Algorytmy tekstowe

Krzysztof Turowski

1 Notacja

1.1 Podstawowe pojęcia

Definicja 1.1. ***Alfabet*** $\mathcal{A}$ – (skończony) zbiór symboli.

W większości algorytmów rozmiar alfabetu jest stały i pomijany w analizie złożoności.

Definicja 1.2. ***Słowo*** $w \in \mathcal{A}^*$ to ciąg zbudowany nad alfabetem $\mathcal{A}$.

Słowo puste jest oznaczane symbolem ε. Zbiór słów niepustych to $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$.

Zbiór słów długości n oznaczamy $\mathcal{A}_n$ i definiujemy jako $\mathcal{A}_0 = \{\varepsilon\}$, $A_{n+1} = \biguplus_{x \in \mathcal{A}_n, y \in \mathcal{A}} xy$.

Numerację symboli w słowie zaczynamy od 1, więc $w[i]$ (albo w_i) jest i-tym symbolem w słowie w.

Definicja 1.3. ***Długość słowa*** to funkcja $|\cdot| : \mathcal{A}^* \to \mathbb{N}$ taka, że $|w| = n$ wtedy i tylko wtedy, gdy $w \in \mathcal{A}^n$.

Definicja 1.4. Słowo u jest ***podsłowem*** (ang. *factor*) słowa w, gdy istnieją słowa $v_1 v_2 \in \mathcal{A}^*$ takie, że $w = v_1 u v_2$. Podsłowo jest ***właściwe***, gdy $v_1 v_2 \neq \varepsilon$.

Definicja 1.5. Słowo u jest ***prefiksem*** (***sufiksem***) słowa w, gdy istnieje słowo $v \in \mathcal{A}^*$ takie, że $w = uv$ ($w = vu$). Prefiks (sufiks) jest ***właściwy***, gdy $v \neq \varepsilon$.

Definicja 1.6. Słowo u jest ***podciągiem*** słowa w, gdy istnieją liczby $1 \leq i_1 < i_2 < \ldots < i_k \leq |w|$ takie, że $u = w[i_1]w[i_2]\ldots w[i_k]$.

Definicja 1.7. Słowo $\bar{w}$ jest ***odwrotnością*** słowa w, gdy dla $n = |w|$ mamy $\bar{w} = w[n]w[n-1]\ldots w[1]$.

Definicja 1.8. Słowo w jest ***palindromem***, jeśli istnieją słowa $u, v \in \mathcal{A}^*$ takie, że $|u| \leq 1$ i $w = \bar{v}uv$.

1.2 Porządki i odległości

Złożoność obliczeniowa algorytmów tekstowych obliczana jest przy przyjęciu dwóch (binarnych) operacji atomowych na parach liter z alfabetu: $=$ oraz $\leq$.

Definicja 1.9. Relacja $\preceq$ jest relacją ***porządku prefiksowego*** tj. $u \preceq v$ wtedy i tylko wtedy, gdy u jest prefiksem u.

Definicja 1.10. Relacja $\preceq_R$ jest relacją ***porządku wojskowego*** (*radix*) tj. $u \preceq_R v$ wtedy i tylko wtedy, gdy:

- albo $|u| < |v|$,
- albo $|u| = |v|$ oraz istnieje $1 \leq k \leq |u|$ takie, że $u[k] < v[k]$ i dla wszystkich $1 \leq j < k$ zachodzi $u[j] = v[j]$.

Definicja 1.11. Relacja $<$ jest relacją ***porządku leksykograficznego*** tj. $u < v$ wtedy i tylko wtedy, gdy:

- albo u jest prefiksem v,
- albo istnieje $1 \leq k \leq |u|$ takie, że $u[k] < v[k]$ i dla wszystkich $1 \leq j < k$ zachodzi $u[j] = v[j]$.

Wniosek 1.12. Porządek leksykograficzny i wojskowy są porządkami liniowymi, rozszerzającymi porządek prefiksowy. Dla słów równego długości u, v zachodzi $u < v$ wtedy i tylko wtedy, gdy $u \preceq_R v$.

1.3 Okresy i słowa pierwotne

Definicja 1.13. Liczba $p \in \mathbb{N}_+$ jest ***okresem*** słowa w, jeśli dla każdego $1 \leq i \leq |w| - p$ zachodzi $w[i] = w[i + p]$.
Najmniejszy okres słowa w oznaczamy przez $p(w)$. Z definicji $|w|$ jest okresem słowa, więc $p(w)$ jest dobrze zdefiniowane.

Definicja 1.14. Słowo u jest ***prefikso-sufiksem*** (ang. *border*) słowa w, jeśli istnieją słowa v_1, v_2 takie, że $w = uv_1 = v_2u$.
Z definicji ε jest prefikso-sufiksem każdego słowa w.

Problem 1.1. Pokaż, że następujące warunki są równoważne:

(i) w ma okres p,
(ii) w jest podsłowem pewnego v^k dla $|v| = p$ i $k \geq 1$,
(iii) $w = (uv)^k v$ dla $|uv| = p$, $v \neq \varepsilon$ i $k \geq 1$,
(iv) w ma prefikso-sufiks długości $|w| - p$.

Definicja 1.15. Słowo w jest ***pierwotne***, jeśli nie istnieje słowo v oraz liczba całkowita $k \geq 2$ takie, że $w = v^k$.

Problem 1.2. Słowo w jest słowem pierwotnym wtedy i tylko wtedy, gdy $p(w) = |w|$ lub $p(w)$ nie dzieli $|w|$.

Problem 1.3. Pokaż, że następujące twierdzenia są równoważne:

(i) $p(w^2) = |w|$,
(ii) w jest słowem pierwotnym,
(iii) w^2 zawiera dokładnie dwa wystąpienia w.

Twierdzenie 1.16 (Słaby lemat o okresowości)**.** *Jeśli słowo w ma okresy p i q takie, że $|w| \geq p + q$, to w ma również okres $NWD(p, q)$.*

Dowód. Bez straty ogólności załóżmy, że $p \geq q$. Najpierw wykażemy, że jeżeli słowo w ma okresy p i q oraz $|w| \geq p + q$, to w ma również okres $p - q$.

Dla $1 \leq i \leq q$ mamy $a[i] = a[i+p] = a[i+p-q]$, ponieważ $1 \leq i \leq i+p-q \geq i+p \leq |w|$. Podobnie dla $q + 1 \leq i \leq |w| - (p - q)$ mamy $a[i] = a[i - q] = a[i + p - q]$, ponieważ $1 \leq i - q \leq i + p - q \geq |w|$.

Z algorytmu Euklidesa wynika wprost, że iterując to rozumowanie możemy pokazać, że w ma również okres $NWD(p, q)$. □

Twierdzenie 1.17 (Silny lemat o okresowości)**.** *Jeśli słowo w ma okresy p i q takie, że $|w| \geq p + q - NWD(p, q)$, to w ma również okres $NWD(p, q)$.*

Dowód. Po pierwsze, jeśli słowo w ma okresy $0 < q < p \leq |w|$, to prefiks i sufiks w o długości $|w| - q$ mają okresy $p - q$. Dla prefiksu wystarczy zauważyć, że dla dowolnego $1 \leq i \leq |w| - p$ zachodzi $w[i] = w[i + p] = w[i + p - q]$ – a to właśnie jest definicja okresu dla słowa $w[1..(|w| - q)]$.

Po drugie, jeśli w ma okres q i istnieje podsłowo v słowa w z $|v| \geq q$ takim, że r jest okresem v i r dzieli q, to w ma okres r.

Niech $r = NWD(p, q)$. Dowód przebiega przez indukcję ze względu na $s = \frac{p+q}{r}$. Dla $p = q = r$ – więc twierdzenie jest oczywiście spełnione np. dla $s = 2$.

Dla $s > 2$ i $q < p$ weźmy słowo w mające okresy p i q takie, że $|w| \geq p + q - r$. Niech $u = w[1..q]$ oraz $w = uv$. Wówczas z pierwszego faktu wiemy, że v ma okres $p - q$. Jednocześnie v jest podsłowem w oraz $|v| = |w| - q \geq p - r \geq q$, więc v ma również okres q.

Dalej wiemy, że $r = NWD(p - q, q)$, $s > \frac{(p-q)+q}{r}$ oraz

$$|v| = |w| - q \geq (p + q - r) - q = (p - q) + q - NWD(p - q, q).$$

Z założenia indukcyjnego v ma zatem okres r. Ostatecznie z drugiego faktu wiemy, że w ma też okres r. □

Problem 1.4. Korzystając ze słów Fibonacciego pokaż, że warunku $|w| \geq p+q-NWD(p, q)$ w silnym lemacie o okresowości nie da się poprawić.

Definicja 1.18. Liczba $ord(w)$ jest ***rzędem*** słowa w, jeśli $ord(w) = |w|/p(w)$.

Problem 1.5. Pokaż, że słowo Fibonacciego jest słowem pierwotnym.

Dowód. Załóżmy, że istnieją $u, k : u^k = Fib_i, k > 1$. Wprowadźmy oznaczenie $u = st$ gdzie $s = u[1 \dots |u| - 2]$ oraz $t = u[|u| - 1, |u|]$.

Na podstawie poprzedniego zadania wiemy, że słowa Fibonacciego bez ostatnich dwu znaków są palindromem. Dlatego $(st)^{k-1}s$ powinno też być palindromem. Z tego wynika, że $s = \overline{s}$ oraz $t = \overline{t}$. To może zachodzić tylko dla dwu przypadków: $t = 00$ i $t = 11$. Na ćwiczeniach jednak zostało pokazane, że dla słów Fibonacciego jedyne dwie opcje są $t = 01$ i $t = 10$. Otrzymujemy sprzeczność, która pokazuje, że żadne słowo Fibonacciego nie można zapisać jako $u^k, k > 1$, więc każde słowo Fibonacciego jest słowem pierwotnym. □

1.4 Słowa sprzężone

Definicja 1.19. Słowa v, w są ***sprzężone*** wtedy i tylko wtedy, gdy istnieją słowa u_1, u_2 takie, że $v = u_1u_2$ i $w = u_2u_1$.

Wniosek 1.20. Relacja sprzężenia (cyklicznego obrotu) jest relacją równoważności, więc definiuje klasy równoważności w $\mathcal{A}^*$.

Problem 1.6. Pokaż, ile elementów ma klasa równoważności dla słowa v.

Problem 1.7. Pokaż dowód małego twierdzenia Fermata na bazie wiedzy, że słowo pierwotne v należy do klasy równoważności o mocy $|v|$.

Dowód. Przypomnijmy na wstępie dowodu wypowiedź małego twierdzenia Fermata: jeżeli p jest liczbą pierwszą, a n dowolną liczbą naturalną, to $p|(n^p - n)$.

Zdefiniujmy taką relację na słowach, że x jest w relacji z y, jeżeli tylko x jest cyklicznym przesunięciem y. Oczywiście jest to relacja równoważności. Rozważmy słowa unarne długości p, czyli słowa postaci a^p, gdzie a jest pewną literą. Niech K będzie zbiorem wszystkich nieunarnych słów długości p nad alfabetem $\{1, \cdots, n\}$. Wszystkie te słowa są pierwotne, ponieważ ich długość jest liczbą pierwszą oraz nie są unarne. Na podstawie założeń dostajemy, że każda klasa równoważności naszej relacji ma dokładnie p elementów. Dodatkowo zauważmy, że zbiór K ma dokładnie $n^p - n$ elementów. Ponieważ K może być podzielony na rozłączne podzbiory mające po p elementów, to $p|(n^p - n)$, co kończy dowód. □

Definicja 1.21. Słowo w jest ***słowem Lyndona*** wtedy i tylko wtedy, gdy jest minimalnym (leksykograficznie, wojskowo) słowem w ramach klasy sprzężenia.

1.5 Morfizmy i klasy słów

Definicja 1.22. Funkcja $f : \mathcal{A}^* \to \mathcal{B}^*$ jest ***morfizmem*** (***podstawieniem***) jeśli $f(xy) = f(x)f(y)$ dla wszystkich $x, y \in \mathcal{A}^*$.
Morfizm jest dosłowny (*literal*), gdy dla każdego $x \in \mathcal{A}$ zachodzi $|f(x)| = 1$.
Morfizm jest nieusuwający (*non-erasing*), gdy dla każdego $x \in \mathcal{A}$ zachodzi $f(x) \neq \varepsilon$.

1.5.1 Słowa Fibonacciego

Definicja 1.23 (Lothaire 2002, s. 10-11)**.** Słowo f_n jest ***słowem Fibonacciego***, gdy $f_0 = 0$, $f_1 = 01$ oraz $f_{k+2} = f_{k+1}f_k$ dla $k = 0, 1, \ldots$.
Równoważnie, $f_k = \phi^n(0)$ dla morfizmu $\phi(0) = 01$, $\phi(1) = 0$.

Problem 1.8. Dla $k \geq 1$ niech $u = f_k f_{k+1}$ i $v = f_{k+1}f_k$. Pokaż, że u powstaje z v przez zamianę dwóch ostatnich liter.

Problem 1.9. Jeśli dla $u \in \mathcal{A}^+$ słowo u^2 jest podsłowem pewnego f_k, to $|u|$ jest pewną liczbą Fibonacciego i u jest sprzężone z pewnym f_l.

1.5.2 Słowa Thuego-Morse'a

Definicja 1.24 (Lothaire 2002, s. 11)**.** Słowo u_n jest ***słowem Thuego-Morse'a***, gdy $u_0 = 0$, $v_0 = 1$ oraz $u_{k+1} = u_k v_k$, $v_{k+1} = v_k u_k$ dla $k = 0, 1, \ldots$.
Równoważnie, $u_k = \mu^n(0)$ dla morfizmu $\mu(0) = 01$, $\mu(1) = 10$.

Problem 1.10. Udowodnić że słowa Thuego-Morse'a u_n są słowami pierwotnymi.

Problem 1.11. Udowodnić że słowa Thuego-Morse'a u_{2n} są palindromami.

Dowód. Zgodnie z definicją rekurencyjną mamy $u_0 = 0, v_0 = 1$ oraz $u_{k+1} = u_k v_k$, $v_{k+1} = v_k u_k$. Rozumować będziemy indukcyjnie. Dla $n = 0$, słowo $u_0 = 0, v_0 = 1$ są oczywiście palindromami. Załóżmy zatem, że dla wszystkich $k < n$ prawdą jest, że u_{2k} oraz v_{2k} są palindromami. Będziemy chcieli wykazać, że jest to prawdą także dla u_{2n} i v_{2n}. Z definicji rekurencyjnej dostajemy, że $u_{2n} = u_{2n-1}v_{2n-1} = u_{2n-2}v_{2n-2}v_{2n-2}u_{2n-2}$. Ponieważ u_{2n_2} oraz v_{2n-2} są palindromami na mocy założenia indukcyjnego, to u_{2n} też jest palindromem. Dokładnie taki sam argument pokazuje, że v_{2n} również jest palindromem. Zatem zakończyliśmy dowód kroku indukcyjnego, a więc też cały dowód. □

Problem 1.12. Sprawdzić czy słowa Thuego-Morse'a zawierają podsłowa u^3 lub $(uv)^2u$ dla pewnych $u, v \in \mathcal{A}^+$.

1.6 Kody

Definicja 1.25. $X \subset \mathcal{A}^+$ jest ***kodem***, gdy dla dowolnych $x_1, \ldots, x_n, y_1, \ldots, y_m \in X$ jeśli dla $x_1 \ldots x_n = y_1 \ldots y_m$, to $n = m$ oraz $x_i = y_i$ dla $i = 1, \ldots, n$.

Definicja 1.26. $X \subset \mathcal{A}^+$ jest ***kodem prefiksowym*** gdy X jest kodem i dla żadnych $x, y \in X$ słowo x nie jest prefiksem y.

Problem 1.13. Zbiór $X \subset \mathcal{A}^+$ jest kodem dla $\mathcal{B}^*$ wtedy i tylko wtedy, gdy dowolny morfizm $\phi : \mathcal{B}^* \to \mathcal{A}^*$ indukuje iniekcję z $\mathcal{B}$ na X.

2 Dokładne dopasowanie wzorca

Problem 1: Dokładne dopasowanie wzorca
Wejście: Słowa $t, w \in \mathcal{A}^+$ ($\|t\| = n$, $\|w\| = m$)
Wyjście: TRUE jeśli w jest podsłowem t, FALSE w przeciwnym przypadku

Problem 2: Wyszukiwanie wszystkich wystąpień wzorca w tekście
Wejście: Słowa $t, w \in \mathcal{A}^+$ ($\|t\| = n$, $\|w\| = m$)
Wyjście: Zbiór liczb $S = \{1 \leq i \leq n : t[i..(i+m-1)] = w\}$.

2.1 Algorytm naiwny

Algorytm 1: Naiwne szukanie wzorca w w tekście t

```
def naive_string_matching(t, w, n, m):
  for i in range(n - m + 1):
    j = 0
    while j < m and t[i + j + 1] == w[j + 1]:
      j = j + 1
    if j == m:
      return True
  return False
```

Problem 2.1. Dla ustalonego n i m znajdź przykład słowa w i tekstu t z $|t| = n$, $|w| = m$, dla którego Algorytm 1 wykonuje najwięcej porównań.

Dowód. Najprostszym przykładem jest poszukiwanie wzorca $w = a^{m-1}b$ w tekście $t = a^n$. Dla takiego zestawu algorytm naiwny nie znajdzie dopasowania, a wcześniej będzie dopasowywał prawie całe słowo w, rozpoczynając od każdej pozycji słowa t. Prowadzi to do wykonania dokładnie $m(n - m + 1)$ porównań. □

Można również zaimplementować algorytm naiwny, który dopasowywałby wzorzec do tekstu od końca.

Algorytm 2: Naiwne szukanie wzorca w w tekście t

```
def brute_force(t, w, n, m):
  i = 1
  while i <= n - m + 1:
    j = m
    while j > 0 and t[i + j - 1] == w[j]:
      j = j - 1
    if j == 0:
      yield i
    i = i + 1
```

2.2 Algorytm Morrisa-Pratta

Algorytm naiwny dokonuje w razie wykrycia niedopasowania przesunięcia wzorca tekstu o 1. Załóżmy, że zachodzi $t[i+k+1] = w[k+1]$ dla pewnego k oraz $t[i+j] = w[j]$ dla $1 \leq j \leq k$ tj. mamy zgodność wzorca z tekstem na k literach.

Jasne jest jednak, że jeśli v nie jest prefikso-sufiksem $w[1..k]$, to istnieje pozycja $0 \leq l \leq k-1$ taka, że $v[|v|-l] \neq w[k-l] = t[i+k-l]$ – a zatem nie ma sensu dopasowywać w. Z kolei jeśli v jest prefikso-sufiksem $w[1..k]$, to z góry wiemy, że jest zgodne z $t[(i+k-|v|)..(i+k)]$.

Wobec tego najmniejsze sensowne przesunięcie to przejście do sprawdzania dla największego prefikso-sufisku $w[1..k]$ – czyli przesunięcie o $p(w[1..k])$.

Dla słowa w można zdefiniować tablice najdłuższych prefikso-sufiksów dla każdego prefiksu:

$$B[i] = \begin{cases} -1 & \text{dla } i = 0, \\ \max\{k \geq 0 : w[1 \ldots k] \text{ jest właściwym sufiksem } w[1 \ldots i]\} & \text{dla } 1 \leq i \leq |w|. \end{cases}$$

Lemat 2.1. *Jeśli u jest prefikso-sufiksem v i v jest prefikso-sufiksem w, to u jest prefikso-sufiksem w.*

Wniosek 2.2. Ciąg $(B[|w|], B[B[|w|]], \ldots, 0)$ zawiera długości wszystkich prefikso-sufiksów słowa w w kolejności malejącej.

Wniosek 2.3. Ciąg $(|w|, \ldots, |w| - B[B[|w|]], |w| - B[|w|])$ zawiera długości wszystkich okresów słowa w w kolejności malejącej.

Algorytm 3: Tablica prefikso-sufiksów

```
def prefix_suffix(w, m):
  B, t = [-1] + [0] * m, -1
  for i in range(1, m + 1):
    while t >= 0 and w[t + 1] != w[i]:
      t = B[t] # prefikso-sufiks to relacja przechodnia
    t = t + 1
    B[i] = t
  return B
```

Problem 2.2. Pokaż, jaki dokładny pesymistyczny czas działania ma Algorytm 3.

Algorytm 4: Algorytm Morrisa-Pratta

```
def morris_pratt_string_matching(t, w, m, n):
  B = prefix_suffix(w)
  i = 0
  while i <= n - m + 1:
    j = 0
    while j < m and t[i + j + 1] == w[j + 1]:
      j = j + 1
    if j == m:
      return True
    i, j = i + j - B[j], max(0, B[j])
  return False
```

Problem 2.3. Pokaż, ile razy w najgorszym razie pojedynczy symbol tekstu t może zostać porównany z wzorcem w Algorytmie 4.

Problem 2.4. Pokaż, ile dokładnie porównań wykonuje w najgorszym przypadku zmodyfikowany Algorytm 4, zwracający wszystkie wystąpienia (tj. pozycje indeksów początkowych) w w t.

2.3 Algorytm Knutha-Morrisa-Pratta

Przesunięcie o $p(w[1..k]) = k - B[k]$ nie musi być najlepsze możliwe. Jeśli niedopasowanie zaszło na znakach $t[i + k + 1] \neq w[j + 1]$ oraz wiadomo z analizy wzorca, że $w[k + 1] = w[k - p(w) + 1]$, to wiadomo, że nie znajdziemy wzorca przez przesunięcie tylko o $p(w)$. Za to możemy szukać prefikso-sufiksu w' dla słowa $w[1..j]$ takiego, że następuje po nim znak inny niż $w[j + 1]$ – czyli tzw. silnego prefikso-sufiksu.

Dla słowa w można zdefiniować tablice silnych prefikso-sufiksów następująco:

$$sB[i] = \begin{cases} \max\{0 \leq k < i : w[1 \ldots k] \text{ jest sufiksem } w[1 \ldots i] \\ \qquad\qquad \text{i } w[k+1] = w[i+1]\} & \text{dla } 1 \leq i \leq |w| - 1, \\ B[i] & \text{dla } i = |w|, \\ -1 & \text{w pozostałych przypadkach.} \end{cases}$$

Algorytm 5: Tablica silnych prefikso-sufiksów

```
def strong_prefix_suffix(w, m):
  sB, t = [-1] + [0] * m, -1
  for i in range(1, m + 1): # niezmiennik: t = B[i - 1]
    while t >= 0 and w[t + 1] != w[i]:
      t = sB[t]
    t = t + 1
    if i == m or w[t + 1] != w[i + 1]:
      sB[i] = t
    else:
      sB[i] = sB[t]
  return sB
```

Problem 2.5. Pokaż, jaki dokładny pesymistyczny czas działania ma Algorytm 5.

Algorytm 6: Algorytm Knutha-Morrisa-Pratta

```
def knuth_morris_pratt_string_matching(t, w, m, n):
  sB = strong_prefix_suffix(w)
  i = 0
  while i <= n - m + 1:
    j = 0
    while j < m and t[i + j + 1] == w[j + 1]:
      j = j + 1
    if j == m:
      return True
    i, j = i + j - sB[j], max(0, sB[j])
  return False
```

Problem 2.6. Pokaż, ile razy w najgorszym razie pojedynczy symbol tekstu t może zostać porównany z wzorcem w Algorytmie 6.

Problem 2.7. Pokaż, ile dokładnie porównań wykonuje w najgorszym przypadku zmodyfikowany Algorytm 6, zwracający wszystkie wystąpienia (tj. pozycje indeksów początkowych) w w t.

2.4 Algorytm Boyera-Moore'a

Algorytm Boyera-Moore'a polega na ulepszeniu naiwnego dopasowywania wzorca od prawej do lewej. Schemat jest taki sam jak dla algorytmów Morrisa-Pratta i Knutha-Morrisa-Pratta: wykonujemy dopasowanie dla danej pozycji okna, w razie znalezienia niedopasowania przesuwamy okno o pewną wartość.

Sprawdzanie dopasowania wzorca od prawej do lewej umożliwia, żeby algorytm wykonał $O\left(\frac{m}{n}\right)$ porównań np. dla $t = 0^n$, $w = 0^{m-1}1$. Wystarczy np. wiedzieć, że wzorzec nie jest okresowy tj. $p(w) = |w|$ oraz że zachodzi $t[i] \neq w[m]$, aby następne sprawdzanie rozpocząć od pozycji $i + m$. Zauważmy, że w algorytmie KMP zawsze musimy porównać każdą literę tekstu ze wzorcem.

W praktyce to powoduje, że algorytm Boyera-Moore'a jest zauważalnie szybszy niż KMP lub MP.

Sednem algorytmu Boyera-Moore'a są 2 niezmienniki:

- $cond_1(j, s)$ – dla każdego $j < k \leq m$ zachodzi $s \geq k$ lub $w[k - s] = w[k]$,
- $cond_2(j, s)$ – dla $s < j$ zachodzi $w[j - s] \neq w[j]$.

Wówczas możemy zdefiniować tablicę BM w następujący sposób:

$$BM[j] = \begin{cases} \min\{s > 0 : cond_1(j, s) \wedge cond_2(j, s)\} & \text{dla } 1 \leq j < m, \\ p(w) & \text{dla } j \in \{0, m\}. \end{cases}$$

Tablica ta, podobnie jak tablica silnych prefikso-sufiksów w algorytmie KMP, określa o ile możemy przesunąć wzorzec względem tekstu w razie niedopasowania – tylko od końca.

Aby efektywnie obliczyć tablicę BM będzie potrzebna tablica prefikso-prefiksów:

$$PREF[j] = \begin{cases} \max\{s \geq 0 : w[j : j + s - 1] \text{ jest prefiksem } w\} & \text{dla } 1 \leq j \leq m, \\ -1 & \text{dla } j = 0. \end{cases}$$

Algorytm 7: Tablica prefiksowo-prefiksowa

```
def prefix_prefix(w, m):
  def naive_scan(i, j):
    r = 0
    while i + r <= m and j + r <= m and w[i + r] == w[j + r]:
      r += 1
    return r
  PREF, s = [-1] * 2 + [0] * (m - 1), 1
  for i in range(2, m + 1):
    # niezmiennik: s jest takie, ze s + PREF[s] - 1 jest maksymalne i PREF[s] > 0
    k = i - s + 1
```

```
    s_max = s + PREF[s] - 1
    if s_max < i:
      PREF[i] = naive_scan(i, 1)
      if PREF[i] > 0:
        s = i
    elif PREF[k] + k - 1 < PREF[s]:
      PREF[i] = PREF[k]
    else:
      PREF[i] = (s_max - i + 1) + naive_scan(s_max + 1, s_max - i + 2)
      s = i
  PREF[1] = m
  return PREF
```

Twierdzenie 2.4. *Algorytm 7 wyznacza poprawną tablicę prefikso-prefiksów dla słowa w długości m.*

Dowód. Niech tablica $PREF$ będzie poprawnie wypełniona dla wszystkich $2 \leq j < i$ oraz niech $1 \leq s < i$ będzie wartością, dla której $s + PREF[s]$ jest maksymalne. Niech ponadto $s_{max} = s + PREF[s] - 1$, $k = i - s + 1$ oraz $r = PREF[k]$

Wówczas mamy 3 sytuacje:

1. $s + PREF[s] < i$ – wówczas obliczamy wartość $PREF[i]$ naiwnie porównując w i $w[i : m]$,
2. $s + PREF[s] \geq i$ oraz $PREF[k] + k - 1 < PREF[s]$ dla $k = i - s + 1$ – wówczas z definicji s wiemy, że $w[s : s_{max}] = w[1 : (s_{max} - s + 1)]$ oraz $i \in [s, s_{max}]$ (z pierwszego) oraz $w[1 : r] = w[k : (r + k - 1)]$, $w[r + 1] \neq w[r + k]$ (z drugiego), wobec czego również $w[1 : r] = w[i : (r + i - 1)]$ i $w[r + 1] \neq w[r + i]$, a więc $PREF[i] = r$,
3. $s + PREF[s] \geq i$ oraz $PREF[k] + k - 1 \geq PREF[s]$ dla $k = i - s + 1$ – wówczas podobnie jak powyżej mamy $w[1 : (s_{max} - i + 1)] = w[i : s_{max}]$, więc sprawdzamy wydłużanie zgodności, naiwnie porównując $w[s_{max} - i : m]$ oraz $w[s_{max} + 1 : m]$.

□

Twierdzenie 2.5. *Algorytm 7 dla słowa długości m działa w czasie $O(m)$.*

Dowód. Wystarczy zauważyć, że w funkcji `naive_scan` kolejne wartości $j + r$ w ramach wszystkich wywołań tworzą ciąg rosnący. □

Algorytm 8: Tablica maksymalnych sufiksów

```
def maximum_suffixes(w, m):
  w_rev = w[0] + w[-1:0:-1]
  PREF = prefix.prefix_prefix(w_rev, m)
  S = [PREF[0]] + PREF[-1:0:-1]
  return S
```

Algorytm 9: Tablica przesunięć według dobrych sufiksów

```
def boyer_moore_shift(w, m):
  B = prefix.prefix_suffix(w, m)
  BM = [m - B[m]] + [m] * m
  S, j = maximum_suffixes(w, m), 0
  for k in range(m - 1, -1, -1):
    if k == S[k]:
      while j < m - k:
        BM[j] = m - k
        j += 1
  for k in range(1, m):
    BM[m - S[k]] = m - k
  return BM
```

Problem 2.8. Pokaż, że Algorytm 9 poprawnie oblicza wartości tablicy BM w czasie $O(m)$.

Problem 2.9. Pokaż, jak policzyć tablicę $wBM = \min\{s > 0 : cond_1(j, s)\}$ w czasie $O(m)$.

Algorytm 10: Algorytm Boyera-Moore'a

```
def boyer_moore(t, w, n, m):
  BM = suffix.boyer_moore_shift(w, m)
  i = 1
  while i <= n - m + 1:
    j = m
    while j > 0 and t[i + j - 1] == w[j]:
      j = j - 1
    if j == 0:
      yield i
    i = i + BM[j]
```

Problem 2.10. Pokaż, że dla wszystkich n, m istnieje tekst t i wzorzec w ($|t| = n$, $|w| = m$) takie, że wyszukanie wzorca w w tekście t z użyciem algorytmu Boyera-Moore'a wymaga czasu $O(n + m)$ a dla algorytmu wykorzystującego tablicę $wBM = \min\{s > 0 : cond_1(j, s)\}$ zamiast BM wymaga czasu $\Omega(nm)$.

2.4.1 Znajdowanie wszystkich wystąpień wzorca

Tak jak algorytm KMP, tak algorytm BM można łatwo zmodyfikować, by zwracał wszystkie wystąpienia wzorca w tekście. Wystarczy przyjąć, że $BM[0] = p(w)$.

Problem 2.11. Pokaż, że jeśli tekst t zawiera r wystąpień wzorca w, to czas działania algorytmu Boyera-Moore'a wynosi $\Omega(rm)$.

Wniosek 2.6. Jeśli wzorzec występuje w tekście $\Theta(n)$ razy, to czas działania algorytmu Boyera-Moore'a wynosi $\Omega(nm)$.

Rozwiązaniem jest wykorzystanie zmiennej *memory*, zawierającej numer pierwszego indeksu, którego nie musimy sprawdzać. W większości obrotów pętli $memory = 0$, jedynym wyjątkiem jest sytuacja, gdy zaszło dopasowanie – wtedy wiemy, że $m - p(w)$ pierwszych symboli już jest dopasowanych i wystarczy porównać $w[(m - p(w) + 1)..m]$ z tekstem.

Algorytm 11: Algorytm Boyera-Moore'a-Galila

```
def boyer_moore_galil(t, w, n, m):
  BM = suffix.boyer_moore_shift(w, m)
  i, memory = 1, 0
  while i <= n - m + 1:
    j = m
    while j > memory and t[i + j - 1] == w[j]:
      j = j - 1
    if j == memory:
      yield i
      i, memory = i + BM[0], m - BM[0]
    else:
      i, memory = i + BM[j], 0
```

2.4.2 Dowód liniowej złożoności algorytmu

Dowód Cole'a

Twierdzenie 2.7. *Algorytm 10 wymaga $O(n' + m)$ czasu do stwierdzenia pierwszego wystąpienia wzorca w z tekście t, gdy to wystąpienie zostało znalezione na pozycji n'.*

Wniosek 2.8. Algorytm 10 wymaga $O(n + m)$ czasu do stwierdzenia, czy słowo t zawiera podsłowo w.

Twierdzenie 2.9. *Algorytm 11 działa w czasie $O(n + m)$.*

Dowód. Z poprzedniego twierdzenia łatwo wykazać, że znalezienie wszystkich wystąpień wymaga czasu $O(n + rm)$ dla r wystąpień wzorca w tekście dla Algorytmu 10, a więc również dla Algorytmu 11.

Jeśli $p(w) \geq \frac{m}{2}$, to z faktu $r \leq \frac{n}{p(w)}$ wynika, że całkowita złożoność wynosi $O(n)$.

Jeśli $p(w) < \frac{m}{2}$, to możemy pogrupować wystąpienia wzorca jeśli dwa kolejne w ciągu są odległe o $p(w)$. W każdym takim łańcuchu w Algorytmie 11 przejrzymy każdy symbol w łańcuchu tylko raz. Jeśli natomiast wykryjemy niedopasowanie, to wiemy, że następne przesuniecie zgodnie z dobrym sufiksem wynosi $|w| - p(w) \geq \frac{m}{2}$. Wobec każdy symbol poza łańcuchami odczytamy co najwyżej 2 razy, a zatem złożoność również będzie liniowa. □

2.4.3 Heurystyka *occurrence shift*

W oryginalnym artykule Boyera i Moore'a zaproponowano również dodatkową heurystykę *occurrence shift*, opartą na znajomości wystąpień liter z alfabetu we wzorcu.

Zdefiniujmy tablicę $LAST$ w następujący sposób:

$$L[x] = \begin{cases} \max\{1 \leq i \leq m : w[i] = x \wedge \forall_{i<j\leq m} w[j] \neq m\} & \text{gdy } x \text{ występuje w } w, \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$$

Jeśli wiemy, że niedopasowanie nastąpiło na $w[j] \neq t[i+j]$, to wiemy również, że nie ma sensu sprawdzać przesunięć krótszych niż $j - LAST[t[i+j]]$. Mówiąc prościej, jeśli $t[i+j] = x$ oraz x występuje w w najpóźniej na pozycji k (tj. $k = LAST[t[i+j]]$), to mamy 2 sytuacje:

- albo $k > j$, wtedy wiedza z tablicy $LAST$ nam nic nie pomaga,
- albo $k < j$, wtedy wiemy, że pierwszą okazją na znalezienie dopasowania jest zestawienie ze sobą $w[k]$ i $t[i+j]$ – czyli właśnie przesunięcie o $j - k$.

Dla znanego, skończonego alfabetu widać wprost, że wyznaczenie tablicy $LAST$ wymaga czasu i pomięci $O(m + |\mathcal{A}|)$.

Ponieważ heurystyka nie wyklucza się z obserwacjami dokonanymi na bazie tablicy BM, jej zastosowanie polega na zastąpieniu przesunięć o $BM[j]$ przesunięciami o $\max\{BM[j], j - LAST[t[i+j]]\}$.

W praktyce heurystyka ma sens wtedy, gdy alfabety są większe, ale dla alfabetu binarnego jest mało skuteczna. Co ciekawe, brakuje teoretycznych analiz skuteczności użycia tej heurystyki np. względem podstawowego algorytmu Boyera-Moore'a.

Problem 2.12. Niech w Algorytmie 10 w linii ??? będzie miał zamiast $i = i + BM[j]$ polecenie $i = i + \max\{1, j - LAST[t[i+j]]\}$. Pokaż dla dowolnych n i m przykłady słów t i w ($|t| = n$, $|w| = m$) takich, że zmodyfikowany algorytm wymaga $\Omega(nm)$ porównań.

Ten sam pomysł pojawia się u Horspoola, który postanowił korzystać tylko z przesuwania o *occurrence shift* zgodnie z ostatnim znakiem aktualnie przetwarzanego tekstu.

Algorytm 12: Algorytm Horspoola

```
def horspool(t, w, n, m):
  LAST = suffix.last_occurrence(w[:-1])
```

```
  i = 1
  while i <= n - m + 1:
    c = t[i + m - 1]
    if w[m] == c:
      j = 1
      while j < m and t[i + j - 1] == w[j]:
        j = j + 1
      if j == m:
        yield i
    bad_character = LAST.get(c, 0)
    i = i + (m - bad_character)
```

Problem 2.13. Pokaż dla dowolnych n i m przykłady słów t i w ($|t| = n$, $|w| = m$) takich, że algorytm Horspoola wymaga $\Omega(nm)$ porównań.

Raita zaproponował dodatkowo, żeby zamiast przechodzić do porównania tekstu z wzorcem za każdym razem wykonać najpierw porównanie symbolu ostatniego, pierwszego i środkowego. Uzasadniał to tym, że wyszukiwanie słów w językach naturalnych często natrafia na problem częstych sufiksów (np. w angielskim *-ing*, *-ion*, *-ed*) Raita, 1992, aczkolwiek dalsze eksperymenty na tekstach losowych również przyniosły podobne rezultaty, co sugeruje że korzyści mogą być spowodowane czymś innym Smith, 1994.

2.4.4 Quick search

Można zaproponować również inną heurystykę opartą o *occurrence shift*: jeśli zaszło niedopasowanie na fragmencie tekstu $t[(i-j)..(i-1)]$, to wiemy, że następne porównanie tekstu z wzorcem będzie zawierało $t[i]$. Wobec tego można od razu przesunąć tekst tak, aby uzgodnić ostatnie wystąpienie litery $t[i]$ we wzorcu z tekstem.

Algorytm 13: Algorytm *quick search*

```
def quick_search(t, w, n, m):
  LAST = suffix.last_occurrence(w)
  i = 1
  while i <= n - m + 1:
    j = 1
    while j <= m and t[i + j - 1] == w[j]:
      j = j + 1
    if j == m + 1:
      yield i
    bad_character = LAST.get(t[i + m], 0) if i + m <= n else 0
    i = i + (m + 1 - bad_character)
```

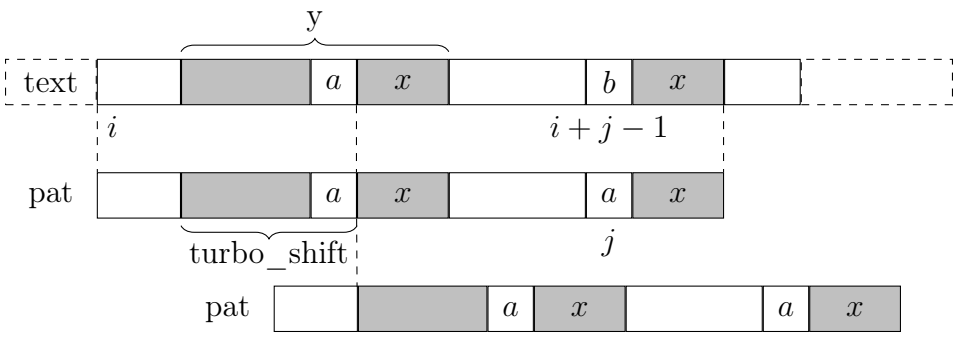
Smith zauważył, że można połączyć podejście Horspoola i Quick Search tj. brać maksimum z obu zalecanych przesunięć, aby otrzymać praktyczne przyspieszenie algorytmu Smith, 1991.

2.5 Algorytm Turbo_BM

Algorytm Turbo_BMjest modyfikacją algorytmu Boyera-Moore'a, która zużywa tylko stałą dodatkową pamięć oraz przyspiesza złożoność pesymistyczną do $2n$. Nie będziemy wykonywać żadnego dodatkowego przetwarzania wstępnego – tak jak w oryginalnym algorytmie korzystać będziemy z tablicy BM obliczonej dla wzorca.

Sam pomysł algorytmu Turbo_BMjest bardzo prosty – podczas skanowania będziemy zapisywać jedno podsłowo wzorca (*memory*), o którym wiemy że pasuje do tekstu na obecnej pozycji. Dzięki temu możemy pominąć ten fragment przy sprawdzaniu dopasowania, oraz w przypadku jego braku możemy wykonać tzw. *Turbo-shift*.

Opiszmy sytuację, w której możemy wykonać Turbo-shift. Niech x będzie najdłuższym sufiksem wzorca, który pasuje do tekstu na danej pozycji, a będzie pierwszą literą wzorca która nie pasuje, a niech y (*memory*) będzie poprzednio zapamiętanym podsłowem, które również pasuje do wzorca (patrz fig. 1). Załóżmy również, że x i y są rozłączne i y jest dłuższy od x. Po wykonaniu Turbo-shift zapominamy o y (patrz kod *boyer_moore_turbo*), więc w poprzednim kroku wykonaliśmy zwykłe przesunięcie (zgodnie z tablicą BM), znane z algorytmu Boyera-Moore'a. W tej sytuacji ax jest suffixem y, oraz litery a i b tekstu są o siebie oddalone o $|y|$. Ale suffix $y \dots x$ ma okres długości $|y|$ (z definicji BM). Możemy więc przesunąć wzorzec o $|y| - |x|$ do przodu i to właśnie przesunięcie nazywać będziemy *Turbo-shift*.



Rysunek 1: Turbo-shift

Analiza złożoności algorytmu Turbo_BMjest dużo prostsza niż analiza algorytmu Boyera Moore'a bez żadnych modyfikacji.

Twierdzenie 2.10. *Algorytm Turbo_BMwykonuje co najwyżej* $2n$ *porównań.*

Dowód. Podzielimy wyszukiwanie wzorca na etapy, każdy etap będzie się składał z dwóch operacji: skanowania i przesuwania wzorca. Podczas etapu k niech Suf_k oznacza suffix wzorca, który pasuje do tekstu, a $shift_k$ niech oznacza długość o którą przesuniemy wzorzec podczas etapu k.

Przsunięcie na etapie k nazwiemy *krótkim*, gdy $2 * shift_k < |Suf_k| + 1$. Wyróżnimy 3 typy etapów:

(1) Etap, po którym następuje przeskok przy skanowaniu dzięki *memory*.
(2) (1) nie zachodzi i wykonujemy długie przesunięcie.
(3) (1) nie zachodzi i wykonujemy krótkie przesunięcie.

Zastosujemy analizę kosztu zamortyzowanego. Zdefiniujmy $cost_k$: dla etapu typu (1) $cost_k = 1$ – będzie to tylko koszt porównania litery, która się nie zgadza. Resztę kosztu przenosimy na pozostałe typy etapów. Dla etapów (2) i (3) $cost_k = |Suf_k| + 1$. Wystarczy nam pokazać, że $\sum_k cost_k < 2 * \sum_k shift_k$. To nam wystarczy, bo $\sum_k shift_k \leq |t|$.

Dla etapu k typu (1) $cost_k = 1$ jest trywialnie mniejszy od $2 * shift_k$. Dla etapu k typu (2) $cost_k = |Suf_k| + 1 \leq 2 * shift_k$ z definicji długiego przesunięcia.

Wystarczy rozważyć etapy typu (3). Jedyna możliwość wykonania krótkiego przesunięcia zachodzi, gdy nie wykonujemy Turbo-shiftu. Ustawiamy wtedy zmienną *memory*, co prowadzi do potencjalnego Turbo-shiftu na etapie $k + 1$. Rozważmy dwa przypadki etapu (3):

(a) $|Suf_k| + shift_k \leq |pat|$. Wtedy z definicji Turbo-shifta mamy: $|Suf_k| - |Suf_{k+1}| \leq shift_{k+1}$, a więc: $cost_k = |Suf_k| + 1 \leq |Suf_{k+1}| + shift_{k+1} + 1 \leq shift_k + shift_{k+1}$.
(b) $|Suf_k| + shift_k > |pat|$. Wtedy mamy: $|Suf_{k+1}| + shift_k + shift_{k+1} \geq |pat|$, oraz: $cost_k \leq |pat| \leq 2 * shift_k - 1 + shift_{k+1}$.

Możemy założyć, że na etapie $k+1$ zachodzi przypadek (b), bo daje on gorsze ograniczenie na $cost_k$. Jeśli etap $k + 1$ jest typu (1), mamy: $cost_k + cost_{k+1} \leq 2 * shift_k + shift_{k+1}$. Gdy na etapie $k + 1$ zachodzi nierówność $|Suf_{k+1}| \leq shift_{k+1}$ wtedy mamy: $cost_k + cost_{k+1} \leq 2 * shift_k + 2 * shift_{k+1}$.

Wystarczy rozważyć sytuację, gdy na etapie $k + 1$ mamy $|Suf_{k+1}| > shift_{k+1}$. To oznacza, że na etapie $k + 1$ wykonujemy standardowe przesunięcie (a nie Turbo-shift). Wtedy aplikujemy indukcyjnie nasze powyższe rozumowanie do etapu $k + 1$, a ponieważ wtedy może zajść tylko przypadek (a) otrzymujemy: $cost_{k+1} \leq shift_{k+1} + shift_{k+2}$, a więc: $cost_k + cost_{k+1} \leq 2 * shift_k + 2 * shift_{k+1} + shift_{k+2}$.

Musimy jeszcze tylko domknąć indukcję: jeśli na wszystkich etapach i od k do $k+j$ mamy $|Suf_i| > shift_i$, wtedy: $cost_k + \ldots + cost_{k+j} \leq 2 * shift_k + \ldots + 2 * shift_{k+j} + shift_{k+j+1}$.

Niech k' będzie pierwszym etapem po etapie k na którym $|Suf_{k'}| \leq shift_{k'}$. Wtedy otrzymujemy $cost_k + \ldots + cost_{k'} \leq 2 * shift_k + \ldots + 2 * shift_{k'}$ co kończy dowód. □

2.6 Algorytm *fast-on-average*

Algorytm 14: Algorytm *fast-on-average*

```
def fast_on_average(t, w, n, m):
  S = build_suffix_tree(w)
  i, r = m, 2 * math.ceil(math.log(m, 2))
  while i <= n:
    if S.contains(t[(i - r):(i + 1)]):
      subt = t[0] + t[(i - m + 1):min(i - r + m - 1, n + 1)]
      subn = min(i - r + m - 1, n + 1) - (i - m + 1)
      for out in knuth_morris_pratt(subt, w, subn, m)
        yield out
    i = i + m - r
```

Twierdzenie 2.11 (Crochemore i Rytter 2002, Theorem 2.5)**.** *Algorytm 14 działa w czasie oczekiwanym* $O\left(n\frac{\log m}{m}\right)$ *i pesymistycznym* $O(n)$*. Przetwarzanie wstępne wzorca wymaga czasu* $O(m)$*.*

Dowód. Stworzenie struktury do sprawdzania, czy słowo $t[(i-r)..i]$ jest podsłowem w (np. drzewa sufiksowego lub DAWG) wymaga czasu $O(m)$.

Jeśli $t[(i-r)..i]$ nie jest podsłowem w, to w nie może być podsłowem t rozpoczynającym się na żadnej z pozycji od $i-m+1$ do $i-r$.

Słowo $t[(i-r)..i]$ ma $2^{r+1} \geq m^2$ możliwych wartości. Z drugiej strony, wiemy że w zawiera mniej niż m podsłów długości $r+1$. Jeśli t jest losowy, to prawdopodobieństwo, że $t[(i-r)..i]$ jest podsłowem w jest mniejsze niż $\frac{1}{m}$. Wobec tego po sprawdzeniu czy słowo $t[(i-r)..i]$ jest podsłowem w (w czasie $O(r)$) jedynie z prawdopodobieństwem mniejszym niż $\frac{1}{m}$ będziemy musieli wykonać algorytm KMP na tekście $t[(i-m+1)..(i-r+m-1)]$ oraz wzorcu w (w czasie $O(m)$). Łączny oczekiwany czas działania tej procedury jest równy zatem $O(r)$, natomiast pesymistyczny $O(m)$.

Każda pojedyncza iteracja rozstrzyga dopasowanie dla $m-r$ możliwych początków indeksów. Wystarczy zatem uruchamiać algorytm dla $i = 1, m-r+1, 2(m-r)+1, \ldots$ – a zatem $O\left(\frac{n}{m}\right)$ razy – i rozwiązać problem niezależnie dla każdego podprzypadku. □

2.7 Algorytm *two-way*

Definicja 2.12. Dla dowolnych słów u, v niepuste słowo w jest *powtórzeniem* (repetition) (u, v), jeśli:

- w jest sufiksem u lub u jest sufiksem w,
- w jest prefiksem v lub v jest prefiksem w.

Zwróćmy uwagę, że dla dowolnych słów u, v zawsze istnieje co najmniej jedno powtórzenie (u, v): $w = vu$.

Definicja 2.13. Dla dowolnych słów u, v lokalny okres jest zdefiniowany jako minimalna długość słowa będącego powtórzeniem (u, v).

$$lp(u, v) = \min\{|w| : w \text{ jest powtórzeniem } (u, v)\}$$

Problem 2.14. Pokaż, że $1 \leq lp(u, v) \leq p(uv)$ dla dowolnych słów u, v.

Problem 2.15. Pokaż jak w czasie $O(n)$ dla dowolnego słowa w obliczyć tablicę $lp(u, v)$ dla wszystkich u, v takich, że $w = uv$.

Definicja 2.14. Faktoryzacją krytyczną słowa w nazywamy parę słów (u, v) takich, że $uv = w$ i $lp(u, v) = p(w)$.

Problem 2.16. Pokaż, że dla dowolnego niepustego słowa w istnieje faktoryzacja krytyczna (u, v) spełniająca $|u| < p(w)$.

Potrzebne będzie znalezienie faktoryzacji krytycznej. Okazuje się, że można to zrobić na bazie algorytmu znajdującego maksymalny sufiks słowa.

Algorytm 15: Wyznaczanie faktoryzacji krytycznej dla tekstu t

```
def critical_factorization(text, n):
  index_lt, p_lt = maximum_suffix(text, n)
  index_gt, p_gt = maximum_suffix(
      text, n, less = operator.__gt__)
  return (index_lt, p_lt) if index_lt >= index_gt else (index_gt, p_gt)
```

Twierdzenie 2.15 (Apostolico i Galil 1997, Theorem 1.18, s. 40)**.** *Algorytm 15 zwraca faktoryzację krytyczną (u, v) dla słowa w. Dodatkowo, $|u| < p(w)$.*

Dowód. Niech $\leq$ będzie porządkiem leksykograficznym zgodnym z kolejnością liter w alfabecie, natomiast $\leq^*$ porządkiem leksykograficznym zgodnym z odwrotną kolejnością liter w alfabecie.

Dla dowolnego niepustego słowa w niech $w = uv$ z v jako maksymalnym sufiksem według $\leq$ oraz $w = u'v'$ z v' jako maksymalnym sufiksem według $\leq^*$.

Jeśli $w = a^n$ dla pewnego $a \in \mathcal{A}$, to każde (u, v) takie, że $w = uv$ jest faktoryzacją krytyczna. Jeśli nie, to $v \neq v'$. Bez straty ogólności załóżmy, że $|v| < |v'|$.

Niech x będzie najkrótszym powtórzeniem dla (u, v). Wystarczy tylko dowieść, że $|x|$ jest okresem w, bo z lematu wyżej wiadomo, że $|x| = lp(u, v) \leq p(w)$.

Możemy wyróżnić cztery przypadki:

1. x jest sufiksem u, v jest prefiksem x – ale wtedy $v < x < xv$ i xv jest sufiksem w, więc v nie byłoby maksymalnym sufiksem w,
2. x jest sufiksem u, x jest właściwym prefiksem v z $v = xz$ dla pewnego $z \in \mathcal{A}^+$ – wtedy z jest również sufiksem w, wobec tego $z < v$ i $v = xz < xv$, więc v nie byłoby maksymalnym sufiksem w,
3. u jest właściwym sufiksem x, v jest prefiksem x – wtedy $|x|$ jest okresem w,
4. u jest właściwym sufiksem x, x jest właściwym prefiksem v z $v = xz$ dla pewnego $z \in \mathcal{A}^+$ – wtedy $v' = yv$ dla pewnego $y \in \mathcal{A}^+$. Skoro v' jest sufiksem $w = uv$, to y jest sufiksem u, a zatem jest też sufiksem x. Wobec tego yz jest sufiksem v oraz całego w, a więc $yz \leq^* v' = yv$, czyli $z \leq^* v$. Zarazem $z \leq v$, bo z definicji z też jest sufiksem w – a zatem z jest prefiksem v. Wobec tego z jest prefikso-sufiksem v a $|x|$ jest okresem v. $|x|$ jest ostatecznie okresem całego w, bo w jest podsłowem x^2z.

Ponieważ pierwsze dwa przypadki kończą się sprzecznością, to wiemy, że u jest właściwym sufiksem x – a więc $|u| < |x| = p(w)$. □

Powyższe obserwacje stały się podstawą pierwszego optymalnego (tj. działającego w czasie liniowym i ze stałą dodatkową pamięcią) algorytmu wyszukiwania wzorca z tekście, tzw. *two-way algorithm*.

Załóżmy, że mamy (u, v), faktoryzację krytyczną wzorca w. Wówczas możemy najpierw sprawdzać dopasowanie odpowiedniej części tekstu do v od lewej do prawej (jak w algorytmie Morrisa-Pratta), a następnie sprawdzać dopasowanie wcześniejszej części do u od prawej do lewej (jak w algorytmie Boyera-Moore'a). Jak zwykle, w razie niedopasowania wykonujemy odpowiednie przesunięcia.

Algorytm 16: Algorytm Two-Way

```
def two_way(text, word, n, m):
  index, p, use_memory = *critical_factorization(word, m), True
  if index - 1 > m / 2 or not word[index:index + p].endswith(word[1:index]):
    p, use_memory = max(len(word[1:index]), len(word[index:])) + 1, False
  i, memory = 1, 0
  while i <= n - m + 1:
    j = max(index - 1, memory)
    while j < m and text[i + j] == word[j + 1]:
      j = j + 1
    if j < m:
      i, memory = i + j + 2 - index, 0
      continue
    j = max(index - 1, memory)
    while j > memory and text[i + j - 1] == word[j]:
      j = j - 1
    if j == memory:
```

```
      yield i
    i, memory = i + p, m - p if use_memory else 0
```

Jak widać, dla jednego z przypadków zostało wykorzystane zapamiętywanie dopasowania, analogicznie jak w algorytmie Boyera-Moore'a-Galila. Okazuje się, że dla drugiego nie jest ono potrzebne, ponieważ okres wzorca jest dostatecznie długi.

Lemat 2.16 (Apostolico i Galil 1997, Lemma 1.19, s. 43)**.** *Niech (u, v) jest faktoryzacją krytyczną w taką, że $|u| < p(w)$. Jeśli u nie jest sufiksem $v[1 \dots p(v)]$, to u występuje dokładnie raz w w.*

Dowód. Załóżmy, że u nie jest sufiksem $v[1 \dots p(v)]$, ale występuje w w drugi raz. Z okresowości v wiemy, że v musi zaczynać się pewnym sufiksem u – ale wtedy $lp(u, v) \leq |u|$. To przeczy jednak założeniu, że (u, v) jest faktoryzacją krytyczną, a więc $lp(u, v) = p(w) > |u|$. □

Wniosek 2.17. Niech (u, v) jest faktoryzacją krytyczną w taką, że $|u| < p(w)$. Jeśli u nie jest sufiksem $v[1 \dots p(v)]$, to

$$p(w) \geq \max\{|u|, |v|\} + 1 > \frac{|w|}{2}.$$

Dowód. Wprost z założenia wynika, że $p(w) \geq |u| + 1$. Z poprzedniego lematu wiemy, że u występuje dokładnie raz w w, a zatem największy możliwy prefikso-sufiks w jest nie dłuższy niż $|u| - 1$. Wobec tego $p(w) \geq |w| - (|u| - 1) = |v| + 1$.

Druga nierówność jest oczywista na mocy faktu $w = uv$. □

Twierdzenie 2.18 (Apostolico i Galil 1997, Theorem 1.18, s. 43)**.** *Algorytm 16 zwraca poprawnie wszystkie wystąpienia podsłów w tekście.*

Dowód. Przy przeglądaniu w prawo w razie niedopasowania po prostu przesuwamy wzorzec o tyle znaków, ile przejrzeliśmy.

Przy przeglądaniu w lewo mamy dwa przypadki, w zależności od tego, czy dla wzorca w i jego faktoryzacji krytycznej (u, v) rzeczywiście u jest sufiksem $v[1 \dots p(v)]$. Jeśli tak, to postępujemy dokładnie jak w algorytmie 11. W tym przypadku zachodzi $p(w) = p(v)$.

Jeśli nie, to z 2.17 wiemy, że $p(w) \geq \max\{|u|, |v|\} + 1$ – a zatem możemy ustawić minimalne przesunięcie na tę drugą wartość. Gwarantuje to zarazem, że nie musimy się martwić o przypadek nakładających się wystąpień wzorca na siebie, więc nie potrzebujemy zapamiętywania prefiksu zgodnie z regułą Galila. □

Twierdzenie 2.19. *Algorytm 16 działa w czasie $O(n + m)$ i wymaga $O(1)$ dodatkowej pamięci.*

Dowód. W dowodzie zakładamy, że wyznaczanie maksymalnego sufiksu jest wykonywane w czasie $O(n+m)$ i w $O(1)$ dodatkowej pamięci np. wykorzystując 25. Wykorzystanie pamięci $O(1)$ przez cały algorytm jest oczywiste.

Jeśli patrzymy na pozycje tekstu t porównywane tylko przy przeglądaniu w prawo w ciągu całego algorytmu, to tworzą one ciąg rosnący. Wobec tego takich porównań mamy co najwyżej n.

Jeśli patrzymy na pozycje tekstu t porównywane tylko przy przeglądaniu w prawo w ciągu całego algorytmu, to żadna nie zostanie porównana dwa razy, ponieważ między kolejnymi iteracjami głównej pętli przesuwamy okno o $p > |u|$ (na mocy dodatkowego warunku dla faktoryzacji krytycznej), a w każdym oknie sprawdzamy co najwyżej $|u|$ znaków w ten sposób. Takich porównań również mamy co najwyżej n. □

2.8 Algorytm Karpa-Rabina

Alternatywne podejście można oprzeć o obliczanie funkcji skrótu h dla poszczególnych podsłów długości m. Jeśli funkcja jest zwykłą liniową funkcją modulo, to łatwo obliczyć $h(t[i+1..m+i+1])$ na podstawie $h(t[i..m+i])$.

Algorytm 17: Algorytm Karpa-Rabina

```
def karp_rabin(text, word, n, m):
  MOD = 257
  A = random.randint(2, MOD - 1)
  Am = pow(A, m, MOD)
  def generate(t):
    return sum(ord(c) * pow(A, i, MOD) for i, c in enumerate(t[::-1])) % MOD
  text += '$'
  hash_text, hash_word = generate(text[1:m + 1]), generate(word[1:])
  for i in range(1, n - m + 2):
    if hash_text == hash_word and text[i:i + m] == word[1:]:
      yield i
    hash_text = (A * hash_text + ord(text[i + m]) - Am * ord(text[i])) % MOD
```

Warto zwrócić uwagę, że algorytm można zaimplementować w dwóch wersjach:

- Monte Carlo – zwracamy dopasowanie od razu, gdy wykryjemy równość haszy, więc czas działania to $O(n)$, ale możliwe fałszywe dopasowania,
- Las Vegas – zwracamy dopasowanie jedynie, gdy wykryjemy równość haszy i sprawdzimy z całym wzorcem, stąd brak błędów, ale pesymistyczny czas działania to $O(nm)$.

2.9 Algorytm Galila–Seiferasa

Algorytm G–S jest algorytmem wyszukiwania wzorca który wymaga jedynie stałej pamięci dodatkowej (eliminując tablicę „mismatch" używaną przez algorytmy typu Morrisa–Pratta) a jednocześnie wciąż działa w asymptotycznym czasie liniowym — osiągając tym samym teoretyczne optimum pod względem zarówno czasu jaki i pamięci.

W tym opisie, tak samo jak w załączonej implementacji algorytmu, stringi indeksowane są od 1. Wszelki kod należy interpretować zgodnie z semantyką języka Python.

Niech `m = len(word)`, `n = len(text)`. Rozważmy schemat algorytmu dopasowania wzorca

```
p,q= 0,0
\textbf{while} True:
  \textbf{while} text[p+q+1] == word[q+1]: q+= 1
  \textbf{if} q == m: \textbf{yield} p+1
  p,q= \textit{p'},\textit{q'}
```

W powyższym pseudokodzie `p` to jest obecny kandydat na pozycję wzorca, a `q` to prefiks o którym wiemy że się zgadza.

Naiwny algorytm *brute-force* używa `p'= p+1, q'= 0`. Ponieważ wiemy jednak że `text[p+1:p+q+1] == word[1:q+1]` to rozważanie `p'= p+x` ma sens tylko jeśli `word[1:q-x+1] == word[x+1:q+1]`; na tej obserwacji jest oparty algorytm Knutha–Morrisa–Pratta, obliczający `p'= p+shift[q], q'= q-shift[q] if q>0 else 0`, gdzie `shift[q]` jest długością najkrótszego okresu `word[1:q+1]`.

Algorytm Galila–Seiferasa jest oparty na tym schemacie oraz następującym twierdzeniu:

Twierdzenie 2.20. *Dla ustalonego (odpowiednio dużego) k, każde słowo* `x` *można przedstawić jako* `u+v`*, gdzie* `v` *ma co najwyżej jeden k-okres prefiksu a* `u` *jest długości* $O($*długość najkrótszego okresu* `v`$)$.

Słowo `z` nazywamy k-okresem prefiksu słowa `w` jeśli `z` jest słowem pierwotnym a $\texttt{z}^{\texttt{k}}$ jest prefiksem `w`.

Do skonstruowania tej dekompozycji przyda nam się następujący lemat:

Lemat 2.21. *Dla słowa pierwotnego* `w` *istnieje dekompozycja* `w == w1 + w2` *taka że dla dowolnego słowa* `w'`*, słowo* `w2 + w`$^{\texttt{k-1}}$ `+ w'` *nie ma k-okresu prefiksu krótszego niż* `len(w)`.

Pierwsza część algorytmu jest poświęcona znalezieniu dekompozycji wzorca takiej jak w twierdzeniu o dekompozycji.

```
s= 0
\textbf{while} word[s+1:] \textit{ma więcej niż jeden k-okres prefiksu}:
  p2= \textit{długość drugiego najkrótszego k-okresu prefiksu}
```

```
  \textit{Korzystając z lematu, znajdź}
   s' < s + p2 \textit{takie że} word[s'+1:] \textit{nie ma k-okresu prefiksu krótszego
  s= s'
```

Niech `l(s)` będzie długością najkrótszego okresu `word[s+1:]` a `p1(s)` długością najkrótszego k-okresu prefiksu. Indukcyjnie można dowieść, że w każdej iteracji pętli `s < c * min(l(s), p1(s))` dla `c = (k-1)/(k-2)` — więc w szczególności na końcu `len(u) < c * l(s)`.

W szczególności istnieje następujący prosty algorytm znajdowania `s'`:

```
s'= s
\textbf{while} word[s'+1:] \textit{ma k-okres prefiksu krótszy niż p2}:
  usuń najkrótszy okres prefiksu
```

Przeanalizujmy złożoność ostatecznego kodu programu:

```
K= 4
s,p1,q1= 0,1,0
p2,q2= 0,0
mode=0
\textbf{while} True:
  \textbf{if} mode==1:
    \textit{#Znajdź najkrótszy k-okres prefiksu word[s+1:]}
    \textbf{while} s+p1+q1 < m \textbf{and} word[s+p1+q1+1] == word[s+q1+1]: q1+= 1
    \textbf{if} p1+q1 >= p1*K: p2, q2= q1,0; mode=2; \textbf{continue}
    \textbf{if} s+p1+q1 == m: \textbf{break}
    p1+= (1 \textbf{if} q1==0 \textbf{else} (q1+K-1)//K); q1= 0
  \textbf{elif} mode==2:
    \textit{#Znajdź drugi najkrótszy k-okres prefiksu word[s+1:]. Jeśli nie istnieje to
    \textbf{while} s+p2+q2 < m \textbf{and} word[s+p2+q2+1] == word[s+q2+1] \textbf{and}
    \textbf{if} p2+q2 == p2*K: mode= 0; \textbf{continue};
    \textbf{if} s+p2+q2 == m: \textbf{break}
    \textbf{if} q2 == p1+q1:
      p2+= p1; q2-= p1;
    \textbf{else}:
      p2+= (1 \textbf{if} q2==0 \textbf{else} (q2+K-1)//K); q2= 0
  \textbf{else}:
    \textit{#Zinkrementuj s}
    \textbf{while} s+p1+q1 < m \textbf{and} word[s+p1+q1+1] == word[s+q1+1]: q1+= 1
    \textbf{while} p1+q1 >= p1*K: s+= p1; q1-= p1;
    p1+= (1 \textbf{if} q1==0 \textbf{else} (q1+K-1)//K); q1= 0
    \textbf{if} p1 >= p2: mode= 1
```

W celu przeanalizowania złożoności tej części algorytmu rozważmy wyrażenie

$$2s + (k+1)p_1 + q_1 + (k+1)p_2 + q_2$$

Jego wartość jest zawsze $O(\texttt{len(word)})$ a każde przypisanie ją zwiększa.

```
p2,q2= 0,0
\textbf{while} True:
  \textbf{while} p2+s+q2 < n \textbf{and} s+q2 < m \textbf{and} text[p2+s+q2+1] == word[
  \textbf{if} q2 == m-s \textbf{and} text[p2+1 : p2+s+1] == word[1 : s+1]:
    yield p2+1
  \textbf{if} q2 == p1+q1:
    p2+= p1; q2-= p1
  \textbf{else}:
    p2+= (1 \textbf{if} q2==0 \textbf{else} (q2+K-1)//K); q2= 0
  \textbf{if} p2+s > n: \textbf{return}
```

Przed wykonaniem drugiej części algorytmu jeśli `word[s+1:]` ma k-okres prefiksu, to jego długość wynosi p_1. Ponadto twierdzimy, że $s < (k-1)p_1/(k-2)$ co nam zagwarantuje że bezpośrednie sprawdzanie równości podciągów zajmie łącznie $O(\texttt{len(text)})$ czasu.

3 Indeksowanie tekstu

Problem 3: Indeksowanie tekstu

Wejście: Słowo $t \in \mathcal{A}^+$ ($|t| = n$)

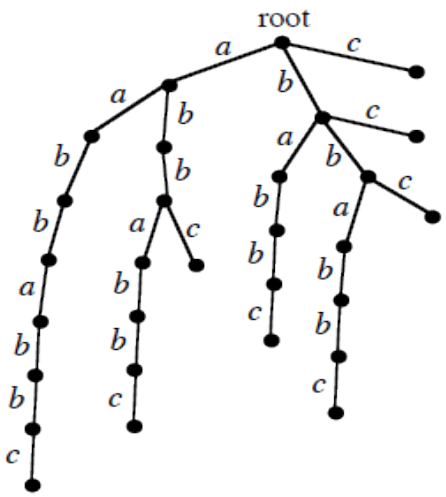
Wyjście: Struktura danych, na której można wykonywać zapytania *czy w jest podsłowem t* dla dowolnego $w \in \mathcal{A}^*$ ($|w| = m$).

Typowo zakładamy, że $m \ll n$ oraz wymagamy, aby zapytanie było wykonywalne w czasie $O(m)$, niezależnym od n. Czasem dopuszczamy, aby czas wykonania zapytań był logarytmiczny względem n.

Dodatkowo wstępne przetwarzanie tekstu powinno było wykonalne w czasie liniowym lub przynajmniej liniowo-polilogarytmicznym względem n (oraz rozmiaru alfabetu $\mathcal{A}$).

3.1 Drzewo sufiksowe

Drzewo sufiksowe (*suffix trie*) to jedna ze struktur danych, która umożliwia łatwe sprawdzanie podsłów. W najprostszej, nieskompresowanej wersji jest to zwykłe drzewo słownikowe $Trie(t)$ zawierające wszystkie sufiksy słowa t.



Sprawdzenie, czy słowo w jest podsłowem t polega po prostu na sprawdzeniu, czy w drzewie sufiksowym $Trie(t)$ istnieje ścieżka zgodna z kolejnymi literami słowa w.

Pewnym ułatwieniem rozważań na drzewach sufiksowych jest zastosowanie specjalnego znaku $\$ \notin \mathcal{A}$ na końcu t. W ten sposób zapewniamy, że każdy sufiks t kończy się w liściu tj. żaden sufiks nie jest prefiksem innego sufiksu.

Problem 3.1. Dla dowolnego n pokaż tekst t długości n taki, że rozmiar drzewa sufiksowego jest możliwie największy.

Ponieważ podstawowe drzewo sufiksowe może mieć rozmiar większy niż liniowy, to wygodniejsze jest przejście na skompresowane drzewo sufiksowe (*suffix tree*) $ST(t)$ tj. drzewo, w którym ściągnięto wszystkie wierzchołki stopnia 2 różne od korzenia. Każda krawędź w drzewie zamiast odpowiadać pojedynczej literze, odpowiada teraz pewnemu słowu z $\mathcal{A}^+$. Dowolne dwie krawędzie wychodzące z tego samego wierzchołka odpowiadają słowom, których największych prefiks jest pusty (tj. różnią się pierwszym znakiem). Oczywiście sprawdzenie, czy w jest podsłowem t przebiega dokładnie tak samo jak poprzednio.

Twierdzenie 3.1. *Skompresowane drzewo sufiksowe dla dowolnego słowa t o długości n ma rozmiar $O(n)$.*

Dowód. Wiemy, że drzewo $ST(t)$ ma co najwyżej n liści (jeśli kończy się symbolem specjalnym, to ma dokładnie n). Z definicji każdy wierzchołek wewnętrzny drzewa skompresowanego ma co najmniej dwa poddrzewa, wobec tego liczba wierzchołków wewnętrznych musi być mniejsza od liczby liści. □

Z powyższych twierdzeń jasne jest, że konstrukcja skompresowanego drzewa sufiksowego przez najpierw zbudowanie, a następnie kompresję drzewa nieskompresowanego jest nieoptymalna.

Optymalne podejścia przedstawione poniżej polegają na iteracyjnym rozbudowywaniu drzewa sufiksowego przez wstawianie kolejnych sufiksów we właściwej kolejności, od najdłuższego lub od najkrótszego. Okazuje się, że samo to jednak nie wystarczy:

Problem 3.2. Dla dowolnego n pokaż tekst t ($|t| = n$) taki, że wstawianie sufiksów do drzewa sufiksowego w dowolnej kolejności zaczynając zawsze od korzenia wymaga $\Theta(n^2)$ operacji.

3.1.1 Algorytm McCreighta

W omawianiu algorytmów będziemy korzystać oznaczenia $ST(t, i)$ jako (tymczasowego) drzewa słownikowego, zawierającego i najdłuższych sufiksów t. Oznaczmy przez $leaf(i)$ liść wstawiony przy przejściu z $ST(t, i-1)$ do $ST(t, i)$, odpowiadający sufiksowi $t[i..n]$. Dodatkowo, $head(i) = parent(leaf(i))$.

Ponadto, zdefiniujmy dla każdego wierzchołka v funkcję $word(v)$ przypisującą mu słowo złożone z etykiet ścieżki od korzenia do v. Jest to szczególnie użyteczne do definicji linków sufiksowych, wykorzystywanego w algorytmie: jeśli $word(v) = aw$ ($a \in \mathcal{A}$, $w \in \mathcal{A}^*$), to $S[v] = u$, gdzie u jest wierzchołkiem, dla którego $word(u) = w$.

Algorytm 18: Algorytm McCreighta budowania drzewa sufiksowego

```
def break_node(parent, child, index):
  u = trie.TrieNode(child.label[:index])
  child.label = child.label[index:]
  u.add_child(child)
  parent.add_child(u)
  return u

def fast_find(v, w):
  index = 0
  while index < len(w) and w[index] in v.children:
    child = v.children[w[index]]
    if index + len(child.label) > len(w):
      return break_node(v, child, len(w) - index)
    v, index = child, index + len(child.label)
  return v

def slow_find(v, w):
  index = 0
  while index < len(w) and w[index] in v.children:
    child = v.children[w[index]]
    for i, c in enumerate(child.label):
      if w[index + i] != c:
```

```
        return break_node(v, child, i), len(w) - (index + i)
    v, index = child, index + len(child.label)
  return v, len(w) - index

def mccreight(text, n):
  text = text + '$'
  root, leaf = trie.TrieNode(""), trie.TrieNode(text[1:])
  root.add_child(leaf)
  S, head = { }, root
  for i in range(2, n + 1):
    # niezmiennik: S[v] jest zdefiniowane dla wszystkich v != head(i - 1)
    if head == root:
      # wyjątek 1: drzewo z jednym liściem
      beta, gamma, v = "", head.children[leaf.label[0]].label[1:], root
    else:
      if head.parent == root:
        # wyjątek 2: head.parent jest rootem
        beta = head.parent.children[head.label[0]].label[1:]
      else:
        beta = head.parent.children[head.label[0]].label
      gamma = head.children[leaf.label[0]].label
      v = fast_find(S[head.parent], beta)
    S[head] = v
    head, remaining = slow_find(v, gamma)
    leaf = trie.TrieNode(text[-remaining:])
    head.add_child(leaf)
  return root
```

W dowodzie optymalności algorytmu McCreighta wykorzystywane będą następujące trzy własności:

Problem 3.3. Pokaż, że $head(i)$ jest potomkiem $S[head(i-1)]$ w $ST(t,i)$.

Problem 3.4. Pokaż, że $S[v]$ jest potomkiem $S[parent(v)]$ dla dowolnego v w każdym $ST(t,i)$.

Problem 3.5. Pokaż, że $depth(v) \leq depth(S[v]) + 1$ dla dowolnego $v \neq head(i)$ w każdym $ST(t,i)$.

Twierdzenie 3.2. *Algorytm McCreighta zwraca poprawne drzewo sufiksowe.*

Dowód. Na początek załóżmy, że dla dowolnego $i = 2, 3, \ldots, n$ mamy zbudowane $ST(t, i-1)$ oraz wszystkie wierzchołki wewnętrzne $ST(t, i-1)$ inne niż $head(i-1)$ mają dobrze

zdefiniowane $S[v]$ (samo $S[head(i-1)]$ też może być zdefiniowane wcześniej, ale nie musi). Ten niezmiennik jest utrzymany przez cały algorytm.

Oczywiście $ST(t,1)$ złożone z korzenia r, liścia l_1 i krawędzi między nimi z etykietą t spełnia te warunki.

Korzystając obu własności z problemów powyżej wiemy, że $S[parent(head(i-1)]$ jest dobrze zdefiniowany oraz istnieje ścieżka z $S[parent(head(i-1)]$ do $S[head(i-1)]$ w $ST(t,i-1)$ – jedynie z tym zastrzeżeniem, że może kończyć się „w środku" krawędzi tj. z $head(i-1)$ jako wierzchołkiem skompresowanym.

Wystarczy zatem dodać wierzchołek $leaf(i)$, być może rozbijając krawędź i tworząc nowy wierzchołek $head(i-1)$, oraz nadać wartość $S[head(i-1)]$ tak, żeby $S[head(i-1)]$ było przodkiem $head(i)$, aby stworzyć drzewo, które jest $ST(t,i)$ oraz ma dobrze zdefiniowane $S[v]$ dla wszystkich wierzchołków wewnętrznych v poza $head(i)$. □

Twierdzenie 3.3. *Algorytm McCreighta ma złożoność* $O(n \log |\mathcal{A}|)$.

Dowód. Złożoność czasowa algorytmu wynika wprost z trzech faktów.

Po pierwsze, poza funkcjami `fast_find` i `slow_find` algorytm wykonuje stałą liczbę operacji w każdej iteracji – a zatem $O(n)$ operacji łącznie. Dla alfabetu $\mathcal{A}$ znalezienie w wierzchołku odpowiedniej krawędzi zaczynającej się od danego symbolu wynosi $O(\log |\mathcal{A}|)$. Wszystkie pozostałe operacje wykonywane są w czasie stałym.

Po drugie, niech funkcja `fast_find` schodzi od wierzchołka $S[parent(head(i-1))]$ w dół do wierzchołka $head(i)$ po dokładnie d_i krawędziach. Wówczas wiemy, że

$$depth(head(i-1)) \leq depth(S[head(i-1)]) + 1 = depth(head(i)) - d_i + 1.$$

Wobec tego całkowita liczba przejść po krawędziach w funkcji `fast_find` wynosi

$$\sum_{i=2}^{n+1} d_i \leq \sum_{i=2}^{n+1} (depth(head(i)) - depth(head(i-1)) + 1) \leq depth(head(n)) + n \leq 2n$$

Po trzecie, funkcja `slow_find` w i-tej iteracji algorytmu wykonuje s_i porównań, gdzie zachodzi $s_i \leq |word(head(i))| - |word(head(i-1))|$. Analogicznie jak powyżej, łączna liczba operacji jest ograniczona z góry przez $O(n)$. □

Warto zauważyć, że jeśli $S[head(i-1)]$ jednak jest dobrze określone w i-tym kroku algorytmu, to możemy od razu przejść do tego wierzchołka, zamiast wykonywać przejście przez rodzica $head(i-1)$ (tzw. *up-link-down*). Taka optymalizacja nie zmienia jednak asymptotycznego czasu wykonania algorytmu.

Problem 3.6. Pokaż, że jeśli w Algorytmie 18 zamiast `fast_find` użylibyśmy funkcji `slow_find`, to dla dowolnego n istnieje tekst t długości n taki, że konstrukcja $ST(t)$ wymaga czasu $\Omega(n^2)$.

3.1.2 Algorytm Ukkonena

Algorytm Ukkonena polega na tym, że w tym przypadku budowane jest w zasadzie poprawne drzewo sufiksowe dla kolejnych prefiksów słowa t. W trakcie wykonania algorytmu wyznaczamy dokładnie taką samą tablicę linków sufiksowych jak w algorytmie McCreighta.

Algorytm 19: Algorytm Ukkonena budowania drzewa sufiksowego

```
def ukkonen(text, n):
  text = text + '$'
  root, leaf = trie.TrieNode(''), trie.TrieNode(text[1:])
  root.add_child(leaf)
  S, head, shift = {root : root}, root, 0
  for i in range(2, n + 2):
    # niezmiennik: S[v] jest zdefiniowane dla wszystkich v != head(i - 1)
    child = head.children.get(text[i - shift])
    if (child is None or shift >= len(child.label)
        or text[i] != child.label[shift]):
      previous_head = None
      while shift > 0 or text[i] not in head.children:
        v = (break_node(head, head.children[text[i - shift]], shift)
             if shift > 0 else head)
        v.add_child(trie.TrieNode(text[i:]))
        if head == root:
          shift -= 1
        if previous_head is not None:
          S[previous_head] = v
        previous_head, head = v, S[head]
        if shift > 0:
          head, shift = fast_find(head, text[i - shift:i], split = False)
      if previous_head is not None:
        S[previous_head] = head
      child = head.children.get(text[i - shift]) if shift >= 0 else None
    if child is not None and len(child.label) == shift + 1:
      head, shift = head.children[text[i - shift]], 0
    else:
      shift += 1
  return root, S
```

Problem 3.7. Pokaż, że wartości $S[v]$ wyznaczane w algorytmach McCrieghta i Ukkonena są identyczne.

Twierdzenie 3.4. *Algorytm Ukkonena zwraca poprawne drzewo sufiksowe.*

Dowód. Przede wszystkim zauważmy, że każdy nowododany liść pozostaje zawsze liściem do samego końca wykonania algorytmu – w ten sposób $leaf(i)$ nie tylko reprezentuje sufiks $t[i]$ w $ST(t,i)$, ale również sufiks $t[i..j]$ w $ST(t,j)$ dla dowolnego $j \geq i$.

Należy przy tym pamiętać, że w rzeczywistości, inaczej niż w powyższym algorytmie, pamiętamy nie całą etykietę na krawędzi, tylko parę liczb oznaczających początek i koniec etykiety w tekście. Wystarczy zatem ustawić indeksy końcowe w liściach na $+\infty$, unikając w ten sposób problemu z dodawaniem kolejnych znaków na końcach liści.

Aby z $ST(t,i-1)$ otrzymać $ST(t,i)$ wystarczy wstawić do drzewa wszystkie sufiksy słowa $t[1..i]$. Załóżmy jednak, że pewnym momencie natrafimy na (być może skompresowany) wierzchołek v, dla którego już istnieje (być może skompresowany) wierzchołek v' spełniający $word(v) = word(v')\,t[i]$. Wówczas wiemy, że w drzewie nieskompresowanym dla wszystkich wierzchołków u odpowiadających sufiksom słowa $word(v)$ istnieją wierzchołki u' takie, że $word(u) = word(u')\,t[i]$. Wobec tego wszystkie mniejsze sufiksy słowa $t[1..i]$ już muszą istnieć.

Idea algorytmu jest zatem taka: w i-tej iteracji wychodzimy od $head(i-1)$ (zob. dowód algorytmu McCreighta) i chodząc po linkach sufiksowych rozbudowujemy drzewo o nowe liście zgodnie z $t[i]$ aż do momentu, gdy natrafimy na istniejący (może skompresowany) wierzchołek. Tak otrzymane drzewo (gdybyśmy zastąpili wszystkie indeksy końcowe $+\infty$ przez i) jest drzewem sufiksowym $ST(t,i)$. □

Twierdzenie 3.5. *Algorytm Ukkonena ma złożoność* $O(n \log |\,\mathcal{A}\,|)$.

Dowód. Wszystkie operacje poza `fast_find` wymagają czasu $O(\log |\,\mathcal{A}\,|)$ na każdą iterację głównej pętli.

Pętla `while` za każdym razem dodaje jeden liść, więc podczas całego wykonania algorytmu funkcja `fast_find` zostanie wykonana $O(n)$ razy. Ponieważ wiemy, że $S[v]$ przyjmuje dokładnie takie same wartości, jak w algorytmie McCreighta, to również tu całkowita liczba odwiedzionych wierzchołków w funkcji `fast_find` wyniesie $O(n)$. □

3.1.3 Dalsze zagadnienia

Giegerich i Kurtz (1997) pokazali, że chociaż teoretyczne idee u podstaw wszystkich trzech algorytmów się różnią, to istnieje głębokie podobieństwo strukturalne. W szczególności można pokazać, że algorytmy McCreighta i Ukkonena wykonują dokładnie te same ciągi abstrakcyjnych operacji, a zatem możliwe jest tłumaczenie krok po kroku wykonania jednego algorytmu na drugi. Również algorytm Weinera przypomina strukturalnie algorytm Ukkonena z tą różnicą, że jest budowany „od końca" i przez to korzysta jedynie z „przybliżonych" linków.

Okazuje się, że możliwa jest konstrukcja drzewa sufiksowego w czasie $O(n)$, niezależnym od $\mathcal{A}$ (Farach, 1997). Wymaga to co prawda utożsamienia alfabetu z podzbiorem liczb naturalnych zamiast z dowolnym zbiorem z liniowym porządkiem, ale w przeciwieństwie do sortowania nie stanowi żadnego problemu.

Drzewa sufiksowe umożliwiają nie tylko szybkie obliczanie zapytań o przynależność, ale mogą również służyć do rozwiązywania kilku innych problemów.

Problem 3.8. Pokaż, jak w czasie $O(m)$ odpowiadać na zapytania, ile razy słowo w ($|w| = m$) występuje w tekście t, gdy mamy skonstruowane $ST(t)$.

Dowód. Mając zbudowane drzewo sufiksowe, idziemy od korzenia w dół czytając kolejne symbole w, a czasami posuwamy się po wewnętrznej etykiecie pewnej krawędzi. Wówczas zbiór wystąpień odpowiada zbiorowi liści w poddrzewie węzła, do którego udało nam się w ten sposób dojść. Liczbę takich liści możemy ustalić bardzo łatwo: budując drzewo sufiksowe możemy od razu obliczyć w każdym węźle trzymać ile liści jest w jego poddrzewie. Jeżeli po drodze nie dało się dalej schodzić po drzewie, oznacza to, że w nie jest podsłowem słowa t, więc odpowiedzią jest zero. □

Problem 3.9. Pokaż, jak w czasie $O(n)$ policzyć, ile różnych podsłów zawiera słowo t ($|t| = n$).

Dowód. Gdybyśmy zbudowali takie pełne drzewo sufiksowe bez kompresji, to wówczas jest to po prostu drzewo trie wszystkich sufiksów danego słowa. Wówczas liczba różnych podsłów słowa n jest po prostu równa liczbie wierzchołków w stworzonym drzewie. W przypadku, gdy budujemy drzewo sufiksowe z kompresją, aby nie mieć kwadratowego rozmiaru całej struktury, należy również policzyć wierzchołki w zbudowanym drzewie, ale dodatkowo licząc wierzchołki liczymy je z odpowiednimi wagami, które wynikają z tego, że wierzchołki w skompresowanym drzewie sufiksowym są tak naprawdę ścieżkami wierzchołków i je też należy uwzględnić, co możemy zrobić bardzo łatwo, poprzez popatrzenie jakie etykiety są trzymane w wierzchołkach i dodając odpowiednie krotności. Możemy to wszystko zrobić jednym przejściem po drzewie, skąd wynika liniowa złożoność względem długości słowa. □

Problem 3.10. Pokaż, jak w czasie $O(nk)$ wyznaczyć najdłuższe wspólne podsłowo dla zbioru słów $\{t_i : 1 \leq i \leq k\}$ długości co najwyżej n.

Dowód. Rozwiązanie jest oczywiście bardzo proste. Wystarczy zbudować drzewo sufiksowe, ale nie dla każdego słowa osobno, ale jedno wspólne. Robimy to w ten sposób, że najpierw budujemy drzewo sufiksowe dla pierwszego słowa. Następnie wstawiamy do niego sufiksy drugiego słowa, trzeciego itd aż do k-tego, z tą różnicą, że wstawiając słowa, w wierzchołkach pamiętamy sobie informacje jaki sufiks tędy przechodzi w dół, tzn. każdy wierzchołek ma informacje na temat tego, sufiksy którego słowa przez niego przechodziły. Wówczas odpowiedzią, czyli najdłuższym wspólnym podsłowem, będzie najgłębiej położony węzeł w naszym drzewie zbudowanym dla n słów, przez który przeszedł jakiś sufiks każdego słowa. Oczywiście można tego LCSa bardzo łatwo odtworzyć jak już mamy taki najgłębszy wierzchołek o tej własności. Wystarczy przejść się od korzenia do niego i zobaczyć jakie etykiety mijaliśmy po drodze. □

3.1.4 Algorytm Weinera

Algorytm Weinera, historycznie najwcześniejszy algorytm optymalny, polega na budowaniu drzewa sufiksowego od prawej do lewej. Zauważmy, że dzięki przeglądaniu tekstu w tej kolejności gwarantujemy, że w i-tym kroku budowana struktura danych jest pełnoprawnym drzewem sufiksowym dla słowa $t[(n-i-1)..n]$.

Tak jak w algorytmie McCreighta, najważniejszą częścią algorytmu jest efektywne wyznaczanie linków między wierzchołkami drzewa – z tą różnicą, że w tym przypadku są to linki odwrotne do linków sufiksowych. Definiujemy je następująco: jeśli $word(v) = w$ ($w \in \mathcal{A}^*$) oraz $t[i] = a$, to $link_a[v] = u$, gdzie u jest pewnym wierzchołkiem, dla którego $word(u) = a\,word(v)$.

Niniejsza implementacja jest oparta na uproszczonej wersji algorytmu, zamieszczonej w Breslauer i Italiano (2013).

Algorytm 20: Algorytm Weinera budowania drzewa sufiksowego

```
def weiner(text, n):
  text = text + '$'
  root = trie.TrieNode("")
  link, head = { (root, ""): root }, root
  for i in range(n + 1, 0, -1):
    # niezmiennik: link[v][c] = u dla wewnętrznych u i v takich, że word(u) = c word(v)
    v, depth = head, n + 2
    while v != root and link.get((v, text[i])) is None:
      v, depth = v.parent, depth - len(v.label)
    u = link.get((v, text[i]))
    if u is None or text[depth] in u.children:
      if u is None:
        u, remaining = slow_find(root, text[depth - 1:])
      else:
        u, remaining = slow_find(u, text[depth:])
      v, _ = fast_find(v, text[depth:-remaining], False)
      depth = len(text) - remaining
      if u != root:
        link[(v, text[i])] = u
    leaf = trie.TrieNode(text[depth:])
    u.add_child(leaf)
    head = leaf
  return root, link
```

Problem 3.11. Pokaż, że Algorytm 20 wyznacza w każdej iteracji poprawne przejścia $link[v, c] = u$.

Problem 3.12. Pokaż, że Algorytm 20 zwraca poprawne drzewo sufiksowe.

Problem 3.13. Pokaż, że Algorytm 20 działa w czasie $O(n \log |\mathcal{A}|)$.

3.2 Tablica sufiksowa

Tablica sufiksowa jest dużo prostszą strukturą, wprowadzoną jako alternatywa dla drzew sufiksowych równolegle w (Gonnet, Baeza-Yates i Snider, 1992) i (Manber i Myers, 1993). Wartości tablicy dla słowa t długości n definiujemy następująco:

$$SA[i] = |\{j : t[j..n] \leq t[i..n]\}|,$$

czyli na i-tej pozycji mamy początkowy indeks i-tego najmniejszego sufiksu w porządku leksykograficznym.

Naiwny algorytm liczenia tablicy SA wymaga czasu $O(n^2 \log n)$, gdy wykorzystujemy sortowanie wymagające $O(n \log n)$ porównań, ponieważ każde porównanie wymaga w najgorszym przypadku czasu $O(n)$.

Można również obliczyć tablicę SA na podstawie zadanego drzewa sufiksowego w czasie $O(n)$: wystarczy przejrzeć drzewo wgłąb lub rekurencyjnie zgodnie z kolejnością liter w alfabecie. Zwróćmy uwagę, że Algorytm 21 wyznacza długości słów na podstawie znajomości tekstu i aktualnej głębokości przeszukiwania.

Algorytm 21: Algorytm budowania tablicy sufiksowej na podstawie drzewa sufiksowego

```
def suffix_array_from_suffix_tree(text, n):
  def traverse(v, depth):
    depth -= len(v.label)
    if len(v.children) == 0:
      return [depth]
    return [d for _, child in sorted(v.children.items())
            for d in traverse(child, depth)]
  ST, _ = suffix_tree.mccreight(text, n)
  return traverse(ST, n + 2)
```

3.3 Cel algorytmu i format wejścia

Wejściem do algorytmu jest słowo w o długości n nad alfabetem $\Sigma = [n]$ - liczb naturalnych od 1 do n. Zakładać będziemy, że $n = 2^k$ - jeśli nie, to można słowo w wydłużyć o czynnik liniowy, tak żeby ta równość zachodziła, dodając na koniec odpowiednią ilość $'\$'$, a po wykonaniu algorytmu "uciąć" z drzewa sufiksowego poddrzewa odpowiadające tym symbolom.

Algorytm tworzy drzewo sufiksowe dla tego słowa w czasie $O(n)$. Od drzewa wymagane jest, aby dla każdego wierzchołka, krawędzie prowadzące do jego dzieci posortowane były leksykograficznie względem ich etykiet (w przypadku liczb naturalnych porządek leksykograficzny rozumiemy jako zdefiniowany przez zwykłą relację $<$).

3.4 Idea algorytmu

Algorytm Faracha działa rekurencyjnie i każdy jego etap można zapisać w trzech krokach:

1. Zbuduj drzewo sufiksowe T_o dla *nieparzystych* sufiksów, czyli takich, które zaczynają się od nieparzystego indeksu (numerujemy od 1).
2. Korzystając z drzewa T_o, zbuduj drzewo T_e - drzewo sufiksowe dla *parzystych* sufiksów (bez wywołania rekurencyjnego!).
3. Połącz drzewa T_o i T_e w jedno drzewo T.

Jeśli kroki 2. i 3 będziemy w stanie wykonać w liniowej złożoności, to wówczas cały algorytm będzie miał złożoność:

$$T(n) = T\left(\frac{n}{2}\right) + O(n),$$

a więc liniową.
Aby uniknąć dodatkowego wywołania rekurencyjnego w równaniu (co zwiększyłoby złożoność do $O(nlgn)$), drzewo T_e konstruowane będzie bezpośrednio z drzewa T_o.

3.5 Tworzenie drzewa T_o

Tworzenie drzewa nieparzystych sufiksów przebiega w kilku krokach:

1. $\forall i \in [\frac{n}{2}]$ stwórz parę $\langle w[2i-1], w[2i]\rangle$ i wszystkie takie pary umieść w liście S' i każdą z nich zamień na jej *rangę* (indeks w posortowanej leksykograficznie liście). Na przykład, dla słowa `121112212221`, lista S' wygląda tak:
   `[(1,2), (1,1), (1,2), (2,1), (2,2), (2,1), #]`, a po zamianie elementów na ich rangi, tak:
   `[2,1,2,3,4,3,#]`.
2. Rekurencyjnie oblicz $T_{S'}$ - drzewo sufiksowe dla słowa S' oraz jego tablicę sufiksową składającą się z $A_{T_{S'}}$ - posortowanej tablicy sufiksów i z $LCP_{T_{S'}}$ - tablicy najdłuższych wspólnych prefiksów dla $A_{T_{S'}}$.
3. Korzystając z $A_{T_{S'}}$, oblicz A_{T_o} w następujący sposób:

   $$A_{T_o}[i] = 2A_{T_{S'}}[i] - 1,$$

   co bazuje na obserwacji, że każdy nieparzysty sufiks $w[2i-1]...w[n]$ jest równoważny sufiksowi $S'[i]...S'[\frac{n}{2}]$ (co wynika z konstrukcji S'), a zatem leksykograficzny porządek $A_{T_{S'}}$ jest taki sam jak porządek A_{T_o}. Jedyne co trzeba zmienić, to indeksy, odwracając przekształcenie z punktu 1.: $\langle w[2i-1], w[2i]\rangle \rightarrow S'[i]$.

4. Oblicz $LCP_{A_{T_o}}$, korzystając z $LCP_{A_{T_{S'}}}$, według wzoru:

$$LCP_{T_o}[i] = 2LCP_{T_{S'}}[i] + \begin{cases} 1 \text{ if } w[A_{T_o}[i] + 2LCP_{T_{S'}}] = w[A_{T_o}[i+1] + 2LCP_{T_{S'}}[i]] \\ 0 \text{ otherwise} \end{cases},$$

co również wynika z konstrukcji słowa S'.
5. Na podstawie A_{T_O} i LCP_{T_o} skonstruuj drzewo T_o.

3.6 Tworzenie drzewa T_e

1. Wstępnie przetwórz drzewo T_o (w czasie liniowym od jego rozmiaru) tak aby dało się odpowiadać na zapytania o `lca` na tym drzewie w czasie stałym.
2. Korzystając z obserwacji, że każdy parzysty sufiks to nieparzysty sufiks poprzedzony jednym znakiem, stwórz A_{T_e}. W tym kroku możemy wykorzystać obliczoną już listę A_{T_o}, "dokleić" odpowiedni znak przed każdym elementem A_{T_o} i użyć sortowania pozycyjnego tylko dla tego pierwszego znaku.
3. Oblicz tablicę LCP_{T_e} korzystając z zapytań o `lcp(word(a),word(b))`, czyli zapytań o `lca(a,b)` w drzewie sufiksowym:

$$lcp(w[2i,n], w[2j,n]) = \begin{cases} lcp(w[2i+1,n], w[2j+1,n]) + 1 \text{ if } w[2i] = w[2j] \\ 0 \text{ otherwise} \end{cases},$$

4. Stwórz T_e w oparciu o A_{T_e} i LCP_{T_e}.

3.7 Łączenie drzew T_o i T_e

Zauważmy, że aby stworzyć drzewo T z drzew T_o i T_e, wystarczy zbudować tablice A_T i LCP_T. Do uzyskania tych tablic potrzebujemy móc szybko (w czasie stałym) odpowiadać na pytania o najdłuższy wspólny prefiks dwóch sąsiadujących sufiksów w tablicy A_T. Umożliwiającą nam to wyrocznię uzyskamy poprzez *zachłanne* połączenie drzew T_o i T_e.

Zachłanne połączenie T' rozumiemy jako takie połączenie dwóch drzew (T_e i T_o), w którym:

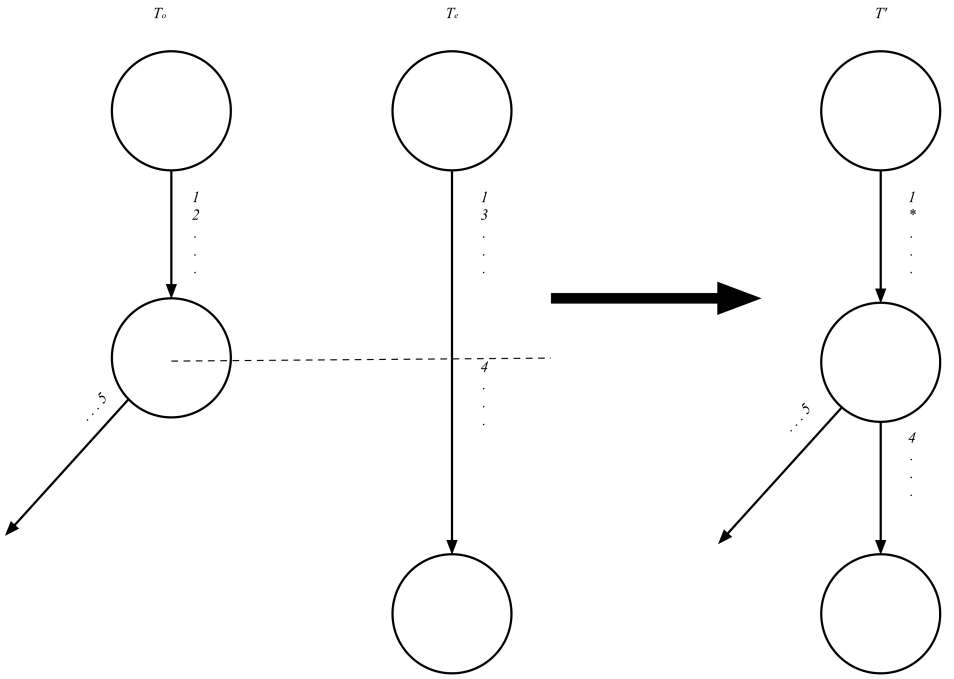
- Każdy wierzchołek z tych drzew ma odpowiadający sobie wierzchołek w drzewie T', a ponadto, jeśli $e \in T_e$, $o \in T_o$ i `word(e) = word(o)`, to wierzchołkom e i o odpowiada ten sam wierzchołek $t \in T$.
  Wierzchołek u *odpowiada* wierzchołkowi $v \in T_o \vee v \in T_e$ wtedy i tylko wtedy gdy $|word(u)| = |word(v)|$, a ponadto dla każdego wierzchołka a z T_o (lub T_e), który jest na drodze z korzenia do wierzchołka v zachodzi:

  $$word(v)[|word(a)| + 1] = word(u)[|word(a)| + 1],$$

  czyli $word(v)$ i $word(u)$ są tej samej długości i zgadzają się przynajmniej na indeksach, które odpowiadają pierwszym znakom na każdej krawędzi na drodze z korzenia do v.

- Z każdego wierzchołka $t \in T$, dla dowolnych dwóch etykiet w, v krawędzi z niego wychodzących zachodzi: $w[0] \neq v[0]$ ($w[1] \neq v[1]$ przy numeracji od 1).
- Drzewo T' nie zawiera żadnych innych wierzchołków.

W skrócie można powiedzieć, że T' to drzewo, które powstaje dokładnie w ten sposób co naiwnie wykonane poprawne połączenie T_o i T_e do T, z tą tylko różnicą, że zamiast porównywać wszystkie znaki na odpowiednich krawędziach (powiedzmy u i v) (i ewentualnie dzielić je jeśli różnią się na jakiejś pozycji), porównujemy tylko pierwsze znaki, a podział robimy tylko w miejscu, w którym skończy się krótsza z nich. Procedurę tworzenia T' (czyli "nałożenia na siebie" drzew T_o i T_e patrząc tylko na pierwszy znak krawędzi) przedstawia poniższy rysunek:



Wiedząc jak jest konstruowane drzewo T', zapiszmy kroki do stworzenia poprawnego drzewa T:

1. Stwórz T' tak jak zostało to wyżej opisane.
2. Wierzchołek drzewa T', który ma odpowiadający wierzchołek w T_o nazwijmy *nieparzystym*, a jeśli ma odpowiadający wierzchołek w T_e, to nazwijmy go *parzystym*. Pewne wierzchołki (np. korzeń) mogą być zarówno parzyste jak i nieparzyste.
   Teraz, dla każdego wierzchołka $u \in T'$ znajdź parę wierzchołków (jeśli istnieje) (a, b), które są jego potomkami i liśćmi drzewa, a ponadto a jest parzysty, b jest nieparzysty oraz `lca(a,b) = u`. Załóżmy, że `word(a) = w[2i,n], word(b) = w[2j-1,n]`. Dodatkowo, każdemu liściowi c takiemu, że `word(c) = w[k,n]` nadajmy etykietę l_k. Zgodnie z tą konwencją $a = l_{2i}$, $b = l_{2j-1}$.
   Zdefiniujmy funkcję d taką, że $d(u) = lca(l_{2i+1}, l_{2j})$ i policzmy ją dla każdego u. Zauważmy, że jeśli T' byłoby poprawnym drzewem, to $d(u) = suffix\_link(u)$.

3. Zakładając, że funkcja d definiuje drzewo, dla każdego wierzchołka u policz głębokość $L(u)$ w tym drzewie. Skorzystaj z faktu, że $lcp(w[2i,n],w[2j-1]) = L(lca(l_{2i},l_{2j-1}))$ i stwórz tablicę LCP_T.

3.8 Dowód poprawności

Wystarczy udowodnić poprawność punktów 2. i 3.

Twierdzenie 3.6. *Najdłuższy wspólny prefiks dowolnej pary liści (a,b), takich, że a - parzysty, b - nieparzysty, i takich, że* $\texttt{lca}(a,b) = u$*, jest taki sam.*

Dowód. Weźmy wierzchołek parzysty u mający zarówno parzystego jak i nieparzystego potomka. Weźmy teraz dwóch parzystych potomków (być może takich samych) u: l_{2i}, l_{2j}. Wówczas, $\texttt{lcp}(w[2i,n],w[2j,n]) \geq |word(u)|$, ponieważ u, l_{2i}, l_{2j} były w drzewie T_e.

Teraz, weźmy parę $l_{2i'-1}$, $l_{2j'-1}$ potomków u. Zachodzi wtedy $\texttt{lcp}(w[2i'-1,n],w[2j'-1,n]) \geq |word(u)|$. Jest tak, ponieważ, jeśli u był również w T_o, to działa ten sam argument co poprzednio. Jeśli nie był w T_o, to ponieważ $l_{2i'-1}$, $l_{2j'-1}$ są liśćmi, to musi istnieć wierzchołek $u' = \texttt{lca}(l_{2i'-1},l_{2j'-1}) \in T_o$, który jest potomkiem u w T'.

Rozważmy teraz parę liści $l_{2i''}$ i $l_{2j''-1}$ w T', takich że $u = \texttt{lca}(l_{2i''},l_{2j''-1})$. Zachodzi wtedy $\texttt{lcp}(w[2i'',n],w[2j''-1,n]) \leq |word(u)|$. Wynika to z faktu, że jeśli byłoby inaczej, to wtedy więcej niż jedna krawędź wychodząca z u zaczynała by się tym samym znakiem, co jednak w T' nie ma miejsca. Analogiczne rozumowanie możemy przeprowaddzić dla u będącego wierzchołkiem nieparzystym i wówczas każda z powyższych nierówności również będzie zachodziła.

Weźmy w końcu liście $l_{2i'},$, $l_{2i''}$, $l_{2j'-1}$, $l_{2j''-1}$ takie, że $\texttt{lca}(l_{2i'},l_{2j'-1}) = \texttt{lca}(l_{2i''},l_{2j''-1}) = u$. Wtedy, $\texttt{lcp}(w[2i',n],w[2j'-1,n]) = k \leq |word(u)|$, co udowodniliśmy przed chwilą. Wiemy jednak, że $\texttt{lcp}(w[2i',n],w[2i'',n]) \geq |word(u)| \geq k$, a zatem musi zachodzić także $\texttt{lcp}(w[2i''\ldots n],w[2j'-1\ldots n]) = k$. Podobnie, $\texttt{lcp}(w[2i''\ldots n],w[2j''-1\ldots n]) = k$. Wobec tego:

$$lcp(w[2i',n],w[2j'-1,n]) = lcp(w[2i'',n],w[2j''-1,n]) = k.$$

□

Twierdzenie 3.7. *Funkcja d definiuje drzewo na wierzchołkach T', a ponadto dla dowolnych l_{2i} i l_{2j-1} zachodzi* $L(\texttt{lca}(l_{2i},l_{2j-1})) = \texttt{lcp}(w[2i,n],w[2j-1,n])$.

Dowód. Dowód jest indukcyjny ze względu na długość $\texttt{lcp}$. Jeśli $\texttt{lcp}word[2i,n],word[2j-1,n]) = 0$, to wówczas $word[2i,n]$ i $word[2j-1,n]$ różnią się na pierwszym znaku, więc $\texttt{lca}(l_{2i},l_{2j-1}) = \texttt{root}$, co wynika ze sposobu konstruowania drzewa T'.

Załóżmy teraz, że twierdzenie zachodzi dla pary sufiksów parzystych i nieparzystych, takich, że ich $\texttt{lcp} < k$. Niech l_{2i}, l_{2j-1} będą liśćmi takimi, że $\texttt{lca}(w[2i,n],w[2j-1,n]) = k > 0$. Ponadto, niech $u = \texttt{lca}(l_{2i},l_{2j-1})$. Wtedy u nie jest korzeniem, bo $k > 0$.

Niech teraz $l_{2i'}$, $l_{2j'-1}$ będą liśćmi użytymi do zdefiniowania $d(u)$. Zatem, z definicji, $d(u) = \texttt{lca}(l_{2i'+1}, l_{2j'})$.

Z założeń indukcyjnych wiemy, że funkcja d definiuje drzewo na dotychczas rozważonych wierzchołkach, a ponadto, że funkcja L jest zdefiniowana na tych wierzchołkach i że zwraca poprawne wartości `lcp`. Zatem, z faktu, że funkcja d definiuje drzewo, wiemy że $L(u) = 1 + L(d(u))$. Ponadto, z faktu, że funkcja L zwraca poprawne wartości dla dotychczas rozważonych wierzchołków wiemy, że $1 + L(d(u)) = 1 + \texttt{lcp}(w[2i' + 1, n], w[2j', n])$. Teraz, ponieważ $k > 0$ (w szczególności oznacza to, że $w[2i'] = w[2j' - 1]$), zachodzi $1 + \texttt{lcp}(w[2i' + 1, n], w[2j', n]) = \texttt{lcp}(w[2i', n], w[2j' - 1, n])$. A z poprzedniego twierdzenia wiemy, że $\texttt{lcp}(w[2i', n], w[2j' - 1, n]) = \texttt{lcp}(w[2i, n], w[2j - 1, n])$.

Wobec tego, $L(u) = \texttt{lcp}(w[2i, n], w[2j - 1, n])$, a ponadto struktura zdefiniowana przez funkcję d jest drzewem. □

3.9 Złożoność

3.9.1 Tworzenie T_o

1. Sortowanie pozycyjne dla par, w których maksymalna liczba jest mniejsza lub równa n (tak jest, bo początkowe słowo spełnia to założenie, a każde następne powstaje przez zastąpienie elementu przez jego rangę) zajmuje $O(n)$ czasu.
2. Jeśli wszystkie inne kroki będą liniowe, to ten krok zajmie $T(n) = T(\frac{n}{2}) + O(n) = O(n)$ czasu.
3. Każdy element tablicy A_{T_o} liczymy w czasie stałym, więc ten krok zajmuje $O(n)$ czasu.
4. Podobnie jak w poprzednim punkcie, mamy stały czas na przetworzenie jednego elementu tablicy, więc całość robimy w $O(n)$.
5. Jest możliwe skonstruowanie drzewa sufiksowego w czasie liniowym z tablic A i LCP.
   Idea: wstawiamy kolejne sufiksy w kolejności leksykograficznej.
   Po wstawieniu każdego sufiksu, zatrzymujemy się w liściu, który go reprezentuje i dopóki $\texttt{lcp}(word(v), next\_suffix) < |word(v)|$ wykonujemy, $v = v.parent$, a następnie dodajemy nową krawędź z wierzchołka v albo dzielimy już wychodzącą.

3.9.2 Tworzenie T_e

1. Da się w czasie liniowym zrobić preprocessing drzewa tak aby w czasie stałym odpowiadać na zapytania `lca`. Jest to nietrywialny algorytm, który jest opisany w `https://www.ics.uci.edu/~eppstein/261/BenFar-LCA-00.pdf`.
2. Sortowanie pozycyjne na liczbach nieprzekraczających n działą w $O(n)$.
3. Ponieważ zapytania o `lcp` przetwarzane są w czasie stałym, czas działania tego kroku to $O(n)$.
4. Analogicznie jak dla T_o - $O(n)$.

3.9.3 Łączenie drzew T_o i T_e

1. Ten krok to zwykły DFS działający liniowo względem sumarycznej ilości wierzchołków w drzewach T_o i T_e, a ponieważ są to poprawne drzewa sufiksowe, to mają liniowe rozmiary względem $O(\frac{n}{2})$.
2. Odpowiednią parę liści dla każdego wierzchołka z T' możemy znaleźć przy pomocy jednego DFS-a, korzystając z obliczonych wcześniej wartości dla potomków tego wierzchołka.
   Funkcję d liczymy dla każdego wierzchołka w czasie stałym, po wcześniejszym liniowym przetworzeniu drzewa T' do zapytań o `lca`.
3. Głębokość w drzewie liczymy ponownie używając algorytmu DFS i korzystając z obliczonych wartości, każdy element LCP_T liczymy w czasie stałym. Odtworzenie drzewa z tablic A_T i LCP_T wykonujemy w czasie $O(n)$, jak wyżej.

3.9.4 Algorytm Karpa-Millera-Rosenberga

Algorytm *prefix-doubling*, zaproponowany w (Karp, Miller i Rosenberg, 1972), opiera się na pomyśle prostej rekurencji. Załóżmy, że dla podsłów $t[i : i + k]$[1] dla pewnego $k \geq 1$ oraz $1 \leq i \leq n$ mamy dostęp do funkcji $rank(i, k)$, zwracającej numer $t[i : i + k]$ w porządku leksykograficznym. Wówczas wystarczy umieć wyznaczać na jej podstawie efektywnie wartości $rank(i, 2k)$ dla wszystkich $1 \leq i \leq n$.

Algorytm 22: Algorytm Karpa-Millera-Rosenberga budowania tablicy sufiksowej

```
def reverse(SA):
  reverse = [0] * len(SA)
  for i in range(len(SA)):
    reverse[SA[i] - 1] = i + 1
  return reverse

def prefix_doubling(text, n):
  '''Computes suffix array using Karp-Miller-Rosenberg algorithm'''
  text += '$'
  mapping = {v: i + 1 for i, v in enumerate(sorted(set(text[1:])))}
  R, k = [mapping[v] for v in text[1:]], 1
  while k < 2 * n:
    pairs = [(R[i], R[i + k] if i + k < len(R) else 0) for i in range(len(R))]
    mapping = {v: i + 1 for i, v in enumerate(sorted(set(pairs)))}
    R, k = [mapping[pair] for pair in pairs], 2 * k
  return reverse(R)
```

[1] Ściślej chodzi nam o $t[i : \min\{i + k, n\}]$ – albo można równoważnie założyć, że na końcu dodajemy nieskończony ciąg symboli \$.

Lemat 3.8 (Crochemore, Hancart i Lecroq 2007, Lemma 4.8, s. 169-170)**.** *Dla dowolnych* $1 \leq i \leq n$ *oraz* $k \geq 1$ *wartość* $rank(i, 2k)$ *jest równa pozycji pary* $(rank(i, k), rank(i + k, k))$ *w porządku leksykograficznym na wszystkich takich parach.*

Dowód. Z założenia, $rank(i, k) < rank(j, k)$ wtedy i tylko wtedy, gdy $t[i : i+k] < t[j : j+k]$ oraz $rank(i, k) = rank(j, k)$ wtedy i tylko wtedy, gdy $t[i : i + k] = t[j : j + k]$ dla dowolnych i, j, k.

Jeśli $t[i : i+k] < t[j : j+k]$, to oczywiście zachodzi $t[i : i+2k] < t[j : j+2k]$ dla dowolnych i, j, k. Podobnie jeśli $t[i : i + k] = t[j : j + k]$ oraz $t[i + k + 1 : i + 2k] = t[j + k + 1 : j + 2k]$, to również $t[i : i + 2k] < t[j : j + 2k]$ dla dowolnych i, j, k.

Składając te wszystkie obserwacje razem, otrzymujemy wprost, że $t[i : i+2k] < t[j : j+2k]$ wtedy i tylko wtedy, gdy $rank(i, 2k) < rank(j, 2k)$ dla dowolnych i, j, k.

Aby zakończyć dowód wystarczy zaobserwować, że dla dwóch sąsiednich elementów w porządku $rank(i, 2k)$ nie może również istnieć żaden element rozdzielający je w porządku par $(rank(i, k), rank(i + k, k))$. □

Wniosek 3.9. Algorytm 22 zwraca poprawną tablicę sufiksową.

Dowód. Po $\log n$ iteracjach głównej pętli otrzymujemy z $rank(i, 1)$ tablicę wartości $rank(i, n)$ – a to z definicji jest dokładnie tablica sufiksowa. □

Twierdzenie 3.10. *Algorytm 22 wykonuje się w czasie* $O(n \log n)$.

Dowód. Całkowita złożoność algorytmu 22 zależy od zastosowanego sortowania. Jeśli jest to sortowanie pozycyjne (*radix sort*), to pojedyncza iteracja wymaga czasu $O(n)$. Iteracji mamy dokładnie $\log n$. □

3.9.5 Algorytm Kärkkäinena-Sandersa

Algorytm *skew*, wprowadzony w (Kärkkäinen i Sanders, 2003), umożliwia zejście do liniowego czasu obliczania tablicy sufiksowej. Wymaga on założenia o tym, że alfabet można utożsamić ze zbiorem liczb naturalnych. W przeciwieństwie jednak np. do problemu sortowania w algorytmach tekstowych jest to założenie bardzo naturalne i często spotykane w praktyce (podobnie jak założenie, że $|\mathcal{A}| = O(1)$).

Sednem algorytmu Kärkkäinena-Sandersa jest zamiana słowa t na słowo t' takie, że $|t'| = \frac{2}{3}|t|$. Zauważmy, że bez straty ogólności możemy założyć, że słowo t jest długości podzielnej przez 3 – wystarczy dołożyć na koniec odpowiednią liczbę znaków $\# \notin \mathcal{A}$ takich, że $\# > a$ dla każdego $a \in \mathcal{A}$.

Dokładniej, niech $triple(i)$ oznacza pozycję $t[i : i+3]$ wśród wszystkich unikalnych trzyliterowych podsłów t posortowanych leksykograficznie. Tworzymy nowe słowo t' w następujący

sposób:

$$t' = triple(1) \cdot triple(4) \ldots \cdot triple(2) \cdot triple(5) \ldots$$

Zwrócony wynik dla tego słowa umożliwia odrębne posortowanie sufiksów t osobno dla zbiorów

$$P_{12} = \{1 \leq i \leq n : i \mod 3 \in \{1, 2\}\}$$
$$P_0 = \{1 \leq i \leq n : i \mod 3 = 0\}.$$

Co więcej, na podstawie danego t' możliwe jest scalenie również częściowych tablic sufiksowych dla P_{12} i P_0.

Algorytm 23: Algorytm Kärkkäinena-Sandersa budowania tablicy sufiksowej

```
def skew(text, n):
  '''Computes suffix array using Kärkkäinen-Sanders algorithm'''
  def convert(data):
    # zamiana tablicy liczb na string UTF-32 w tym samym porzadku znakow
    return '#' + ''.join(chr(ord('0') + v) for v in data)
  def compare(i, j):
    if i % 3 == 1:
      return (text[i], S.get(i + 1, 0)) >= (text[j], S.get(j + 1, 0))
    return (text[i:i + 2], S.get(i + 2, 0)) >= (text[j:j + 2], S.get(j + 2, 0))

  if n <= 4:
    return naive(text, n)
  text += '$'
  P12 = [i for i in range(1, n + 2) if i % 3 == 1] \
      + [i for i in range(1, n + 2) if i % 3 == 2]
  triples = _rank([text[i:i + 3] for i in P12])
  recursion = skew(convert(triples), (2 * n + 1) // 3 + 1)[1:]
  L12 = [P12[v - 1] for v in recursion]

  mapping = {v: i + 1 for i, v in enumerate(L12)}
  S = {v: mapping[v] for v in P12}

  P0 = [i for i in range(1, n + 2) if i % 3 == 0]
  tuples = [(text[i], S.get(i + 1, 0)) for i in P0]
  L0 = [P0[i - 1] for i in _reverse(_rank(tuples))]
  return _merge(L12, L0, compare = compare)
```

Problem 3.14. Pokaż że 23 na podstawie t' wyznacza poprawnie tablicę sufiksów dla P_{12}.

Problem 3.15. Pokaż że 23 na podstawie t' wyznacza poprawnie tablicę sufiksów dla P_0.

Twierdzenie 3.11. *Algorytm 23 na podstawie t' poprawnie scala tablice sufiksów dla P_{12} i P_0.*

Dowód. Na mocy założenia mamy dostępne posortowane listy sufiksów osobno dla indeksów należących do P_{12} i P_0.

Scalenie odbywa się zgodnie z funkcją *merge* identyczną jak w algorytmie mergesort. Mamy 2 przypadki:

1. jeśli mamy $i \in P_{12}$ takie, że $i \mod 3 = 2$ oraz pewne $j \in P_0$, to wiemy, że $i+1, j+1 \in P_{12}$ – wystarczy zatem porównać ze sobą pary $(t[i], S[i+1])$ i $(t[j], S[j+1])$,
2. jeśli mamy $i \in P_{12}$ takie, że $i \mod 3 = 1$ oraz pewne $j \in P_0$, to wiemy, że $i+2, j+2 \in P_{12}$ – wówczas postępujemy podobnie, tylko porównujemy trójki $(t[i], t[i+1], S[i+2])$ i $(t[j], t[j+1], S[j+2])$.

Zauważmy, że $S[i+1]$, $S[i+2]$ lub $t[i+1]$ mogą nie istnieć (podobnie dla j). Wówczas wystarczy zwracać wartości, które będą zawsze mniejsze od wartości odpowiednio z S lub $\mathcal{A}$. □

Twierdzenie 3.12. *Czas działania algorytmu 23 wynosi $O(n)$.*

3.9.6 Wykonywanie zapytań w czasie $O(m \log n)$

Sprawdzanie, czy słowo w o długości m występuje w tekście t dla danej tablicy sufiksowej $SA(t)$ jest wykonalne w czasie $O(m \log n)$: wystarczy zwykłe binarne wyszukiwanie słowa w w $SA(t)$.

Co więcej, tablica sufiksowa umożliwia nie tylko zliczenie wszystkich wystąpień słowa w w tekście w czasie $O(m \log n)$, ale również odnalezienie ich w czasie $O(m \log n + k)$. Wystarczy tylko wyszukać pozycje odpowiadające słowom w oraz $w\#$, gdzie $\# \notin \mathcal{A}$ oraz $a < \#$ dla każdego $a \in \mathcal{A}$. Pozycje te wyznaczają w tablicy $SA(t)$ sekwencję początków sufiksów, dla których w musi być prefiksem – chociaż niekoniecznie w kolejności rosnącej.

3.9.7 Wykonywanie zapytań w czasie $O(m + \log n)$

Tablica sufiksowa SA zawiera początkowe indeksy leksykograficznie uporządkowanych sufiksów słowa t. Oznaczmy taki sufiks wskazany przez indeks a w SA jako $t_a = t[SA[a] \dots n]$. Mając tablicę sufiksową, można ją wykorzystać do indeksowania tekstu – poprzez zwykłe wyszukiwanie binarne można odnaleźć pierwszą (l_w) i ostatnią (r_w) pozycję w SA taką, że w jest prefiksem $t_i, i \in l_w \leq i \leq r_w$. Wtedy $SA[i]$ wskazują wszystkie wystąpienia w w t. Jednak to podejście wymaga $O(m \log n)$ czasu.

Manber i Myers w 1993 r. zaproponowali prosty algorytm przyśpieszający takie zapytania do $O(m + \log n)$. Wykorzystywana jest do tego struktura LCP-LR zawierająca najdłuższe wspólne prefiksy (lcp) sufiksów w.

3.9.8 Zapytania do SA z wykorzystaniem LCP-LR

Załóżmy, że mamy tablicę SA oraz LCP-LR (jej konstrukcja zostanie omówiona w następnej sekcji). Informację o lcp sufiksów t można wykorzystać do przyśpieszenia wyszukiwania binarnego, ograniczając liczbę wykonywanych porównań symboli.

Oznaczmy jako l i r lewą i prawą granicę rozpatrywanego przedziału SA w wyszukiwaniu binarnym. $c = (l + r)/2$ jest zatem pozycja oznaczająca sufiks t_c, z którym porównywamy wyszukiwane słowo w. Dalej oznaczmy $L = lcp(t_l, w)$ i $R = lcp(t_r, w)$. Podczas biegu algorytmu będą wartości L i R aktualizowane (niezmiennik), i to z wykorzystaniem minimalnej ilości porównań. Niech $H = \max(L, R)$ i $C = lcp(t_c, w)$.

Bez straty ogólności załóżmy, że w danym cyklu wyszukiwania $H = L \geq R$. Mogą zajść trzy przypadki (w praktyce można pierwsze dwa implementować razem):

1. $lcp(t_l, t_c) > L$
   Wtedy $C = L$.
   w ma pierwszych L znaków zgodnych z t_l i różni się $L + 1$-szym. Skoro t_l i t_c zgadzają się przynajmniej w pierwszych $L + 1$ znakach, w i t_c będą również mieli pierwszych L znaków identycznych i $L + 1$-szy inny, więc $C = L$.
2. $lcp(t_l, t_c) = L$
   W takim przypadku musimy porównać $L + 1, L + 2 \ldots$-ty symbol w i t_c póki nie znajdziemy i takie, że $w[L + i] \neq t_c[L + i]$. Wtedy $C = L + i - 1$.
   Podobnie jak w przypadku 1, skoro w i t_l są identyczne w pierwszych L znakach oraz t_l i t_c są też identyczne w pierwszych L znakach, to pierwszych L znaków w i t_c będzie również identycznych. Jednak $lcp(w, t_c)$ może być dłuższy od L, więc musimy zrobić skan po znakach dalej.
3. $lcp(t_l, t_c) < L$
   Wtedy $C = lcp(t_l, t_c)$.
   Analogicznie do przypadku 1, tylko na odwrót.

Kiedy znamy C, wystarczy porównać $C + 1$-szy symbol w i t_c, żeby zdecydować, czy przesunąć lewą lub prawą granicę przeszukiwania oraz czy zaktualizować wartość L lub $R = C$. W przypadku, kiedy $C = m$, decydujemy według tego, czy szukamy l_w (przesuwamy r) albo r_w (przesuwamy l).

Twierdzenie 3.13. *Algorytm poprawnie znajduje l_w/r_w.*

Dowód. Algorytm działa na takiej samej zasadzie, co podstawowe wyszukiwanie binarne, tylko z szybką identyfikacją pierwszego różniącego się symbolu. □

Twierdzenie 3.14. *Algorytm działa w czasie $O(m + \log n)$.*

Dowód. Tak samo, jak w wyszukiwaniu binarnym, będziemy musieli wykonać w najgorszym przypadku $O(\log n)$ cykli. Zauważmy, że poza jednym obowiązkowym porównaniem znaku w każdym cyklu, dodatkowe porównywania są wykonywane tylko w opcji 2 z wyżej wymienionych. Biorąc pod uwagę, że jest to jedyna opcja, przy której się zmienia H (jeżeli zachodzi opcja 3, wtedy szukane l_w/r_w znajduje się w lewej połowie i w związku z tym aktualizujemy R, czyli mniejszą z wartości L, R), że H może tylko rosnąć (przybliżamy się do szukanej pozycji) oraz że H jest z definicji ograniczone przez m, takich porównań będzie maksymalnie m. Łącznie więc otrzymujemy $O(m + \log n)$ porównań. □

LCP-LR i jej konstrukcja

LCP-LR pozwala na zapytania o *lcp* dwu sufiksów t w czasie $O(1)$. Jeżeli mielibyśmy przechowywać tę informację o każdych dwu sufiksach t, *LCP-LR* zajmowałaby $O(n^2)$ pamięci, jednak ograniczenie się do par potrzebnych w trakcie wyszukiwania binarnego redukuje wymagania pamięciowe do $O(n)$.

Manber i Meyers proponują implementację *LCP-LR* jako dwu tablic *Llcp* i *Rlcp*, gdzie dla każdej trójki pozycji (r, c, l), która może zostać rozpatrzona w trakcie wyszukiwania binarnego, $Llcp[c] = lcp(t_l, t_c)$ i $Rlcp[c] = lcp(t_r, t_c)$. Alternatywnie można zastosować słownik (hash table) indeksowany dwójkami $LCP\text{-}LR[(a, b)] = lcp(t_a, t_b)$.

Mając tablicę LCP (obliczoną np. algorytmem Kasai), potrzebne *lcp* można łatwo uzyskać z wykorzystaniem rekursji:

$$lcp(t_l, t_r) = \begin{cases} LCP[r] & \text{jeżeli } r = l + 1 \\ min(lcp(t_l, t_c), lcp(t_c, t_r)) & \text{wpp} \end{cases}$$

gdzie $c = (l + r)/2$.

Twierdzenie 3.15. *Powyższy algorytm zwraca poprawne wartości LCP-LR dla wszystkich potrzebnych* (a, b).

Dowód. Algorytm wybiera pary (a, b) w dokładnie ten sam sposób co wyszukiwanie binarne, uzyskamy więc dokładnie te same (a, b). Biorąc pod uwagę, że sufiksy w SA są uporządkowane leksykograficznie, $lcp(t_a, t_b) = \min(lcp(t_a, t_{a+1}), lcp(t_{a+1}, t_{a+2}) \dots lcp(t_{b-1}, t_b)) = \min_{a+1 \le i \le b} LCP[i]$, co gwarantuje poprawność wyników. □

Twierdzenie 3.16. *LCP-LR uzyskujemy w czasie* $O(n)$.

Dowód. Ilość porównań jest z góry ograniczona przez

$$\sum_{i=0}^{\log_2 n} \frac{n}{2^i} = 2n \in O(n)$$

□

3.9.9 Dalsze zagadnienia

Li, Li i Huo (2018) pokazali algorytm obliczania tablicy sufiksowej w czasie $O(n)$ wymagający zaledwie $O(1)$ dodatkowej pamięci.

4 Algorytm small-large

4.1 Tablica sufiksów

Dla słowa s niech T_i oznacza sufiks rozpoczynający się na indeksie i, a t_i to znak na i-tej pozycji. Tablica sufiksów słowa s to tablica zawierająca indeksy sufiksów s posortowanych według porządku leksykograficznego tych sufiksów. Tablica ta przydaje się np. przy budowaniu drzew sufiksowych. Poniżej przedstawiony jest algorytm budujący tablicę sufiksów w czasie i pamięci liniowej.

W poniższych rozważaniach bez straty ogólności zakładamy, że każde słowo s kończy się znakiem unikalnym i najmniejszym w słowie. Zakładamy również że alfabet to $\{1, 2, ..., n\}$.

4.2 SL - kategoryzacja sufiksów

Na początek przedstawimy kategoryzacje prefiksów na typy S (small) i L (large). Główna idea algorytmu jest oparta właśnie na tej kategoryzacji i jej własnościach.

Definicja 4.1. Sufiks i jest typu S/L, gdy jest mniejszy/większy od sufiksu $i + 1$. Sufiks n jest S i L (zależnie od potrzeby).

Można, używając poniższego faktu, w czasie i pamięci liniowej dokonać kategoryzacji sufiksów na typy S/L.

Obserwacja 4.2. Dla sufiksu T_i $(i < n)$:

- jeśli $t_i \neq t_{i+1}$, to wystarczy porównać jeden znak aby dowiedzieć się jakiego typu jest T_i
- jeśli $t_i = t_{i+1}$, to znajdź pierwszy $j > i$, taki że $t_j \neq t_i$, wtedy
  $t_j > t_i \Rightarrow T_i, T_{i+1}, ..., T_{j-1}$ są typu S
  $t_j < t_i \Rightarrow T_i, T_{i+1}, ..., T_{j-1}$ są typu L

Przedstawmy również ważną własność kategoryzacji SL.

Lemat 4.3. *Sufiks typu S jest leksykograficznie większy od każdego sufiksu typu L, który rozpoczyna się tym samym znakiem.*

Dowód. Dowód nie wprost - załóżmy że istnieją sufiksy: $T_i = c\alpha c_1\beta$ typ S i $T_j = c\alpha c_2\gamma$ typ L, $c, c_1, c_2 \in \Sigma$, $\alpha, \beta, \gamma \in \Sigma^*$, $c_1 \neq c_2$, takie że $T_i < T_j$.

Case 1: α zawiera znak inny niż c
Niech c_3 - znak $\neq c$ najbardziej na lewo w α. T_i jest typu S, więc $c_3 > c$. Podobnie z T_j typu L otrzymujemy $c_3 < c$, sprzeczność.

Case 2: α zawiera tylko c lub jest puste
Z typów T_i i T_j mamy $c_1 \geq c$ i $c \geq c_2$ (dowód podobnie jak case 1). Stąd $c_1 \geq c_2$, co daje sprzeczność z $T_i < T_j$.

□

4.3 Algorytm small-large

Na algorytm można patrzeć w dwóch fazach - sortowanie sufiksów S/L, w zależności których z nich jest mniej. Następnie wykorzystujemy otrzymane informacje utworzenie porządku na wszystkich sufiksach. Wybieranie gałęzi z mniejszą ilością prefiksów danego typu jest konieczne aby zachować złożoność, ale nie wpływa na poprawność algorytmu.

procedure SMALL-LARGE(S)
$SL \leftarrow$ SL-CATEGORIZATION(s)
if $SL.count(\acute{S}") < len(n)/2$
$sortedS \leftarrow$ SORTS(s, SL)
return FINISHS(s, SL, $sortedS$)
Else
$sortedL \leftarrow$ SORTL(s, SL)
return FINISHL(s, SL, $sortedS$)
end procedure

W kolejnych sekcjach opisane zostaną bardziej szczegółowo funkcje $sortS$ i $finishS$. Procedury dla typu L są bardzo podobne - należy przeglądać niektóre listy w odwrotnej kolejności, operacje przeniesienia na początek/koniec grupy zamienić na przeniesienie na koniec/początek i zastąpić odniesienia do typu S odniesieniami do typu L.

4.4 Sortowanie S sufiksów - faza 1

Definicja 4.4. S odległość - dystans od i do najbliższego na lewo S sufiksu (nie wliczając i).

Definicja 4.5. S podsłowo - podsłowo $t[i, i+1, ..., j]$, gdzie i oraz j to S sufiksy, a pomiędzy są same L sufiksy.

Definicja 4.6. Jeśli słowo $t[i, i+1, ..., j]$ jest S podsłowem, to każdy jego prefiks nazywamy prefiksem S podsłowa.

S podsłowa są ściśle powiązane z S sufiksami - gdy weźmiemy ciąg kolejnych S podsłów i skleimy je ze sobą, otrzymamy S sufiks modulo duplikaty znaków na zetknięciu podsłów.

procedure SORTS(s, SL)
$SsubstrPrefixes \leftarrow$ S substring prefixes, divided by Sdist, bucketed and sorted by first character
$Ssubstr \leftarrow$ S substrings, bucketed and sorted by first character
For $d \leftarrow 1, ..., max\_Sdist$
For j in $SsubstrPrefixes[d]$
move $j - d$ to the front of its group in $Ssubstr$
EndFor
update groups in $Ssubstr$ using old $Ssubstr$ and $SsubstrPrefixes[d]$
EndFor
$s' \leftarrow$ map $Ssubstr$ to its buckets number, ordered as in s
return unmap SMALL-LARGE(s')
end procedure

Po pętli for S podsłowa są uporządkowane w kolejności prawie leksyko-graficznej. Jedyny wyjątek to gdy α jest podsłowem β, wtedy $\alpha > \beta$ w tym porządku.

Oznaczmy jako T'_i sufiks w s' odpowiadający S sufiksowi T_i. Poprawność sortowania wynika z poniższego faktu.

Lemat 4.7. $T_i < T_j \iff T'_i < T'_j$

Dowód.

$T_i < T_j \Rightarrow T'_i < T'_j$ - trywialnie z mapowania

$T_i < T_j \Leftarrow T'_i < T'_j$ - opis idei
Rozważa się pierwszy znak różniący $T'_i < T'_j$. Jest to numer jakiejś grupy, która indukuje S podsłowo. Odzyskujemy te podsłowa α, β i rozważamy dwa przypadki - β to prefiks α lub nie. Z tego można wywnioskować $T_i < T_j$. □

4.5 Sortowanie całości - faza 2

procedure FINISHS(s, SL, sortedS)
$A \leftarrow$ bucket and sort s suffixes by first character
For x in $reversed(sortedS)$ ▷ Lemat 4.3
move x to the end of its bucket in A
move that buckets end to the left by one
EndFor
For $i \leftarrow 1, ..., n$ ▷ lemat 4.8
If $SL[A[i] - 1] ==$ "$\ddot{L}$"
move $A[i] - 1$ to the front of its bucket
move that buckets front to the right by one
EndIf

EndFor
return A
end procedure

Po pierwszej pętli sufiksy typu S są już na poprawnych pozycjach. Pętla druga porządkuje resztę.

Lemat 4.8. *W kroku i drugiej pętli algorytmu, sufiks $T_{A[i]}$ jest już na poprawnej pozycji w tablicy sufiksów.*

Dowód. Indukcja.

Baza indukcji: wynika z tego, że najmniejszy sufiks jest typu S wiec jest już na poprawnej pozycji

Krok indukcyjny $i \to i+1$: nie wprost - istnieje sufiks $k > i+1$, który w tablicy sufiksów powinien zająć miejsce $A[i+1]$.
Wprost z założenia $T_{A[k]} < T_{A[i+1]}$ i obydwa te sufiksy są typu L - sufiksy S są na poprawnych miejscach.
$T_{A[k]}$ i $T_{A[i+1]}$ muszą rozpoczynać się tą samą literą c - przez wcześniejsze dzielenie na grupy przez pierwszą literę $T_{A[i+1]}$ ma jako pierwszą najmniejszą literę z sufiksów $A[i+1], A[i+2], ..., A[n]$. Niech $T_{A[i+1]} = c\alpha$ i $T_{A[k]} = c\beta$.

$$T_{A[k]} \; is \; type \; L \Rightarrow \beta < T_{A[k]}$$
$$T_{A[k]} < T_{A[i+1]} \Rightarrow \beta < \alpha$$
$$(T_{A[k]} \; is \; A[i+1]th \; in \; suffix \; array) \wedge \beta < T_{A[k]} \Rightarrow \beta \in \{A[1], A[2], ..., A[i]\}$$

$\beta < \alpha$, więc $T_{A[k]}$ był przesunięty na początek grupy (w gałęzi if) przed $T_{A[i+1]}$. Sufiksy te należą do tej samej grupy, zatem $T_{A[k]}$ leży na lewo od $T_{A[i+1]}$, sprzeczność. □

Z Lemat 4.3 i Lemat 4.8 wynika poprawność algorytmu.

4.6 Złożoność

Ze względu na rekurencję $T(n) = T(n/2) + \mathcal{O}(n)$ otrzymujemy złożoność czasową i pamięciową liniową w stosunku do długości słowa.

4.7 Tablica największych wspólnych prefiksów

Konstrukcja tablicy LCP.

Twierdzenie 4.9. *Algorytm Kasai zwraca poprawną tablicę sufiksową.*

Dowód. Weźmy dwa dowolne sufiksy słowa t sąsiednie w porządku leksykograficznym tj. zaczynające się na pozycjach $SA[i]$ oraz $SA[i+1]$ dla pewnego $1 \leq i \leq n-1$. Niech długość najdłuższego wspólnego prefiksu $LCP(t[SA[i] : n], t[SA[i+1] : n]) = k$ dla pewnego $k \geq 0$.

Weźmy teraz j takie, że $SA[j] = SA[i] + 1$, tj. $t[SA[j] : n] = t[SA[i] + 1 : n]$. Wówczas wiemy, że

$$LCP(t[SA[j] : n], t[SA[j+1] : n]) \geq \max\{LCP(t[SA[i] : n], t[SA[i+1] : n]) - 1, 0\},$$

ponieważ $t[SA[j] : n] < t[SA[j+1] : n] \leq t[SA[i+1] + 1 : n]$ oraz

$$\begin{aligned} LCP(t[SA[j] : n], t[SA[i+1] + 1 : n]) &= LCP(t[SA[i] + 1 : n], t[SA[i+1] + 1 : n]) \\ &= \max\{LCP(t[SA[i] : n], t[SA[i+1] : n]) - 1, 0\}. \end{aligned}$$

Wykorzystujemy tu prosty fakt, że jeśli mamy dowolne słowa $w_1 \leq w_2 \leq w_3$, to $LCP(w_1, w_2) \geq LCP(w_1, w_3)$.

Jeśli nie istnieje szukane j, to $SA[i] = n$, a zatem rzecz jasna $t[SA[i]+1 : n] = \varepsilon$, a zatem jego najdłuższy wspólny prefiks z dowolnym innym podsłowem wynosi $k = 0$. □

Twierdzenie 4.10. *Liczba różnych podsłów dla słowa t ($|t| = n$) wynosi*

$$\binom{n}{2} - \sum_{i=1}^{n-1} LCP[i]$$

Dowód. Wystarczy policzyć podsłowa zaczynające się na pozycjach $SA[i+1]$ dla $0 \leq i \leq n-1$ tj. prefiksy $t[SA[i+1] : n]$ niebędące prefiksami słów $t[SA[j] : n]$ dla $1 \leq j \leq i$.

Ponieważ sufiksy są posortowane, to wiemy, że każdy prefiks $t[SA[i+1] : n]$ długości $0 \leq k < LCP[i]$ już wystąpił wcześniej – w szczególności jako prefiks $t[SA[i] : n]$.

Dalej, wiemy że $t[SA[i] : n] < t[SA[i+1] : SA[i+1] + LCP[i] + 1]$, ponieważ słowa są posortowane oraz $t[SA[i+1] + LCP[i] + 1]$ jest z definicji LCP pierwszym znakiem różnym w słowach $t[SA[i] : n]$ i $t[SA[i+1] : n]$. Oczywiście z przechodniości relacji $<$ wynika, że również

$$t[SA[j] : n] \leq t[SA[i] : n] < t[SA[i+1] : SA[i+1] + LCP[i] + k]$$

dla wszystkich $k \geq 1$ oraz $1 \leq j \leq i$, a zatem każdy prefiks $t[SA[i+1] : n]$ długości większej niż $LCP[i]$ na pewno nie jest prefiksem $t[SA[j] : n]$ dla $1 \leq j \leq i$.

$t[SA[i+1] : n]$ ma łącznie $n - SA[i+1]$ prefiksów, więc takich prefiksów jest oczywiście $n - SA[i+1] - LCP[i]$. Dla $i = 0$ oczywiście będzie to tylko $n - SA[1]$. Ostatecznie, całkowita liczba różnych słów wynosi

$$\sum_{i=0}^{n-1} (n - SA[i+1]) - \sum_{i=1}^{n-1} LCP[i-1] = \binom{n}{2} - \sum_{i=1}^{n-1} LCP[i],$$

ponieważ $\sum_{i=0}^{n-1} (n - SA[i+1])$ to zarazem liczba wszystkich (niekoniecznie różnych) podsłów w tekście. □

4.8 Algorytm SA-IS

4.8.1 Notacja

Jako S będziemy oznaczać słowo, którego tablicę sufiksową mamy obliczyć. Przyjmujemy, że S kończy się znakiem \$, który nie występuje nigdzie indziej w słowie i w porządku leksykograficznym jest najmniejszy.

Dodatkowo niech n będzie długością słowa S, a znaki będą numerowane od 1. Wtedy $S_i = S[i..n]$ dla $i \in \{1, ..., n\}$ jest sufiksem S rozpoczynającym się na i-tej pozycji.

4.8.2 Algorytm

Ogólny sposób działania SA-IS polega na sortowaniu sufiksów na podstawie tablicy sufiksowej krótszego słowa S', obliczanej rekurencyjnie. Słowo S' powstaje przez wybór pewnych podsłów S, i przepisaniu każdego z nich na pojedynczy znak zachowując kolejność leksykograficzną. Aby osiągnąć czas $O(n)$, obie te czynności wykonywane są za pomocą *sortowania indukowanego*.

Aby przedstawić dokładne działanie algorytmu, koniecznie są następujące pojęcia:

Definicja 4.11. S_i jest sufiksem typu S (typu L) gdy $S_i < S_{i+1}$ ($S_i > S_{i+1}$).

$S_n =' \$'$ jest typu S.

Typ znaku $S[i]$ określany jest jako ten sam co typ sufiksu S_i.

Definicja 4.12. Znak $S[i]$, $i \in \{2, ..., n\}$, jest LMS-znakiem (leftmost S-type) gdy $S[i-1]$ i $S[i]$ są odpowiednio typu L i typu S.

Definicja 4.13. LMS-podsłowo to 1. $S[n] =$ \$; lub 2. podsłowo $S[i..j]$, gdzie $S[i]$, $S[j]$ to kolejne LMS-znaki w S.

Podsłowa początkowo wybierane z S to właśnie LMS-podsłowa. Sortowanie indukowane zarówno dla LMS-podsłów (celem zredukowania podsłów do S'), jak i sufiksów S, działa opierając się na podziale sufiksów na typ S i typ L.

Całość algorytmu zawiera się w poniższych krokach:

1. Określ typ każdego z sufiksów S
2. Wyznacz LMS-podsłowa S
3. Posortuj znalezione LMS-podsłowa za pomocą sortowania indukowanego
4. Przepisz każde LMS-podsłowo na literę odpowiadającą jego pozycji w porządku i utwórz z nich słowo S' zachowując kolejność występowania w S
5. Oblicz tablicę sufiksową SA' słowa S'
6. Wyznacz tablicę sufiksową SA słowa S za pomocą sortowania indukowanego na podstawie SA'

W kroku 5, jeżeli nie występują dwa identyczne LMS-podsłowa (wszystkie znaki S' są unikalne), SA' można uzyskać bezpośrednio jako odwrotność permutacji reprezentowanej przez S'; w przeciwnym przypadku, jest ona pozyskiwana rekurencyjnie.

4.8.3 Wyznaczanie LMS-podsłów

Typ każdego z sufiksów można ustalić, przeglądając S od końca:

1. \$ jest typu S
2. jeśli $S[i] = S[i+1]$, typ S_i jest ten sam co S_{i+1}
3. wpp. $S[i] \neq S[i+1]$, a typ S_i zależy bezpośrednio od nich.

Tablica przechowująca typy sufiksów S pozwala wskazać pozycje, na których zaczynają się kolejne LMS-podsłowa. W dalszej części algorytmu LMS-podsłowa będą reprezentowane przez indeks ich pierwszego znaku.

4.8.4 Redukcja do mniejszego problemu

Na LMS-podsłowach jest zdefiniowany następujący porządek: porównując parami kolejne znaki obu podsłów, jeśli się różnią, decyduje ich kolejność leksykograficzna; w przepiwnym przypadku znak typu L jest mniejszy od znaku typu S. Ten porządek odpowiada temu, że jeśli S_i jest typu S, S_j typu L, i pierwsze litery są sobie równe, to $S_j < S_i$.

Sortowanie indukowane pracuje na wynikowej tablicy SA, podzielonej na kubełki dla każdego znaku w S. W trakcie sortowania wyłania się dodatkowy podział każdego kubełka na te przeznaczone dla sufiksów różnego typu w kolejności L, S; ale nie jest on reprezentowany bezpośrednio. Dla każdej litery w tablicy B rozmiaru alfabetu przechowywany jest wskaźnik na pewne miejsce w odpowiadającym kubełku. Wskazywane miejsce zależy od etapu sortowania.

Samo sortowanie odbywa się w trzech krokach:

1. wyznacz koniec każdego z kubełków. Wstaw każde z LMS-podsłów do jego kubełka (odp. pierwszemu znakowi podsłowa);
2. wyznacz początek każdego z kubełków. Dla każdego $i = 1, ..., n$: jeśli $c = S[SA[i] - 1]$ jest typu L, wstaw $SA[i] - 1$ do kubełka c;
3. wyznacz koniec każdego z kubełków. Dla każdego $i = n, ..., 1$: jeśli $c = S[SA[i] - 1]$ jest typu S, wstaw $SA[i] - 1$ do kubełka c.

Dokładniej, wyznaczenie początku (końca) kubełka c oznacza zapisanie odpowiedniego indeksu do $B[c]$, natomiast wstawianie elementu odbywa się przez zapisanie go do $SA[B[c]]$ i przesunięcie $B[c]$ o jeden w lewo (prawo).

S' jest utworzone przez ułożenie LMS-podsłów tak, jak występują w S i zamianę każdego z nich na kolejne liczby zgodnie z porządkiem uzyskanym z powyższego sortowania (identyczne podsłowa otrzymują tę samą liczbę).

4.8.5 Sortowanie wszystkich sufiksów

Procedura odtworzenia tablicy sufiksowej S przez sortowanie indukowane jest niemal identyczna z sortowaniem LMS-podsłów. Jedyna różnica występuje w pierwszym kroku: na końce kubełków należy wstawić LMS-podsłowa (indeksy pierwszych znaków) w zgodzie z kolejnością uzyskaną w SA'. Po jego wykonaniu zawartość SA jest tablicą sufiksową S.

4.8.6 Poprawność sortowania

Rozważmy ostatni krok całego algorytmu:

Lemat 4.14. *Mając dane posortowane wszystkie sufiksy typu L, krok 3 sortuje wszystkie sufiksy S w czasie $O(n)$.*

Dowód. Pokażemy przez indukcję, że w momencie przeglądania $SA[i]$, $S_{SA[i]}$ został już zapisany na swoim miejscu.

Dla $i = n$, największy sufiks musi być typu L, zatem był posortowany w kroku 2.

Dla $i < n$, gdy wszystkie $SA[i+1], ..., SA[n]$ są uzupełnione poprawnie, załóżmy że sufiks, który powinien znajdować się w $SA[i]$ leży pod $SA[k]$, $k < i$. Ponieważ sufiksy typu L zostały posortowane, w $SA[i]$, $SA[k]$ znajdują się sufiksy typu S. Sufiksy są już w swoich kubełkach, więc $SA[i] = c\alpha$, $SA[k] = c\beta$ dla jakiegoś c. $c\beta < \beta < \alpha$, zatem poprawne pozycje β, α znajdują się wśród już wstawionych i przeglądniętych $SA[i+1], ..., SA[n]$, a $c\alpha$ powinno było zostać wstawione na koniec swojego kubełka przed $c\beta$, co jest sprzeczne z obecnym stanem tablicy SA. □

Gdyby najpierw posortować sufiksy typu S, a następnie wykonać krok 2, wszystkie sufiksy typu L również zostały by posortowane, co można wykazać w analogiczny sposób. Ze względu na to, że tylko LMS-sufiksy biorą udział w ich sortowaniu, wystarczy zacząć od wstawienia do SA LMS-sufiksów w kolejności leksykograficznej, aby uzyskać ten sam rezultat. Tym samym, o ile tablica SA' wyznacza poprawny porządek LMS-sufiksów, ostatni etap algorytmu jest poprawny.

Niech $P[i]$ wskazuje na początek i-tego LMS-podsłowa. Wtedy

Lemat 4.15. *$S'_i < S'_j$ jest równoważne z $S_{P[i]} < S_{P[j]}$.*

Dowód. Jeśli $S'[i] \neq S'[j]$, to znak świadczący o różnicy w odpowiadających im LMS-podsłowach wyznacza kolejność sufiksów zaczynających się na tych samych pozycjach, natomiast porządek na LMS-podsłowach jest zgody z porzadkiem leksykograficznym na sufiksach. W przeciwynm wypadku, zauważmy, że $S'[i] = S'[j]$ oznacza równą długość odpowiednich LMS-podsłów, zatem możliwe jest przeprowadzenie powyższego rozumowania na pierwszym różniącym się znaku sufiksów w S'. □

Pozostaje wykazać, że pierwsze użycie sortowania indukowanego sortuje LMS-podsłowa. Mechanizm dowodu jest podobny, z kilkoma różnicami: w pierwszym kroku chcemy otrzymać poprawny porządek na LMS-podsłowach obciętych do pierwszego znaku. Wystarczy, że zostaną wstawione do odpowiednich kubełków, co zapewnia krok 1. Te z kolei wystarczą do posortowania sufiksów typu L obciętych do pierwszego LMS-znaku w kroku 2. Analogicznie dzieje się dla sufiksów typu S w kroku 3, z wyjątkiem LMS-sufiksów, które są rozważane do miejsca wystąpienia *drugiego* LMS-znaku, czyli LMS-podsłów.

4.8.7 Złożoność

Zarówno wyznaczanie LMS-podsłów, jak i sortowanie indukowane działa w oczywisty sposób w czasie $O(n)$. Ponieważ LMS-podsłów jest nie więcej niż połowa długości S (z wyjątkiem \$ w środku każdego z nich musi się znajdować przynajmniej jeden znak typu L), całkowity czas wykonania można określić równaniem $T(n) = T(\lfloor n/2 \rfloor) + O(n)$, które daje $O(n)$.

4.9 Algorytm obliczania tablicy sufiksowej autorstwa N. Larssona i K. Sadakane

Problem który będzie rozwiązywany definiujemy następująco. Niech $X = x_1 x_2 \dots x_n\$$ będzie ciągiem znaków złożonym z ciągu wejściowego długości n z dopisanym symbolem \$, który jest leksykograficznie przed wszystkimi symbolami ciągu wejściowego. Jako S_i, $1 \leq i \leq n+1$ będziemy oznaczać sufiks X zaczynający się w pozycji i. Wyjściem algorytmu ma być tablica indeksów I (*tablica sufiksowa*) taka że $S_{I[k-1]}$ jest leksykograficznie mniejsze od $S_{I[k]}$ dla $0 < k \leq n$.

Algorytm tworzenia tablicy sufiksowej zaprezentowany przez N. Larssona i K. Sadakane działa w czasie $O(n \log n)$ gdzie n jest długością wejściowego tekstu. Autorzy rozbudowują podejście **basicAlgo** i korzystają z techniki 'podwajania' znanej z algorytmu Karpa, Millera, Rosenberga która polega na użyciu pozycji sufiksów po fazie sortowania jako kluczy do sortowania dwukrotnie dłuższych sufiksów w kolejnej fazie. Autorzy formalizują to podejście definiując pojęcie *h-order*, czyli porządek leksykograficzny sufiksów rozważając jedynie h początkowych znaków każdego sufiksu. Zauważmy, że *h-order* niekoniecznie jest jednoznaczny jeśli $h < n$.

Lemat 4.16. *Sortowanie sufiksów przy użyciu klucza w postaci pary złożonej z pozycji sufiksu S_i w h-order oraz pozycji sufiksu S_{i+h} w h-order daje sortowanie sufiksów w 2h-order.*

Definicja 4.17. Dla tablicy sufiksów I która jest w *h-order* definiujemy

- Maksymalna (względem długości) spójna sekwencja sufiksów w I które mają te same początkowe h znaków to **grupa**
- Grupa składająca się z co najmniej 2 sufiksów jest **grupą nieposortowaną**
- Grupa składająca się z jednego sufiksu jest **grupą posortowaną**

- Maksymalna spójna sekwencja posortowanych grup jest **wspólną grupą posortowaną**

Grupy będziemy numerować żeby móc przeprowadzać obliczenia na poszczególnych grupach z osobna. Dla grupy $I[f \dots g]$ numer tej grupy to g. Ponadto tablica $V[i] = g$ zawiera informację, że sufiks S_{i+1} znajduje się w grupie o numerze g. Dodatkowo będziemy korzystać z tablicy L która trzyma długości nieposortowanych grup oraz długości wspólnych grup posortowanych. Dla rozróżnienia, długości tych późniejszych będą trzymane jako liczba ujemna. Tablicę L użyjemy w podstawowej wersji algorytmu. Udoskonalenia omówione pod koniec tego omówienia pozwolą nam obyć się bez osobnej tablicy na długości grup.

Lemat 4.18. *Gdy I jest w h-order, każdy sufiks w **wspólnej grupie posortowanej** jest unikalnie rozróżnialny od wszystkich innych sufiksów przy pomocy pierwszych h symboli.*

4.9.1 Algorytm

Algorytm po każdej iteracji konstruuje z tablicy sufiksów w porządku *h-order* tablicę w porządku *2h-order* korzystając z obserwacji 4.16. Zgodnie z obserwacją 4.18, sufiksy znajdujące się we wspólnych grupach posortowanych są już na swoich miejscach. Mając to na uwadze możemy wykorzystać to w algorytmie i przy procesowaniu nieposortowanych grup sufiksów 'przeskakiwać' nad wspólnymi grupami posortowanymi. Po każdej iteracji jest szansa, że pojawią się nowe grupy posortowane więc dodatkowo będziemy pilnować aby spójne sekwencje grup posortowanych były przez nas łączone we wspólne grupy posortowane.

Podstawowy algorytm będzie się prezentował następująco:

1. Umieść sufiksy $1 \dots n+1$ w tablicy I. Posortuj I używając x_i jako klucza dla sufiksu i (pierwsza litera). Ustaw $h = 1$
2. Dla każdego sufiksu S_{i+1} ustaw numer grupy do której należy w $V[i]$
3. Dla każdej nieposortowanej grupy lub wspólnej grupy posortowanej zapisz jej długość (lub zanegowaną długość) do tablicy L
4. Przesortuj każdą nieposortowaną grupę z osobna za pomocą *ternary quick sorta* (czyli takiego co dzieli na podtablice $[< pivot, = pivot, > pivot]$), używając $V[S_k + h - 1]$ jako klucza dla każdego sufiksu S_k w grupie
5. Zaznacz pozycje podziału pomiędzy nierównymi kluczami w dla każdej przetworzonej nieposortowanej grupy na potrzeby aktualizacji tablic V oraz L
6. Podwój h. Stwórz nowe grupy poprzez podział na zaznaczonych pozycjach (aktualizując odpowiednio V i L)
7. Zakończ jeśli I składa się tylko z jednej posortowanej grupy. W przeciwnym przypadku idź do kroku 4.

W podstawowej wersji algorytmu tablice V oraz L aktualizujemy po wykonaniu sortowania. Możemy to wykonać np. zaznaczając pozycje podziałów w procedurze quicksort np. używając bitów znaku w tablicy I. Na tej podstawie aktualizujemy V - ujemny wpis oznacza nową grupę - a następnie tablicę L.

Na pierwszy rzut oka można by wyciągnąć wniosek, że algorytm działa w czasie $O(n \log n^2)$. Dokładniejsza analiza pozwala na ograniczenie przez $O(n \log n)$. Na potrzeby analizy zakładamy, że *ternary quicksort* dzieli zawsze w medianie (możliwe do uzyskania przy wydajnym algorytmie znajdowania mediany, nieefektywne w praktyce). Proces sortowania całej tablicy sufiksowej (nie tylko jednego quicksorta) przedstawimy jako ternarne drzewo obliczeń.

Lemat 4.19. *Długość ścieżki od korzenia do liścia jest ograniczona z góry przez* $2 \log n + 3$

Dowód. Każdy węzeł 'środkowy' na ścieżce odpowiada podwojeniu *h-order* w którym znajduje się rozważana tablica. Takich węzłów może być więc $\log n + 1$ Z faktu, że używamy mediany po zejściu lewym lub prawym dzieckiem długość rozważanej tablicy zmniejsza się o połowę. Znów takich węzłów będzie maksymalnie $\log n + 1$ □

Na danym poziomie drzewa ilość wykonanej pracy będzie $O(n)$ bo to obliczenie odpowiada wykonaniu podziału na rozłączne podtablice. Stąd dostajemy złożoność $O(n \log n)$ bo aktualizacja podtablic odpowiedzialnych za grupy również jest wykonywana w czasie liniowym od rozmiaru podtablicy.

4.9.2 Usprawnienia

W algorytmie autorzy wprowadzają usprawnienia które nie wpływają negatywnie na czas wykonania. Opiszemy tutaj większość z nich (te które zostały wykorzystane w implementacji).

Eliminacja tablicy długości L. Tablica długości dla grup nieposortowanych jest nam zbędna ze względu na to jak numerujemy grupy. Przypomnijmy, numer grupy $I[f \dots g]$ to g, czyli $V[f-1] = g$ a w konsekwencji rozmiar grupy to $V[f-1] - g + 1$. Do spamiętania rozmiarów wspólnych grup posortowanych możemy użyć tablicy I. Wpólne grupy posortowane nie pojawią się w wywołaniu quicksorta a tylko tam korzystamy z I w trakcie działania algorytmu. Jedynym problemem pozostaje odzyskanie porządku w I na samym końcu algorytmu, ale tutaj wystarczy, że skorzystamy z tablicy V trzymającej numery grup (indywidualne grupy posortowane mają różne numery), więc $I[V[i]] = i$ dla $0 \leq i < n + 1$ odzyska tablicę sufiksową na końcu. Ostatecznie otrzymujemy następującą procedurę odzyskania długości grupy: jeśli $I[i] < 0$ to $I[i \dots i - I[i] - 1]$ jest wspólną grupą posortowaną. W przeciwnym wypadku $I[i \dots i + V[I[i] - 1]]$ jest nieposortowaną grupą.

Połączenie sortowania i aktualizowania tablic. Zauważmy, że wykonanie połączenia dwóch następujących wspólnych grup posortowanych możemy wykonywać dopiero przy kroku 4 algorytmu gdy iterujemy się po kolejnych nieposortowanych grupach w I przeskakując nad wspólnymi grupami posortowanymi. Aktualizacja tablic dla grup nieposortowanych w czasie wykonania sortowania musi być wykonana z uwagą na to żeby nie zmienić wartości w V które będą wykorzystane jako klucze sortowania. Fakt wykorzystania quicksorta zapewnia nam, że ustalenie kolejności aktualizacji tablic jest proste. Niektóre elementy w pierwszej partycji będą wykorzystywać jako klucz numery grup elementów z środkowej partycji (bo

bierzemy $V[I[currentElement]+h-1]$ jako klucz). Tym samym aktualizację numerów grup z partycji środkowej wykonujemy po rekurencyjnym posortowaniu pierwszej partycji.

1. Podziel tablice względem *pivot* na trzy partycje
2. Wywołaj się rekurencyjnie na sekcji '$<$ *pivot*'
3. Zaktualizuj numery grup w sekcji '$=$ *pivot*' która się staje w całości nową grupą
4. Wywołaj się rekurencyjnie na sekcji '$>$ *pivot*'

4.10 Suffix trays

Uzupełnić – ale z niskim priorytetem

5 Największy sufiks

Problem 4: Największy leksykograficznie sufiks

Wejście: Słowo $w \in \mathcal{A}^+$ ($|w| = m$)
Wyjście: Liczba $1 \leq i \leq m$ taka, że $w[i..m]$ jest największym leksykograficznie sufiksem w.

5.1 Algorytm na bazie tablicy prefikso-sufiksów

Lemat 5.1. *Jeśli dla dowolnego słowa w słowo u jest maksymalnym sufiksem $w[1..(j-1)]$ a słowo v jest takim sufiksem słowa $w[1..(j-1)]$, że $uw[j] < vw[j]$, to v jest prefikso-sufiksem u.*

Dowód. Skoro u jest maksymalnym sufiksem $w[1..(j-1)]$, to $u > v$. Jeśli v nie było prefiksem u, to istniałaby pozycja $1 \leq i \leq |v|$ taka, że $v[i] < u[i]$. Ale wówczas $uy \geq u > vy$ dla dowolnego $y \in \mathcal{A}^*$ – sprzeczność z tym, że $uw[j] < vw[j]$. □

Lemat 5.2. *Jeśli dla dowolnego słowa w słowo u jest maksymalnym sufiksem $w[1..(j-1)]$ oraz dla pewnego prefikso-sufiksu v słowa u zachodzi $uw[j] < vw[j]$, to dla wszystkich słów v' takich, że v' jest prefikso-sufiksem u i v jest prefikso-sufiksem v' zachodzi $uw[j] < v'w[j] < vw[j]$.*

Dowód. Dla dowolnego v' będącego prefikso-sufiksem u istnieje słowo x takie, że $u = v'x$. Ponieważ u jest maksymalnym sufiksem $w[1..(j-1)]$, to $u = v'x > vx$, ponieważ vx też jest sufiksem $w[1..(j-1)]$. Wówczas zachodzi $vx < u < uw[j] < vw[j]$, a zatem $x < w[j]$. Wobec tego $uw[j] = v'xw[j] < v'w[j]$.

Wiemy, że $vw[j] > uw[j]$. Skoro $|u| > |v|$, to $vw[j] > u[1..(|v|+1)]y$ dla dowolnego $y \in \mathcal{A}^*$. Łatwo sprawdzić, że każde v' takie, że v' jest prefikso-sufiksem u i v jest prefikso-sufiksem v jest postaci $u[1..(|v|+1)]y$. □

Wniosek 5.3. Jeśli dla dowolnego słowa w słowo u jest maksymalnym sufiksem $w[1..(j-1)]$ a słowo $vw[j]$ jest maksymalnym sufiksem słowa $w[1..j]$, to wystarczy schodzić po prefikso-sufiksach u dopóki to możliwe.

Algorytm 24: Wyszukiwanie największego leksykograficznie sufiksu

```
def from_prefix_suffix(text, n, less = operator.__lt__):
  def equal(a, b):
    return not less(a, b) and not less(b, a)
  B, t, out = [-1] + [0] * n, -1, 1
  for i in range(1, n + 1):
    # niezmiennik: out = maximum_suffix(text[0..i - 1]))
    # niezmiennik: B[0..i - out] = prefix_suffix(text[out..i - 1])
    # niezmiennik: t = B[i - out]
    while t >= 0 and less(text[out + t], text[i]):
      out, t = i - t, B[t]
    while t >= 0 and not equal(text[out + t], text[i]):
      t = B[t]
    t = t + 1
    B[i - out + 1] = t
  return out, (n + 1 - out) - B[n + 1 - out]
```

Problem 5.1. Pokaż, że $B[j]$ to taka liczba, że $w[1..B[j]]$ jest największym prefikso-sufiksem maksymalnego sufiksu $w[1..j]$.

5.2 Algorytm o stałej pamięci

Algorytm 25: Wyszukiwanie największego leksykograficznie sufiksu w pamięci $O(1)$

```
def constant_space(text, n, less = operator.__lt__):
  def equal(a, b):
    return not less(a, b) and not less(b, a)
  out, p, i = 1, 1, 2
  while i <= n:
    r = (i - out) % p
    if equal(text[i], text[out + r]):
      i = i + 1
    elif less(text[i], text[out + r]):
      i, p = i + 1, i + 1 - out
```

```
    else:
      out, i, p = i - r, i - r + 1, 1
  return out, p
```

Problem 5.2. Pokaż, że algorytm 25 wyznacza poprawnie największy leksykograficznie sufiks.

6 Odległość edycyjna

Problem 5: Odległość edycyjna	
Wejście:	Słowa $t_1, t_2 \in \mathcal{A}^+$ ($\lvert t_i \rvert = n_i$ dla $i = 1, 2$)
Wyjście:	Minimalna liczba operacji typu *insert*, *delete* i *substitute* przeprowadzająca słowo t_1 w t_2.

Zwróćmy uwagę, że analogiczny problem, w którym dozwolona jest tylko operacja *substitute* to obliczanie odległości Hamminga. Można ją wprost wyznaczyć przeglądając t_1 i t_2 od lewej do prawej, zliczając pozycje, dla których $t_1[i] \neq t_2[i]$.

Podstawowy algorytm obliczania odległości edycyjnej sprowadza się do sekwencyjnego obliczania wartości funkcji rekurencyjnej

$$d_e(i,j) = \begin{cases} 0 & \text{gdy } i = 0 \text{ lub } j = 0, \\ d_e(i-1, j-1) & \text{gdy } t_1[i] = t_2[j], \\ \min\{d_e(i-1,j-1), d_e(i-1,j), d_e(i-1,j-1)\} + 1 & \text{gdy } t_1[i] \neq t_2[j]. \end{cases}$$

Poprawność algorytmu wynika z indukcji ze względu na porządek par (i, j). Czas działania algorytmu to $O(n_1 n_2)$. Jak widać z powyższego wzoru, do obliczeń wystarczy zapamiętanie poprzedniego wiersza lub kolumny, więc wymagana pamięć wynosi $O(\min\{n_1, n_2\})$.

Algorytm 26: Obliczanie odległości edycyjnej w czasie $O(n_1 n_2)$ i pamięci $O(\min\{n_1, n_2\})$

```
def edit_distance(text_1, text_2, n_1, n_2):
  if n_1 < n_2:
    return edit_distance(text_2, text_1, n_2, n_1)
  previous_row, current_row = None, range(n_2 + 1)
  for i, ci in enumerate(text_1[1:]):
    previous_row, current_row = current_row, [i + 1] + [None] * n_2
    for j, cj in enumerate(text_2[1:]):
      insertion = previous_row[j + 1] + 1
      deletion = current_row[j] + 1
      substitution = previous_row[j] + (ci != cj)
      current_row[j + 1] = min(insertion, deletion, substitution)
  return current_row[-1]
```

Technika może być łatwo uogólniona do dowolnych macierzy ocen, przypisujących różne wagi za odległości dla operacji *insert*, *delete* i *substitute*. Przykładem takiego wykorzystania są rodziny macierzy PAM i BLOSUM dla algorytmów dopasowań ciągów w zastosowaniach bioinformatycznych.

Można również łatwo zamienić problem na obliczanie funkcji podobieństwa jako maksymalnej (zamiast minimalnej) wartości dla danej macierzy operacji, tym razem interpretowanej jako macierz nagród (Sellers, 1974).

6.1 Technika czterech Rosjan

Dla obliczeń macierzowych istnieje ogólna technika przyspieszania obliczeń, nazywana *Four Russians speedup*[2].

Główna idea tej metody polega na podziale wejścia na małe bloki o niewielkim zakresie wartości. Załóżmy, że pewien problem jest w ogólności rozwiązywany przez funkcję $y_n = f(y_{n-1}, x_n)$, a jedna iteracja wymaga czasu $F(n)$ – więc całość wykonuje się w czasie $\sum_{i=1}^{n-1} F(i)$.

Załóżmy teraz, że każda pozycja wejściowa pochodzi z małego zbioru X, a każda pozycja wyjściowa również z małego zbioru Y. Wówczas dla dowolnego k zawsze możemy obliczyć z wyprzedzeniem tablicę wszystkich $|X|^k|Y|$ wartości funkcji $f^{(k)}$, czyli k-krotnego złożenia f. Wówczas można policzyć f_n wykorzystując $\frac{n}{k}$ wyszukiwania w tablicy. Całkowity czas działania (zakładając stały dostęp do tablicy wartości) wynosi zatem

$$|X|^k|Y| \sum_{i=1}^{k-1} F(i) + \frac{n}{k}.$$

Przykładowo dla $F(n) = n^2$, $|X| = O(1)$ i $|Y| = O(n)$ wystarczy dobrać $k = \log_{|X|} n$, aby zmniejszyć czas wykonania z $O(n^3)$ do $O(n^2 \log^3 n)$.

Metoda ta używana jest m.in. do mnożenia i odwracania macierzy binarnych, a także do efektywnego liczenia domknięcia przechodniego grafu. Można ją również wykorzystać do optymalizacji algorytmu obliczającego odległość edycyjną (Masek i Paterson, 1980). Technika ta również uogólnia się na różnorodne macierze nagród i kar.

Algorytm i dowód

[2]Najbardziej znanym z autorów był Jefim Dinic, znany również z algorytmu dla znajdowania maksymalnego przepływu w grafach.

7 Masek, Paterson „A Faster Algorithm Computing String Edit Distances" - omówienie algorytmu

Odległość edycyjna pomiędzy dwoma słowami o długościach n i m jest definiowana jako minimalny koszt sekwencji operacji edytujących (dodania znaku, usunięcia znaku, zamiany jednego znaku w inny), które prowadzą do transformacji jednego słowa w drugie. Szczególnym przypadkiem problemu znajdywania odległości edycyjnej jest szukanie najdłuższego wspólnego podciągu dwóch słów. Rozwiązanie tego podproblemu w czasie szybszym niż $O(n^{2-\epsilon})$ (gdy słowa mają tę samą długość n), prowadziłoby do obalenia SETH. Autorzy pracy pokazują algorytm na szukanie odległości edycyjnej dwóch słów działający w czasie $O(n \cdot max(1, m/logn))$ pod warunkiem, że koszty operacji edycyjnych są całkowitymi wielokrotnościami tej samej liczby rzeczywistej, a alfabet jest skończony.

7.1 Przyjęte oznaczenia

A_n - n-ty znak słowa A
$A^{i,j}$ - słowo $A_i...A_j$
D_a - koszt usunięcia znaku a
I_a - koszt wstawienia znaku a
$R_{a,b}$ - koszt zamiany znaku a na znak b
δ - odległość edycyjna, minimum po sumie kosztów wszystkich możliwych sekwencji edycyjnych, $\delta_{i,j}$ - odległość edycyjna między słowami $A^{1,i}$ oraz $B^{1,j}$

7.2 Algorytm Wagnera i Fischera

Algorytm opisany w pracy opiera się na pomyśle z innego algorytmu, autorstwa Wagnera i Fischera. Algorytm ten wyznacza odległość edycyjną dynamicznie, w czasie proporcjonalnym do iloczynu długości słów. Tworzy w tym celu pomocniczą macierz o rozmiarze $(|A| + 1) \times (|B| + 1)$, w której komórka (i,j) odpowiada kosztowi $\delta_{i,j}$. Poniższe twierdzenia prezentują sposób obliczania bazy i kolejnych komórek:

Twierdzenie 1

$\delta_{0,0} = 0 \ \forall i, j : 1 \leq i \leq |A|, 1 \leq j \leq |B|$
$\delta_{i,0} = \sum_{1 \leq r \leq i} D_{A_r}$ oraz $\delta_{0,j} = \sum_{1 \leq r \leq j} I_{B_r}$

Twierdzenie 2

$\delta_{i,j} = min(\delta_{i-1,j-1} + R_{A_i,B_j}, \delta_{i-1,j} + D_{A_i}, \delta_{i,j-1} + I_{B_j})$

7.3 Przyspieszenie metodą czterech Rosjan

Technika ćzterech Rosjan"pochodzi z algorytmu obliczania przechodniego domknięcia grafu skierowanego poprzez podział macierzy sąsiedztwa na podmacierze o niewielkiej ilości wierszy. Okazuje się, że podobny pomysł można wykorzystać do asymptotycznego przyspieszenia powyższego algorytmu, właśnie dzięki podziałowi obliczenia na dużo mniejszych obliczeń.

Zauważmy, że podmacierz (i, j, m) macierzy edycyjnej z algorytmu Wagnera-Fischera - zaczynająca się w komórce $(i + 1, j + 1)$ i mająca rozmiar $m \times m$ - jest zdeterminowana tylko i wyłącznie poprzez koszt $\delta(i, j)$ oraz wektory inicjalne $\delta(i, j + 1)...\delta(i, j + m)$ oraz $\delta(i + 1, j)...\delta(i + m, j)$. Obliczając wartość w komórce (i, j) odwołujemy się bowiem tylko do trzech sąsiednich komórek $(i - 1, j - 1), (i - 1, j), (i, j - 1)$. Możemy więc wcześniej dla każdej możliwej podmacierzy (i, j, m) obliczyć i zapamiętać jej wektory finalne: $\delta(i + m, j + 1)...\delta(i + m, j + m)$ oraz $\delta(i + 1, j + m)...\delta(i + m, j + m)$.

Aby tego dokonać, musimy być w stanie wyenumerować wszystkie możliwe macierze rozmiaru m. Zakładamy, że alfabet jest skończony, więc słów długości m jest skończona ilość. Jednak generowanie wszystkich możliwych wektorów inicjalnych może zająć zbyt dużo czasu - wartości komórek mają tendencję do wzrostu wraz z rozszerzaniem się macierzy głównej. Kiedy jednak ograniczymy funkcję kosztu do przypisywania operacjom edycyjnym wyłącznie wielokrotności jakiejś liczby rzeczywistej, okaże się, że mamy tylko skończoną ilość różnic pomiędzy sąsiednimi komórkami w macierzy edycyjnej.

Będziemy więc operować na wektorach różnic (kroków) pomiędzy komórkami sąsiadującymi ze sobą w poziomie lub pionie. Prowadzi to do następującej modyfikacji twierdzenia 2:

Twierdzenie 2b

$\delta_{i,j} - \delta_{i-1,j} =$
$min(R_{A_i,B_j} - (\delta_{i-1,j} - \delta_{i-1,j-1}), D_{A_i}, I_{B_j} + (\delta_{i,j-1} - \delta_{i-1,j-1}) - (\delta_{i-1,j} - \delta_{i-1,j-1}))$

$\delta_{i,j} - \delta_{i,j-1} =$
$min(R_{A_i,B_j} - (\delta_{i,j-1} - \delta_{i-1,j-1}), D_{A_i} + (\delta_{i-1,j} - \delta_{i-1,j-1}) - (\delta_{i,j-1} - \delta_{i-1,j-1}), I_{B_j})$

Proponowany w pracy **Algorytm Y** wykonuje pierwszy krok - generuje wszystkie możliwe pary C, D słów długości m oraz wszystkie możliwe wektory inicjalne różnic: R - pomiędzy sąsiednimi wierszami (wertykalny) i S - pomiędzy sąsiednimi kolumnami (horyzontalny). Następnie oblicza odpowiadające danej podmacierzy wektory finalne różnic - R' oraz S', wyliczając kolejne wektory T (wertykalne) i U (horyzontalne), zgodnie ze wzorem z twierdzenia 2b. Obliczenie i zaindeksowanie każej możliwej podmacierzy zajmuje $O(m^2 logm)$ czasu.

Następnie **Algorytm Z** łączy ze sobą podwyniki wygenerowane przez Algorytm Y i oblicza rzeczywistą odległość edycyjną. W tym celu dynamicznie oblicza kolejne finalne wektory dla podmacierzy (i, j, m) dla $i \in \{0, m, ...|A| * (m-1)\}$ oraz $j \in \{0, m, ...|B| * (m-1)\}$, wykorzystując fakt, że wektor finalny danej podmacierzy jest wektorem inicjalnym dla sąsiadującej (odpowiednio pionowym lub poziomym). Rzeczywista odległość edycyjna może być następnie uzyskana jako suma wartości komórek na dowolnej ścieżce od komórki początkowej do końcowej (dodając kolejne różnice, docierając do ostatniej komórki otrzymamy jej wartość). Czas działania tej części to $O(|A| \cdot |B| \cdot (m + log|A|)/m^2)$. Jeśli wybierzemy $m = \lfloor log_k|A| \rfloor$, całość (Algorytm Y i Z) działa w czasie $O(|A| \cdot |B|/m)$.

8 Najdłuższy wspólny podciąg

Powyższego algorytmu można również użyć do obliczenia najdłuższego wspólnego podciągu (LCS). W tym celu jako funkcje kosztu wybieramy: $D_a = I_a = 1$ oraz $R_{a,b} = 0$ jeśli a=b oraz $R_{a,b} = 2$ w przeciwnym wypadku. Najtańszą ścieżką edycyjną jest w takim przypadku usunięcie z jednego słowa znaków z nie-LCS i dodanie do drugiego słowa znaków z LCS. Do odtworzenia LCS można wykorzystać pomysł analogiczny do odtwarzania ze standardowego algorytmu (Wagnera-Fischera). Zaczynając od ostatniej komórki w macierzy, patrzymy skąd otrzymaliśmy daną wartość i cofamy się. Za każdym razem kiedy operacją, którą wykonywaliśmy jest zamiana, a znaki się zgadzają - mamy do czynienia z literą z LCS. W przypadku algorytmu prezentowanego w pracy nie mamy obliczonej całej macierzy, ale możemy odtworzyć z wektorów inicjalnych podmacierze różnic przecinane przez ścieżkę. Wybór znaków należących do LCS będzie taki sam jak w przypadku rzeczywistej macierzy kosztów.

9 Najdłuższy wspólny podciąg

Problem 6: Najdłuższy wspólny podciąg

Wejście: Zbiór słów $T = \{t_1, t_2, ..., t_k\} \subseteq \mathcal{A}^+$ takich, że $|t_i| \leq n$ dla $1 \leq i \leq k$.
Wyjście: Słowo t będące najdłuższym podciągiem wszystkich słów z T.

Problem najdłuższego wspólnego podciągu jest powiązany z problemem odległości edycyjnej. Dla $|T| = 2$ niech $d(t_1, t_2)$ oznacza odległość dla dopuszczalnych operacji *insert* i *delete*. Wówczas zachodzi

$$|t_1| + |t_2| = 2|t| + d(t_1, t_2),$$

wystarczy bowiem zauważyć, że t_1 w t przeprowadzają wszystkie operacje *delete* a t w t_2 – *insert*.

9.1 Algorytm Needlemana-Wunscha

Podstawowy algorytm obliczania najdłuższego wspólnego podciągu dla dwóch słów jest prostym rozszerzeniem algorytmu 26. Jedyna różnica polega na konieczności trzymania całej tablicy odległości. Odtwarzanie wyniku następuje na podstawie przejścia po ścieżce od ostatniej wartości tj. $d(t_1, t_2)$ do początku, odpowiadającemu $d(\epsilon, \epsilon)$. Wybieramy rzecz jasna tylko te przejścia, które spełniają relację min w równaniu rekurencyjnym.

Oczywiście można odtwarzać kolejne przejścia i uzgodnione litery na podstawie t_1, t_2 porównując odpowiednie wartości macierzy d – lub też można zapisywać odpowiednie informacje równolegle w trakcie wypełniania tablicy d.

Algorytm 27: Obliczanie najdłuższego wspólnego podsłowa w czasie i pamięci $O(n_1 n_2)$

```
def needleman_wunsch(text_1, text_2, n_1, n_2):
  Data = namedtuple('Data', ['distance', 'previous', 'letter'])
  d = { (0, 0): Data(0, None, '') }
  for i, ci in enumerate(text_1[1:]):
    d[(i + 1, 0)] = Data(i + 1, (i, 0), ci)
  for i, ci in enumerate(text_2[1:]):
    d[(0, i + 1)] = Data(i + 1, (0, i), ci)
  for i, ci in enumerate(text_1[1:]):
    for j, cj in enumerate(text_2[1:]):
      insertion = Data(d[(i, j + 1)].distance + 1, (i, j + 1), '')
      deletion = Data(d[(i + 1, j)].distance + 1, (i + 1, j), '')
      if ci == cj:
        agreement = Data(d[(i, j)].distance, (i, j), ci)
        if agreement.distance <= min(insertion.distance, deletion.distance):
          d[(i + 1, j + 1)] = agreement
          continue
      if insertion.distance <= deletion.distance:
        d[(i + 1, j + 1)] = insertion
      else:
        d[(i + 1, j + 1)] = deletion
  text, p = '', (n_1, n_2)
  while p != (0, 0):
    p_next = d[p].previous
    if p[0] - p_next[0] == 1 and p[1] - p_next[1] == 1:
      text = d[p].letter + text
    p = p_next
  return text, d[(n_1, n_2)].distance
```

9.2 Algorytm Hirschberga

Ponieważ dla dostatecznie dużych n_1 i n_2 w praktyce pamięć jest bardziej dotkliwą barierą niż czas, można zastanowić się nad usprawnieniem algorytmu, aby wykorzystywał tylko $O(\min\{n_1, n_2\})$ pamięci.

Okazuje się, że wystarczy zastosować podejście dziel-i-rządź. Z jednej strony jasne jest, że odpowiadająca najdłuższemu podciągowi optymalna ścieżka od punktu $(0, 0)$ do (n_1, n_2) w macierzy d musi przejść przez pewien punkt (i, j) dla dowolnego $0 \leq i \leq n_1$. Możemy zatem wybrać i w połowie aktualnie rozpatrywanego przedziału.

Znalezienie j nie jest trudne. Okazuje się, że wystarczy do tego obserwacja z poniższego lematu:

Problem 9.1. Punkt (i, j) leży na (pewnej) optymalnej ścieżce w macierzy d wtedy i tylko wtedy, gdy

$$j = \arg\max\{0 \leq k \leq n_2 : d(t_1[1..i], t_2[1..k]) + d(t_1[\bar{i}..n_2], t_2[\bar{k}..n_2])\}.$$

Algorytm 28: Obliczanie najdłuższego wspólnego podsłowa w czasie $O(n_1 n_2)$ i pamięci $O(\min\{n_1, n_2\})$

```
def hirschberg(text_1, text_2, n_1, n_2):
  if n_1 < n_2:
    return hirschberg(text_2, text_1, n_2, n_1)
  if n_2 == 0:
    return '', n_1
  if n_2 == 1:
    return needleman_wunsch(text_1, text_2, n_1, n_2)
  split_1 = n_1 // 2
  distance_previous = distance.indel_distance_row(
      text_1[:split_1 + 1], text_2, split_1, n_2)
  distance_next = distance.indel_distance_row(
      text_1[0] + text_1[n_1:split_1:-1], text_2[0] + text_2[n_2:0:-1],
      n_1 - split_1, n_2)[::-1]
  distance_sum = [d_1 + d_2 for d_1, d_2 in zip(
      distance_previous, distance_next)]
  split_2 = distance_sum.index(min(distance_sum))
  out_previous = hirschberg(
      text_1[:split_1 + 1], text_2[:split_2 + 1], split_1, split_2)
  out_next = hirschberg(
      text_1[0] + text_1[split_1 + 1::], text_2[0] + text_2[split_2 + 1:],
      n_1 - split_1, n_2 - split_2)
  return out_previous[0] + out_next[0], out_previous[1] + out_next[1]
```

9.3 Inne wyniki

W ramach wyników negatywnych dla problemu odległości edycyjnej wiadomo, że nie istnieje żaden algorytm dla obliczania odległości edycyjnej o złożoności $\Theta(n^{2-\varepsilon})$ dla dowolnego $\varepsilon > 0$, o ile Strong Exponential Time Hypothesis jest prawdziwa (Backurs i Indyk, 2015). Twierdzenie to zachodzi nawet dla przypadku alfabetu binarnego (Bringmann i Künnemann, 2015).

9.4 LCS w czasie oczekiwanym O(ND)

Algorytm omówiony w pracy służy do poszukiwania najdłuższego wspólnego podciągu. Powstał w oparciu o pracę **An O(ND) Diffrence Algorithm and Its Variations** autorstwa **Eugune W.Myers**.

9.4.1 Definicje

Zanim przejdziemy do algorytmu zdefiniujmy graf edycji, definicje wspólnego podciągu oraz pewne pomocnicze określenia.

9.4.2 Graf edycji

Mając dane słowa $A = a_1a_2 \dots a_N$ oraz $B = b_1b_2 \dots b_M$ jako graf edycji G oznaczamy skierowany graf, dla którego zbiór wierzchołków to punkty siatki (x, y), $x \in [0, N]$ *i* $y \in [0, M]$, natomiast na krawędzi składają się:

- $(x-1, y) \to (x, y)$ dla każdego $x \in [1, N]$ oraz $y \in [0, M]$.
- $(x, y-1) \to (x, y)$ dla każdego $x \in [0, N]$ oraz $y \in [1, M]$.
- $(x-1, y-1) \to (x, y)$ jeśli $a_x = b_y$, taka krawędź definiuje pewne dopasowanie pomiędzy wzorcami, punkty dla których $a_x = b_y$ nazwiemy punktami dopasowania.

Definicja 9.1. Podciąg słowa A to każde słowo powstałe poprzez usunięcie 0 lub więcej symboli z A. Natomiast wspólny podciąg dwóch słów A i B to podciąg każdego z nich.

9.4.3 Graf edycji, a wspólny podciąg

Definicja 9.2. Śladem długości L nazywamy sekwencje L punktów dopasowania $(x_1, y_1), (x_2, y_2) \dots (x_L, y_L)$ gdzie dla każdego $1 \leq i < L : \ x_i < x_{i+1} \wedge y_i < y_{i+1}$. Jest to pewna sekwencji diagonalnych krawędzi. Ślad możemy powiązać z pewną ścieżka pomiędzy $(0, 0)$ a (N, M).

Każdy ślad grafu edycji definiuje pewien wspólny podciąg słów A i B, który powstaje poprzez odpowiednie symbole zdefiniowane przez punkty dopasowania.

Definicja 9.3. Odległość edycyjna D to ilość operacji edycji które przeprowadzamy na tekście A żeby przeprowadzić go w tekst B. Jeśli spojrzymy na ścieżkę która zawiera pewien ślad L to każda krawędź pozioma w tej ścieżce definiuje pewną operację usunięcia symbolu z tekstu A, natomiast krawędzi pionowe definiują operację dodania symboli do tekstu A za odpowiednim indeksem w ciągu A. Możemy zauważyć że wartość odległości edycyjnej jest powiązana z długością śladu L, tj $D = N + M - 2L$.

Idea algorytmu będzie opierała się na wyszukiwaniu najdłuższego śladu pomiędzy punktami (0,0) oraz (N,M), czyli znalezieniu ścieżki która zawiera największą ilość krawędzi diagonalnych, powyższy ślad wyznaczy nam najdłuższy wspólny podciąg słów A i B.

Pozostały nam jeszcze dwie definicję, które będą wykorzystywane w algorytmie:

Definicja 9.4. Jako D-ścieżkę definiujemy ścieżkę w grafie edycji rozpoczynającą się w punkcie $(0, 0)$, posiadającą dokładnie D krawędzi niediagonalnych.

Definicja 9.5. W celu wprowadzenia algorytmu zdefiniujemy numerację diagonali. K-diagonala, to taka krawędź diagonalna, która składa się z punktów (x, y), takich że $x-y = k$. Zgodnie z tym krawędzi są numerowane od $-M$ do N.

Na podstawię definicji k-diagonali zauważmy, że krawędź pionowa (pozioma) której punkt startowy leży na k-diagonali, posiada drugi koniec krawędzi na (k-1)-diagonali (odpowiednio (k+1)-diagonali dla poziomej). Zbiór diagonali znajdujących się na końcu krawędzi pionowej/poziomej nazywamy ogonem (w przypadku gdy koniec krawędzi nie jest początkiem żadnej diagonali, wtedy ogon nazywamy pustym).

9.4.4 Algorytm: Poszukiwanie najdłuższej ścieżki

Algorytm będzie opierał się na wyszukiwaniu najdalszych D-ścieżek w k-diagonali. Przed wprowadzeniem pseudokodu, najpierw omówimy pewne właściwości związane z diagonalami oraz ścieżkami

Lemat 9.6. *Każda D-ścieżka musi kończyć się na k-diagonali, takiej że* $k \in \{-D, -D + 2, \ldots, D - 2, D\}$

Dowód. Dowód indukcyjny. Dla D = 0 zauważmy, że 0-ścieżka może kończyć się tylko na diagonali 0, ponieważ oba indeksy (x, y) na tej ścieżce muszą być sobie równe (nie możemy wykorzystać żadnej krawędzi poziomej lub pionowej z definicji 0-ścieżki). Następnie zauważmy, że D-ścieżka składa się z pewnej (D-1)-ścieżki, pewnej poziomej/pionowej krawędzi oraz ogona. Z założenia indukcji mamy że ścieżka D-1 kończy się na pewnej k krawędzi gdzie $k \in \{-D + 1, -D + 3, \ldots D - 3, D - 1\}$, natomiast nowy ogon będzie składał się z pewnej krawędzi $k \pm 1$(bazując na obserwacji z k-diagonalami), stąd zbiór diagonali na których kończy się D-ścieżka to $k_D \in \{-D + 1 \pm 1, -D + 3 \pm 1, \ldots D - 3 \pm 1, D - 1 \pm 1\}$, czyli mamy $k_D \in \{-D, -D + 2, \ldots, D - 2, D\}$. □

Definicja 9.7. D-ścieżkę nazywamy najdalej sięgającą w diagonali k, jeśli jest to ścieżka której końcowy punkt ma największy numer kolumny (odpowiednio numer wiersza z definicji k-diagonali) spośród wszystkich D-ścieżek w k diagonali.

Lemat 9.8. *Najdalej sięgająca D-ścieżka w diagonali k może być rozbita na najdalej sięgającą (D-1)-ścieżkę kończącą się na diagonali k - 1, krawędź poziomą oraz najdłuższy ogon z tej krawędzi, lub w najdalej sięgającą (D-1)-ścieżkę w diagonali k + 1, krawędź pionową oraz najdłuższy ogon z tej krawędzi.*

Dowód. Wybranie odpowiedniej najdalszej (D-1)-ścieżki w diagonali k+1/k-1 związane jest ze wcześniej wspomnianą obserwacją na temat ogonu krawędzi poziomej/pionowej. Natomiast w celu udowodnienie faktu że to najdalej sięgająca (D-1)-ścieżka załóżmy że (D-1)-ścieżka składająca się na kompozycję najdalej sięgającej D-ścieżki nie jest najdalej sięgająca w diagonali k, ale możemy zaobserwować że skoro D-ścieżka musi sięgać dalej niż (D-1)-ścieżka, to oznacza że do ogona tej D-ścieżki będziemy mogli się też dostać z końca ogona najdalszej (D-1)-ścieżki przy pomocy odpowiedniej krawędzi niediagonalnej, więc zawsze możemy najdalszą D-ścieżkę rozłożyć na najdalszą (D-1)-ścieżkę. □

Na bazie obu lematów wprowadzamy algorytm obliczania punktu końca najdalej sięgającej D-ścieżki w diagonali k. Zauważmy że odległość edycyjna $D \in [0, M+N]$, więc będziemy kolejno badali możliwe odległości edycyjne, aż dojdziemy do punktu (N, M).

```
\caption{Obliczanie odległości edycyjnej}
\begin{algorithmic}
\STATE $V:[0] + [0] * 2(m+n)$
\STATE $V[1] \leftarrow 0$
\FOR{$D \leftarrow 0$ \TO $M+N$}
    \FOR{$k \leftarrow -D$ \TO $D$ in steps of 2}
        \IF{$k= -D$ or $k \neq D$ and $V[k-1] < V[k+1]$}
            \STATE $x \leftarrow V[k+1]$
        \ELSE
            \STATE $x \leftarrow V[k-1] + 1$
        \ENDIF
        \STATE $y \leftarrow x-k$
        \WHILE{$x<N$ and $y<M$ and $a_{x+1}=b_{y+1}$}
            \STATE $V[k] \leftarrow x$
        \ENDWHILE
        \IF{$x \geq N$ and $y \geq M$}
            \RETURN D
        \ENDIF
    \ENDFOR
\ENDFOR
\end{algorithmic}
```

Powyższy algorytm pozwala nam w czasie O((M + N)D) wyznaczyć odległość edycyjną (a co za tym idzie również długość najdłuższego wspólnego podciągu) przy wykorzystaniu O(D) pamięci.

Niestety problem pojawia się przy wyznaczeniu symboli na które składa się najdłuższy wspólny podciąg. Musielibyśmy po każdej iteracji zapisywać tablicę V, w celu wyznaczenia z których punktów przechodziliśmy w tej ścieżce, co sprawiałoby że musielibyśmy wykorzystać $O(D^2)$ pamięci.

W tym celu w następnej części omówimy algorytm bazujący na przeszukiwaniu najdalszych ścieżek, który będzie działał w O(D) pamięci.

9.4.5 Algorytm w liniowej pamięci

Finalny algorytm na poszukiwanie najdłuższego wspólnego podciągu będzie opierał się na strategii dziel i rządź. Algorytm będzie korzystał z poszukiwania najdalej sięgającej ścieżki z punktu $(0, 0)$ oraz odwrotnej ścieżki z punktu (N, M). W celu obliczenia odwrotnej ścieżki będziemy korzystać z poprzedniego algorytmu, z tym że tym razem będziemy szukali ścieżki z (N, M) w kierunku $(0, 0)$ odwracając krawędzie z grafu edycji oraz minimalizując wartość Y w przeciwieństwie do maksymalizowania X z poprzedniego algorytmu (natomiast ścieżka w przód liczona jest analogicznie do poprzedniego algorytmu). Przed zaprezentowaniem algorytmu udowodnimy następujący lemat:

Lemat 9.9. *Podział D-ścieżek Pomiędzy punktami* $(0, 0)$ *i* (N, M) *istnieje D-ścieżka wtedy i tylko wtedy gdy istnieje* $\lceil D/2 \rceil$*-ścieżka z* $(0, 0)$ *do punktu* (x, y) *oraz* $\lfloor D/2 \rfloor$*-ścieżka z punktu* (u, v) *do* (M, N)*, taka że*

- $u + v \geq \lceil D/2 \rceil$ *oraz* $x + y \leq N + M - \lfloor D/2 \rfloor$
- $x - y = u - v$ *oraz* $x \geq u$.

Generalnie powyższy lemat opiera się na podziale D-ścieżki na pewne 2 ścieżki, jedna ścieżka z $(0, 0)$ a druga ścieżka, którą możemy traktować jako odwrotną ścieżkę z (N, M) w kierunku $(0, 0)$. Zauważmy że jak rozłożymy D-ścieżkę na 2 takie przeciwne ścieżki, to ogon znajdujący się na końcu ścieżki do przodu będzie pokrywał się z ogonem znajdującym się na końcu ścieżki przeciwnej, będziemy właśnie z tej właściwości korzystać szukając ścieżkę do przodu i ścieżkę od końca, aż nasze ścieżki pokryją się w pewnej diagonali. Właśnie te diagonale z ogona które się pokrywają wyznaczą nam symbole które będą "środkiemńaszego najdłuższego wspólnego podciągu.

Następnie wykorzystamy strategię dziel i rządź rozdzielimy sobie słowa na podstawie środkowego ogona i wykonamy się rekurencyjnie na naszych podsłowach, największy wspólny podciąg to rekurencyjne wywołanie lewej części słów, znaków z ogona oraz wywołania na prawej części słów.

Algorytm będzie opierał się na odpowiedniej adaptacji poprzedniego algorytmu do przeszukiwania wprzód D-ścieżek oraz wstecznych D-ścieżek, w przypadku gdy dla jakiejś diago-

nali k ogony ścieżek się pokryją zwracamy odpowiednią odległość edycyjną oraz ogon który znaleźliśmy. W przypadku gdy dla jakiegoś D, D-ścieżka wprzód pokrywa się z (D-1)ścieżką przeciwną, to zwracamy $2D-1$, natomiast gdy D-ścieżka przeciwna pokrywa się z D-ścieżką wprzód zwracamy $2D$.

Nasz finalny algorytm na znalezienie największego wspólnego podciągu prezentuję się następująco:

```
\caption{LCS(A,B,N,M)}
\begin{algorithmic}
\IF{$N > 0$ and $M > 0$}
    \STATE{ Uruchamiamy naszą procedurę szukania środkowego ogona}
    \STATE{$(X,Y) \xrightarrow{} (U,V)$ to nasz środkowy ogon, D - odległość edycyjna }
    \IF {$D > 1$}
        \STATE $left = LCS(A[1\dots x],B[1\dots y],x,y)$
        \STATE $middle = A[x+1\dots u]$
        \STATE $right = LCS(A[u+1\dots N],B[v+1\dots M],N-u,M-v)$
        \RETURN $left + middle + right$
    \ELSIF {$M > N$}
        \RETURN $A[1\dots N]$
    \ELSE
        \RETURN $B[1\dots M]$
    \ENDIF
\ENDIF
\end{algorithmic}
```

9.4.6 Złożoność

Oznaczmy jako $T(P, D)$ czas działania algorytmu gdzie $P = N + M$. Nasz czas działania opisuje następująco zależność

$$T(P,D) \leq \begin{cases} \alpha PD + T(P_1, \lceil D/2 \rceil) + T(P_2, \lfloor D/2 \rfloor) & \text{if } D > 1 \\ \beta P & \text{if } D \leq 1 \end{cases}$$

gdzie $P_1 + P_2 \leq P$. Korzystając z zależności $\lceil D/2 \rceil \leq \frac{2D}{3}$ dla $D \geq 2$, możemy udowodnić że $T(P, D) \leq 3\alpha PD + \beta P$, co dowodzi że algorytm wykonuje się w $O((M + N)D)$ czasie.

Natomiast nasza złożoność pamięciowa, zauważmy, że procedura poszukiwania środkowego ogonu zajmuje $O(D)$ miejsca, ze względu na wykorzystanie dwóch wektorów V, dodatkowo procedura poszukiwania środkowego ogona działa niezależnie od rekurencji, więc na całą rekurencje potrzebujemy miejsca tylko na 2 wektory, natomiast rekurencja składa

się z $O(logD)$ poziomów więc zajmuje $O(logD)$ pamięci, stąd mamy złożoność $O(m+n)$ pamięciową.

10 Kiran Kumar-Pandu Rangan: A Linear Space Algorithm for the LCS Problem

Algorytm zaprezentowany w pracy opiera się na zastosowaniu techniki *dziel i zwyciężaj*. Pozwala ona znaleźć szukany najdłuższy wspólny podciąg w złożoności $O(n(m-p))$ oraz przy wykorzystaniu liniowego rozmiaru pamięci. W poniższym opisie A oraz B oznaczają słowa o długości odpowiednio m oraz n, które są argumentami algorytmu, natomiast C to szukany najdłuższy wspólny podciąg (LCS) o długości p. Zapis $T(i:j)$ oznacza podsłowo słowa T z początkiem o indeksie i oraz końcem o indeksie j. W opisie pracy zakładam że słowa indeksowane są od cyfry 1, implementacja algorytmu ze względu praktycznych wykorzystuje indeksowanie słowa od 0 - przy wywołaniach rekurencyjnych na podsłowach trzeba byłoby przekazywać za każdym razem słowa o jeden znak dłuższe.
Pierwszą rzeczą, którą wykonuje algorytm jest obliczenie długości najdłuższego wspólnego podciągu. Znając jego długość możemy obliczyć *perfekcyjne cięcie*, które definiowany jest w następujący sposób:

Definicja 10.1. Parę (u, v) nazywamy *perfekcyjnym cięciem* dla słów A, B jeżeli:

1. (u, v) jest prawidłowym cięciem dla słów A, B; to znaczy jeśli $LCS(A, B)$ możemy przedstawić jako $C = C_1C_2$ wtedy C_1 to $LCS(A(1:u), B(1:v))$ oraz C_2 to $LCS(A(u+1:m), B(v+1:n))$,
2. $W(A(1:u), B(1:v)) = (m-p)/2$, gdzie $W(A, B) = |A| - |C|$

Perfekcyjne cięcie pozwala podzielić zadany problem zgodnie z definicją na dwa podproblemy: *LCS(A(1:u), B(1:v))* oraz *LCS(A(u+1:m), B(v+1:n))*, a z otrzymanych wyników utworzyć rozwiązanie. Głównym problemem algorytmu jest zatem znalezienie perfekcyjnego cięcia. Do rozwiązania tego problemu wykorzystywane są dwie procedury: *calmid* oraz *fillone*. By lepiej zrozumieć ich ideę wprowadźmy najpierw następujące definicje:

Definicja 10.2. $L_i(k)$ oznacza największe h dla którego słowa *A(i:m), B(h:n)* mają najdłuższy wspólny podciąg długości k.

Definicja 10.3. $L_i^*(k)$ oznacza najmniejsze h dla którego słowa *A(i:m), B(h:n)* mają najdłuższy wspólny podciąg długości k.

W pracy przedstawiona została zależność pomiędzy *perfekcyjnym cięciem (u, v)* a wartościami $L_i(k)$ oraz $L_i^*(k)$. Pozwala ona znaleźć perfekcyjne cięcie słów A, B jeśli zna się wartości jeśli zna się wartości $L_i(k)$ oraz $L_i^*(k)$ dla odpowiednich i, k:

Twierdzenie 10.4. *Para (u, v) jest perfekcyjnym cięciem wtedy i tylko wtedy gdy* $L_{u+1}(m-u-w') \geq v \geq L_u^*(u-w)$, *gdzie* $w = \lfloor \frac{m-p}{2} \rfloor$ *oraz* $w' = \lceil \frac{m-p}{2} \rceil$.

Procedura *calmid(A, B, m, n, x, LL, r)* oblicza wartości $L_{m-x+1-j}(j)$ za pomocą procedury *fillone(A, B, m, n, R1, R2, r, s)* i zapisuje je w tablicy LL. Zmienna r po wykonaniu funkcji oznacza natomiast największe j dla którego dana wartość jest poprawnie zdefiniowana. Procedura *fillone(A, B, m, n, R1, R2, r, s)* używa dwóch tablic by obliczyć wartości $L_{s+1}(0), L_s(1), L_{s-1}(2), ...$ Obliczenia te wykonywane są iteracyjne, z wykorzystaniem poprzednich wartości. Poprawność tych funkcji oraz dokładne ich działanie opiszę przy dowodzie ich poprawności.

11 Opis dowodu poprawności oraz złożoności algorytmu

Prezentowany w pracy dowód poprawności algorytmu podzielony został na dowód poprawności dla każdej z procedur zaprezentowanych przez autorów. Na początek przedstawmy dowody poprawności dla procedur *fillone* oraz *calmid* których działanie jest kluczowe dla większości pozostałych funkcji algorytmu. Następnie przyjrzymy się metodzie znajdowania *perfekcyjnego cięcia*. Pozostałe procedury działają w sposób dość jasny do zrozumienia.

11.1 Procedura *fillone*

Tak jak zostało już wspomniane procedura *fillone(A, B, m, n,* R_1, R_2*, r, s)* używa dwóch tablic by obliczyć wartości $L_{s+1}(0), L_s(1), L_{s-1}(2), ...$ Sposób w jaki uzyskiwane są kolejne wartości wynika z następującego lematu:

Lemat 11.1. *Niech h będzie największą liczbą taką że* $A[i] = B[h]$ *oraz* $h < l_{i+1}(j-1)$*, wtedy liczba* $L_i(j)$ *jest definiowana przez h oraz* $L_{i+1}(j)$*:*

$$y = \begin{cases} h & \text{gdy } L_{i+1} \text{ jest zdefiniowane} \\ L_{i+1}(j) & \text{gdy } h \text{ nie istnieje} \\ max(h, L_{i+1}(j)) & \text{gdy obie wartości są zdefiniowane} \end{cases}$$

W procedurze wykorzystywana jest zmienna lokalna $lower_b$, która odpowiada za to, by nie przekroczyć wartości dla których h oraz $L_{i+1}(j)$ z powyższego lematu dla danego i, j są niezdefiniowane.
Dodatkowo wprowadzona zostaje zmienna *over*, która odpowiada za to by przerwać działanie funkcji gdy wszystkie szukane wartości zostały już obliczone. Aby uzasadnić liniową złożoność pamięciową funkcji, wystarczy zauważyć że jedyne czego używa procedura *fillone* to liniowej wielkości tablice R_1 oraz R_2, zatem złożoność pamięciowa to $O(n+m)$. Złożoność czasowa natomiast wynika z tego że maksymalna liczba porównań którą możemy wykonać w pętli *while* to n, ponieważ nigdy nie zwiększamy zmiennej pos_b, zatem złożoność to $O(n)$.

11.2 Procedura *calmid*

Procedura składa się z dwóch części. Pierwsza to pętla for, która w i-tej iteracji odpowiada za wyliczanie wartości $L_{m-i+2-j}(j)$ i zapisanie jej do $R_1[j]$. Przepisywanie wyników z tablicy R_2 do tablicy R_1 na koniec każdej iteracji zapewnia, że będą znajdowały się tam poprawne wartości dla wszystkich $0 \leq j \leq r$. Poprawność tych wartości wynika z poprawności działania procedury *fillone*
Druga część to przepisanie ostatecznych wartości $L_{m-x-j+1}(j)$ które zapisane są w tablicy $R_1(j)$ do zwracanej tablicy $LL(j)$.
Podobnie jak przy procedurze *fillone* złożoność pamięciowa oraz czasowa są dosyć proste do uzasadnienia. Wykorzystanie pamięci to ponownie jedynie linowego rozmiaru tablice R_1, r_2, LL, zatem złożoność pamięciowa to $O(n+m)$.
Złożoność czasowa to natomiast $(x+1)$ wywołanie procedury *fillone* w pętli for. Przypominając że złożoność procedury *fillone* wynosi $O(n)$, zatem złożoność czasowa procedury *calmid* to $O((x+1)n)$.

11.3 Znajdowanie perfekcyjnego cięcia

Procedura znajdowania *perfekcyjnego cięcia* polega na dwukrotnym wywołaniu procedury calmid odpowiednio dla odwróconych słów A, B oraz dla nieodwróconych. Po zakończeniu tych wywołań otrzymujemy w tablicach LL_1, LL_2 wyliczone wartości odpowiednio: $n+1-L_{m-w+1-k}(k)$ dla $k \leq r_1$ oraz $L_{w+k+1}(m-(w+k)-w')$ $(= L_{m-w'+1-(p-k)}(p-k))$ dla $p-k \leq r_2$. Z twierdzenia 1.1 wynika zatem, że wystarczy zatem dobrać $u = k+w$ oraz $v = LL_1(k)$ by dostać poprawne perfekcyjne cięcie.
Złożoność czasowa i pamięciowa to ta sama, która miała procedura calmid, gdyż dodatkowo wykonywana jest jedynie pętla *for* by dobrać odpowiednie (u, v), która ma złożoność $O(r_1) \leq O(n)$.

11.4 Złożoność czasowa i pamięciowa algorytmu

Patrząc na główną procedurę *Longest common subsequence*, która działa zgodnie z techniką *dziel i zwyciężaj* opisaną na początku dokumentu możemy obliczyć złożoność czasową algorytmu. Mamy do rozpatrzenia dwa przypadki:

- Przypadku bazowego: jego rozwiązanie to wywołanie funkcji *calmid* z parametrem $x = m - p$, gdzie $m - p \leq 2$, zatem jest to złożoność $O(n)$ oraz dodatkowe operacje w pętlach *while*, których maksymalnie zostanie wykonanych $O(m)$. Zatem całkowita złożoność przypadku bazowego to $O(n+m)$.
- Znajdowanie *perfekcyjnego cięcia* oraz dwa wywołania rekurencyjne: pierwsze to dwa wywołania procedury *calmid* z parametrem $x = w$ lub $x = w'$ odpowiednio, zatem ich złożoność to $O(n(w+1))$ i $O(n(w'+1))$. Złożoność ta to inaczej $O(n(m-p))$, ponieważ tak dobrany był parametry w i w'. Dodając dwa wywołania rekurencyjne

zgodne z *perfekcyjnym cięciem* (u, v) dostajemy złożoność:

$$T(m, n, p) = O(n(m - p)) + T(u, v, u - w) + T(m - u, n - v, m - u - w')$$

która po przeliczeniach okazuje się wynosić $O(n(m - p))$.

Jedyną operacją wykonywaną poza procedurą *Longest common subsequence* jest znalezienie na początku algorytmu długości najdłuższego wspólnego podciągu, która również wykonywana jest w czasie $O(n(m - p))$. Podsumowując całkowita złożoność czasowa algorytmu to:

$$O(n(m - p)) + O(n + m) + O(n(m - p)) = O(n(m - p))$$

Złożoność pamięciowa algorytmu jest natomiast liniowa względem rozmiaru słów A i B, czyli wynosi $O(n + m)$. By uzyskać taką złożoność przy wywołaniach rekurencyjnych zamiast podsłów przekazujemy jedynie indeksy początku i końca podsłowa. Tablice LL_1 oraz LL_2 wykorzystywane do obliczania i przechowywanych wartości mają zawsze wielkość liniową. Maksymalna głębokość rekursji jaka może wystąpić w algorytmie to $O(log(m - p))$, zatem całkowita złożoność pamięciowa algorytmu pozostaje liniowa.

12 Dopasowanie wzorca z wieloznacznikami

Wprowadźmy teraz symbol specjalny ? jako wieloznacznik lokalny (*don't care symbol*), pasujący do każdego pojedynczego symbolu z $\mathcal{A}$.

Definicja 12.1. Relacja $\simeq$ jest relacją ***dopasowania z wieloznacznikiem lokalnym*** tj. $u \simeq v$ wtedy i tylko wtedy, gdy $|u| = |v|$ oraz dla wszystkich $1 \leq i \leq |u|$ zachodzi $u[i] = v[i]$ lub $u[i] = ?$, lub też $v[i] = ?$.

Problem 7: Wyszukiwanie wszystkich wystąpień wzorca z wieloznacznikami lokalnymi w tekście

---

Wejście: Słowa $t, w \in (\mathcal{A} \cup \{?\})^+$ ($|t| = n$, $|w| = m$)
Wyjście: Zbiór liczb $S = \{1 \leq i \leq n : t[i..(i + m - 1)] \simeq w\}$.

Istnieje bardzo pomysłowy algorytm obliczania wszystkich wystąpień wzorca z wieloznacznikami lokalnymi w tekście, oparty o szybką transformatę Fouriera. Jak wiadomo, FFT umożliwia obliczanie dla dwóch wektorów X i Y o długości odpowiednio n i m (zakładając, że $n \geq m$) w czasie $O(n \log n)$ operacji splotu tj. takiego Z, że dla wszystkich $0 \leq i \leq n - m$ zachodzi

$$Z[i] = \sum_{j=0}^{m-1} X[i + j]Y[m - 1 - j].$$

Zwróćmy uwagę, że – wyjątkowo – mamy w tym przypadku indeksowanie od 0.

Algorytm 29: Obliczanie wszystkie wystąpień wzorca z wieloznacznikami lokalnymi w tekście

```
def basic_fft(text, word, n, m):
  if n < m:
    return
  A = set(list(text[1:] + word[1:]))
  A.discard('?')
  mismatches = [0] * (n - m + 1)
  for a in A:
    text_binary = [int(c == a) for c in text[1:]]
    for b in A:
      if a == b:
        continue
      word_binary = [int(c == b) for c in reversed(word[1:])]
      mismatches_ab = scipy.signal.convolve(
        text_binary, word_binary, mode = 'valid', method = 'fft')
      mismatches = [x + y for x, y in zip(mismatches, mismatches_ab)]
  yield from (i + 1 for i, v in enumerate(mismatches) if v == 0)
```

Twierdzenie 12.2. *Algorytm 29 wyznacza poprawnie wszystkie wystąpień wzorca z wieloznacznikami lokalnymi w tekście.*

Dowód. Kluczowa obserwacja jest następująca: wybierzmy dwie różne litery $a, b \in \mathcal{A}$ i wyznaczmy odpowiednio dla wszystkich $0 \leq i \leq n-1$ oraz $0 \leq j \leq m-1$

$$X[i] = 1 \text{ gdy } t[i+1] = a, \text{w przeciwnym przypadku } X[i] = 0,$$
$$Y[j] = 1 \text{ gdy } w[m-j] = b, \text{w przeciwnym przypadku } Y[j] = 0.$$

Wówczas

$$Z[i] = \sum_{j=0}^{m-1} X[i+j]Y[m-1-j] = |\{0 \leq j \leq m-1 : t[i+j+1] = a \wedge w[j+1] = b\}|,$$

a zatem $Z[i]$ jest po prostu zliczeniem wszystkich niedopasowań między tekstem $t[i+1..i+m]$ a wzorcem w, takich że w tekście występuje a, a we wzorcu b. Powtarzając to dla wszystkich par liter w alfabecie możemy zliczyć wszystkie niedopasowania między $t[i+1..i+m]$ a w – a zatem dopasowania będą na na tych pozycjach, na których niedopasowań brak. □

Jak łatwo zauważyć, algorytm 29 działa w czasie $O(n \log n |\mathcal{A}|^2)$.

12.1 Wyszukiwanie z wildcardami w tekście i wzorcu

Na wejściu do algorytmu dostajemy tekst i wzorzec w których oprócz liter mogą występować wildcardy (?) pasujące do dowolnego znaku. Na wyjściu ma się znaleźć lista pozycji w których wzorzec pasuje do tekstu.

12.1.1 Wersja bez wildcardów

W rozwiązaniu problemu użyty jest splot obliczany za pomocą szybkiej transformaty Fouriera (FFT) zdefiniowany następująco:

$$p \otimes t \stackrel{\text{def}}{=} (\sum_{j=0}^{m-1} p_j t_{i+j}, 0 \leq i \leq n-m)$$

Normalne (bez wildcardów) dopasowanie wzorca p do tekstu t na pozycjach od i do i+m możemy obliczyć za pomocą splotu. Korzystamy z własności wzoru:

$$\sum_{j=0}^{m-1} (p_j - t_{i+j})^2 = \sum_{j=0}^{m-1} (p_j^2 - 2p_j t_{i+j} + t_{i+j}^2)$$

Zobaczmy że lewa część równania równa 0 to znaleźliśmy dopasowanie. Tekst nie różni się od wzorca na żadnej z m kolejnych pozycji. Prawą stronę potrafimy szybko obliczyć ponieważ podniesienie każdej pozycji do kwadratu wykonywane jest w czasie stałym a środkowy składnik liczmy za pomocą FFT.

12.1.2 Wersja z wildcardami

Zdefiniujmy ciągi:

$$p'_j = \begin{cases} 0 & p_j = '?' \\ 1 & wpp \end{cases}$$

$$t'_j = \begin{cases} 0 & t_j = '?' \\ 1 & wpp \end{cases}$$

Wtedy łatwo jest widać że następujące równanie jest naturalnym rozszerzeniem rozwiązania poprzedniego problemu.

$$\sum_{j=0}^{m-1} p'_j t'_{i+j} (p_j - t_{i+j})^2 = 0$$

Zobaczmy że tekst część pod sumą jest zawsze nieujemna p'_j oraz t'_{i+j} przyjmują wartości {0,1} oraz kwadrat różnicy jest zawsze nieujemny. Więc zastanówmy się kiedy suma jest równa zero. Wtedy kiedy wszystkie składniki są równe zero. Kiedy składnik jest równy zero? W jednym z 3 przypadków. $p'_j = 0$ wtedy mamy wildcard we wzorcu, $t'_{i+j} = 0$ wtedy

mamy wildcard w tekście, $(p_j - t_{i+j})^2$ wtedy litery wzorca i tekstu są równe. Zobaczmy że to zachodzi wtedy i tylko wtedy gdy mamy do czynienia z dopasowaniem.

Po rozpisaniu nawiasu dostajemy formę:

$$\sum_{j=0}^{m-1}(p'_j p_j^2 t'_{i+j} - 2p'_j p_j t_{i+j} t'_{i+j} + p'_j t_{i+j}^2 t'_{i+j})$$

Takie rozwinięcie łatwo jest obliczyć wykorzystując 3 razy FFT oraz podstawowe operacje na ciągach.

12.1.3 Złożoność

Trywialnie jest pokazać że algorytm działa $O(n \log n)$ gdy użyjemy FFT dla tekstu i patternu rozszerzonego do długości tekstu. Można jednak osiągnąć $O(n \log m)$ gdy podzielimy tekst na n/m zachodzących na siebie kawałków wielkości $2m$ i dla każdego policzymy matching ze wzorcem a następnie wyniki złączymy. Zobaczmy że jest to poprawne ponieważ dla każdego miejsca początkowego istnieje kawałek, który zawiera część od długości co najmniej m od tego miejsca. Złożoność: $n/m * O(m \log m) = O(n \log m)$.

13 Przybliżone dopasowanie wzorca

13.1 Landau-Vishkin – Efficient string matching with k mismatches

W podanym tekście $T[1, \ldots, n]$ chcemy znaleźć wszystkie wystąpienia wzorca $A[1, \ldots, m]$, które różnią się z nim na nie więcej niż k pozycjach, w złożoności $O(k(m\ log\ m\ +\ n))$

13.1.1 Główna idea

Niech PAT-MISMATCH będzie dwuwymiarową tablicą o rozmiarze $(m-1)(2k+1)$, gdzie i-ty wiersz tablicy zawiera $2k+1$ pierwszych pozycji, idąc od lewej strony, na których podsłowa $A[i+1 : m]$ oraz $A[1 : m-i]$ się różnią (PAT-MISMATCH$[i][j] = f$ oznacza, że $A[i+f] \neq A[f]$ oraz, że jest to niedopasowanie numer j). Domyślną wartością w tej tablicy jest $m+1$.

Niech TEXT-MISMATCH będzie dwuwymiarową tablicą o rozmiarze $(n-m+1)(k+1)$, w której i-ty wiersz zawiera informacje na temat $k+1$ pierwszych pozycji, idąc od lewej strony, na których podsłowo $T[i : i+m]$ różni się od wzorca A (TEXT-MISMATCH$[i][j] = f$ oznacza, że $T[i+f] \neq A[f]$ oraz, że jest to niedopasowanie numer j). Domyślną wartością w tej tablicy jest $m+1$.

Dzięki tablicy TEXT-MISMATCH jesteśmy w stanie sprawdzić, czy podsłowo $T[i+1 :$

$i+m]$ jest dopasowaniem wzorca A z k niedopasowaniami. Odpowiedź jest twierdząca, wtedy i tylko wtedy gdy TEXT-MISMATCH$[i][k+1] = m+1$.

Pokażemy teraz jak wykorzystać tablicę PAT-MISMATCH do wyliczania wartości w tablicy TEXT-MISMATCH. Niech j będzie najdalszą pozycją niedopasowania litery wzorca, które znaleźliśmy w tekście. Niech r indeksem początku podsłowa tekstu, dla którego pozycja j jest niedopasowaniem, tzn. $T[j] \neq A[j-r]$. Obliczając wartości dla wiersza i-tego w tablicy TEXT-MISMATCH, takiego że $r \leq i < j$ możemy skorzystać z następującej obserwacji:

Lemat 13.1. *Pozycje na których T[i+1:j] będzie się różniło z A[1:j-i] są pozycjami rozróżniającymi podsłowa A[i-r+1:j-r] i A[1:j-i] (przypadek 1) lub T[i+1:j] i A[i-r+1:j-r] (przypadek 2).*

Dowód. Niech x będzie pozycją w tekście oraz $i+1 \leq x < j$. Rozważmy następujące przypadki:

1. A[x-r] = A[x-i] oraz T[x] = A[x-r], wtedy T[x] = A[x-i],
2. A[x-r] = A[x-i] oraz $T[x] \neq A[x-r]$, wtedy $T[x] \neq A[x-i]$,
3. $A[x-r] \neq A[x-i]$ oraz T[x] = A[x-r], wtedy $T[x] \neq A[x-i]$,
4. $A[x-r] \neq A[x-i]$ oraz $T[x] \neq A[x-r]$, wtedy może zajść $T[x] \neq A[x-i]$ i T[x] = A[x-i].

Tak, więc jedynymi pozycjami na których T[i+1:j] będzie się różniło z A[1:j-i], są te na których różnią się podsłowa A[i-r+1:j-r] i A[1:j-i] lub T[i+1:j] i A[i-r+1:j-r]. □

Zauważmy, że poszukiwane pozycje są obliczone w tablicach PAT-MISMATCH[i-r] oraz TEXT-MISMATCH[r].

Niech operacja MERGE poszukuje pozycji rozróżniających słowa T[i+1:j] oraz A[1:j-i] przeglądając kolejne komórki w tablicach PAT-MISMATCH[i-r] oraz TEXT-MISMATCH[r], reprezentujące pozycje nie wykraczające poza j.

Lemat 13.2. *Jeżeli jest $\geq k+1$ niedopasowań w pozycjach $\leq j$, wtedy MERGE znajduje pierwsze k+1 z nich. Jeżeli niedopasowań jest <k+1 w pozycjach $\leq j$, wtedy MERGE znajdzie je wszystkie.*

Dowód. Zauważmy, że pozycji spełniających przypadek 2 jest nie więcej niż k. Niech y będzie liczbą pozycji spełniających przypadek 1. Wykażemy, że nie potrzebujemy więcej niż 2k+1 pozycji spełniających przypadek 1, aby procedura MERGE działa poprawnie. Jeżeli:

1. PAT-MISMATCH[i-r][2k+1] >= j-i, to korzystając z obserwacji 13.1 znajdziemy wszystkie lub k+1 pierwszych niedopasowań.
2. PAT-MISMATCH[i-r][2k+1] < j-i, tzn. mamy 2k+1 pozycji >i oraz <j, które rozróżniają podsłowa wzorca. Przypomnijmy, że pozycji spełniających warunek 1 jest nie więcej niż k. Wtedy co najmniej k+1 pozycji spełnia przypadek 1, a nie spełnia

przypadeku 2. Korzystając z obserwacji 13.1 procedura MERGE znajdzie k+1 pozycji rozróżniających słowa T[i+1:j] oraz A[1:j-i].

□

13.1.2 Algorytm analiza tekstu

Niech procedura EXTEND(i, j, b) iteracyjnie przegląda słowa T[j:] oraz A[j-i:] dopóki nie znajdzie k+1 niedopasowań. Liczba b to ilość niedopasowań znaleziona przez procedurę MERGE.

Pokażemy teraz algorytm wyznaczający wartości tablicy TEXT-MISMATCH, bazując na tablicy PAT-MISMATCH (której wyliczenie zaprezentujemy w następnej sekcji).

```
TEXT-MISMATCH[0,...,n-m][1,...,k+1] = m+1
r, j = 0, 0
for i in range(0, n-m):
    b = 0
    if i < j:
        MERGE(i, r, j, b)
    if b < k+1:
        r = i
        EXTEND(i, j, b)
```

13.1.3 Analiza wzorca

W tej sekcji zaprezentujemy obliczanie tablicy PAT-MISMATCH. Załóżmy, bez strat ogólności, że m jest potęgą dwójki. Podzielmy zbiór wierszy tablicy PAT-MISMATCH na log m następujących zbiorów: [1], [2,3], [4,5,6,7], ... [$\frac{1}{2}$m,...,m].

W kolejnych krokach analizy wzorca, będziemy wyliczali tablicę PAT-MISMATCH dla kolejnych zbiorów wierszy. Analiza wzorca dla l-tego zbioru wierszy, tzn. dla wierszy [2^{l-1}, ..., $2^l - 1$] przebiega w taki sam sposób jak analiza tekstu, przy czym rolę tablicy PAT-MISMATCH spełnia tablica PAT-MISMATCH[1:$2^{l-1} - 1$], rolę tablicy TEXT-MISMATCH spełnia PAT-MISMATCH[2^{l-1}, ..., $2^l - 1$], a k zastępujemy przez $\min(2^{log\ m\ -\ l}2k+1, m-2^l)$ (dowód, że takie k jest odpowiednie, jest analogiczny jak 13.2)

13.1.4 Złożoność

Każda faza analizy wzorca wykonywana jest w złożoności O(km). Ponieważ mamy logm faz analizy wzorca dostajemy sumaryczną złożoność O($km\ log\ m$). Analizy tekstu wykonana jest w złożoności O(kn). Złożoność całego algorytmu to $O(k(m\ log\ m\ +\ n))$.

13.2 Algorytm aproksymacyjnego dopasowania wzorca względem odległości edycyjnej – podejście na podstawie algorytmu Boyer'a-Moore'a.

Problem 8: k - przybliżone wyszukiwanie wzorca w tekście względem odległości edycyjnej

Wejście: Słowa $t, p \in \mathcal{A}^+$ ($|t| = n$, $|p| = m$)
Wyjście: Zbiór liczb $S = \{j :$ istnieje podsłowo T zakończone na t_j o odległości edycyjnej od p równej co najwyżej $k\}$.

13.2.1 Algorytm programowania dynamicznego.

Podstawowym algorytmem rozwiązującym zadany wyżej problem jest następujący algorytm programowania dynamicznego wyliczający tablicę $D(i, j)$ oznaczającą minimalną odległość edycyjną między słowami $p_1...p_i$ oraz dowolnym podsłowem T zakończonym na t_j:

$$D(0, j) = 0,\ 0 \leq j \leq n$$
$$D(i, j) = min \begin{cases} D(i-1, j) + 1 \\ D(i-1, j-1) + \text{ if } p_i = t_j \text{ then 0 else 1} \\ D(i, j-1) + 1 \end{cases}$$

Rozwiązaniem są wszystkie j takie, że $D(m, j) \leq k$. Algorytm ten działa w czasie $O(mn)$.

13.2.2 Idea algorytmu ABM (Approximate Boyer-Moore)

Algorytm oparty na pomyśle analogicznym do algorytmu Boyer'a-Moore'a będzie będzie działać w dwóch głównych fazach: *skanowania* i *sprawdzania*. W fazie skanowania przechodzimy przez dane słowo t i zaznaczamy pewne komórki pierwszego rzędu tablicy D (tj komórki postaci $D(0, j)$). Po zakończeniu fazy skanowania rozpoczynamy fazę sprawdzania, czyli dla każdej zaznaczonej komórki D będziemy obliczali przekątną tablicy D poprzez zwykłe programowanie dynamiczne. Kiedykolwiek nasza procedura dynamiczna będzie odnosić się do niewyliczonej komórki możemy przyjąć, że jej wartość wynosi ∞.
Faza sprawdzania wydaje się dość intuicyjna dlatego skupmy się na fazie skanowania.

Faza skanowania powtarza w pętli dwie czynności. *Zaznacza* (mark) i *przesuwa* (shift) indeks który skanujemy. Operacja przesuwania jest analogiczna do operacji przesuwania w algorytmie $Boyer'a - Moore'a$. Operacja zaznaczania wykonywana jest wtedy gdy obecny indeks wymaga dokładniejszego sprawdzenia przez wyliczenie tablicy D.

13.2.3 Faza skanowania algorytmu ABM

Definicja 13.3. Mówimy, że dla każdego $D(i,j)$ istnieje ***łuk minimalizujący*** z $D(i-1,j)$ do $D(i,j)$ jeśli $D(i,j) = D(i-1,j)+1$, z $D(i,j-1)$ do $D(i,j)$ jeśli $D(i,j) = D(i,j-1)+1$, oraz z $D(i-1,j-1)$ do $D(i,j)$ jeśli $D(i,j) = D(i-1,j-1)$ gdy $p_i = t_j$ albo $D(i,j) = D(i-1,j-1)+1$ gdy $p_i \neq t_j$.

Definicja 13.4. ***Minimalizującą ścieżką*** nazywamy dowolną ścieżkę złożoną z minimalizujących łuków zaczynającą w $D(0,j)$ w pierwszym rzędzie tablicy D do komórki $D(m,h)$ znajdującej się w ostatnim rzędzie. Minimalizującą ścieżkę nazywamy ***dobrą*** gdy prowadzi do $D(m,h) \leq k$.

Lemat 13.5. *Komórki dowolnej dobrej ścieżki minimalizującej zawarte są wśród* $\leq k+1$ *kolejnych przekątnych tablicy* D.

Definicja 13.6. Dla $i = 1,...m$ przez k***-otoczenie*** znaku p_i ze wzorca p oznaczmy słowo $C = p_{i-k}...p_{i+k}$, gdzie $p_j = \epsilon$ dla $j < 1$ oraz $j > m$.

Lemat 13.7. *Jeśli dobra ścieżka minimalizująca przechodzi przez pewne komórki przekątnej* h *tablicy* D*, wtedy dla co najwyżej* k *indeksów* i*,* $1 \leq i \leq m$*, znak* t_{h+i} *nie występuje w* k*-otoczeniu* C_i.

Definicja 13.8. Kolumnę j, $h+1 \leq j \leq h+m$ tablicy D nazywamy ***złą*** jeśli t_j nie należy do k-otoczenia C_{j-h}.

Obserwacja 13.9. Z lematu 1.5 wynika, że dla danego t_j złych kolumn jest co najwyżej k o ile

Na podstawie powyższej obserwacji i lematu możemy skonstruować następującą procedurę zaznaczania. Dla przekątnej h, dla $i = m, m-1, ..., k+1$ lub dopóki nie znaleźliśmy $k+1$ złych kolumn sprawdź czy t_{h+1} należy do C_i. Jeśli znaleźliśmy $\leq k$ złych kolumn wtedy zaznaczmy komórki $D(0,h-k), ..., D(0,h+k)$.
Aby szybko odpowiadać na to czy kolumna jest zła możemy obliczyć tablicę $BAD(i,a)$ dla $1 \leq i \leq m$, $a \in \mathcal{A}$ taką, że

$$Bad(i,a) = \quad \textbf{true}, \text{ wtw gdy } a \textbf{ nie } \text{należy do } k\text{-otoczenia } C_i.$$

Taką tablicę dla wzorca p możemy obliczyć w czasie $O((\mid \mathcal{A} \mid + k)m)$.
Po zbadaniu przekątnej h możemy ustalić przesunięcie d tak aby rozpocząć sprawdzanie od przekątnej d. Oczywistym jest, że minimalnie d może wynosić $k+1$ i niczego nie pominiemy, jednak oby zrobić to lepiej, możemy skorzystać z podejścia analogicznego do algorytmu $Boyer'a - Moore'a$. Skorzystajmy tutaj z tablicy przesunięć $BM_k[i,a] = min\{s : \ s = m \text{ lub } (1 \leq s < morazp_{i-s} = a)\}$ wymiaru $(k+1) \times \mid \mathcal{A} \mid$. Poniższy algorytm oblicza ją w czasie $O(m + k \mid \mathcal{A} \mid)$:

Algorytm 30: Obliczanie tablicy przesunięć Boyer'a-Moore'a

```
def basic_fft(text, word, n, m):
  if n < m:
    return
  A = set(list(text[1:] + word[1:]))
  A.discard('?')
  mismatches = [0] * (n - m + 1)
  for a in A:
    text_binary = [int(c == a) for c in text[1:]]
    for b in A:
      if a == b:
        continue
      word_binary = [int(c == b) for c in reversed(word[1:])]
      mismatches_ab = scipy.signal.convolve(
        text_binary, word_binary, mode = 'valid', method = 'fft')
      mismatches = [x + y for x, y in zip(mismatches, mismatches_ab)]
  yield from (i + 1 for i, v in enumerate(mismatches) if v == 0)
```

Cała faza skanowania opisana jest w następującym kodzie:

Algorytm 31: Faza skanowania algorytmu ABM

```
def basic_fft(text, word, n, m):
  if n < m:
    return
  A = set(list(text[1:] + word[1:]))
  A.discard('?')
  mismatches = [0] * (n - m + 1)
  for a in A:
    text_binary = [int(c == a) for c in text[1:]]
    for b in A:
      if a == b:
        continue
      word_binary = [int(c == b) for c in reversed(word[1:])]
      mismatches_ab = scipy.signal.convolve(
        text_binary, word_binary, mode = 'valid', method = 'fft')
      mismatches = [x + y for x, y in zip(mismatches, mismatches_ab)]
  yield from (i + 1 for i, v in enumerate(mismatches) if v == 0)
```

13.2.4 Złożoność.

Obliczanie tablic BAD i BM_k przy założeniu, że $k < m$ zajmuje $O((k + |\mathcal{A}|)m)$. Cały algorytm skanowania w najgorszym przypadku zajmuje $O(\frac{mn}{k})$. Autorzy pracy `https://www.`

researchgate.net/publication/220616992_Approximate_Boyer-Moore_String_Matching udowodnili (dość nietrywialnie) następujące twierdzenie:

Twierdzenie 13.10. *Dla* $2k + 1 < |\mathcal{A}|$ *oczekiwana czas trwania Algorytmu 2 zajmuje* $O(\frac{|\mathcal{A}|}{|\mathcal{A}|-2k} \cdot kn \cdot (\frac{k}{|\mathcal{A}|+2k^2} + \frac{1}{m}))$.

Fazę sprawdzania można zaimplementować tak by działała w czasie $O(kn)$, co sprawia, że cały algorytm ABM rozwiązuje problem k-przybliżonego dopasowania względem odległości edycyjnej w oczekiwanym czasie $O(kn \cdot (\frac{1}{m-k} + \frac{k}{|\mathcal{A}|}))$

14 Kompresja

15 Najkrótsze wspólne nadsłowo

Problem 9: Najkrótsze wspólne nadsłowo

Wejście: Zbiór słów $T = \{t_1, t_2, \ldots, t_k\} \subseteq \mathcal{A}^+$ takich, że $|t_i| \leq n$ dla $1 \leq i \leq k$.
Wyjście: Słowo t będące najkrótszym nadsłowem wszystkich słów z T.

Bez straty ogólności zakładamy, że dla żadnej pary słów t_i i t_j nie zachodzi sytuacja, że t_i jest podsłowem t_j. Gdyby taka sytuacja wystąpiła, to moglibyśmy usunąć t_i z T bez zmiany optymalnego rozwiązania.

W ogólności problem jest trudny, zarówno dla krótkich słów, jak i dla małych alfabetów.

Twierdzenie 15.1 (Gallant, Maier i Storer 1980)**.** *Niech* $T = \{t_1, t_2, ..., t_k\}$ *i* $|t_i| = 3$ *dla* $1 \leq i \leq k$. *Problem znalezienia najkrótszego nadsłowa dla* T *jest NP-trudny.*

Twierdzenie 15.2 (Gallant, Maier i Storer 1980)**.** *Niech* $T = \{t_1, t_2, ..., t_k\}$ *oraz* $|\mathcal{A}| = 2$. *Problem znalezienia najkrótszego nadsłowa dla* T *jest NP-trudny.*

Pokazano, że problem znalezienia najkrótszego nadsłowa należy do klasy złożoności MAX-SNP, a zatem istnieje stała $c > 1$, dla której nie istnieje algorytm c-przybliżony dla tego problemu. Vassilevska (2005) dowiodła, że $c \geq \frac{1217}{1216}$, Karpinski i Schmied (2013) poprawili to do $c \geq \frac{333}{332}$.

Problem 15.1. Niech $T = \{t_1, t_2, ..., t_k\}$ przy $|t_i| \leq 2$ dla $1 \leq i \leq k$. Pokaż, jak możliwe jest wyznaczenie najkrótszego nadsłowa t dla T w czasie wielomianowym.

15.1 Algorytm zachłanny

Kod i dowód złożoności

Niech $T = \{c(ab)^k, (ba)^k, (ab)^k c\}$, wówczas algorytm zachłanny najpierw złoży ze sobą dwa skrajne ciągi, a zatem $t_{APX} = (ba)^k c(ab)^k c$, $|t_{APX}| = 4k + 2$, podczas gdy $t_{OPT} = c(ab)^{k+1}c$, $|t_{OPT}| = 2k + 4$.

Została postawiona hipoteza, że algorytm zachłanny jest 2-przybliżony, ale dotychczas udało się tylko pokazać współczynnik aproksymacji równy 4 (Blum i in., 1994), a następnie poprawić współczynnik go do 3.5 (Kaplan i in., 2005).

Zamiast porównywać długości słowa optymalnego i zwróconego przez algorytm możemy też zmienić kryterium optymalizacji. Naturalnym alternatywnym kryterium wydaje się wielkość kompresji:

Problem 10: Wspólne nadsłowo o największej kompresji

Wejście: Zbiór słów $T = \{t_1, t_2, \ldots, t_k\} \subseteq \mathcal{A}^+$ takich, że $|t_i| \leq n$ dla $1 \leq i \leq k$.
Wyjście: Słowo t będące nadsłowem wszystkich słów z T i maksymalizujące $d = \sum_{i=1}^{k} |t_i| - |t|$.

Oczywiście zbiór rozwiązań optymalnych dla obu problemów jest identyczny.

Okazuje się, że przy tych założeniach można pokazać, że algorytm zachłanny jest 2-przybliżony tj. osiąga kompresję taką, że $2d_{APX} \geq d_{OPT}$ (Tarhio i Ukkonen, 1988).

15.2 Inne wyniki

Oprócz algorytmu zachłannego opracowywano inne algorytmy o współczynnikach aproksymacji kolejno $\frac{8}{3}$ (Armen i Stein, 1996), $\frac{5}{2}$ (Kaplan i in., 2005; Sweedyk, 2000), $\frac{57}{23}$ (Mucha, 2013) i – obecnie najlepszego – $\frac{71}{30}$ (Paluch, 2014). Większość z tych rezultatów polega na odpowiednio sprytnej redukcji do odpowiednio efektywnego algorytmu dla problemu maksymalnej ścieżki komiwojażera w grafie asymetrycznym.

Dodatkowo, jeśli przyjmiemy, że $|t_i| = r \geq 3$, to istnieje algorytm $\frac{r^2+r-4}{4r-6}$-przybliżony (Golovnev, Kulikov i Mihajlin, 2013). W Braquelaire i in. (2018) ten wynik został nieznacznie poprawiony – pozostaje on nadal jednak lepszy niż algorytm ogólny tylko dla $2 \leq r \leq 7$.

15.3 Aproksymacja najkrótszego nadsłowa

Problem najkrótszego nadsłowa w wersji decyzyjnej jest problemem NP zupełnym. W związku z tym pojawiało się wiele podejść do aproksymacji optymalizacyjnej wersji problemu. Tarhio i Ukkonen oraz Turner napisali dwie niezależne prace naukowe, w których opisano algorytmy aproksymacyjne dla rozważanego problemu. Mimo, że autorzy twierdzili, że skonstruowane przez nich algorytmy powinny być 2-aproksymacyjne, to nie udało im się udowodnić żadnego ograniczenia.

W tym dokumencie przeanalizujemy algorytm opisany przez Minga Li, który jest pierwszą aproksymacyją, dla której udało się udowodnić współczynnik - $log(n)$. Autor pracy twierdzi,

że współczynnik w rzeczywistości wynosi co najwyżej 2.

15.3.1 Algorytm

Zakładamy, że podane na wejściu słowa spełniają warunek, iż żadne nie jest podsłowem innego. Możemy tak zrobić, ponieważ wyeliminowanie tego typu sytuacji jest niezmiennicze ze względu na wynik i zajmuje wielomianowy czas.

Definicja 15.3. Dla danych słów s_1, s_2 słowo $m(s_1, s_2)$, długości nie większej niż $|s_1| + |s_2|$, nazywamy scaleniem, jeżeli s_1 jest jego prefiksem, a s_2 sufiksem (lub s_2 prefiksem, a s_1 sufiksem).

Lemat 15.4. *Dla danych słów s_1, s_2 istnieje co najwyżej $2min(|s_1|, |s_2|)$ scaleń.*

Dowód powyższego lematu jest trywialny.

Algorytm

1. Input to $S = \{s_1, \ldots, s_n\}$. Jeżeli $|S| = 1$, to zwróć s_1.
2. Zdefiniujmy $T = \emptyset$.
3. Niech s, s' będą słowami minimalizującymi
$$\min_{m-\text{scalenie}} \frac{|m(s, s')|}{v(m(s, s'))},$$
gdzie
$$v(m(s, s')) = \sum_{a \in A(m(s,s'))} |a|$$
dla $A(m(s, s'))$ będącego zbiorem podsłów $m(s, s')$ z S. Wrzucamy $m(s, s')$ do T dla m realizującego powyższe minimum, natomiast wszystkie słowa z $A(m(s, s'))$ wyrzucamy z S.
4. Jeżeli $|S| \leq 1$, to $S = T \cup S$ i przechodzimy do 1.
5. W przeciwnym przypadku przechodzimy do 3.

Złożoność jest oczywiście wielomianowa - przy każdym powrocie do punktu 1, moc zbioru S zmniejszyła się przynajmniej dwukrotnie, natomiast sumaryczna długość słów znajdujących się w nim może się tylko zmniejszyć.

Wyszukiwanie minimum i modyfikacja zbiorów w punktach 3 i 4 jest trywialnie realizowane w czasie wielomianowym.

Pozostaje ograniczyć współczynnik aproksymacji algorytmu.

15.3.2 Współczynnik aproksymacji

Twierdzenie 15.5. *Algorytm przedstawiony powyżej jest $log(n)$ aproksymacyjny.*

Dowód. Niech $s_1, \ldots, s_m$ będzie ułożeniem słów wejściowych w kolejności ich występowania w optymalnym rozwiązaniu - jeżeli słowo występuje kilka razy, to rozważamy najbardziej lewe wystąpienie. Dzielimy słowa na następujące grupy:
Grupa G_1 zawiera $s_1, \ldots, s_i$, gdzie s_i to ostatnie słowo takie, że pierwsze wystąpienie s_1 nachodzi na pierwsze wystąpienie s_i w optymalnym rozwiązaniu.
Grupa G_2 budowana jest w analogiczny sposób, z tym że zaczynamy konstrukcje od s_{i+1}. W ten sposób dzielimy całe wejście $s_1, \ldots, s_m$ na rozłączne grupy $G_1, \ldots, G_k$.

Dla każdej grupy G_i wyróżniamy elementy b_i, t_i - pierwsze i ostatnie słowa (we wcześniej wspomnianym porządku) budujące grupę. Jako że b_i nachodzi na t_i, to istnieje scalenie m_i takie, że wszystkie słowa z grupy G_i są podsłowami $m_i(b_i, t_i)$. Zauważmy, że

$$\sum_{i=1}^{k} |m_i(b_i, t_i)| \leq 2n,$$

gdzie n to długość optymalnego rozwiązania, ponieważ każda litera słowa optymalnego jest pokrywana przez maksymalnie dwie grupy.

Wprowadźmy teraz pojęcie kosztu słowa w kontekście analizowanego przez nas algorytmu. Załóżmy, że słowo s odpadało ze zbioru S na rzecz scalenia $m(s', s'')$.

$$cost(s) = \frac{|m(s', s'')|}{v(m(s', s''))} \cdot |s|.$$

Zauważmy, że koszt całego algorytmu (długość słowa wynikowego) jest równa nie więcej niż długość wszystkich słów, które powstały w trakcie pierwszej fazy opróżniania zbioru S - elementów $s_1, \ldots, s_m$. Niech $S_h = A(M_h(B_h, T_h))$ to zbiór elementów wyrzuconych h-tym kroku wspomnianej fazy.

$$COST \leq \sum_{h=1}^{r} |M_h(B_h, T_h)| = \sum_{i=h}^{r} \frac{|M_h(B_h, T_h)|}{v(M_h(B_h, T_h))} \cdot v(M_h(B_h, T_h)) =$$

$$\sum_{h=1}^{r} \sum_{s \in S_h} \frac{|M_h(B_h, T_h)|}{v(M_h(B_h, T_h))} \cdot |s| = \sum_{h=1}^{r} \sum_{s \in S_h} cost(s) = \sum_{s \in S} cost(s).$$

□

Rozważmy teraz grupę G_j i koszt, jaki płacimy za wyrzucenie jej elementów. Grupę G_j dokładnie przed fazą h będziemy oznaczać G_j^h. Sumę długości słów w zbiorze A oznaczamy $||A||$.

$$Cost(G_j) = \sum_{h=1}^{k} \sum_{s \in (G_j^{h+1} \setminus G_j^h)} cost(s) = \sum_{h=1}^{k} \frac{|M_h(B_h, T_h))|}{v(M_h(B_h, T_h))} \cdot (||G_j^{h+1}|| - ||G_j^h||).$$

Jako że nasz algorytm wybierał scalenia $M_h(B_h, T_h)$ realizujące minimalne wartości, to

$$\sum_{h=1}^{k} \frac{|M_h(B_h,T_h))|}{v(M_h(B_h,T_h))} \cdot (||G_j^{h+1}|| - ||G_j^h||) \leq \sum_{h=1}^{k} \frac{|m_j(b_j,t_j))|}{v(m_j(b_j,t_j))} \cdot (||G_j^{h+1}|| - ||G_j^h||) =$$

$$\sum_{h=1}^{k} \frac{|m_j(b_j,t_j))|}{||G_j^h||} \cdot (||G_j^{h+1}|| - ||G_j^h||) \leq |m_j(b_j,t_j))| \cdot Harm(||G_j||).$$

Jako że n jest co najwyżej wielomianowo większe niż każde $||G_j||$, to dostajemy oszacowanie

$$Cost(G_j) \leq |m_j(b_j,t_j))| \cdot log(n),$$

co z poprzednim lematem skutkuje w

$$COST \leq 2nlog(n).$$

16 Dokładne dopasowanie wielu wzorców

16.1 Algorytm Aho-Corasick

Niech $\mathcal{W} = \{w_1, ..., w_k\}$ będzie zbiorem wzorców (niepustych słów nad alfabetem Σ). Otrzymawszy słowo $T \in \Sigma^n$ chcemy wyznaczyć wszystkie indeksy z $[n]$, na których w T zaczyna się pewien wzorzec z $\mathcal{W}$. Naturalnym rozwiązaniem jest k-krotne użycie jednego z wydajnych (liniowych) algorytmów do szukania pojedynczego wzorca. Podejście takie oczywiście jest poprawne, niestety skutkuje złożonością czasową $\Theta(kn)$. Okazuje się, że na podstawie $\mathcal{W}$ jesteśmy w stanie skonstruować w czasie $\Theta(\sum_{i\in[k]} |w_i|)$ odpowiednią strukturę, która będzie potrafiła znaleźć naraz wszystkie wystąpienia wzorców oglądając T wyłącznie raz. Przed nami: automat Aho-Corasick.

Pomysł polega na zbudowaniu odpowiedniego deterministycznego automatu skończonego $\mathcal{A}$. Czytając T będziemy podróżować zgodnie z krawędziami $\mathcal{A}$. W każdym stanie s będziemy mieć zapisaną listę tych wzorców z $\mathcal{W}$, które *kończą się* w s – tzn. aby znaleźć się w stanie s musieliśmy przejść skądś krawędziami etykietowanymi kolejnymi literami pewnego wzorca. Spacerowanie po $\mathcal{A}$ wzdłuż T możemy interpretować jako poruszanie się *jednocześnie* po wszystkich słowach z $\mathcal{W}$.

Zaczniemy konstrukcję od utworzenia *drzewa trie* na podstawie $\mathcal{W}$. Następnie dodamy do każdego stanu informacje o wzorcach które się w nim kończą i podniesiemy funkcję przejścia z częściowej do totalnej. Na koniec uprościmy automat. Po tych zabiegach otrzymamy strukturę, dzięki której wyszukiwanie wielu wzorców okaże się bardzo proste:

```
def find_occurrences(self, text, n):
  state = self._root
```

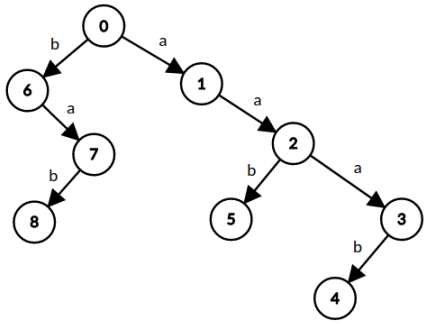
```
for i in range(1, n + 1):
  state = state.nxt(text[i])
for keyword_len in state.output():
  start_pos = i - keyword_len + 1
  yield text[start_pos:i + 1], start_pos
```

Warto zwrócić uwagę na fakt, że jeśli dany zbiór wzorców chcemy wyszukać w wielu różnych tekstach $T_1, ..., T_t$, to wystarczy tylko raz skonstruować automat Aho-Corasick i użyć go niezależnie na każdym z T_j.

Uwaga: sednem naszych rozważań jest opis sposobu, w jaki można przechodzić pomiędzy stanami automatu. Wygodnie nam będzie utożsamić ze sobą różne podejścia realizacji – równoważne dla nas będą pojęcia zbioru etykietowanych krawędzi (skierowanych) pomiędzy dwoma stanami, funkcji mapujących pary (stan, symbol) w stan oraz lokalne w stanach tablice routingu.

16.2 Drzewo trie

Mając zbiór słów $\mathcal{W}$ nad alfabetem Σ możemy w prosty sposób zbudować tzw. *drzewo trie*, ozn. $\mathcal{T}_{\mathcal{W}}$. Jest to skierowane, ukorzenione drzewo, którego krawędzie etykietowane są symbolami z Σ. Każde słowo $w \in \mathcal{W}$ ma odpowiadającą mu ścieżkę z korzenia, etykietowaną kolejnymi literami w. Jeśli $w \in \mathcal{W}$ nie jest prefiksem żadnego innego słowa z $\mathcal{W}$, to jego ścieżka kończy się w liściu. Inaczej ujmując, istnieje bijekcja pomiędzy ścieżkami w $\mathcal{T}_{\mathcal{W}}$ a wszystkimi (różnymi) prefiksami słów z $\mathcal{W}$. Krawędź etykietowaną symbolem a pomiędzy s_1 a s_2 możemy kodować jako trójkę (s_1, a, s_2).



Rysunek 2: $\mathcal{T}_{\{aaab,aab,bab,ba\}}$

Zacznijmy temat konstrukcji od zdefiniowania funkcji, które będą nam potrzebne. Niech $\mathcal{W}$ oznacza zbiór wzorców, $\mathcal{T}$ oznacza drzewo trie dla $\mathcal{W}$ (skrót dla $\mathcal{T}_{\mathcal{W}}$), $\mathfrak{r}$ oznacza korzeń $\mathcal{T}$, $\mathcal{S}$ oznacza zbiór wierzchołków $\mathcal{T}$ (czyli docelowo zbiór stanów $\mathcal{A}$).

- `goto` $:: \mathcal{S} \times \Sigma \mapsto \mathcal{S}$ – ta funkcja w zasadzie reprezentuje zbiór krawędzi $\mathcal{T}$; dodatkowo jednak, dla każdego symbolu w Σ, który nie etykietuje żadnej krawędzi wychodzącej z

$\mathfrak{r}$, dodajemy pętlę w $\mathfrak{r}$; czyli:

$$\texttt{goto}(s_1, a) = \begin{cases} s_2 & \text{jeśli } (s_1, a, s_2) \in E[\mathcal{T}] \\ \mathfrak{r} & \text{jeśli } s_1 = \mathfrak{r} \ \wedge \forall_s \ (\mathfrak{r}, a, s) \notin E[\mathcal{T}] \\ \texttt{None} & \end{cases}$$

- `output` $:: \mathcal{S} \mapsto \wp(\mathcal{W})$ – jak zostało już wspomniane, chcemy w każdym stanie s przechowywać listę wzorców, których wystąpienie w T będzie poświadczone odwiedzeniem stanu s; w sporym zakresie możemy to obliczyć już przy konstrukcji $\mathcal{T}$ (np. wracając do rysunku 2, `output`$(1) = \emptyset$, `output`$(5) = \{aab\}$, `output`$(7) = \{ba\}$); będziemy musieli jeszcze jednak obsłużyć przypadki, gdy jeden wzorzec będzie sufiksem drugiego (np. `output`$(4) = \{aaab, aab\}$)
- `fail` $:: \mathcal{S} \mapsto \mathcal{S}$ – ostatecznie $\mathcal{A}$ będzie skonstruowane na wierzchołkach $\mathcal{T}$ więc już teraz możemy spróbować zinterpretować obecność w stanie s po przeczytaniu j-tego symbolu T – otóż etykiety na ścieżce pomiędzy $\mathfrak{r}$ a s tworzą najdłuższy możliwy prefiks któregoś ze wzorców z $\mathcal{W}$, który da się dopasować w T kończąc na pozycji j; funkcja `goto` jest częściowa, a więc dla niektórych stanów i niektórych symboli nie mamy zdefiniowanego przejścia (na razie otrzymamy `None`); aby nie utknąć i zachować poprawność interpretacji, to `fail`(s) musi wskazywać na taki stan s', żeby słowo powstałe ze ścieżki $\mathfrak{r} \rightsquigarrow s'$ było najdłuższym możliwym sufiksem słowa odpowiadającemu $\mathfrak{r} \rightsquigarrow s$; w dalszej części utożsamiać będziemy ścieżki ze słowami;
  Chodzenie krawędziami `fail` opiera się na identycznym pomyśle co rekurencyjne używanie tablic *Border* w algorytmie Morrisa-Pratta. Warto również zauważyć, że `fail` nie ma sensu definiować dla $\mathfrak{r}$.
- `next` $:: \mathcal{S} \times \Sigma \mapsto \mathcal{S}$ – trzy zdefiniowane funkcje powyżej zupełnie wystarczają, aby nasz docelowy automat robił to, czego pragniemy; jak jednak łatwo zauważyć, próba przeczytania jednego znaku z T może wymagać więcej niż jednego przejścia krawędzią korzystając z `goto` oraz `fail` – możemy musieć schodzić wzdłuż `fail` wielokrotnie; aby uniknąć tego, pod sam koniec utworzymy funkcję `next`, która skompresuje nam tego typu ścieżki i zastąpi `goto` oraz `fail`

16.3 Obliczanie `goto`, `output`, `fail`

Jak już wiemy *co* chcemy policzyć, to możemy przejść do tego *jak* to zrobić. Funkcję `goto` oraz część `output`, jak już zauważyliśmy, możemy obliczyć konstruując $\mathcal{T}_{\mathcal{W}}$:

```
def _construct_goto(self, keywords, alphabet):
  for k, k_len in keywords:
    self._enter(k, k_len)

  for a in alphabet:
    if self._root.goto(a) is None:
```

```
      self._root.update_goto(a, self._root)

def _enter(self, keyword, keyword_len):
  current_state = self._root
  j = 1

  while j < keyword_len and current_state.goto(keyword[j]) is not None:
    current_state = current_state.goto(keyword[j])
    j += 1

  for a in keyword[j:keyword_len + 1]:
    next_state = AhoCorasickAutomaton.Node()
    current_state.update_goto(a, next_state)
    current_state = next_state

  current_state.append_outputs([(keyword, keyword_len)])
```

Funkcja def _enter(...) uzupełnia częściowo skonstruowane drzewo trie o ścieżkę dla podanego wzorca, w razie potrzeby tworząc nowe węzły. W pętli w wierszu 5. uzupełniamy definicję `goto` dla korzenia.

Funkcję `fail` powinniśmy obliczać dla coraz bardziej odległych od $\mathfrak{r}$ węzłów – podobnie jak w algorytmie Morrisa-Pratta długości najdłuższych borderów obliczaliśmy na podstawie wartości dla krótszych słów:

```
def _construct_fail(self, alphabet):
  q = Queue()
  for s in (self._root.goto(a) for a in alphabet):
    if s != self._root:
      q.put(s)
      s.update_fail(self._root)

  while not q.empty():
    current = q.get()
    for a, child in ((a, current.goto(a)) for a in alphabet):
      if child is not None:
        q.put(child)

        fallback = current.fail()
        while fallback.goto(a) is None:
          fallback = fallback.fail()

        child_fallback = fallback.goto(a)
```

```
        child.update_fail(child_fallback)
        child.append_outputs(child_fallback.output())
```

Przechodzimy zatem po $\mathcal{T}$ za pomocą algorytmu BFS. Będąc w wierzchołku `current`, widząc krawędź etykietowaną przez `a` do wierzchołka `child`, chcemy obliczyć `fail`(`child`). Szukamy więc najdłuższego sufiksu słowa generowanego przez $\mathfrak{r} \rightsquigarrow$ `child` obecnego w $\mathcal{T}$. Będzie to najdłuższy sufiks ścieżki $\mathfrak{r} \rightsquigarrow$ `current`, który można przedłużyć o symbol `a`.

Znalazłszy odpowiedni wierzchołek `child_fallback` powiększamy zbiór `output`(`child`) o wartości z `output`(`child_fallback`) – z indukcji wynika, że `output`(`child_fallback`) zawiera już wszystkie wzorce, które są sufiksami $\mathfrak{r} \rightsquigarrow$ `child_fallback`.

16.4 Obliczanie `next`

Jeśli `goto` była określona dla $s \in \mathcal{S}$ oraz $a \in \Sigma$, to `next`(s, a) = `goto`(s, a). W przeciwnym wypadku będąc w s i chcąc przejść symbolem a, powinniśmy schodzić krawędziami `fail` tak długo, aż znajdziemy się w s', dla którego `goto`$(s', a) \neq$ `None`. Jest to podobny zabieg co poprawka Knutha do algorytmu Morrisa-Pratta.

```
def _construct_next(self, alphabet):
  q = Queue()
  for a in alphabet:
    a_child = self._root.goto(a)
    self._root.update_next(a, a_child)
    if a_child != self._root:
      q.put(a_child)
  self._root.use_only_next()

  while not q.empty():
    current = q.get()
    for a, child in ((a, current.goto(a)) for a in alphabet):
      if child is not None:
        q.put(child)
        current.update_next(a, child)
      else:
        fallback = current.fail()
        current.update_next(a, fallback.next(a))
    current.use_only_next()
```

Instrukcje w wierszach 8. oraz 19. pozwalają zwolnić pamięć zapominając o wartościach funkcji `goto` i `fail`. Formalnie dopiero teraz, struktura $\mathcal{A}$ ze zbiorem stanów $\mathcal{S}$ oraz funkcją przejścia `next` jest deterministycznym automatem skończonym.

16.4.1 Analiza poprawności

Z tego, co już zostało przedstawione, powinna wynikać poprawność zastosowanej metody. Aby jednak formalnie jej dowieść, wystarczy postawić trzy proste lematy:

Lemat 16.1. *Niech $s, t \in \mathcal{S}$ i niech u, v będą słowami generowanymi przez ścieżki $\mathfrak{r} \rightsquigarrow s$ i $\mathfrak{r} \rightsquigarrow t$. Wówczas $\texttt{fail}(s) = t \iff v$ jest najdłuższym właściwym sufiksem u, który jest prefiksem pewnego $w_i \in \mathcal{W}$.*

Lemat 16.2. *Słowo k należy do $\texttt{output}(s) \iff w \in \mathcal{W}$ oraz k jest sufiksem słowa generowanego przez $\mathfrak{r} \rightsquigarrow s$.*

Lemat 16.3. *Po przeczytaniu j znaków z T znajdujemy się w stanie $s \iff$ słowo generowane przez $\mathfrak{r} \rightsquigarrow s$ jest najdłuższym sufiksem $T[1..j]$, który jest prefiksem pewnego $w \in \mathcal{W}$.*

Ich uzasadnienia są prostymi dowodami rekurencyjnymi zapisanymi we fragmentach kodu powyżej.

16.5 Analiza złożoności

Konstrukcja całego automatu polega na sekwencyjnym wywołaniu procedur obliczających funkcje przejścia:

```
def basic_fft(text, word, n, m):
  if n < m:
    return
  A = set(list(text[1:] + word[1:]))
  A.discard('?')
  mismatches = [0] * (n - m + 1)
  for a in A:
    text_binary = [int(c == a) for c in text[1:]]
    for b in A:
      if a == b:
        continue
      word_binary = [int(c == b) for c in reversed(word[1:])]
      mismatches_ab = scipy.signal.convolve(
        text_binary, word_binary, mode = 'valid', method = 'fft')
      mismatches = [x + y for x, y in zip(mismatches, mismatches_ab)]
  yield from (i + 1 for i, v in enumerate(mismatches) if v == 0)
```

Niech $m = \max_i |w_i|$. Bardzo łatwo zauważyć, że każdą z powyższych operacji wykonać można w czasie $\mathcal{O}(km)$ (dokładnie $\Theta(\sum_{i \in [k]} |w_i|)$) – za każdym razem przechodzimy po całym drzewie $\mathcal{T}$ odwiedzając każdy wierzchołek dokładnie raz oraz wykonując $\Theta(1)$ operacji w

każdym z nich. Zakładamy również, że łączenie zbiorów `output` wykonuje się w czasie stałym (jak np. listy wiązane). Warto dodać, że moc Σ wlicza się w zapotrzebowanie (zarówno pamięciowe jak i czasowe) liniowo – rozmiar grafu jest z góry ograniczony przez $|\Sigma| \cdot km$ (z każdego wierzchołka może wychodzi $|\Sigma|$ krawędzi). Konstrukcja $\mathcal{A}$ oraz wyszukiwanie w T odbywają się w czasie liniowo proporcjonalnym do wielkości $\mathcal{T}$.

Pozostaje zastanowić się, ile czasu zajmie przeczytanie tekstu T ($|T| = n$) i wypisanie wszystkich wystąpień wzorców. Moc Σ traktujemy jak stałą.

Najpierw rozważmy sytuację, w której nie zgłaszamy żadnego wystąpienia wzorca (możemy np. tylko zaznaczyć pozycje T o których wiemy, że na nich coś się kończy):

- używając funkcji `goto` oraz `fail` wykonamy co najwyżej $2n$ przejść w $\mathcal{A}$ – po przeczytaniu dowolnego symbolu przejdziemy dokładnie raz według funkcji `goto` i być może wielokrotnie krawędziami `fail`; łatwo jednak zaobserwować, że każde przejście `fail` skraca odległość z obecnego wierzchołka do `t`, a z `t` zawsze można wyjść używając `goto`; zatem nie możemy przejść krawędziami `fail` więcej razy niż przeszliśmy `goto`;
  można również spojrzeć na to w ten sposób, że przechodząc automatem $\mathcal{A}$ wzdłuż T, pamiętamy dwa wskaźniki w T – słowo pomiędzy nimi to właśnie to, które jest generowane przez ścieżkę z korzenia do obecnego wierzchołka; przejście `goto` przesuwa 'prawy' wskaźnik o 1 w prawo, przejście `fail` przesuwa 'lewy' wskaźnik o dodatnią liczbę pozycji w prawą, ale na tyle małą by nie przeskoczyć prawego;
- używając funkcji `next`, przy każdym przeczytanym symbolu z T wykonujemy zawsze jedno przejście – n ruchów

Problem może się pojawić, gdy będziemy chcieli wypisywać wszystkie wystąpienia wzorców. Pesymistycznym przykładem jest $\mathcal{W} = \{a, a^2, a^3, ..., a^k\}$ oraz $T = a^n$. Liczba wystąpień będzie $\Theta(kn)$. Jednakże, nie jest to wada zastosowanego algorytmu – każdy inny algorytm zgłaszający wszystkie wystąpienia wzorców musiałby wykonać co najmniej tyle samo pracy.

Zatem możemy określić złożoność czasową wyszukiwania wielu wzorców w tekście za pomocą automatu Aho-Corasick jako liniową w stosunku do sumy długości wszystkich wzorców oraz ilości ich wystąpień.

16.5.1 Zastosowanie do wzorców 2D

Okazuje się, że bardzo prosto można użyć automatu Aho-Corasick do wyszukiwania wzorców dwuwymiarowych. W skrócie:

1. niech T będzie kwadratową tablicą rozmiaru n^2, P - kwadratową tablicą rozmiaru m^2 o różnych kolumnach, $m < n$
2. niech $\mathcal{W}$ będzie zbiorem kolumn P
3. automatem Aho-Corasick szukamy $\mathcal{W}$ w każdej z kolumn T; jeśli fragment j-tej kolumny od indeksu i pokrywa się z k-tą kolumną P, to niech $T_P[i][j] = k$
4. algorytmem KMP szukamy w każdym wierszu T_P ciągu $(1, 2, ..., m)$

Całość działa w czasie liniowym od wielkości wejścia i rozmiaru zbioru z którego pochodzą elementy w T oraz P.

16.6 Algorytm Commentz-Walter

Algorytm Commentz-Walter to algorytm tekstowy służący do wyszukiwania wystąpień wzorca w tekście opracowany przez Beate Commentz-Walter (Uniwersytet Kraju Saary w Saarbrücken) w 1979 roku. Podobnie jak algorytm Aho-Corasick znajduje on w tekście wystąpienia słów ze słownika (pewnego zadanego zbioru wzorców), czyli jest w stanie jednocześnie w zadanym tekście wyszukiwać wiele wzorców.

Algorytm Commentz-Walter jest merytorycznie ciekawy z tego powodu, że stanowi bardzo sprytną fuzję dwóch innych algorytmów, łącząc idee leżące u podstaw wspomnianego już algorytmu Aho-Corasick oraz algorytmu Boyera-Moore'a. Algorytm, o którym sobie dziś opowiemy, dla tekstu o długości n i takiego zestawu wzorców, że najdłuższy spośród nich ma długość m, ma pesymistyczny czas działania $O(mn)$. W praktyce okazuje się jednak bardzo często, że czas działania tego algorytmu jest dużo lepszy. Stąd też nazwa pracy autorstwa Beate Commentz-Walter - „A String Matching Algorithm - Fast on the Average".

Najprawdopodobniej z powodu bardzo optymalnego średniego czasu działania, w GNU grep jest zaimplementowany algorytm wyszukujący wiele wzorców, bardzo podobny do oryginalnego algorytmu Commentz-Walter.

Najpierw przyjrzymy się nieco dokładniej dwóm głównym algorytmom, z których idei korzysta algorytm Commentz-Walter. Pierwszy z nich, algorytm Aho-Corasick jest jednym z algorytmów wyszukiwania wzorca w tekście. Znajduje on w tekście wystąpienia słów ze słownika (pewnego zadanego zbioru wzorców). Wszystkie wzorce są szukane jednocześnie, co powoduje, że złożoność obliczeniowa algorytmu jest liniowa od sumy długości wzorców, długości tekstu i liczby wystąpień wzorców w tekście. W tekście może jednak występować nawet kwadratowa od długości tekstu liczba wystąpień wzorców.

Ideą algorytmu jest stworzenie drzewa trie o sufiksowych połączeniach pomiędzy wierzchołkami reprezentującymi różne słowa. Inaczej mówiąc tworzone jest drzewo (automat) o etykietowanych krawędziach, w którym każdy wierzchołek reprezentuje pewne słowo, składające się ze złączonych etykiet krawędzi znajdujących się na ścieżce od korzenia do tego węzła. Dodatkowo dołączane są krawędzie od wierzchołka do innego wierzchołka reprezentującego jego najdłuższy sufiks (fail).

W czasie fazy wyszukiwania algorytm porusza się po tym drzewie zaczynając od korzenia i przechodząc do wierzchołka po krawędzi etykietowanej znakiem odczytanym z tekstu. W przypadku gdy takiej nie ma w drzewie, algorytm korzysta z krawędzi fail i tam próbuje przejść do wierzchołka po krawędzi etykietowanej odczytanym znakiem. W przypadku natrafienia na węzeł oznaczony jako koniec słowa, algorytm wypisuje je jako znalezione w tekście oraz sprawdza, czy czasem w tym miejscu nie kończy się więcej wzorców przechodząc krawędziami fail aż do korzenia sprawdzając, czy nie przechodzi po wierzchołku będącym

końcem wzorca.

Reasumując algorytm Aho-Corasick składa się z dwóch głównych faz, jedna to skonstruowanie odpowiedniego automatu dla zestawu podanych wzorców, a druga to użycie tego automatu do wyszukiwania tych wzorców w już podanym tekście. Oczywiście raz skonstruowany automat można wykorzystywać dowolnie wiele razy dla wielu różnych tekstów.

Drugi algorytm, z którego idei będziemy czerpać, to algorytm Boyera i Moore'a. Jest to algorytm wyszukiwania jednego wzorca w tekście. Polega na porównywaniu, zaczynając od ostatniego elementu wzorca. Jego główną zaletą jest to, że jeżeli okaże się, że znak, który aktualnie sprawdzamy, nie należy do wzorca, to możemy przeskoczyć w analizie tekstu o całą długość wzorca. Z reguły skoki wzorca są większe od 1. Między innymi to sprawia, że algorytm ten jest często znacznie lepszy niż algorytm KMP, a także w przypadku, gdy algorytm Aho-Corasick jest uruchamiany z jednym wzorcem, to prawie na pewno algorytm Boyera-Moore'a poradzi sobie z nim szybciej niż algorytm Aho-Corasick. Asymptotycznie czas potrzebny na wyszukanie wszystkich wystąpień wzorca długości m, w tekście długości n wynosi bardzo często średnio $O(n/m)$. Istnieją również ulepszenia oryginalnego algorytmu Booyera-Moore'a, które sprawiają, że faza wyszukiwania ma czas liniowy względem długości tekstu, nawet w przypadku pesymistycznym.

Uzbrojeni w to drobne przypomnienie dotyczące tych dwóch bazowych algorytmów jesteśmy gotowi przyjrzeć się algorytmowi Commentz-Walter. Algorytm ten podobnie jako Aho-Corasick składa się z dwóch faz. Pierwsza faza to zbudowanie struktury pozwalającej wyszukiwać wiele wzorców jednocześnie, a druga faza to już samo wyszukiwanie tych wzorców w konkretnym tekście. Wyszukiwanie zachowuje się dość podobnie jak wyszukiwanie w algorytmie Boyera-Moore'a, również osiągając średnio czas podliniowy względem rozmiaru tekstu. W kolejnych sekcjach tego omówienia przyjrzymy się z detalami kolejnym fazom tego algorytmu, a na koniec skomentujemy szerzej czas jego działania.

Do przechowywania w pamięci pewnego zestawu wzorców w użyteczny sposób pomocna będzie nam następująca struktura:

Definicja 16.4. Drzewem trie nazywamy takie ukorzenione drzewo T, że:

1. Każdy wierzchołek $v \in T$, poza korzeniem r ma pewną etykietę $a = l(v)$. Jest to element ustalonego alfabetu Σ.
2. Korzeń r ma przypisaną etykietę ϵ, za pomocą której oznaczamy puste słowo.
3. Jeżeli wierzchołki v_1, v_2 są różnymi dziećmi tego samego wierzchołka v, to $l(v_1) \neq l(v_2)$.

Mówimy, że ścieżka $v_1, \cdots, v_k$ (gdzie v_{i+1} jest synem v_i) w drzewie trie T **reprezentuje** słowo $l(v_1) \cdots l(v_k)$. Słowo to oznaczamy za pomocą $w(v_k)$, jeżeli $v_1 = r$. Dodatkowo

Definicja 16.5. dla każdego węzła $v \in T$ definiujemy jego **głębokość** $d(v) := 0$, jeżeli $v = r$ oraz $d(v) := d(v') + 1$, jeżeli wierzchołek v jest synem v'. Dodatkowo wprowadzamy oznaczenie na **głębokość całego drzewa trie** $d(T) := \max\{d(v) : v \in T\}$.

Niech $\mathcal{W} = \{w_1, \cdots, w_r\}$ będzie zbiorem wzorców nad pewnym alfabetem Σ, które będziemy chcieli wyszukiwać w pewnym tekście t. Podobnie jak w algorytmie Aho-Corasick

reprezentujemy zbiór $\mathcal{W}$ za pomocą drzewa trie T. **Główna różnica polega na tym, że nie trzymamy w drzewie trie wzorców, ale odwrócone wzorce.** Dokładniej dla każdego $h = 1, \cdots, r$ istnieje wierzchołek $v_h \in T$ reprezentujący odwrócony wzorzec w_h^R. Dodatkowo każdy wierzchołek drzewa trie jest wyposażony w tak zwaną **funkcję wyjścia** zdefiniowaną jako $out(v) := \{w : w^R = w(v)\}$.

Główna idea tego algorytmu zawiera się w dodaniu do drzewa trie dwóch funkcji **shift1** i **shift2**. Są to funkcje, które przypisują wierzchołkom drzewa trie liczby naturalne, które to liczby będą nam bardzo pomocne w fazie wyszukiwania, aby symulować przesunięcia znane z algorytmu Boyera-Moore'a. Formalna definicja tych funkcji jest następująca i wykorzystuje definicje dwóch typów zbiorów. Mianowicie

Definicja 16.6. dla każdego $v \neq r, v \in T$, definiujemy zbiór $set1(v)$ jako zbiór tych wszystkich wierzchołków v', że $w(v)$ jest poprawnym sufiksem słowa $w(v')$, tzn. $w(v') = uw(v)$, dla pewnego niepustego słowa u. Definiujemy również zbiór $set2(v)$ jako zbiór tych wszystkich wierzchołków v' takich, że $v' \in set1(v)$ oraz $out(v') \neq \emptyset$.

Przydadzą nam się jeszcze oznaczenia $wmax := \max_{1 \leq h \leq r} |w_h|$ oraz $wmin := \min_{1 \leq h \leq r} |w_h|$. Teraz jesteśmy już gotowi zdefiniować funkcje przesunięcia i definicje te są następujące.

Definicja 16.7. dla każdego $v \in T$ definiujemy funkcję **shift1** jako $shift1(v) := 1$ jeżeli $v = r$. W przeciwnym przypadku

$$shift1(v) := min\left(\{d(v') - d(v) : v' \in set1(v)\} \cup \{wmin\}\right)$$

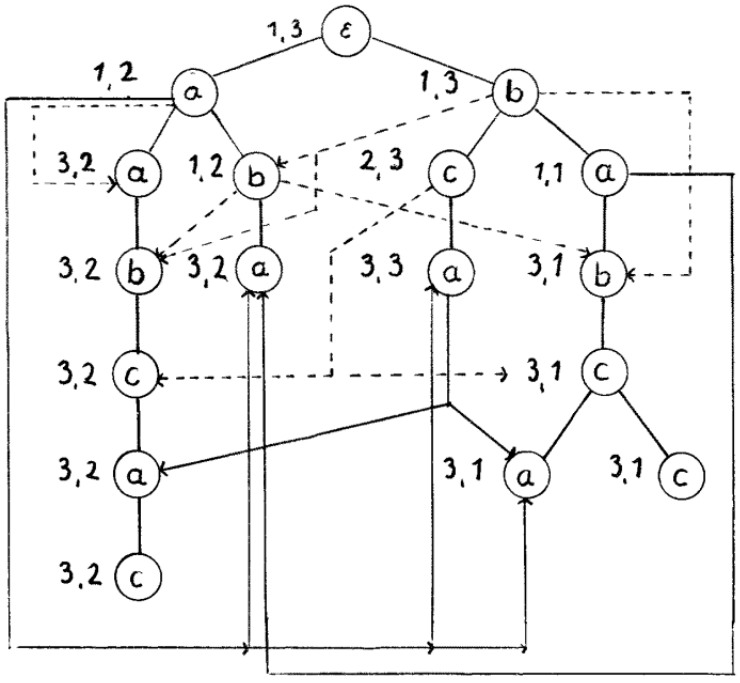
Definicja 16.8. dla każdego $v \in T$ definiujemy funkcję **shift2** jako $shift2(v) := wmin$ jeżeli $v = r$. W przeciwnym przypadku

$$shift2(v) := min\left(\{d(v') - d(v) : v' \in set2(v)\} \cup shift2(parent(v))\right)$$

Ostatnia funkcja, którą będziemy potrzebować to funkcja $char : \Sigma \longrightarrow \mathrm{N}$ zdefiniowana jako

$$char(a) := \min(\{d(v) : l(v) = a\} \cup \{wmin + 1\})$$

Poniżej mamy przykład (pochodzący z oryginalnej pracy) tak zbudowanej struktury dla zbioru wzorców $\{cacbaa, acb, aba, acbab, ccbab\}$, gdzie $wmin = 3$. Na poniższej ilustracji dla każdego wierzchołka $v \neq r$, ostrzałkowane linie ciągłe wskazują na elementy zbioru $set2(v)$, natomiast ostrzałkowane, jak i przerywane linie wskazują na elementy zbioru $set1(v)$.



16.6.1 Faza wyszukiwania w tekście

Wynikiem tej fazy jest lista par postaci (w, i), gdzie w to któryś z szukanym wzorców, natomiast i to pozycja w tekście, na której został znaleziony wzorzec w. Faza ta działa następująco. Załóżmy, że mamy już zbudowane drzewo trie składające się z wszystkich wzorców, ale wstawionych do drzewa trie w odwróconej kolejności. Cała sztuczka polega teraz na tym, że dopasowywanie robimy jak w algorytmie Aho-Corasick chodząc po drzewie trie, ale korzystając z pomysłu zapożyczonego z algorytmu Boyera-Moore'a, w przypadku niezgodności znaków używamy funkcji shift1 oraz shift2, aby wykonać odpowiedni skok w tekście, w którym dopasowujemy wzorce z drzewa trie. Zamieszczony poniżej pseudokod dość dokładnie tłumaczy jak wygląda ta kluczowa faza algorytmu.

Pozostaje uzasadnić, że ten algorytm na pewno zwraca wszystkie wystąpienia wzorców w tekście. Nie ma wątpliwości co do tego, że każda zwrócona para (w, i) jest na pewno dobrym wystąpieniem. Pozostaje więc się upewnić, że na pewno żadne wystąpienie któregoś z wzorców nie jest pomijane. Z budowy algorytmu wynika, że jedynym podejrzanym miejscem jest moment robienia przesunięcia w tekście. Potrzebujemy, więc podać argument, że $t[i-j+1]\cdots t[i] = w_t^R(v)$ dla pewnego $v \in T$ implikuje, że nie istnieje takie i', że $i < i' < s(v, t[i-j])$ oraz $t[i' - |w| + 1]\cdots t[i']$ dla pewnego wzorca w. Jednak z definicji $s(v, t[i-j])$ mamy tę własność po prostu za darmo.

Problem 11: Commentz-Walter, faza wyszukiwania

procedure COMMENTZ-WALTER-SEARCH($text, n, trie$) ▷ tekst, długość, drzewo trie wzorców
$v := root(r)$ ▷ obecnie przetwarzany wierzchołek trie
$i := wmin$ ▷ obecnie przetwarzana pozycja w tekście
$j := 0$ ▷ głębokość v w drzewie trie
While $j \leq n$
While $\exists$ dziecko v' wierzchołka v takie, że $l(v') = t[i-j]$ ▷ faza skanowania
$v = v'$
$j = j + 1$
yield (w, i) dla każdego $w \in out(v)$
EndWhile
$shift(v, t[i-j]) := \min(\max(shift1(v), char(t[i-j]) - j - 1), shift2(v))$
$i = i + shift(v, t[i-j])$ ▷ faza przesuwania tekstu
$j = 0$
EndWhile
end procedure

16.6.2 Faza przygotowania wzorców

W tej fazie algorytmu na wejściu mamy zbiór wzorców, a na wyjściu chcemy mieć drzewo trie dla tego zestawu wzorców oraz obliczone dla niego funkcje out, shift1, shift2 oraz char. Chcielibyśmy to wszystko zrobić w czasie liniowym względem sumy długości wzorców. Oczywiście konstrukcja trie w takim czasie jest natychmiastowo. Po prostu dodajemy kolejne słowa budując jednocześnie drzewo. Z definicji funkcji char i out również wynika prostota ich obliczania wprost z definicji. Kłopotliwe mogą się wydawać na pierwszy rzut oka kluczowe funkcje shift1 oraz shift2.

Jednak cały problem mamy już rozwiązany za pomocą algorytmu Aho-Corasick. Mianowicie rozważmy pewną funkcje f zdefiniowaną na wierzchołkach drzewa trie T za pomocą formuły $f(v') = v$, jeżeli $w(v)$ jest maksymalnym sufiksem właściwym $w(v')$ w drzewie T. Funkcja ta idealnie współgra z funkcją fail z automatu Aho-Corasick, więc to co musimy zrobić, aby ją obliczyć, to odpalić fazę konstrukcji z algorytmu Aho-Corasick i w czasie jednego przejścia algorytmem bfs po drzewie trie obliczyć funkcję fail. Okazuje się bowiem, że funkcja $set1'$ odwrotna do funkcji f, czyli $set1'(v) = \{v' : f(v') = v\}$, to dokładnie podzbiór szukanego przez nas zbioru $set1$, który potrzebujemy do obliczenia funkcji przesunięć. Ponadto zawiera on wierzchołki $v' \in set1(v)$ takie, że $d(v') - d(v)$ jest minimalne. Wynika stąd, że funkcję shift1 można istotnie obliczyć w czasie liniowym. Analogicznie obliczamy funkcję shift2 rozważając zbiór $set2'(v) = \{v' : v' \in set1'(v), out(v') \neq \emptyset\}$.

16.7 Złożoność obliczeniowa algorytmu

Faza konstrukcji drzewa trie opisana powyżej jest wykonywana w czasie liniowym względem sumy długości wszystkich wzorców. Przeszukiwanie tekstu pesymistycznie może mieć czas liniowy względem iloczynu długości tekstu, w którym wyszukujemy, z długością najdłuższego wzorca, ale dzięki temu, że mamy funkcje przesunięć, bardzo często nie skanujemy całego tekstu znak po znaku, tylko jego części. Poprzez silną analogię do algorytmu Boyera-Moore'a stwierdzamy, że algorytm Commentz-Walter, działa średnio podobnie jak algorytm Boyera-Moore'a, czyli bardzo szybko. Pesymistycznie jednak pamiętać należy, że może mieć złożoność $O(nm)$, gdzie n, to długość tekstu, a m to długość najdłuższego wzorca.

16.8 Blum et al., Liniowa aproksymacja najkrótszego wspólnego nadsłowa, algorytm 3-przybliżony

Mając na wejściu słowa $S = \{s_1, ..., s_m\}$, takie, że żadne z nich nie zawiera żadnego innego w całości, możemy poszukać wspólnego nadsłowo s dla danego zbioru S.

Najprostszym przykładem może być słowo $s = s_1 s_2 ... s_m$. Jednakże, tak uzyskane nadsłowo będzie niepotrzebnie długie.

Jako, że problem ten ma duże znaczenie praktyczne (w sekwencjonowaniu DNA) naturalnym jest próba znalezienia najkrótszego wspólnego nadsłowa.

Oczywistym podejściem jest następujący algorytm zachłanny. Po kolei łącz słowa mające największą część wspólną nachodzącą na siebie określaną jako *overlap*. Rób tak, aż do uzyskania jednego słowa. Postawiona jest hipoteza, że takie podejście ma współczynnik aproksymacji równy 2, ale przed opublikowaniem pracy przez Blum et al. nie było wiadome, czy taki algorytm jest w ogóle liniową aproksymacją. Okazuje się, że ten algorytm (określać będziemy go poprzez **GREEDY**) ma współczynnik aproksymacji co najwyżej 4.

Ponadto Blum et al. zaprezentowali alternatywny algorytm, nazywany **TGREEDY**, dla którego udowodnili, że jest on 3-aproksymacyjny. Sam problem znalezienia najkrótszego wspólnego nadsłowa jest NP trudny.

W pierwszej sekcji przedstawimy podstawowe definicje i notacje, a w kolejnych algorytm **MGREEDY**, a w następnej sekcji algorytm **TGREEDY** wykorzystujący algorytm **MGREEDY** by uzyskać pożądany współczynnik aproksymacji. W ostatniej sekcji postaramy się udzielić odpowiedzi na pytanie, który z tych dwóch algorytmów jest lepszy?

16.9 Podstawowe definicje

Mając dwa niekoniecznie różne słowa $s = uv, t = vw$, takie, że u, w są niepuste, a v maksymalnej długości, mianem *overlap*'u słów s, t określamy v. Długośc overlap'u słów s, t oznaczamy jako $ov(s,t) = |v|$. Zauważmy, że dla $s = t$ overlap jest maksymalnym właściwym prefikso-sufiksem.

Ponadto u nazwiemy *prefiksem słowa s względem słowa t* i oznaczymy poprzez $pref(s,t) = u$. *Odległością między s, a t* nazwiemy $d(s,t) = |pref(s,t)|$.

Wtedy słowo postaci $uvw = pref(s,t)t$ jest najkrótszym nadsłowem s i t o długości $d(s,t)+|t| = |s|+|t|-ov(s,t)$. Takie słowo będziemy też określali jako *merge słów s i t*. Dla słów $s_i, s_j \in S$ będziemy używali $pref(i,j), d(i,j), ov(i,j)$ zamiast $pref(s_i,s_j), d(s_i,s_j), ov(s_i,s_j)$.

Mając słowa $s_{i_1}, s_{i_2}, ..., s_{i_r}$ zdefiniujemy $s = \langle s_{i_1}, ..., s_{i_r} \rangle$ jako
$s = pref(i_1,i_2)...pref(i_{r-1},i_r)s_{i_r}$, czyli najkrótsze słowo, w którym słowa $s_{i_1}, ..., s_{i_r}$ pojawiają się dokładnie w tej kolejności.

Określmy s_{i_1} poprzez $first(s)$ oraz s_{i_r} poprzez $last(s)$.

W trakcie algorytmu **GREEDY**, a także **MGREEDY** następujący niezmiennik jest zachowany.

Lemat 16.9. *W dowolnym momencie algorytmu **GREEDY** w aktualnie przetwarzanym zbiorze słów dla każdych dwóch różnych słów s, t z tego zbioru jest, że $first(s)$ ani $last(s)$ nie jest podsłowom słowa t.*

Dowód. Udowodnimy lemat dla $first(s)$, dowód dla $last(s)$ jest zupełnie analogiczny.

Początkowo $first(s) = last(s) = s$ dla każdego słowa s w aktualnym zbiorze. Załóżmy, że niezmiennik jest w pewnym momencie naruszony poprzez złączenie dwóch słów t_1, t_2 w słowo $t = \langle t_1, t_2 \rangle$, czyli, że t ma $first(s)$ jako podsłowo. Wtedy wiemy, że $t = ufirst(s)v$ dla jakichś u, v. Skoro $first(s)$ nie może być podsłowem t_1 ani t_2 (założenie o zbiorze S), to $|first(s)| > ov(t_1,t_2)$ oraz $|u| < d(t_1,t_2)$. Czyli, $ov(t_1,s) > ov(t_1,t_2)$, co daje nam sprzeczność z faktem, że $ov(t_1,t_2)$ jest największym. □

Lemat ten mówi nam, że algorytm wybierając dwa słowa s, t o największym $ov(s,t)$ wybiera de facto słowa o największym $ov(first(s), last(t))$. W wyniku takiego łączenia powstaje słowo $\langle first(s), ..., last(s), first(t), ..., last(t) \rangle$. Możemy, więc powiedzieć, że **GREEDY** porządkuje słowa w ten sposób (dla każdej pary słów porównując ich overlap'y), a następnie znajduje najkrótsze wspólne nadsłowo zawierające te słowa w tej kolejności.

Wprowadzimy teraz pojęcie grafu odległości, który opisuje nam naszą instancję problemu. Mianowicie, $G_S = (V, E)$, gdzie wierzchołkom przypisane są poszczególne słowa z S, a krawędziom będą przypisane słowa $pref(i,j)$ o wadze $|pref(i,j)| = d(i,j)$.

Możemy na takim grafie łatwo stwierdzić, że jest zachowana nierówność: $CYC(G_S) \leq TSP(G_S) \leq OPT(S) - ov(last(s), first(s)) \leq OPT(S)$.

Dzięki temu można pokazać, że znajdując minmialne pokrycie cyklowe ($CYC(G_S)$) możemy skonstruować wspólne nadsłowo o długości co najwyżej $4 \cdot OPT(S)$ poprzez konkatenacje słów wynikających z poszczególnych cyklów w pokryciu cyklowym. Tak opisany algorytm nazywamy **CONCATCYCLES**.

16.10 Algorytm *MGREEDY*

Możemy teraz opisać algorytm **MGREEDY**.

1. Niech S będzie zbiorem słów początkowych, a T zbiorem pustym.
2. Dopóki S jest niepusty, wybierz s, t z S maksymalizujące $ov(s,t)$.
   Dla $s \neq t$, niech $S := (S \setminus \{s,t\}) \cup \{\langle s,t\rangle\}, T := T$.
   Dla $s = t$, niech $S := S \setminus \{s\}$, a $T := T \cup \{s\}$.
3. Jako wspólne nadsłowo zwróc konkatenację słów ze zbioru T.

Rozważmy teraz graf G'_S, analogiczny jak G_S, ale dla każdej krawędzi mamy jako wagę $ov(i,j)$ zamiast $d(i,j)$. Zauważmy, że dla dowonlego pokrycia cyklowego grafu G_S oraz tego samego pokrycia cyklowego grafu G'_S mamy, że suma wag z pokrycia cyklowego z tych dwóch grafów równa się sumie długości słów w S. Jest tak ponieważ $\forall s,t : ov(s,t) + d(s,t) = |s|$. Czyli maksymalne pokrycie cyklowe grafu G'_S, będzie tożsame z minimalnym pokryciem cyklowym grafu G_S. Okazuje się, że **MGREEDY** łącząc dwa słowa $s \neq t \in S$ w jedno, de facto wybiera krawędź w grafie G'_S między $s_\alpha := last(s)$, a $s_\beta := first(t)$ takie, że $ov(\alpha,\beta)$ jest maksymalne wśród wszystkich wolnych $ov(i,j)$.

Wystarczy teraz tylko pokazać, że taka zachłanna strategia dla tego rodzaju grafu konstruuje pokrycie cyklowe o maksymalnej wadze, co będzie oznaczało, że **MGREEDY** jest niegorszy niż **CONCATCYCLES**, czyli aproksymuje ze współczynnikiem 4. To też pokazali w swojej oryginalnej pracy Blum et al.

16.11 Algorytm *TGREEDY*

Algorytm **TGREEDY** najpierw wykonuje kroki od 1 do 2 z algorytmu **MGREEDY**, a następnie na uzyskanym zbiorze T, który odpowiada maksymalnemu pokryciu cyklowemu grafu G'_S, wykonuje algorytm **GREEDY**. Aby pokazać, że **TGREEDY** ma pożądany współczynnik aproksymacj potrzebujemy wprowadzić parę dodatkowych definicji i lematów.

Definicja 16.10. Niech $\equiv$ będzie relacją równoważności dla słów, taką, że $s \equiv t \Leftrightarrow \exists u,v : s = uv, t = vu$. Wtedy, jeżeli c to cykl na wierzchołkach (słowach) $i_0, ..., i_r$ to $strings(c)$ jest klasą równoważności $[pref(i_0,i_1)pref(i_1,i_2)...pref(i_{r-1},i_0)]_\equiv$.
Ponadto, $strings(c,i_k)$ zdefiniujemy jako $pref(i_k,i_{k+1})...pref(i_{k-1},i_k)$, gdzie indeksowanie jest modulo r. Oczywiście, $strings(c,i_k) \in strings(c)$.

Definicja 16.11. Niech $w(c)$, będzie wagą cyklu, czyli $w(c) = |s|, s \in strings(c)$. Dla wygody będziemy pisać, że $s \in c$ zamiast $s \in strings(c)$.

Lemat 16.12. *Niech c_1, c_2 będą dwoma cyklami w minimalnym pokryciu cyklowym C, z $s_1 \in c_1$ oraz $s_2 \in c_2$ Wtedy, $ov(s_1,s_2) \leq w(c_1) + w(c_2)$.*

Lemat 16.13. *Nadsłowo $\langle s_{i_0}, ..., s_{i_{r-1}}\rangle$ jest podsłowem $strings(c,i_0)s_{i_0}$.*

Twierdzenie 16.14. ***TGREEDY*** *dla zbioru $S = \{s_1, ..., s_m\}$ zwraca nadsłowo s nad S o długości co najwyżej $3 \cdot OPT(S)$.*

Dowód. Niech $n := OPT(S)$, pokażemy, że $|s| \leq 3n$. Niech T będzie zbiorem samonachodzących się słów uzyskanych przez **MGREEDY** na S, a C będzie minimalnym pokryciem cyklowym skonstruowanym przez **MGREEDY**. Dla każdego $x \in T$, niech c_x oznacza cykl w C odpowiadający słowie x oraz niech $w_x = w(c_x)$ będzie jego wagą. Dla każdego zbioru słów R niech $||R|| = \sum_{x \in R} |x|$ będzie sumaryczną długością wszystkich słów w zbiorze R. Niech $w = \sum_{x \in T} w_x$. Skoro, $CYC(G_S) \leq TSP(G_S) \leq OPT(S)$, wiemy że $w \leq n$.

Dzięki lematowi lemat 16.12, wiemy, że kompresja uzyskana w najkrótszym nadsłowie nad T jest mniejsza niż $2w$, i.e $||T|| - n_T \leq 2w$, gdzie $n_T := OPT(T)$. Ponadto, znanym rezultatem jest (Tarhio i Ukkonen, 1988), że algorytm zachłanny osiąga kompresję ze współczynnikiem 2. Łącząc te dwa wyniki dostajemy, że: $||T|| - |s| \geq (||T|| - n_T)/2 = ||T|| - n_T - (||T|| - n_T)/2 \geq ||T|| - n_T - w$.

Z tego wynika, że $|s| \leq n_T + w$.

Dla każdego $x \in T$, niech s_{i_x} będzie słowem z cyklu c_x, który jest prefiksem x. Niech $S' = \{s_{i_x} | x \in T\}, n' = OPT(S'), S'' = \{strings(c_x, i_x)s_{i_x} | x \in T\}, n'' = OPT(S'')$.

Z lematu lemat 16.13 wynika, że nadsłowo S'' jest także nadsłowem T, więc $n_T \leq n''$. Dla dowolnej permutacji π prawdą jest, że $|S''_\pi| \leq |S'_\pi| + \sum_{x \in T} w_x$, więc $n'' \leq n' + w$, gdzie S'_π, S''_π jest nadsłowem uzyskanym poprzez połączenie słów z S', S'' w kolejności narzuconej przez π.

Wiemy, że $S' \subset S''$, a więc $n' \leq n$. Możemy teraz połączyć te wyniki i dostać, że

$$n_T \leq n'' \leq n' + w \leq n + w$$

Łącząc to z faktem, że $|s| \leq n_T + w$ dostajemy, że $|s| \leq n + 2w \leq 3n$. □

16.12 Uwagi końcowe

Blum et al. pokazali dodatkowo, że **GREEDY** osiąga współczynnik aproksymacji równy 4, ale hipoteza czy ten współczynnik można obniżyć do 2 jest wciąż otwarta. Możnaby się pokusić wysunąć hipotezę, że **TGREEDY** jest lepszy niż **GREEDY**. Tak nie jest. Są przykłady zbioru S dla których zarówno jeden jak i drugi wypada lepiej.

Dla zbioru $S = \{c(ab)^k, (ab)^{k+1}, (ba)^k c\}$ **TGREEDY** wygeneruje słowo $c(ab)^k ac(ab)^{k+1}a$ o długości $n = 4k+6$, podczas gdy **GREEDY** zwróci słowo optymalne $c(ab)^{k+1}ac$ o długośći $n = 2k + 5$.

Z kolei dla zbioru $S = \{cab^k, ab^k ab^k a, b^k dab^{k-1}\}$ **TGREEDY** wygeneruje słowo $cab^k dab^k ab^k a$ o długości $3k+6$, podczas gdy **GREEDY** wygeneruje słowo $cab^k ab^k ab^k dab^{k-1}$ długości $4k+5$.

16.13 Fast practical multi-pattern matching

Fast practical multi-pattern matching to algorytm tekstowy służący do wyszukiwania w zadanym tekście wystąpień słów z podanego zbioru wzorców.

Algorytm jest ulepszeniem algorytmu Commentz-Waltera, czerpiąc z niego kilka istotnych obserwacji takich jak przeskakiwanie nieinteresujących obszarów tekstu oraz wykorzystanie automatu Aho-Corasick. Dodatkowym elementem zwiększającym efektywność jest wykorzystanie automatu sufiksowego zwanego dalej *DAWG*.

Algorytm, który jest tematem pracy, dla tekstu o długości n ma pesymistyczny czas działania $O(n)$. W przypadku średnim, gdzie najkrótszy spośród wzorców ma długość m a suma ich długości jest wielomianowa w zależności od m oraz prawdopodobieństwo wystąpienia litery z alfabetu (conajmniej dwu-elementowego) jest jednostajne i niezależne, złożoność wynosi $O(\frac{n}{m} * \log m)$, co jest równocześnie dolnym ograniczeniem dla tego problemu. Dodatkowo dla odpowienio dużego m algorytm osiąga dobre rezultaty w praktyce.

16.13.1 Idea

Idea algorytmu oparta jest o dwie fazy zwane *PROCESS1* oraz *PROCESS2*. Pierwsza z nich skanuje test od lewej do prawej przesuwając potencjną pozycję zakończenia wzorca co najmniej o $\frac{m}{2}$. W tej fazie bazuje on na zmiennej γ, która spełnia niezmiennik $\gamma =$ najdłuży suffix od początku tekstu do pozycji i, który jest jednocześnie prefiksem jakiegoś wzorca. Jeśli długość $\gamma \leq \frac{m}{2}$ kończymy pierwszą fazę algorytmu i przechodzimy do fazy drugiej. W implementacji będziemy utorzsamiać γ ze stanem w AC.

Faza druga wykonuje skanowanie tekstu od tyłu od pozycji $i + SHIFT$ na odległość równą m, gdzie $SHIFT = m - |\gamma|$.

Dodatkowo podczas fazy pierwszej, gdy wyznaczaczamy kolejne wartości γ, będziemy używali zbudowanego wcześniej automatu $Aho-Corasick$, aby robić to bardziej efektywnie. Natomiast faza druga w skanowaniu do tyłu używa $DAWGa$, który został zbudowany przy użyciu zbioru odwróconych wzorców.

16.13.2 Struktura dla wzorców

W celu przechowywania w pamięci zadanego zestawu wzorców należy przed przystąpieniem do wyszukiwania zbudować automat $Aho - Corasick$ oraz $DAWG$. Pozwoli to szybko odpowiadać na wymagane zapytania.

```
\caption{Fast practical multi-pattern matching, preprocessing}
\begin{algorithmic}[1]
\Procedure{MULTI-PATTERN-MATCHING-BUILD}{$PATTERNS$} \Comment{lista wzorców}
\State $acm = build\_aho\_corasick\_machine(patterns)$ \Comment{budujemy standardowy aut
\State $reversed\_patterns = reverse(patterns)$ \Comment{odwracamy wzorce}
\State $dawg = build\_dawg(reversed\_patterns)$ \Comment{budujemy standardowy DAWG}
\State \textbf{return} $(acm, dawg)$ \Comment{budujemy standardowy DAWG}
\EndProcedure
\end{algorithmic}
```

Definicja 16.15. Automat $Aho - Corasick$ powinien udostępniać:

- $step(node, a)$ = stan AC po przetworzeniu znaku a zaczynając w $node$.
- $traverse(node, s)$ = stan AC po przetworzeniu słowa s zaczynając w $node$.
- wartość $depth$ oznaczającą głębokość w drzewie dla każdego wierzchołka.
- listę $final$ oznaczającą jakie wzorce powinny zostać zaakceptowane dla danego wierzchołka.

Definicja 16.16. DAWG powinien udostępniać:

- $traverse(node, s)$ = stan $DAWG$ po przetworzeniu słowa s zaczynając w $node$.

Definicja 16.17. Potrzebne zapytania:

- $NEXT1(\gamma, a)$ = najdłuższy sufiks słowa γa będący jednocześnie prefiksem jakiegoś wzorca.
- $NEXT2(j, i)$ = najdłuższy sufiks podsłowa $text[j...i]$ będący jednocześnie prefiksem jakiegoś wzorca.

```
\caption{Fast practical multi-pattern matching, NEXT1}
\begin{algorithmic}[1]
\Procedure{NEXT1}{$\gamma, character$}
\State \textbf{return} $ acm.step(gamma, char)$ \Comment{wykonujemy jeden krok w AC}
\EndProcedure
\end{algorithmic}

\caption{Fast practical multi-pattern matching, NEXT2}
\begin{algorithmic}[1]
\Procedure{NEXT2}{$j, i$} \Comment{indeksy zakresu tekstu}
\State $\gamma_{new} = dawg.traverse(dawg.root, j, i)$ \Comment{przechodzimy DAWG skanuj
\State \textbf{return} $ acm.traverse(acm.root, \gamma_{new})$ \Comment{przechodzimy AC
\EndProcedure
\end{algorithmic}
```

16.13.3 Wyszukiwanie wzorca

Wyszukiwanie wzorca działa na zasadzie przesuwania zakresu tekstu bazując na AC oraz DAWG. Wykonujemy naprzemiennie fazę pierwszą i drugą wspomniane na początku. Pierwsza z nich skanuje test od lewej do prawej, aby przesunąć potencjalną pozycję zakończenia wzorca co najmniej o m2. Przesunięcie opieramy na zmiennej γ, która spełnia niezmiennik γ= najdłuższy suffix od początku tekstu do pozycji i, który jest jednocześnie prefiksem jakiegoś wzorca. Jeśli długość γ, kończymy pierwszą fazę algorytmu i przechodzimy do fazy drugiej. W implementacji będziemy utożsamiać γ ze stanem w AC, a głębokość tego stanu z długością dopasowanego prefiksu wzorca. Tak długo jak nie osiągnęliśmy wymaganej długości prefiksu, skanujemy kolejne znaki tekstu przechodząc jednocześnie po AC. Jeśli w tej

fazie uda nam się dopasować jakiś wzorzec (γ jest stanem akceptującym automatu AC), to zwracamy go poprzez *yield*.

Faza druga wykonuje skanowanie tekstu od tyłu od pozycji $i + SHIFT$ na odległość równą m, gdzie $SHIFT = m - |\gamma|$. Skanowanie odbywa się na zasadzie skanowania tekstu od tyłu jednocześnie symulując przejścia w $DAWG$. Tym sposobem dopasujemy najdłuższy sufiks spośród sufiksów wszystkich wzorców. Dla znalezionego sufiksu musimy jeszcze zaktualizować naszą pozycję w AC i możemy powrócić do fazy pierwszej.

```
\caption{Fast practical multi-pattern matching, faza wyszukiwania}
\begin{algorithmic}[1]
\Procedure{MULTI-PATTERN-MATCHING}{$text, n, S$} \Comment{tekst, długość, struktura}
\State $i := 0$ \Comment{pozycja w tekście}
\State $\gamma := S.acm.root$ \Comment{równoważnie $\gamma := S.acm.root$}

\While{ $True$ } \Comment{faza skanowania, zachowany niezmiennik $\Gamma$}
    \While {$\gamma{.depth} \geq \frac{m}{2}$} \Comment{szukamy odpowiednio długiego pre
        \If{$\gamma$ is final state}
            \State \textbf{yield} $(w, i)$ dla każdego $w \in \gamma{.out}$
        \EndIf
        \State $i = i + 1$
        \If{$i \geq n$}
            \State \textbf{return}
        \EndIf
        \State $\gamma = NEXT1(\gamma, text[i])$  \Comment{wykonujemy krok w AC}
    \EndWhile
    \State $crit\_pos := i - \gamma{.depth} + 1$ \Comment{pierwsza pozycja, której nie w
    \State $SHIFT := m - \gamma{.depth}$
    \State $i = i + SHIFT$ \Comment{przeskakujemy fragment tekstu}
    \If{$i \geq n$}
        \State \textbf{return}
    \EndIf
    \State $\gamma = NEXT2(crit\_pos, i)$ \Comment{szukamy najdłuższego sufiksu w DAWG}
\EndWhile
\EndProcedure
\end{algorithmic}
```

Wynikiem tej częsci jest lista par postaci (w, i), gdzie w to któryś z szukanych wzorców, natomiast i to pozycja w tekście, na której został znaleziony wzorzec w.

16.13.4 Złożoność obliczeniowa algorytmu

Lemat 16.18. *Zakładjąc zbudowaną strukturę AC oraz DAWG:*

- $NEXT1(\gamma, a)$ *działa w czasie* $O(1)$.
  Wykonanie funkcji $NEXT1$ *sprowadza się do wykonania jednego kroku w automacie* AC. □
- $NEXT2(j, i)$ *działa w czasie* $O(min(i - j, i - crit\_pos))$.
  Wykonanie funkcji $NEXT2$ *wymaga przeskanowania tekstu od pozycji* i *do* j *idąc od prawej. Równoczesne symulowanie DAWG wykona taką samą liczbę kroków.* □

Twierdzenie 16.19. *Algorytm w sumie porównuje co najwyżej* $2n$ *znaków.*

Dowód. Zauważmy, że znak na danej pozycji może być przetworzony przez $PROCESS1$ co najwyżej raz. Jeśli tak się stało, to następnie zostanie tylko raz przetworzony przez $PROCESS2$ i już nigdy ponownie do niego nie wrócimy. Jeśli natomiast $PROCESS2$ odwiedzi dany znak dwa razy, mamy wtedy pewność, że dany znak nie został przeczytany przez $PROCESS1$ w ogóle. □ □

Niech σ to prawdopodobieństwo jednostajengo wystapienia symbolu w tekście. Wtedy:

Lemat 16.20. $P(\Delta(i) > (3k + 1) * log_\sigma m) \leq \frac{1}{m^{k+1}}$, *gdzie* P *to prawdopodobieństwo.*

Twierdzenie 16.21. *Zakładając* $M \leq m^k$ *algorytm wykonuje* $O(\frac{n}{m} * log_\sigma m)$ *porównań.*

Dowód. Podczas działania algorytmu wykonywanych jest co najwyżej $O(\frac{n}{m})$ przesunięć. Wystarczy jeszcze udowodnić, że średnio pierwsza i druga faza algorytmu odwiedza zaledwie logarytmiczną liczbę znaków. Niech $K = (2k + 1) * log_\sigma m$. $PROCESS1$ zakończy się z dużym prawdopodobieństwem po K krokach. Jeśli nie, zrobi on m^k kroków z prawdopodobieństwem $1/m^{k+1}$. Dostajemy więc średnią złożoność $O(K)$ co jest logarytmiczne. Sytuacja z $PROCESS2$ jest analogiczna. □ □

Bibliografia

Abouelhoda, Mohamed Ibrahim, Stefan Kurtz i Enno Ohlebusch (2004). „Replacing suffix trees with enhanced suffix arrays". W: *Journal of Discrete Algorithms* 2.1, s. 53–86.

Adamczyk, Zbigniew i Wojciech Rytter (2013). „A note on a simple computation of the maximal suffix of a string". W: *Journal of Discrete Algorithms* 20, s. 61–64.

Aho, Alfred i Margaret Corasick (1975). „Efficient string matching: an aid to bibliographic search". W: *Communications of the ACM* 18.6, s. 333–340.

Apostolico, Alberto i Zvi Galil (1997). *Pattern Matching Algorithms*. Oxford University Press.

Apostolico, Alberto i Raffaele Giancarlo (1986). „The Boyer–Moore–Galil string searching strategies revisited". W: *SIAM Journal on Computing* 15.1, s. 98–105.

Armen, Chris i Clifford Stein (1996). „A 2 2/3-approximation algorithm for the shortest superstring problem". W: *Annual Symposium on Combinatorial Pattern Matching*. Springer, s. 87–101.

Backurs, Arturs i Piotr Indyk (2015). „Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)". W: *Proceedings of the 47th Annual ACM Symposium on Theory of Computing*, s. 51–58.

Berstel, Jean i Juhani Karhumäki (2003). „Combinatorics on Words – A Tutorial". W: *Bulletin of the EATCS* 79, s. 178–228.

Blum, Avrim, Tao Jiang, Ming Li, John Tromp i Mihalis Yannakakis (1994). „Linear approximation of shortest superstrings". W: *Journal of the ACM* 41.4, s. 630–647.

Braquelaire, Tristan, Marie Gasparoux, Mathieu Raffinot i Raluca Uricaru (2018). „On improving the approximation ratio of the r-shortest common superstring problem". W: *arXiv preprint arXiv:1805.00060*.

Breslauer, Dany (1996). „Saving comparisons in the Crochemore-Perrin string-matching algorithm". W: *Theoretical Computer Science* 158.1-2, s. 177–192.

Breslauer, Dany i Giuseppe Italiano (2013). „Near real-time suffix tree construction via the fringe marked ancestor problem". W: *Journal of Discrete Algorithms* 18, s. 32–48.

Bringmann, Karl i Marvin Künnemann (2015). „Quadratic conditional lower bounds for string problems and dynamic time warping". W: *Proceedings of 56th Annual Symposium on Foundations of Computer Science*, s. 79–97.

Chang, William i Eugene Lawler (1990). „Approximate string matching in sublinear expected time". W: *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, s. 116–124.

Clifford, Peter i Raphaël Clifford (2007). „Self-normalised distance with don't cares". W: *Annual Symposium on Combinatorial Pattern Matching*, s. 63–70.

Cole, Richard i Ramesh Hariharan (2002). „Approximate string matching: A simpler faster algorithm". W: *SIAM Journal on Computing* 31.6, s. 1761–1782.

Cole, Richard, Tsvi Kopelowitz i Moshe Lewenstein (2006). „Suffix trays and suffix trists: structures for faster text indexing". W: *International Colloquium on Automata, Languages, and Programming*. Springer, s. 358–369.

Commentz-Walter, Beate (1979). „A string matching algorithm fast on the average". W: *International Colloquium on Automata, Languages, and Programming*. Springer, s. 118–132.

Crochemore, Maxime, Artur Czumaj, Leszek Gąsieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski i Wojciech Rytter (1994). „Speeding up two string-matching algorithms". W: *Algorithmica* 12.4-5, s. 247–267.

Crochemore, Maxime, Artur Czumaj, Leszek Gąsieniec, Thierry Lecroq, Wojciech Plandowski i Wojciech Rytter (1999). „Fast practical multi-pattern matching". W: *Information Processing Letters* 71.3-4, s. 107–113.

Crochemore, Maxime, Christophe Hancart i Thierry Lecroq (2007). *Algorithms on Strings*. Cambridge University Press.

Crochemore, Maxime i Dominique Perrin (1991). „Two-way string-matching". W: *Journal of the ACM* 38.3, s. 650–674.

Crochemore, Maxime i Wojciech Rytter (1994). *Text Algorithms*. Oxford University Press.

— (2002). *Jewels of Stringology*. World Scientific.

Farach, Martin (1997). „Optimal suffix tree construction with large alphabets". W: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, s. 137–143.

Ferragina, Paolo i Giovanni Manzini (2000). „Opportunistic data structures with applications". W: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, s. 390–398.

Fischer, Michael i Michael Paterson (1974). „String matching and other products". W: *Complexity of Computation* 7, s. 113–125.

Franceschini, Gianni i Torben Hagerup (2011). „Finding the maximum suffix with fewer comparisons". W: *Journal of Discrete Algorithms* 9.3, s. 279–286.

Galil, Zvi i Raffaele Giancarlo (1986). „Improved string matching with k mismatches". W: *ACM SIGACT News* 17.4, s. 52–54.

Galil, Zvi i Kunsoo Park (1990). „An improved algorithm for approximate string matching". W: *SIAM Journal on Computing* 19.6, s. 989–999.

Galil, Zvi i Joel Seiferas (1983). „Time-Space-Optimal String Matching". W: *Journal of Computer and System Sciences* 26, s. 280–294.

Gallant, John, David Maier i James Storer (1980). „On finding minimal length superstrings". W: *Journal of Computer and System Sciences* 20.1, s. 50–58.

Giegerich, Robert i Stefan Kurtz (1997). „From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction". W: *Algorithmica* 19.3, s. 331–353.

Golovnev, Alexander, Alexander Kulikov i Ivan Mihajlin (2013). „Approximating shortest superstring problem using de Bruijn graphs". W: *Annual Symposium on Combinatorial Pattern Matching*, s. 120–129.

Gonnet, Gaston, Ricardo Baeza-Yates i Tim Snider (1992). „New Indices for Text: Pat Trees and Pat Arrays." W: *Information Retrieval: Data Structures & Algorithms* 66, s. 82.

Grossi, Roberto i Fabrizio Luccio (1989). „Simple and efficient string matching with k mismatches". W: *Information Processing Letters* 33.3, s. 113–120.

Gusfield, Dan (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge University Press.

Hirschberg, Daniel (1975). „A linear space algorithm for computing maximal common subsequences". W: *Communications of the ACM* 18.6, s. 341–343.

Itoh, Hideo i Hozumi Tanaka (1999). „An efficient method for in memory construction of suffix arrays". W: *6th International Symposium on String Processing and Information Retrieval*. IEEE, s. 81–88.

Jacquet, Philippe i Wojciech Szpankowski (2015). *Analytic Pattern Matching: from DNA to Twitter*. Cambridge University Press.

Jiang, Tao i Ming Li (1996). „DNA sequencing and string learning". W: *Mathematical Systems Theory* 29.4, s. 387–405.

Kalai, Adam (2002). „Efficient pattern-matching with don't cares". W: *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial i Applied Mathematics, s. 655–656.

Kaplan, Haim, Moshe Lewenstein, Nira Shafrir i Maxim Sviridenko (2005). „Approximation algorithms for asymmetric TSP by decomposing directed regular multigraphs". W: *Journal of the ACM* 52.4, s. 602–626.

Kaplan, Haim i Nira Shafrir (2005). „The greedy algorithm for shortest superstrings". W: *Information Processing Letters* 93.1, s. 13–17.

Kärkkäinen, Juha (2007). „Fast BWT in small space by blockwise suffix sorting". W: *Theoretical Computer Science* 387.3, s. 249–257.

Kärkkäinen, Juha i Simon Puglisi (2010). „Medium-space algorithms for inverse BWT". W: *European Symposium on Algorithms*. Springer, s. 451–462.

Kärkkäinen, Juha i Peter Sanders (2003). „Simple linear work suffix array construction". W: *International Colloquium on Automata, Languages, and Programming*, s. 943–955.

Karp, Richard, Raymond Miller i Arnold Rosenberg (1972). „Rapid identification of repeated patterns in strings, trees and arrays". W: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, s. 125–136.

Karpinski, Marek i Richard Schmied (2013). „Improved inapproximability results for the shortest superstring and related problems". W: *Proceedings of the 19th Computing: The Australasian Theory Symposium*. T. 141, s. 27–36.

Kim, Dong Kyue, Jeong Seop Sim, Heejin Park i Kunsoo Park (2003). „Linear-time construction of suffix arrays". W: *Annual Symposium on Combinatorial Pattern Matching*. Springer, s. 186–199.

Ko, Pang i Srinivas Aluru (2003). „Space efficient linear time construction of suffix arrays". W: *Annual Symposium on Combinatorial Pattern Matching*. Springer, s. 200–210.

Kumar, S. Kiran i C. Pandu Rangan (1987). „A linear space algorithm for the LCS problem". W: *Acta Informatica* 24.3, s. 353–362.

Landau, Gad i Uzi Vishkin (1986). „Efficient string matching with k mismatches". W: *Theoretical Computer Science* 43, s. 239–249.

— (1988). „Fast string matching with k differences". W: *Journal of Computer and System Sciences* 37.1, s. 63–78.

— (1989). „Fast parallel and serial approximate string matching". W: *Journal of Algorithms* 10.2, s. 157–169.

Larsson, Jesper i Kunihiko Sadakane (2007). „Faster suffix sorting". W: *Theoretical Computer Science* 387.3, s. 258–272.

Li, Ming (1990). „Towards a DNA sequencing theory (learning a string)". W: *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*. IEEE, s. 125–134.

Li, Zhize, Jian Li i Hongwei Huo (2018). „Optimal in-place suffix sorting". W: *International Symposium on String Processing and Information Retrieval*, s. 268–284.

Lothaire, Monsieur (2002). *Algebraic Combinatorics on Words*. Cambridge University Press.

— (2005). *Applied Combinatorics on Words*. Cambridge University Press.

Manber, Udi i Gene Myers (1993). „Suffix arrays: a new method for on-line string searches". W: *SIAM Journal on Computing* 22.5, s. 935–948.

Masek, William i Michael Paterson (1980). „A faster algorithm computing string edit distances". W: *Journal of Computer and System Sciences* 20.1, s. 18–31.

McCreight, Edward (1976). „A space-economical suffix tree construction algorithm". W: *Journal of the ACM* 23.2, s. 262–272.
Mucha, Marcin (2013). „Lyndon words and short superstrings". W: *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, s. 958–972.
Myers, Eugene (1986). „An $O(nd)$ difference algorithm and its variations". W: *Algorithmica* 1.1-4, s. 251–266.
Navarro, Gonzalo i Mathieu Raffinot (2002). *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press.
Nong, Ge (2013). „Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets". W: *ACM Transactions on Information Systems* 31.3, s. 1–15.
Nong, Ge i Sen Zhang (2007). „Efficient algorithms for the inverse sort transform". W: *IEEE Transactions on Computers* 56.11, s. 1564–1574.
Nong, Ge, Sen Zhang i Wai Hong Chan (2009). „Linear suffix array construction by almost pure induced-sorting". W: *2009 Data Compression Conference*. IEEE, s. 193–202.
Paluch, Katarzyna (2014). „Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring". W: *arXiv preprint arXiv:1401.3670*.
Räihä, Kari-Jouko i Esko Ukkonen (1981). „The shortest common supersequence problem over binary alphabet is NP-complete". W: *Theoretical computer science* 16.2, s. 187–198.
Raita, Timo (1992). „Tuning the Boyer-Moore-Horspool string searching algorithm". W: *Software: Practice and Experience* 22.10, s. 879–884.
Schoenmeyr, Tor i David Yu Zhang (2005). „FFT-based algorithms for the string matching with mismatches problem". W: *Journal of Algorithms* 57.2, s. 130–139.
Sellers, Peter (1974). „On the theory and computation of evolutionary distances". W: *SIAM Journal on Applied Mathematics* 26.4, s. 787–793.
Seward, Julian (2001). „Space-time tradeoffs in the inverse BW transform". W: *Proceedings DCC 2001. Data Compression Conference*. IEEE, s. 439–448.
Smith, Peter (1991). „Experiments with a very fast substring search algorithm". W: *Software: Practice and Experience* 21.10, s. 1065–1074.
— (1994). „On tuning the Boyer-Moore-Horspool string searching algorithm". W: *Software: Practice and Experience* 24.4, s. 435–436.
Stephen, Graham (1994). *String Searching Algorithms*. World Scientific.
Sweedyk, Elizabeth (2000). „2 1/2-Approximation Algorithm for Shortest Superstring". W: *SIAM Journal on Computing* 29.3, s. 954–986.
Szpankowski, Wojciech (2011). *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons.
Tarhio, Jorma i Esko Ukkonen (1988). „A greedy approximation algorithm for constructing shortest common superstrings". W: *Theoretical computer science* 57.1, s. 131–145.
— (1993). „Approximate Boyer–Moore string matching". W: *SIAM Journal on Computing* 22.2, s. 243–260.
Ukkonen, Esko (1995). „On-line construction of suffix trees". W: *Algorithmica* 14.3, s. 249–260.

Valiente, Gabriel (2009). *Combinatorial pattern matching algorithms in computational biology using Perl and R*. Chapman i Hall/CRC.

Vassilevska, Virginia (2005). „Explicit inapproximability bounds for the shortest superstring problem". W: *International Symposium on Mathematical Foundations of Computer Science*, s. 793–800.

Vazirani, Vijay (2005). *Algorytmy aproksymacyjne*. Wydawnictwa Naukowo-Techniczne.

Wyner, Aaron i Jacob Ziv (1994). „The sliding-window Lempel-Ziv algorithm is asymptotically optimal". W: *Proceedings of the IEEE* 82.6, s. 872–877.

Ziv, Jacob i Abraham Lempel (1977). „A universal algorithm for sequential data compression". W: *IEEE Transactions on Information Theory* 23.3, s. 337–343.

— (1978). „Compression of individual sequences via variable-rate coding". W: *IEEE Transactions on Information Theory* 24.5, s. 530–536.