Simple string matching with k mismatches

W oparciu o „Simple and efficient string matching with k mismatches“

R. Grossi, F. Luccio

Maciej Nemś

Maj 2022

1 Wstęp

Problem *string matching with k mismatches* (SMK) polega na tym, że dla tekstu T oraz dla wzorca W takich, że $|T| = n$, $|W| = m$ znajdujemy wszystkie wystąpienia W w T z co najwyżej $k < m$ różnymi symbolami. W pracy autorzy przedstawiają dwa algorytmy rozwiązujące ten problem. Oba opierają się na liczbie permutacji znaków W w T z co najwyżej k błędami. Pierwszy algorytm działa w czasie $O(n \log |A_W| + rm)$, gdzie r to liczba wystąpień permutacji W w T z k błędami, natomiast A_w, to zbiór znaków w W. Drugi algorytm działa w czasie $O(n \log |A_W| + dk)$, gdzie $d \leqslant r$, to liczba różnych wystąpień permutacji W w T z k błędami. Oba algorytmy najpierw szukają wszystkich miejsc, gdzie w T jest jakaś permutacja W z k błędami, a następnie zwracają z tych miejsc wszystkie te, które mają co najwyżej k błędów.

2 Opis algorytmów

2.1 Szukanie permutacji

Algorytm

Oba algorytmy rozwiązujące problem SMK wymagają najpierw obliczenia wszystkich miejsc w tekście T, gdzie zaczyna się jakaś permutacja wzorca W z co najwyżej k błędami. Do szukania permutacji będziemy potrzebować kolejki FIFO Q, która będzie trzymać symbole z T oraz struktury C, która będzie trzymała liczniki dla każdego znaku występującego w A_T. Licznik Z będzie zliczał liczbę błędów.

Algorytm dla każdego znaku w tekście najpierw go dorzuca do kolejki i dekrementuje licznik. Jeśli $C[znak] < 0$, to znaczy, że wykorzystaliśmy już wszystkie wystąpienia tego znaku w W, więc mamy błąd (inkrementujemy K). Teraz tak długo, aż błędów w kolejce jest ponad k, usuwamy znaki z kolejki i odpowiednio dekrementujemy K. Po pętli, jeśli rozmiar Q równa się m, to zebraliśmy tyle znaków, co długość wzorca, mamy co najwyżej K błędów, czyli wiemy, że to jest permutacja wzorca z co najwyżej K błędami i możemy ją zgłosić jako match. Dodatkowo, aby zawsze na początku pętli rozmiar Q wynosił co najwyżej $m - 1$, to usuwamy pierwszy element.

Algorithm 1 Permutation Matching

```
input T, W, n, m, k
  if n < m then
    end with no matches
  end if
  Preprocessing: For each a ∈ A_W set C[a] to count of a in W; For all
  t ∈ T if t ∉ A_W, substitute it with special symbol % ∉ A_W to get new
  word T̃; Set C[%] = 0
  Z := 0
  for j = 1 to n do
    Q.push(T̃[j])
    C[T̃[j]] = C[T̃[j]] − 1
    if C[T̃j]] < 0 then
      Z = Z + 1
    end if
    // Pop from Q, until there are at most k mismatches
    while Z > k do
      x = Q.pop()
      if C[x] < 0 then
        Z = Z − 1
      end if
      C[X] = C[X] + 1
    end while
    // If Q has length m report occurance, as there are at most k mismat-
    ches
    // also prepare for next iteration by poping from Q
    if Q.size() == m then
      Report occurence in position j − m + 1
      x = Q.pop()
      if C[x] < 0 then
        Z = Z − 1
      end if
      C[X] = C[X] + 1
    end if
  end for
```

Poprawność

Dość łatwo zauważyć, że algorytm ten jest poprawny. W każdej iteracji na koniec sprawdzamy warunki konieczne, by pozycja $j-m+1$ była dopasowaniem. Tzn sprawdzamy liczbę zebranych znaków w kolejce i liczbę błędów. Przesuwamy się w każdym wykonaniu pętli o 1, więc sprawdzimy wszystkie możliwe pozycje.

Złożoność

Złożoność takiego dopasowania to $O(n \log |A_W|)$. Musimy stworzyć słownik, który zawiera A_W oraz % w czasie $O(|A_W| \log |A_W|)$. Do kolejki Q dodamy co najwyżej n razy więc i odejmiemy co najwyżej n razy. Oznacza to, że sumarycznie główna pętla zajmie $O(n \log |A_W|)$ czasu, ponieważ n razy zrobimy operacje liniowe typu dodaj do kolejki, odejmij z kolejki, inkrementuj, dekrementuj, oraz n razy zrobimy operacje logarytmiczne dostępu do słownika C.

2.2 Algorytm pierwszy $O(n \log |A_W| + rm)$

Algorytm

Wykonaj algorytm permutation matching. Następnie dla każdej pozycji uznanej za dopasowanie, oblicz odległość Hamminga w czasie liniowym porównując wzorzec do tekstu od tej pozycji. Zwróć wszystkie pozycje, dla których odległość Hamminga jest $\leqslant k$.

Poprawność

Algorytm ten zwróci nam wszystkie pozycje będące dopasowaniem, ponieważ każde dopasowanie z k błędami jest też dopasowaniem jakiejś permutacji W z k błędami. Algorytm permutation matching zwróci nam wszystkie te pozycje, a następnie zostaną one zwrócone przez cały algorytm, ponieważ ich odległość Hamminga będzie $\leqslant k$. Dodatkowo algorytm ten nie zwróci żadnej pozycji nie będącej dopasowaniem, ponieważ dla takich pozycji odległość Hamminga jest $> k$

Złożoność

Permutation matching zajmuje $O(n \log |A_W|)$. Sprawdzenie każdego dopasowania zajmuje $O(m)$ czasu (sprawdzanie odległości Hamminga). Takich dopasowań jest $r < n$. Algorytm ten ma więc złożoność $O(n \log |A_W| + rm)$.

2.3 Algorytm drugi $O(n \log |A_W| + dk)$

2.3.1 Algorytm

Wykonaj algorytm permutation matching i oznacz wszystkie dopasowane miejsca w $\tilde{T}$.

Następnie zbuduj drzewo sufiksowe dla napisu $W@\tilde{T}$, gdzie @ jest separatorem, oraz $@ \notin A_W \cup \{\%\}$. Dla każdego dopasowanego miejsca j w $\tilde{T}$ oznacz liść odpowiadający za sufiks $W@\tilde{T}$, który zaczyna się na pozycji $|\tilde{T}| - j$ od końca (czyli odpowiednik tego samego sufiksu, tylko w połączonym słowie). Dodatkowo oznacz liść odpowiadający za całe słowo $W@\tilde{T}$.

Przeglądaj drzewo sufiksowe w kolejności preorder, aż napotkasz wierzchołek u taki, że odpowiada za prefiks długości $\geqslant m$, natomiast rodzic u odpowiada za prefiks długości $< m$. Następnie znajdź dowolny liść w poddrzewie u. Oznaczmy go x. Zauważmy, że wszystkie liście w tym poddrzewie mają wspólny prefiks długości $\geqslant m$. Więc jeśli jeden z tych liści jest oznaczony, to wszystkie są oznaczone.Jeśli prefiks długości m liścia x na odległość Hamminga $\leqslant k$, to wszystkie liście poddrzewa u mają prefiksy długości m z odległością Hamminga $\leqslant k$ (mają wspólny prefiks długości m). Wszystkie są więc dopasowaniem (oprócz wierzchołka odpowiadającego za $W@\tilde{T}$).

Teraz zdefiniujmy funkcję $MISMATCH(j, m, k, lcp)$, która dla

- indeksu j w $W@\tilde{T}$
- m, k z problemu SMK
- lcp będącego strukturą odpowiadającą na problem longest common prefix dla drzewa sufiksowego $W@\tilde{T}$ w czasie stałym. $lcp.query(i, j)$ zwraca długość najdłuższego wspólnego prefiksu dla sufiksów $W@\tilde{T}$ zaczynających się na pozycjach i oraz j

zwraca **true** wtw, gdy odległość Hamminga dla podsłów zaczynających się na pozycjach 1 oraz j w $W@\tilde{T}$ wynosi $\leqslant k$. Mając taki algorytm, dla każdego znalezionego poddrzewa u, po znalezieniu liścia x, sprawdzamy, czy x jest oznaczony, oraz czy $MISMATCH(j_x, m, k, lcp)$ zwraca **true**. Jeśli tak, to znaczy, że wszystkie liście w poddrzewie u (oprócz liścia odpowiadającego całemu słowu) są dopasowaniami W z k błędami, więc możemy zwrócić ich indeksy w $\tilde{T}$.

Algorithm 2 MISMATCH

input j, m, k, lcp
  $w = 1$ // represents current index in pattern part of $W@\tilde{T}$
  $t = j$ // represents current index in text part of $W@\tilde{T}$
  $c = 0$ // counter for mismatches
  // until there are more than k mistakes, or index in pattern is $> m$
  // we find longest common prefix and update indexes for pattern and text
  while $w \leqslant m$ **and** $c \leqslant k$ **do**
    $q := lcp.query(w, t)$
    if $w + q \leqslant m$ **then**
      // LCP ended before end of pattern, we should increment mistake counter
      $c = c + 1$
    end if
    w = w + q + 1
    t = t + q + 1
  end while
  return $c \leqslant k$

Poprawność

Dlaczego oznaczamy $W@\tilde{T}$? Zauważmy, że to co robi ten algorytm, to szukanie oznaczonych liści, które mają prefiks długości m z odpowiednią odległością Hamminga. Jako, że $W@\tilde{T}$ ma prefiks długości m z odległością Hamminga 0 (część W), to w drzewie sufiksowym może wystąpić w tym samym poddrzewie, co faktyczne liście, które powinniśmy zliczyć. Ten prefiks nie jest jednak dopasowaniem, ponieważ nie leży w całości w T. Pozostałe prefiksy, które są „nielegalne“ (nie leżą w całości w T) nigdy nie wylądują w tym samym poddrzewie, ponieważ mają znak @, który nie występuje w T. Dlatego z „nielegalnych“ liści musimy oznaczyć liść odpowiadający $W@\tilde{T}$.

Zauważmy, że dla każdego poddrzewa u, które rozpatrywaliśmy w algorytmie, albo wszystkie liście są oznaczone, albo wszystkie liście nie są oznaczone (mają ten sam prefiks długości m, więc dla wszystkich z nich permutation matching musiał znaleźć to samo). Przeglądając drzewo w kolejności preorder znajdziemy wszystkie chciane u (odpowiada prefiksowi długości $\geqslant m$). Dodatkowo, łatwo zauważyć, że funkcja MISMATCH poprawnie oblicza, czy odległość Hamminga takiego wspólnego prefiksu jest $\leqslant k$. Jeśli więc dla każdego takiego u sprawdzimy dowolny liść, czy został oznaczony i wartość funkcji MISMATCH, to znajdziemy wszystkie indeksy dopasowań z co najwyżej k błędami.

Złożoność

Konstrukcja struktury drzewa sufiksowego może być zrobiona w czasie $O(n)$. Tak samo strukturę LCP można utworzyć w czasie $O(n)$ [2]. Czas działania algorytmu permutation matching, to $O(n \log |A_W|)$. Przejrzenie drzewa sufiksowego w kolejności preorder zajmuje $O(n)$. Funkcję MISMATCH wykonamy d razy, gdzie d, to upper bound na liczbę takich samych permutacji W w T. Wywołanie funkcji $MISMATCH$ trwa $O(k)$, ponieważ każde wywołanie $lcp.query(i, j)$ jest stałe. Całość ma więc złożoność $O(n \log |A_W| + dk)$.

Literatura

[1] Grossi, R. & Luccio, F. Simple and efficient string matching with k mismatches. *Information Processing Letters.* **33**, 113-120 (1989), https://www.sciencedirect.com/science/article/pii/0020019089901889

[2] Galil, Z. & Giancarlo, R. Data structures and algorithms for approximate string matching. *Journal Of Complexity.* **4**, 33-72 (1988), https://www.sciencedirect.com/science/article/pii/0885064X88900088