Algorytm Bitap Shift-Add
W oparciu o „A new approach to text searching”
Ricardo Baeza-Yates, Gaston H. Gonnet

Juliusz Wajgelt

Maj 2022

1 Wstęp

Algorytm Shift-Add służy do znajdowania przybliżonych dopasowań (approximate matching) wzorca w tekście. Pozwala on na znalezienie wszystkich dopasowań wzorca w odległości (Hamminga) co najwyżej k w czasie $O(nm \log k)$ z dodatkowym preprocessingiem w czasie $O(|\Sigma| m \log k)$ oraz pamięci $O(|\Sigma| m \log k)$. Algorytm może być również łatwo rozszerzony aby pozwalać na przypisanie niedopasowaniom różnych wag (np. wliczanie do odległości niedopasowań między spółgłoskami a samogłoskami z wagą 2 razy większą, niż pozostałych), dopasowania do skończonych klas znaków, oraz dopasowań do wielu wzorców jednocześnie.

Co istotne, złożoność algorytmu zależy od rozmiaru słowa maszynowego komputera, na którym jest uruchamiany. Dla $b = \lceil \log_2(k+1) \rceil + 1$ oraz w równego długości słowa maszynowego, algorytm wykonuje $O(\lceil \frac{mb}{w} \rceil n)$ operacji i używa $O(|\Sigma| \cdot \lceil \frac{mb}{w} \rceil)$ pamięci, co przyczynia się do dobrej wydajności w praktycznych zastosowaniach dla krótkich wzorców.

2 Opis algorytmu

2.1 Dopasowania dokładne

Algorytm

Główna część algorytmu polega na liniowym przejściu po kolejnych znakach tekstu. W trakcie działania utrzymujemy zmienną `state`, będącą wektorem długości m mówiącą o stanie dopasowania wzorca kończącym się w rozpatrywanym miejscu tekstu. Dokładniej, gdy rozpatrujemy j-ty znak tekstu, `state[i]` będzie mówiło, czy wzorzec na pozycjach $1..i$ pasuje do tekstu na pozycjach $(j-i+1)..j$ - jeżeli tak, wartość `state[i]` będzie równa 0, a w przeciwnym przypadku 1. Oznacza to, że mamy do czynienia z dopasowaniem wzorca wtedy i tylko wtedy, gdy $\mathtt{m} = 0$.

Oznaczmy $s_i(j)$ wartość `state[i]` gdy rozpatrujemy j-ty znak tekstu. Nietrudno zauważyć, że

$$s_i(j) = s_{i-1}(j-1) \oplus \begin{cases} 0 & \text{jeżeli } pat_i = text_j \\ 1 & \text{wpp.} \end{cases}$$

gdzie $\oplus$ oznacza dodawanie boolowskie.

Ponieważ dla każdej pozycji `state` w danym kroku iteracji algorytmu potrzebujemy tylko jednego bitu informacji, możemy cały `state` trzymać jako maskę bitową długości m. Jedna iteracja polega na przesunięciu bitowym maski `state` o jedno miejsce w lewo (co odpowiada operacji $s_i(j) = s_{i-1}(j-1)$) a następnie wykonaniu *bitwise-or* z maską bitową `t` t.że:

$$\mathtt{t}_i = \begin{cases} 0 & \text{jeżeli } pat_i = text_j \\ 1 & \text{wpp.} \end{cases}$$

Zauważmy, że (przy ustalonym wzorcu *pat*) ponieważ wartość `t` zależy tylko od j-tego znaku tekstu $text_j$, możemy przed wykonaniem głównej pętli wyliczyć dla każdego znaku alfabetu $c \in \Sigma$ tablicę `T[c]` taką, że

$$\mathtt{T[c]}_i = \begin{cases} 0 & \text{jeżeli } pat_i = c \\ 1 & \text{wpp.} \end{cases}$$

dzięki czemu pojedyncza iteracja algorytmu to po prostu

$$\mathtt{state} := (\mathtt{state} \ \mathtt{<<}\ \mathtt{1}) \oplus \mathtt{T}[text_j]$$

gdzie $\oplus$ oznacza operację *bitwise-or* (tzn. dodawanie boolowskie po współrzędnych) a `<<` przesunięcie bitowe w lewo (ucinające $(m+1)$-szą cyfrę).

To, czy na obecnie rozpatrywanej pozycji kończy się dopasowanie możemy rozstrzygnąć sprawdzając, czy m-ty bit maski `state` jest równy 0, albo, równoważnie, czy $\mathtt{state} < 2^{m-1}$.

Złożoność

Główna pętla algorytmu wykonuje n iteracji, w każdej wykonując dwie czynności:

- wyliczenie nowego stanu $\mathtt{state} := (\mathtt{state} \ \mathtt{<<}\ \mathtt{1}) \oplus \mathtt{T}[text_j]$
- sprawdzenie, czy znaleziono dopasowanie $\mathtt{state} < 2^{m-1}$

Pierwsza z nich wymaga przesunięcia bitowego maski o jedno miejsce, co wymaga $\lceil \frac{m}{w} \rceil$ operacji procesora dla długości słowa maszynowego w oraz policzenia *bitwise-or* z drugą maską długości m, co również wymaga $\lceil \frac{m}{w} \rceil$ operacji.

Druga może być wykonana w jednej operacji - wystarczy porównać najbardziej znaczące słowa zapisu `state` oraz 2^{m-1}.

Daje to w sumie $O(\lceil \frac{m}{w} \rceil n)$ operacji dla głównej pętli algorytmu.

Preprocessing polega na wyliczeniu tablicy T. Możemy to zrobić inicjalizując dla każdego $c \in \Sigma$: $\texttt{T[c]} = 1..1$, a następnie iterując się po wzorcu i ustawiając

$$\texttt{T[}\mathrm{pat}_i\texttt{]} = \texttt{T[}\mathrm{pat}_i\texttt{]} \odot 1..10_i1..1$$

gdzie $\odot$ to iloczyn boolowski po współrzędnych (*bitwise-and*). Ponieważ w każdym kroku modyfikujemy tylko jedną pozycję w jednej masce, możemy to zrobić w czasie stałym, co daje złożoność preprocessingu $O(|\Sigma| \cdot m)$.

W trakcie działania algorytmu musimy utrzymywać w pamięci jedynie stan `state` oraz tablicę T, co daje złożoność pamięciową

$$O(\left\lceil \frac{m}{w} \right\rceil + |\Sigma| \cdot \left\lceil \frac{m}{w} \right\rceil) = O(|\Sigma| \cdot \left\lceil \frac{m}{w} \right\rceil)$$

Co istotne, nie musimy przechowywać w pamięci tekstu wejściowego (możemy go przetwarzać jako strumień znaków).

2.2 Dopasowanie przybliżone

Algorytm

Algorytm dla dopasowania dokładnego można łatwo zmodyfikować tak, aby pozwalał na znajdowanie przybliżonych dopasowań wzorca o odległości Hamminga nieprzekraczającej k.

W tym przypadku stan `state` zamiast trzymać informację, czy na danym prefiksie wzorca nastąpiło niedopasowanie z sufiksem tekstu, będziemy zliczać, ile niedopasowań wystąpiło, to znaczy gdy rozpatrujemy j-ty znak tekstu,

$$\texttt{state[i]} = \#\{k \; : \; 1 \leqslant k \leqslant i, \, pat_k \neq text_{j-i+1+k}\}$$

Ponownie, jeżeli oznaczymy $s_i(j)$ wartość `state[i]` gdy rozpatrujemy j-ty znak tekstu to

$$s_i(j) = s_{i-1}(j-1) + \begin{cases} 0 & \text{jeżeli } pat_i = text_j \\ 1 & \text{wpp.} \end{cases}$$

Podobnie jak w poprzedniej wersji algorytmu, możemy przygotować sobie wartości, które dodajemy w każdej iteracji żeby wyliczyć nowy stan jako

$$\texttt{T[c]}_i = \begin{cases} 0 & \text{jeżeli } pat_i = c \\ 1 & \text{wpp.} \end{cases}$$

Jeżeli pojedynczą komórkę stanu `state[i]` zapiszemy za pomocą b bitów, to pojedynczy krok algorytmu wyglądałby następująco:

$$\texttt{state} := (\texttt{state << } b) + \texttt{T[}text_j\texttt{]}$$

W zależności od wyboru b pojawia się tu problem - ponieważ wykonujemy operację dodawania, może nastąpić przepełnienie `state[i]`, w wyniku czego zmodyfikujemy `state[i+1]`. Możemy w prosty sposób zapobiec temu poprzez dodanie dodatkowego bitu przepełnienia dla każdego `state[i]`, który w każdym kroku algorytmu przepisujemy do osobnej maski bitowej `overflow[i]`, a następnie zerujemy. Daje to zmodyfikowany algorytm:

$$
\begin{aligned}
\texttt{state} &:= (\texttt{state << } b) + \texttt{T[}text_j\texttt{]} \\
\texttt{overflow} &:= (\texttt{overflow << } b) \oplus (\texttt{state} \odot \texttt{mask}) \\
\texttt{state} &:= \texttt{state} \odot \overline{\texttt{mask}}
\end{aligned}
$$

Gdzie `mask` to maska bitowa z `1` na co b-tym bicie (czyli na pozycjach bitów przepełnienia).

W tym algorytmie mamy dopasowanie przybliżone o odległości co najwyżej k, jeżeli $\texttt{state[m]} \leqslant k$ oraz $\texttt{overflow[m]} = 0$.

Ostatni szczegół to ustalenie odpowiedniej wartości b. Ponieważ interesuje nas tylko, czy liczba niedopasowań wzorca nie przekroczyła k, wystarczy, że każda komórka `state[i]` będzie mogła przechować liczby do k oraz dodatkowy bit przepełnienia, co daje wartość b

$$b = \lceil \log_2(k+1) \rceil + 1$$

Złożoność

Analiza złożoności jest analogiczna do wariantu z dopasowaniem dokładnym, z jedyną różnicą, że każda komórka masek `state`, `overflow`, `T[c]` zajmuje b bitów, a więc złożoność każdej operacji zmienia się na $O(\lceil \frac{mb}{w} \rceil)$, dając ostatecznie złożoność głównej pętli algorytmu $O(\lceil \frac{mb}{w} \rceil)$, preprocessingu $O(|\Sigma| \cdot mb)$ oraz złożoność pamięciową $O(|\Sigma| \cdot \lceil \frac{mb}{w} \rceil)$.

Ponieważ stosunek długości wzorca m i liczby dopuszczalnych niedopasowań k do długości słowa maszynowego ma duże przełożenie na wydajność algorytmu, warto rozważyć, dla różnych wartości k, dla jakich długości wzorca m możemy zmieścić w całości maski bitowe używane w algorytmie w jednym słowie maszynowym.

k	$w = 64$	$w = 128$	$w = 256$
1	$m = 32$	$m = 64$	$m = 128$
2-3	$m = 21$	$m = 42$	$m = 85$
4-7	$m = 16$	$m = 32$	$m = 64$
8-15	x	$m = 25$	$m = 51$
16-21	x	x	$m = 42$

2.3 Skończone klasy znaków

Oba algorytmy można łatwo rozszerzyć o możliwość dopasowania na każdej pozycji nie pojedynczego znaku, a jednego z pewnego skończonego zbioru znaków. Zauważmy, że jedyne miejsce, w którym sprawdzamy, czy znak tekstu pasuje do wzorca, to dodanie wartości $\texttt{T[}text_j\texttt{]}$ do stanu `state`. Modyfikacja algorytmu polegać będzie na wyliczeniu tablicy `T` jako:

$$\texttt{T[c]}_i = \begin{cases} 0 & \text{jeżeli } c \in pat_i \\ 1 & \text{wpp.} \end{cases}$$

gdzie pat_i to zbiór znaków, które możemy dopasować na i-tej pozycji wzorca.

2.4 Niedopasowania z wagami

Algorytm dopasowania przybliżonego możemy rozszerzyć tak, aby liczył różne niedopasowania z różnymi wagami. Niech $\mathrm{cost}(p, t)$ oznacza koszt niedopasowania znaku p we wzorcu do znaku t w tekście. Modyfikacja algorytmu polega na wyliczeniu tablicy `T` jako:

$$\texttt{T[c]}_i = \begin{cases} 0 & \text{jeżeli } pat_i = c \\ \mathrm{cost}(pat_i, c) & \text{wpp.} \end{cases}$$

Łatwo zauważyć, że dla takiej tablicy `T`, algorytm dopasowania przybliżonego zwróci podsłowa tekstu takie, że suma kosztów wszystkich niedopasowań wzorca do tego podsłowa jest conajwyżej k.