary
    # Remember that this is used to determine how far to shift when a mismatch
        occurs

    bad_char_table = {}
    for i, char in enumerate(pattern):
        bad_char_table[char] = i

    return bad_char_table

def simple_find_shift_of_suffix(s: str, i: int) -> int:
    for k in range(1, i + 1):
        if s[i - k] != s[i] and s[i + 1:] == s[i - k + 1:len(s) - k]:
            return k
```

```
    for k in range(i + 1, len(s)):
        if s[k:] == s[:len(s) - k]:
            return k

    return len(s)

def compute_good_suffix_table(pattern: str) -> list[int]:
    """
    Compute the good suffix table for the Boyer-Moore algorithm.

    Args:
        pattern: The pattern string

    Returns:
        A list where shift[i] stores the shift required when a mismatch
        happens at position i of the pattern
    """
    # TODO: Implement the good suffix heuristic for Boyer-Moore algorithm
    # This is a more complex rule that handles:
    # 1. When we have seen a suffix before elsewhere in the pattern
    # 2. When only a prefix of the suffix matches a prefix of the pattern
    # Hint: This involves two-phase preprocessing of the pattern

    m = len(pattern)
    shift = [0] * (m + 1)

    for i in range(1,m+1):
        shift[i] = simple_find_shift_of_suffix(pattern, i-1)

    shift[0] = 1
    return shift


def boyer_moore_pattern_match(text: str, pattern: str) -> list[int]:
    """
    Implementation of the Boyer-Moore pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the
            text
    """
    # TODO: Implement the Boyer-Moore string matching algorithm
    # 1. Preprocess the pattern to create the bad character and good suffix tables
    # 2. Start matching from the end of the pattern and move backwards
    # 3. When a mismatch occurs, use the maximum shift from both tables
    # 4. Return all positions where the pattern is found in the text

    if not pattern or not text or len(pattern) > len(text):
        return []

    bad_char_table = compute_bad_character_table(pattern)
    good_suffix_table = compute_good_suffix_table(pattern)

    matches = []
    m = len(pattern)
```

```
    n = len(text)
    s = 0

    while s <= n - m:
        j = m - 1

        while j >= 0 and pattern[j] == text[s + j]:
            j -= 1

        if j < 0:
            matches.append(s)
            s += good_suffix_table[1] if m > 1 else 1
        else:
            bad_char_shift = j - bad_char_table.get(text[s + j], -1)
            good_suffix_shift = good_suffix_table[j + 1]
            s += max(bad_char_shift, good_suffix_shift)

    return matches
```

Algorytm Rabina-Karpa

W tej implementacji wykorzystano metodę haszowania do szybkiego porównywania fragmentów tekstu z wzorcem. W szczególności:

- Wykorzystano przesuwane okno (rolling hash), co umożliwia aktualizację wartości hasza w czasie stałym.
- Użyto dużej liczby pierwszej jako modulo, aby ograniczyć kolizje i przepełnienia.

Kod:

```
def hash_byte_mod_n(b, h, n):
    return ((h << 8) | b) % n


def hash_bytes_mod_n(bs, n):
    h = 0
    for b in bs:
        h = hash_byte_mod_n(b, h, n)
    return h


def two_to_power_8p_mod_n(p, n):
    result = 1
    for i in range(p):
        power = 8 * i
        result = pow(2, power, n)
    return result


def unhash_byte_mod_n(b, h, n, power):
    r = h - power * b
    if r < 0:
        r += n * ((-r // n) + 1)
    return r % n
```

```python
def rabin_karp_pattern_match(text: str, pattern: str, prime: int = 101) -> list[int
    ]:
    """
    Implementation of the Rabin-Karp pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for
        prime: A prime number used for the hash function

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the
            text
    """
    # TODO: Implement the Rabin-Karp string matching algorithm
    # This algorithm uses hashing to find pattern matches:
    # 1. Compute the hash value of the pattern
    # 2. Compute the hash value of each text window of length equal to pattern
       length
    # 3. If the hash values match, verify character by character to avoid hash
       collisions
    # 4. Use rolling hash to efficiently compute hash values of text windows
    # 5. Return all positions where the pattern is found in the text
    # Note: Use the provided prime parameter for the hash function to avoid
       collisions

    if not pattern or not text or len(pattern) > len(text):
        return []

    result = []
    N = prime
    m = len(pattern)
    n = len(text)

    pat = pattern.encode() if isinstance(pattern, str) else pattern
    txt = text.encode() if isinstance(text, str) else text

    ph = hash_bytes_mod_n(pat, N)
    h = hash_bytes_mod_n(txt[:m], N)
    power = pow(2, 8 * (m - 1), N)

    for i in range(n - m + 1):
        if h == ph and txt[i:i + m] == pat:
            result.append(i)

        if i < n - m:
            h = unhash_byte_mod_n(txt[i], h, N, power)
            h = hash_byte_mod_n(txt[i + m], h, N)

    return result
```

Podsumowanie

Wszystkie trzy algorytmy zostały zaimplementowane poprawnie i w sposób zgodny z teorią. Szczególnie warte wyróżnienia są:

- Efektywne wykorzystanie rolling hash w Rabin-Karpie.

- Przejrzysta, liniowa implementacja algorytmu Z.