Równoważność cykliczna ciągów

Definicja problemu i przedstawienie rozwiązań

Mikołaj Juda

2023

W referacie przedstawiono problem równoważności cyklicznej ciągów oraz różne algorytmy do jego rozwiązywaia razem z implementacją w języku Python. Pokrótce omówiono algorytm naiwny oraz algorytm korzystający z wyszukiwania wzorca. Przedstawiono również szybki algorytm sprawdzania równoważności list cyklicznych Shiloacha(1979)[1] zmodyfikowany do obsługi ciągów numerowanych od 0[3]. oraz szczegółowo wyjaśniono dowód jego poprawności i analizę złożoności obliczeniowej.

Spis treści

1 Definicja problemu

Dane są dwa ciągi $A = (a_0, \dots, a_{n-1})$ oraz $B = (b_0, \dots, b_{n-1})$ długości n. A i B są *równoważne cyklicznie* ($A \equiv B$), gdy są równe w sensie list cyklicznych tzn.

Definicja 1.1 (Równoważność cykliczna)**.**

$$A \equiv B \iff \exists_{k_0 \in \mathbb{Z}} \forall_{k \in \{0, \dots, n-1\}}\ a_{(k_0+k) \pmod n} = b_k$$

Dla wygody dalszego zapisu oznaczmy:

$$a_k \coloneqq a_{k \pmod n},\ b_k \coloneqq b_{k \pmod n} \text{ dla wszystkich } k \geq n$$

Zdefiniujmy A_k jako listę powstałą z przesunięcia cyklicznego ciągu A takiego, że a_k jest pierwszym elementem ciągu A_k. Analogicznie dla B_k.[1]

$$A_k = [a_k, \dots, a_n, a_0, \dots, a_{k-1}]$$
$$B_k = [b_k, \dots, b_n, b_0, \dots, b_{k-1}]$$

Definicję Definicja 1.1 można przedstawić równoważnie jako:

Definicja 1.2 (Równoważność cykliczna)**.**

$$A \equiv B \iff \exists_{k_0 \in \mathbb{Z}}\ A_{k_0} = B$$

Podsumowując, problem brzmi: „Czy istnieje takie przesunięcie cykliczne jednego ciągu, że jest po nim równy drugiemu ciagowi?”

[1] $A_0 = [a_0, \dots, a_{n-1}]$, oraz $B_0 = [b_0, \dots, b_{n-1}]$

2 Algorytm naiwny

Z Definicji 1.1 można łatwo zauważyć, że

Lemat 2.1. *Jeżeli nie istnieje* $k_0 \in \{0, \ldots, n-1\}$ *spełniające warunek:*

$$\forall_{k \in \{0,\ldots,n-1\}}\ a_{k_0+k} = b_k$$

to nie istnieje $k_0 \in \mathbb{Z}$ *spełniające ten warunek.*

Dowód. Oczywiste. ∎

Wniosek 2.1. *Żeby ustalić istnienie* k_0 *z Definicji 1.1 wystarczy sprawdzić czy*

$$\exists_{k_0 \in \{0,\ldots,n-1\}} \forall_{k \in \{0,\ldots,n-1\}}\ a_{k_0+k} = b_k$$

Algorytm naiwny sprawdza dla każdego $l \in \{0, \ldots, n-1\}$ czy

$$\forall_{k \in \{0,\ldots,n-1\}}\ a_{l+k} = b_k$$

Jeśli trafi na l spełniające warunek to mamy $k_0 = l$ i algorytm zwraca `True`, w przeciwnym wypadku zwraca `False`. Algorytm ma złożoność kwadratową.[3]

2.1 Implementacja

```python
def rownowazne_cyklicznie(a: list, b: list) -> bool:
    if len(a) != len(b):
        return False
    n = len(a)
    for l in range(n):
        for k in range(n):
            if a[(l + k) % n] != b[k]:
                break
        else:
            return True
    return False
```

3 Algorytm wykorzystujący wyszukiwanie wzorca

Utwórzmy listę
$$AA = [a_0, \ldots, a_{n-1}, a_0, \ldots, a_{n-1}]$$
i zauważmy, że dowolny spójny podciąg AA o długości n rozpoczynający się od indeksu k ma postać A_k, czyli jest przesunięciem cyklicznym ciągu A.

Uwaga 3.1. Łatwo zauważyć, że
$$\{ A_0, \ldots, A_{n-1} \} = \{ A_{\mathbb{Z}} \}$$
Zatem jeśli sprawdzimy czy B jest spójnym podciągiem AA to otrzymamy rozwiązanie problemu równoważnosci cyklicznej.
Można więc wykorzystać tytaj dowolny algorytm wyszukiwania wzorca, jednakże naiwny algorytm wyszukiwania wzorca sprowadza się do wcześniej przedstawionego algorytmu naiwnego i ma złożoność kwadratową. Wykorzystanie algorytmu wyszukiwania wzorca o liniowej złożoności obliczeniowej umożliwia sprawdzenie równoważności cyklicznej w czasie liniowym.[2]
Wersja wykorzystujaca algorytm Knutha-Morrisa-Pratta wykonuje około 5n porównań.[1] Wymaga on jednak dodatkowej pamięci (liniowa złożoność pamięciowa).

3.1 Implementacja[2]

```python
def rownowaznosc_cykliczna(A: list, B: list) -> bool:
    if len(A) != len(B):
        return False
    AA = A + A
    return KMPSearchExists(B, AA)


def KMPSearchExists(pattern, word) -> bool:
    M = len(pattern)
    N = len(word)
    lps = [0] * M
    j = 0
    computeLPSArray(pattern, M, lps)
    i = 0
    while i < N:
        if pattern[j] == word[i]:
            i += 1
            j += 1
        if j == M:
            return True
        elif i < N and pattern[j] != word[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return False


def computeLPSArray(pat, M, lps):
    len = 0
    lps[0] = 0
    i = 1
    while i < M:
        if pat[i] == pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            if len != 0:
                len = lps[len - 1]
            else:
                lps[i] = 0
                i += 1
```

[2]Użyta implementacja algorytmu KMP (z modyfikacjami) pochodzi ze strony: https://www.geeksforgeeks.org/python-program-for-kmp-algorithm-for-pattern-searching-2/

4 Algorytm szybki[3]

Algorytm wymaga istnienia porządku liniowego na zbiorze

$$\{ a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1} \}$$

Niech $a \leq b$ oznacza, że a poprzedza lub jest równe b w tym porządku, a $a < b$ oznacza $x \leq b \wedge a \neq b$.
Wtedy porządek leksykograficzny na zbiorze

$$\{ A_0, \ldots, A_{n-1}, B_0, \ldots, B_{n-1} \}$$

jest dobrym porządkiem.
Niech $A_i \preceq B_j$ oznacza, że A_i poprzedza lub jest równe B_j w porządku leksykograficznym, a $A_i \prec B_j$ oznacza $A_i \preceq B_j \wedge A_i \neq B_j$.
Niech

$$A_{i_0} = \min(\{ A_0, \ldots, A_{n-1} \}) \text{ tzn. } \forall_{0 \leq i < n-1} A_{i_0} \preceq A_i$$

analogicznie

$$B_{j_0} = \min(\{ B_0, \ldots, B_{n-1} \})$$

[4]

Lemat 4.1.

$$A \equiv B \iff A_{i_0} = B_{j_0}$$

Dowód. Oczywiste. ■

Lemat 4.2.

$$\forall_{0 \leq l < k}\ a_{i+l} = b_{j+l} \wedge a_{i+k} < b_{j+k} \implies \forall_{0 \leq l \leq k}\ A_{i+l} \prec B_{j+l}$$

Dowód. Trywialne. ■

Zdefiniujmy

$$D_A = \{ i : \exists_j\ B_j \prec A_i \}$$

analogicznie

$$D_B = \{ j : \exists_i\ A_i \prec B_j \}$$

tzn. D_A to zbiór indeksów wyznaczających takie przesunięcia cykliczne A, dla których istnieje przesunięcie cykliczne B poprzedzające je w porządku leksykograficznym. Analogicznie dla B.

[3] Będę używał nieco innych oznaczeń niż w [1]
[4] Lexicographically minimal string rotation

Zdefiniujmy

$$i_D = \begin{cases} -1 & \text{jeżeli } D_A = \emptyset \\ \max(D_A) & \text{w przeciwnym przypadku} \end{cases}$$

$$j_D = \begin{cases} -1 & \text{jeżeli } D_B = \emptyset \\ \max(D_B) & \text{w przeciwnym przypadku} \end{cases}$$

Lemat 4.3.

$$D_A \supseteq \{ 0, \dots, n-1 \} \vee D_B \supseteq \{ 0, \dots, n-1 \} \implies A \neq B$$

Dowód. Dowód wynika z Lematu 4.1

$$A \equiv B \implies i_0 \notin D_A \wedge j_0 \notin D_B$$

■

4.1 Algorytm

Główną częścią algorytmu jest szybkie zbieranie indeksów przesunięć cyklicznych, które powinny być w zbiorach D_A i D_B.[5] dopóki nie zajdzie jeden z poniższych warunków stopu:

(1) $D_A \supseteq \{ 0, \dots, n-1 \}$

(2) $D_B \supseteq \{ 0, \dots, n-1 \}$

(3) Znajdziemy takie i oraz j, że $\forall_{0 \leq k \leq n-1}\ a_{i+k} = b_{j+k}$ (tzn. $A \equiv B$)

Głównym elementem algorytmu będzie procedura COMPARE, która znajduje pierwszy indeks k, dla którego $a_{i+k} \neq b_{j+k}$ i w zależności od tego, czy $a_{i+k} < b_{j+k}$ dodaje do D_B zbiór $\{ j, \dots, j+k \}$ albo do D_A zbiór $\{ i, \dots, i+k \}$ (Patrz Lemat 4.2). Jeśli nie znajdzie nierówności to zwraca $A \equiv B$.
Procedura ROWNCYKL wykonuje procedurę COMPARE dopóki nie zajdzie jeden z powyższych warunków.

[5] W implementacji te zbiory są niejawne.

$D_a \leftarrow \emptyset$
$D_b \leftarrow \emptyset$
$i_D \leftarrow -1$
$j_D \leftarrow -1$
procedure COMPARE(i, j)
    for $k = 0, \ldots, n-1$ **do**
        if $a_{i+k} \neq b_{j+k}$[6] **then**
            if $a_{i+k} < b_{j+k}$ **then**
                $D_a \leftarrow D_a \cup \{ i, \ldots, i+k \}$
            else
                $D_b \leftarrow D_b \cup \{ j, \ldots, j+k \}$
            end if
            $i_D \leftarrow i + k$
            $j_D \leftarrow j + k$
            return
        end if
    end for
    return $A \equiv B$
end procedure
procedure ROWNCYKL(A, B)
    $D_a \leftarrow \emptyset$
    $D_b \leftarrow \emptyset$
    $i_D \leftarrow -1$
    $j_D \leftarrow -1$
    while $D_A \not\supseteq \{ 0, \ldots, n-1 \} \wedge D_B \not\supseteq \{ 0, \ldots, n-1 \}$ **do**
        if COMPARE$(i_D + 1, j_D + 1)$ returns $A \equiv B$ **then**
            return $A \equiv B$
        end if
    end while
    return $A \not\equiv B$
end procedure

[6] W analizie złożoności będziemy liczyć te porównania

4.2 Poprawność

Lemat 4.4 (Niezmiennik). *Po każdym wykonaniu procedury* COMPARE D_A *oraz* D_B *mają postać*

$$D_A = \{\, 0, \ldots, i_D \,\}$$
$$D_B = \{\, 0, \ldots, j_D \,\}$$

tzn. w D_A *nie brakuje żadnego elementu między* 0 *a* i_D*, analogicznie w* D_B *nie brakuje żadnego elementu między* 0 *a* j_D*.*

Dowód. Dowód wynika trywialnie z faktu, że jedyny sposób, w jaki modyfikowane jest D_A to

$$D_a \leftarrow D_a \cup \{\, i_D + 1, \ldots, i_D + k \,\}$$

analogicznie dla D_B

$$D_b \leftarrow D_b \cup \{\, j_D + 1, \ldots, j_D + k \,\}$$

■

Wniosek 4.1. *Z Lematu 4.4 wynika, że nie musimy przechowywać zbiorów* D_A *i* D_B*, wystarczy nam jedynie* i_D *oraz* j_D*.*

Twierdzenie 4.1. *Procedura* ROWNCYKL *ma własność stopu.*

Dowód. Jeżeli dla jakiegoś i oraz j procedura COMPARE zwróci $A \equiv B$ to ROWNCYKL się zatrzymuje.
Jeżeli procedura COMPARE nie zwróci $A \equiv B$ dla żadnego i oraz j to z każdym wywołaniem COMPARE $i_D + j_D$ rośnie co najmniej o 1, więc w końcu (z Lematu 4.4) warunek stopu 1 lub warunek stopu 2 zostanie spełniony i procedura ROWNCYKL się zatrzyma. ■

Twierdzenie 4.2. *Procedura* ROWNCYKL *zwraca* $A \equiv B$ *wtedy i tylko wtedy gdy* $A \equiv B$*.*

Dowód. Jeżeli procedura ROWNCYKL zwróciła $A \equiv B$ to znaczy, że procedura COMPARE wykazała równość dwóch przesunięć cyklicznych A i B, a więc $A \equiv B$. Jeżeli procedura ROWNCYKL nie zwróci $A \equiv B$ to zwróci $A \not\equiv B$, gdyż z Twierdzenia 4.1 wiemy, że zawsze się zatrzymuje. Zwrócenie $A \not\equiv B$ oznacza, że wystąpił warunek stopu 1 lub warunek stopu 2, więc z Lematu 4.3 $A \not\equiv B$. ■

4.3 Złożoność

Definicja 4.1. Wykonanie procedury COMPARE nazwiemy *udanym*, jeżeli zwróci $A \equiv B$. (Wykonanie, które nie jest udane jest *nieudane*)

Definicja 4.2. Oznaczmy D_A oraz D_B bezpośredionio przed ostatnim wykonaniem procedury COMPARE jako odpowiednio D_A^* oraz D_B^*.

Lemat 4.5. *Łączna liczba porównań wykonanych przez nieudane wywołania procedury* COMPARE *wynosi* $|D_A| + |D_B|$

Dowód. Każde wywołanie procedury COMPARE za wyjątkiem ostatniego jest nieudane i zwiększa $|D_A| + |D_B|$ o dokładnie tyle ile wykonało porównań. Wynika to trywialnie z definicji procedury COMPARE. ■

Lemat 4.6. $\left|D_A^*\right| + \left|D_B^*\right| \leq 2n - 2$

Dowód. Jeśli $\left|D_A^*\right| + \left|D_B^*\right| \geq 2n - 1$ to oznacza, że

$$\{0, \ldots, n-1\} \subset D_A^* \vee \{0, \ldots, n-1\} \subset D_B^*$$

co nie jest możliwe (patrz Definicja 4.2). ■

Twierdzenie 4.3. *Algorytm wykonuje co najwyżej* $3n - 2$ *porównań.*

Dowód. Dowód wynika z Lematu 4.5 oraz Lematu 4.6, gdyż ostatnie wywołanie procedury COMPARE może wykonać co najwyżej n porównań. ■

Wniosek 4.2. *Algorytm ma pesymistyczną złożoność czasową* $\mathcal{O}(n)$ *i dodatkową złożoność pamięciową* $\mathcal{O}(1)$ *(z Wniosku 4.1)*

4.4 Implementacja

Bibliografia

[1] Yossi Shiloach. „A fast equivalence-checking algorithm for circular lists". W: *Information Processing Letters* 8.5 (1979), s. 236–238. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90114-5. URL: https://www.sciencedirect.com/science/article/pii/0020019079901145.

[2] Lech Banachowski, Krzysztof Diks i Wojciech Rytter. *Algorytmy i struktury danych*. pol. Wyd. 5. Warszawa: Wydawnictwa Naukowo-Techniczne, 2006. ISBN: 8320432243.

[3] *Algorytmy i struktury danych/Wstęp: poprawność i złożoność algorytmu*. 2020. URL: https://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych/Wst%C4%99p:_poprawno%C5%9B%C4%87_i_z%C5%82o%C5%BCono%C5%9B%C4%87_algorytmu.