Hubert_Miklas_Sprawozdanie_lab3

April 12, 2025

1 Sprawozdanie z laboratorium 3 z algorytmów tekstowych

Hubert Miklas 11-04-2025

1.1 Wstęp

Tematem laboratorium były algortymy dopasowania wzorca do tekstu. W trakcie trwania laboratorium zaimplementowałem algortym naiwny, algortym Rabina-Karpa i algorytm Z. W sprawozdaniu opisuję algorytm Boyera-Moora i algorytm Knutha-Morrisa-Pratta.

1.2 Algorytm Knutha-Morrisa_Pratta

1.2.1 Opis algorytmu

Algorytm Knutha-Morrisa-Pratta (KMP) służy do efektywnego wyszukiwania wzorca w tekście. Jego główną zaletą jest unikanie wielokrotnego porównywania tych samych znaków poprzez wykorzystanie informacji z wcześniej wykonanych dopasowań. Działa w czasie $O(n+m)$ długości tekstu oraz wzorca.

Algorytm składa się z dwóch głównych kroków:

1. Preprocessing wzorca – obliczany jest tzw. LPS (Longest Prefix-Suffix) - tablica najdłuższych prefikso-sufiks-ów, który dla każdego indeksu `i` wzorca przechowuje długość najdłuższego właściwego prefiksu do `i`, będącego jednocześnie sufiksem `pattern[0...i]`.

```
def compute_lps_array(pattern: str) -> list[int]:
    m = len(pattern)
    lps = [0] * m
    length = 0

    i = 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
```

```
            i += 1
    return lps
```

Opis LPS:

1. Zaczynając od i = 1 (drugi znak wzorca), porównujemy znaki `pattern[i]` i `pattern[length]`.
2. Jeśli znaki są równe, oznacza to, że prefiks może być wydłużony – zwiększamy `length` i przypisujemy jego wartość do `lps[i]`.
3. Jeśli znaki się różnią i `length != 0`, to wracamy do krótszego prefiksu, czyli ustawiamy `length = lps[length - 1]` i próbujemy ponownie bez przesuwania `i`.
4. Jeśli znaki się różnią i `length == 0`, to nie znaleziono pasującego prefiksu – ustawiamy `lps[i] = 0` i przesuwamy `i`.
5. Przeszukiwanie tekstu – wykorzystując tablicę LPS, algorytm porównuje znaki tekstu z wzorcem i w przypadku niezgodności „cofa się" do ostatniego znanego dobrego dopasowania, zamiast wracać do początku wzorca.

```
def kmp_pattern_match(text: str, pattern: str) -> list[int]:
    result = []
    n, m = len(text), len(pattern)
    if m == 0 or n == 0 or m > n:
        return result

    lps = compute_lps_array(pattern)
    i, j = 0, 0

    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1
        if j == m:
            result.append(i - j)
            j = lps[j - 1] # Returns to the closest prefix which is also a suffix
        elif i < n and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1] # Returns to the closest prefix which is also a suffix
            else:
                i += 1
    return result
```

Części wykomentowane działają w następujący sposób: Jeżeli mamy słowo `xyx` i znajdujemy się na miejscu `i` w tekście, gdzie `x` jest prefikso-sufiksem, to jeżeli wiadomo, że w tekście pojawił się `x`, to możemy spokojnie przyjąć, że początek wzorca `x` będący prefiksem również będzie pasował, w takim razie w `i` znajduje się ostatni znak prefikso-sufiksu `x`.

Kod został przetestowany i przeszedł wszystkie testy.

1.3 Algorytm Boyera-Moora

1.3.1 Opis algorytmu

Algorytm Boyera-Moore'a służy do szybkiego wyszukiwania wzorca w tekście w średnim przypadku. Wyróżnia się bardzo dobrą wydajnością w praktyce dzięki zastosowaniu dwóch heurystyk: "bad character" (zły znak) oraz "good suffix" (dobry sufiks). Obie heurystyki pozwalają na pomijanie fragmentów tekstu, które nie mogą zawierać dopasowania, co znacząco ogranicza liczbę porównań. W najgorszym przypadku działa w czasie $O(n \cdot m)$, ale w średnim przypadku osiąga złożoność bliską $O(n)$.

Algorytm składa się z trzech głównych kroków:

1. Obliczenie tablicy złych znaków – dla każdego znaku w wzorcu obliczana jest tablica, która wskazuje na najdalszą od lewej pozycję wystąpienia danego znaku w wzorcu. Tę tablicę wykorzystuje się do przesuwania wzorca w przypadku niezgodności znaków.

```
def compute_bad_character_table(pattern: str) -> dict:
    table = {}
    for i, char in enumerate(pattern):
        table[char] = i
    return table
```

2. Obliczenie tablicy dobrych sufiksów – tablica ta służy do przesuwania wzorca w przypadku, gdy w wyniku porównania tekstu z wzorcem wykryto niezgodność. W przypadku dopasowania części wzorca do sufiksu w tekście, ta heurystyka pomaga określić, o ile należy przesunąć wzorzec, aby przynajmniej część wzorca pasowała.

```
def compute_good_suffix_table(pattern: str) -> list[int]:
    m = len(pattern)
    shift = [0] * (m + 1)
    pos = [0] * (m + 1)

    i = m
    j = m + 1
    pos[i] = j

    while i > 0:
        last_position = j - 1 if j - 1 < m else m - 1  # The last matching position
        while j <= m and pattern[i - 1] != pattern[last_position]:
            if shift[j] == 0:  # If shift[j] is still 0, set it to the value j - i
                shift[j] = j - i
            j = pos[j]  # Update j to the position of the next potential matching suffix
        i -= 1
        j -= 1
        pos[i] = j

    j = pos[0]
    for i in range(m + 1):
        if shift[i] == 0:
            shift[i] = j
```

```
            if i == j:
                j = pos[j]

        return shift
```

3. Przeszukiwanie tekstu – w tej fazie algorytm porównuje tekst z wzorcem, zaczynając od końca wzorca. W przypadku niezgodności, wzorzec jest przesuwany zgodnie z wartościami z tabeli "bad character" lub "good suffix". Gdy wzorzec jest dopasowany do tekstu, jego pozycja jest zapisywana w wynikach, a następnie wzorzec jest przesuwany dalej.

```
def boyer_moore_pattern_match(text: str, pattern: str) -> list[int]:
    result = []
    n = len(text)
    m = len(pattern)

    if n == 0 or m == 0 or m > n:
        return result

    bad_char = compute_bad_character_table(pattern)
    good_suffix = compute_good_suffix_table(pattern)

    i = 0

    while i <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1 # Go as far as possible with j, until character doesn't match or the entire
        if j < 0:
            result.append(i) # There was a match! j should be -1
            i += good_suffix[0] if m > 1 else 1 # Shift to the good suffix, but for patterns o
        else:
            bc_shift = j - bad_char.get(text[i + j], -1) # Shift to the last bad character, be
            gs_shift = good_suffix[j] if j < len(good_suffix) else 1 # Take the best suffix fr
            i += max(bc_shift, gs_shift) # This makes this algorithm good, we shift not by 1,
    return result
```

W komentarzach powyżej opisałem krótko co się dzieje w danych linijkach.

Kod został przetestowany i przeszedł wszystkie testy.