Sprawozdanie z laboratorium 4: Wyszukiwanie wzorców (2)

Jan Kaczmarski

Wstęp

W ramach laboratorium zaimplementowano następujące algorytmy/struktury danych/rozwiązano następujące problemy:

- Algorytm Shift-Or
- Algorytm Fuzzy Matching
- Odległość Levenshtein'a
- Algorytm Aho-Corasick
- Algorytm Two Dimensional Search

Celem laboratorium było zapoznanie się z teoretycznymi podstawami wyrażeń regularnych, implementacja wybranych algorytmów służących do ich dopasowywania oraz praktyczne zastosowanie mechanizmów wyszukiwania wzorców w tekście. Dodatkowo, celem było porównanie efektywności różnych podejść do problemu dopasowania wzorca. Implementacje zostały wykonane w języku Python. W sprawozdaniu przedstawiono opis zaimplementowanych metod oraz analizę ich działania.

Algorytm Shift-Or

Shift-Or (Bitap) to szybki algorytm wyszukiwania wzorca w tekście, wykorzystujący operacje bitowe. Działa efektywnie dla krótkich wzorców (do długości słowa maszynowego, np. 64 bity), przetwarzając wiele stanów dopasowania jednocześnie.

Stan dopasowania 's' reprezentowany jest jako maska bitowa. Początkowo wszystkie bity są ustawione na 1. Dla każdego znaku tekstu 's' jest przesuwany w lewo i łączony bitowo z maską znaku. Dopasowanie zachodzi, gdy najbardziej znaczący bit 's' (odpowiadający końcowi wzorca) przyjmie wartość 0.

Złożoność czasowa algorytmu to $O(m + |\Sigma| + n)$, gdzie 'm' to długość wzorca, $|\Sigma|$ — rozmiar alfabetu, a 'n' — długość tekstu.

Kluczowe elementy algorytmu

- **Stan ('s'):** Stan algorytmu 's' jest również maską bitową o długości 'm'. Jeśli 'j'-ty bit stanu 's' jest ustawiony na 0, oznacza to, że prefiks wzorca o długości 'j+1' (tj. 'pattern[0...j]') pasuje do sufiksu aktualnie przetworzonego fragmentu tekstu (kończącego się na bieżącym znaku). Początkowo wszystkie bity stanu 's' są ustawione na 1 (np. ' 0'), co oznacza brak znalezionych dopasowań.
- **Aktualizacja stanu:** Po wczytaniu kolejnego znaku 'text[i]' z tekstu, stan 's' jest aktualizowany w dwóch krokach:
    1. 's = s « 1': Stan jest przesuwany o jeden bit w lewo. Najmłodszy bit (zerowy) jest ustawiany na 0. To przesunięcie odpowiada "próbie" rozszerzenia wszystkich dotychczasowych częściowych dopasowań o kolejny znak.

2. ‘s = s — M[text[i]]‘: Wynik przesunięcia jest łączony operacją bitową OR z maską ‘M[text[i]]‘ odpowiadającą bieżącemu znakowi tekstu. Jeśli ‘j‘-ty bit ‘M[text[i]]‘ jest równy 1 (co oznacza, że ‘text[i]‘ nie pasuje do ‘pattern[j]‘), to ‘j‘-ty bit stanu ‘s‘ zostanie ustawiony na 1. Jeśli ‘j‘-ty bit ‘M[text[i]]‘ jest równy 0 (co oznacza, że ‘text[i]‘ pasuje do ‘pattern[j]‘), to ‘j‘-ty bit ‘s‘ zachowa swoją wartość po przesunięciu (jeśli był 0, pozostanie 0, sygnalizując dopasowanie prefiksu ‘pattern[0...j]‘).

- **Wykrycie dopasowania:** Pełne dopasowanie wzorca w tekście jest sygnalizowane, gdy ‘(m-1)‘-szy bit stanu ‘s‘ (najbardziej znaczący bit, odpowiadający ostatniemu znakowi wzorca) staje się równy 0. Oznacza to, że cały wzorzec ‘pattern[0...m-1]‘ został dopasowany.

Charakterystyka implementacji

- Wykorzystanie operacji bitowych do szybkiego przetwarzania.
- Wstępne obliczanie tablicy masek dla wszystkich możliwych znaków.
- Złożoność czasowa: $O((m + |\Sigma|) + n)$ dla wzorców o długości ‘m‘ nieprzekraczającej liczby bitów w słowie maszynowym, gdzie $|\Sigma|$ to rozmiar alfabetu, a ‘n‘ to długość tekstu. W ogólnym przypadku, gdy ‘m‘ jest większe od rozmiaru słowa maszynowego, złożoność może wynosić $O(nm/w + |\Sigma|)$, gdzie ‘w‘ to rozmiar słowa maszynowego.
- Złożoność pamięciowa: $O(m + |\Sigma|)$ na przechowanie masek i stanu.

Poniżej znajduje się kod implementujący algorytm Shift-Or.

Kod:

```
def set_nth_bit(n: int) -> int:
    """
    Zwraca maskę bitową z ustawionym n-tym bitem na 1.

    Args:
        n: Pozycja bitu do ustawienia (0-indeksowana)

    Returns:
        Maska bitowa z n-tym bitem ustawionym na 1
    """
    # TODO: Zaimplementuj ustawianie n-tego bitu
    return 1 << n


def nth_bit(m: int, n: int) -> int:
    """
    Zwraca wartość n-tego bitu w masce m.

    Args:
        m: Maska bitowa
        n: Pozycja bitu do odczytania (0-indeksowana)

    Returns:
        Wartość n-tego bitu (0 lub 1)
    """
    # TODO: Zaimplementuj odczytywanie n-tego bitu
    return (m >> n) & 1


def make_mask(pattern: str) -> list:
```

```
    """
    Tworzy tablicę masek dla algorytmu Shift-Or.

    Args:
        pattern: Wzorzec do wyszukiwania

    Returns:
        Tablica 256 masek, gdzie każda maska odpowiada jednemu znakowi ASCII
    """
    # TODO: Zaimplementuj tworzenie tablicy masek dla algorytmu Shift-Or
    # TODO: Utwórz tablicę z maskami dla wszystkich znaków ASCII
    # TODO: Dla każdego znaku w pattern, ustaw odpowiednie bity w maskach
    m = [~0] * 256 # Inicjalizacja masek wartością, gdzie wszystkie bity są 1

    for j, c in enumerate(pattern):
        m[ord(c)] &= ~set_nth_bit(j) # Ustaw j-ty bit na 0 dla znaku c

    return m


def shift_or(text: str, pattern: str) -> list[int]:
    """
    Implementacja algorytmu Shift-Or do wyszukiwania wzorca.

    Args:
        text: Tekst do przeszukania
        pattern: Wzorzec do wyszukiwania

    Returns:
        Lista pozycji (0-indeksowanych), na których znaleziono wzorzec
    """
    # TODO: Zaimplementuj algorytm Shift-Or
    # TODO: Obsłuż przypadki brzegowe (pusty wzorzec, wzorzec dłuższy niż tekst)
    # TODO: Utwórz maski dla wzorca
    # TODO: Zainicjalizuj stan początkowy
    # TODO: Zaimplementuj główną logikę algorytmu
    # TODO: Wykryj i zapisz pozycje dopasowań

    pattern_len = len(pattern)
    text_len = len(text)

    if pattern_len == 0:
        return []
    if pattern_len > text_len:
        return []
    # Ograniczenie: Wzorzec nie może być dłuższy niż 64 znaki dla typowego słowa
        maszynowego
    # W Pythonie liczby całkowite mają dowolną precyzję, ale idea algorytmu
    # zakłada efektywność dla m <= rozmiar słowa maszynowego.
    # Dla celów tej implementacji, zakładamy, że operacje bitowe są efektywne.

    masks = make_mask(pattern)
    current_state = ~0  # Stan początkowy: wszystkie bity ustawione na 1
    output = []

    for i, char_code in enumerate(map(ord, text)):
        current_state = (current_state << 1) | masks[char_code]

        if nth_bit(current_state, pattern_len - 1) == 0:
```

```
            output.append(i - pattern_len + 1)

    return output
```

Fuzzy Shift-Or

Fuzzy Shift-Or to rozszerzenie algorytmu Shift-Or umożliwiające wyszukiwanie przybliżone — dopasowania wzorca do tekstu z maksymalnie k różnicami (np. błędami typu Hamming).

Zamiast jednej maski stanu, algorytm utrzymuje $k + 1$ masek reprezentujących stany dopasowania z różnymi liczbami błędów. Każda kolejna maska s_i odpowiada dopasowaniu z co najmniej i różnicami. Podczas przetwarzania kolejnych znaków tekstu stany są aktualizowane bitowo na podstawie poprzednich stanów oraz maski znaku.

Dopasowanie przy co najwyżej k błędach jest wykrywane, gdy najbardziej znaczący bit s_k (odpowiadający końcowi wzorca) przyjmie wartość 0.

Złożoność czasowa algorytmu to $O(kn)$, a pamięciowa $O(k)$, co czyni go efektywnym dla małych wartości k i krótkich wzorców.

Listing 1: Implementacja Fuzzy Shift-Or w Pythonie

```
def hamming_distance(s1: str, s2: str) -> int:
    """
    Oblicza odległość Hamminga między dwoma ciągami znaków.

    Args:
        s1: Pierwszy ciąg znaków
        s2: Drugi ciąg znaków

    Returns:
        Odległość Hamminga (liczba pozycji, na których znaki się różnią)
        Jeśli ciągi mają różne długości, zwraca -1
    """
    if len(s1) != len(s2):
        return -1

    cnt = 0
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            cnt += 1

    return cnt


def set_nth_bit(n: int) -> int:
    """
    Zwraca maskę bitową z ustawionym n-tym bitem na 1.

    Args:
        n: Pozycja bitu do ustawienia (0-indeksowana)

    Returns:
        Maska bitowa z n-tym bitem ustawionym na 1
    """
    return 1 << n


def nth_bit(m: int, n: int) -> int:
    """
```

```
    Zwraca wartość n-tego bitu w masce m.

    Args:
        m: Maska bitowa
        n: Pozycja bitu do odczytania (0-indeksowana)

    Returns:
        Wartość n-tego bitu (0 lub 1)
    """
    return (m >> n) & 1


def make_mask(pattern: str) -> list:
    """
    Tworzy tablicę masek dla algorytmu Shift-Or.

    Args:
        pattern: Wzorzec do wyszukiwania

    Returns:
        Tablica 256 masek, gdzie każda maska odpowiada jednemu znakowi ASCII
    """
    m = [0xFF] * 256

    for j, c in enumerate(pattern):
        m[ord(c)] &= ~set_nth_bit(j)

    return m


def fuzzy_shift_or(text: str, pattern: str, k: int = 2) -> list[int]:
    """
    Implementacja przybliżonego wyszukiwania wzorca przy użyciu algorytmu Shift-Or.

    Args:
        text: Tekst do przeszukania
        pattern: Wzorzec do wyszukiwania
        k: Maksymalna dopuszczalna liczba różnic (odległość Hamminga)

    Returns:
        Lista pozycji (0-indeksowanych), na których znaleziono wzorzec
        z maksymalnie k różnicami
    """
    if len(pattern) == 0 or len(pattern) > len(text) or k < 0:
        return []

    res = []
    m = make_mask(pattern)
    s = [~0] * (k + 1)

    for i, c in enumerate(text):
        prev = s[0]
        s[0] = ((s[0] << 1) | m[ord(c)])

        for d in range(1, k + 1):
            temp = s[d]
            s[d] = ((s[d] << 1) | m[ord(c)]) & (prev << 1)
            prev = temp
```

```
        if nth_bit(s[k], len(pattern) - 1) == 0:
            res.append(i - len(pattern) + 1)

    return res
```

Odległość Levenshteina

Odległość Levenshteina to klasyczna miara podobieństwa dwóch ciągów znaków. Definiuje się ją jako minimalną liczbę operacji edycyjnych (wstawienie, usunięcie, zamiana znaku) potrzebnych do przekształcenia jednego ciągu w drugi.

Algorytm opiera się na programowaniu dynamicznym — konstruowana jest macierz $dp[i][j]$ oznaczająca koszt przekształcenia pierwszych i znaków ciągu $s1$ w pierwsze j znaków ciągu $s2$. Komórki wypełniane są rekurencyjnie na podstawie operacji edycyjnych.

Złożoność czasowa algorytmu to $O(m \cdot n)$, a pamięciowa również $O(m \cdot n)$, gdzie m i n to długości ciągów wejściowych.

Listing 2: Obliczanie odległości Levenshteina w Pythonie

```
def levenshtein_distance(s1: str, s2: str) -> int:
    """
    Oblicza odległość Levenshteina między dwoma ciągami znaków.

    Args:
        s1: Pierwszy ciąg znaków
        s2: Drugi ciąg znaków

    Returns:
        Odległość Levenshteina (minimalna liczba operacji wstawienia, usunięcia
        lub zamiany znaku potrzebnych do przekształcenia s1 w s2)
    """
    m, n = len(s1), len(s2)

    # Przypadki brzegowe
    if m == 0:
        return n
    if n == 0:
        return m

    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Inicjalizacja pierwszego wiersza i kolumny
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Wypełnianie macierzy
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if s1[i - 1] == s2[j - 1] else 1
            dp[i][j] = min(
                dp[i - 1][j] + 1,       # usunięcie
                dp[i][j - 1] + 1,       # wstawienie
                dp[i - 1][j - 1] + cost # zamiana
            )

    return dp[m][n]
```

Algorytm Aho-Corasicka

Algorytm Aho-Corasicka służy do jednoczesnego wyszukiwania wielu wzorców w jednym przebiegu przez tekst. Działa w czasie liniowym względem długości tekstu oraz sumarycznej długości wszystkich wzorców.

Zbudowany jest z automatu deterministycznego, który łączy drzewo Trie ze wskaźnikami niepowodzenia (failure links), co pozwala na szybkie przejścia przy nieudanym dopasowaniu. Każdy wierzchołek przechowuje listę wzorców kończących się w tym miejscu.

Złożoność czasowa: $O(N + M + Z)$, gdzie:

- N — długość tekstu wejściowego,
- M — łączna długość wszystkich wzorców,
- Z — liczba znalezionych dopasowań.

Listing 3: Algorytm Aho-Corasicka w Pythonie

```
from collections import deque
from typing import Dict, List, Optional, Tuple

class AhoCorasickNode:
    def __init__(self):
        self.children: Dict[str, AhoCorasickNode] = {}
        self.fail: Optional[AhoCorasickNode] = None
        self.output: List[str] = []

class AhoCorasick:
    def __init__(self, patterns: List[str]):
        self.root = AhoCorasickNode()
        self.patterns = [p for p in patterns if p]  # usuń puste wzorce
        self._build_trie()
        self._build_failure_links()

    def _build_trie(self):
        for pattern in self.patterns:
            node = self.root
            for char in pattern:
                if char not in node.children:
                    node.children[char] = AhoCorasickNode()
                node = node.children[char]
            node.output.append(pattern)

    def _build_failure_links(self):
        queue = deque()
        for child in self.root.children.values():
            child.fail = self.root
            queue.append(child)

        while queue:
            current_node = queue.popleft()
            for char, child in current_node.children.items():
                fail_node = current_node.fail
                while fail_node and char not in fail_node.children:
                    fail_node = fail_node.fail
                child.fail = fail_node.children[char] if fail_node and char in
                    fail_node.children else self.root
                child.output += child.fail.output
                queue.append(child)
```

```
    def search(self, text: str) -> List[Tuple[int, str]]:
        result = []
        node = self.root

        for i, char in enumerate(text):
            while node and char not in node.children:
                node = node.fail
            if not node:
                node = self.root
                continue
            node = node.children[char]
            for pattern in node.output:
                result.append((i - len(pattern) + 1, pattern))
        return result
```

Wyszukiwanie wzorca dwuwymiarowego (2D)

Algorytm dwuwymiarowego wyszukiwania wzorca znajduje wystąpienia mniejszej macierzy znaków (wzorca) wewnątrz większej macierzy (tekstu). W tym podejściu wykorzystujemy algorytm Aho-Corasicka do znajdowania kolumn wzorca w kolumnach tekstu. Następnie sprawdzamy, czy znalezione kolumny występują w odpowiedniej kolejności w sąsiadujących wierszach.

Złożoność czasowa: $\mathcal{O}(HW + hw + Z)$, gdzie:

- H, W — wysokość i szerokość tekstu,
- h, w — wysokość i szerokość wzorca,
- Z — liczba dopasowań.

Listing 4: Wyszukiwanie wzorca 2D z użyciem Aho-Corasicka

```
from typing import List, Tuple
from .aho_corasick_algorithm import AhoCorasick

def transpose(matrix: List[str]) -> List[str]:
    return ["".join(row[i] for row in matrix) for i in range(len(matrix[0]))]

def find_pattern_in_column(text_column: str, pattern_columns: list[str]) -> list[
    tuple[int, int]]:
    """
    Wyszukuje wszystkie kolumny wzorca w kolumnie tekstu.
    """
    ac = AhoCorasick(pattern_columns)
    matches = ac.search(text_column)
    return [(i, k) for (i, p) in matches for k, pat in enumerate(pattern_columns)
        if pat == p]

def find_pattern_2d(text: list[str], pattern: list[str]) -> list[tuple[int, int]]:
    """
    Wyszukuje wzorzec dwuwymiarowy w tekście dwuwymiarowym.

    Args:
        text: Tekst jako lista wierszy (ciągów znaków)
        pattern: Wzorzec jako lista wierszy

    Returns:
        Lista współrzędnych (i, j)     lewy górny róg dopasowania wzorca w tekście.
    """
```

```
    if not text or not pattern or not text[0] or not pattern[0]:
        return []

    H, W = len(text), len(text[0])
    h, w = len(pattern), len(pattern[0])
    if h > H or w > W:
        return []

    text_cols = transpose(text)
    pattern_cols = transpose(pattern)

    T = [[-1] * W for _ in range(H)]

    for j, col_text in enumerate(text_cols):
        matches = find_pattern_in_column(col_text, pattern_cols)
        for i, pattern_col_idx in matches:
            if 0 <= i <= H - h:
                T[i][j] = pattern_col_idx

    result = []
    expected_sequence = list(range(w))

    for i in range(H - h + 1):
        row = T[i]
        for j in range(W - w + 1):
            if row[j : j + w] == expected_sequence:
                result.append((i, j))
    return result
```