Fast practical multi-pattern matching

Mateusz Pabian

1 Wprowadzenie

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 (co najmniej dwuelementowego) jest jednostajne i niezależne, złożoność wynosi $O(\frac{n}{m} * \log m)$, co jest równocześnie dolnym ograniczeniem dla tego problemu. Dodatkowo dla odpowiednio dużego m algorytm osiąga dobre rezultaty w praktyce.

2 Idea

Idea algorytmu oparta jest o dwie fazy zwane *PROCESS1* oraz *PROCESS2*. Pierwsza z nich skanuje test od lewej do prawej przesuwając potencjalną pozycję zakończenia wzorca co najmniej o $\frac{m}{2}$. W tej fazie bazuje on na zmiennej γ, która spełnia niezmiennik $\gamma =$ najdłuższy suffix od początku tekstu do pozycji i, który jest jednocześnie prefiksem jakiegoś wzorca. Jeśli długość $\gamma \leqslant \frac{m}{2}$ kończymy pierwszą fazę algorytmu i przechodzimy do fazy drugiej. W implementacji będziemy utożsamiać 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 wyznaczamy 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.

3 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.

Algorithm 1 Fast practical multi-pattern matching, preprocessing

```
procedure MULTI-PATTERN-MATCHING-BUILD(PATTERNS)              ▷ lista wzorców
    acm = build_aho_corasick_machine(patterns)    ▷ budujemy standardowy automat AC
    reversed_patterns = reverse(patterns)                   ▷ odwracamy wzorce
    dawg = build_dawg(reversed_patterns)             ▷ budujemy standardowy DAWG
    return (acm, dawg)                               ▷ budujemy standardowy DAWG
end procedure
```

Definicja 3.1. *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 3.2. *DAWG powinien udostępniać:*

- *$traverse(node, s)$ = stan $DAWG$ po przetworzeniu słowa s zaczynając w node.*

Definicja 3.3. *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.*

Algorithm 2 Fast practical multi-pattern matching, NEXT1

```
procedure NEXT1(γ, character)
    return acm.step(gamma, char)            ▷ wykonujemy jeden krok w AC
end procedure
```

Algorithm 3 Fast practical multi-pattern matching, NEXT2

```
procedure NEXT2(j, i)                          ▷ indeksy zakresu tekstu
    γ_new = dawg.traverse(dawg.root, j, i)     ▷ przechodzimy DAWG skanując znaki od i do j
    return acm.traverse(acm.root, γ_new)       ▷ przechodzimy AC aby móc utożsamić γ ze stanem
end procedure
```

4 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 $\frac{m}{2}$. Przesunięcie opieramy na zmiennej γ, która spełnia niezmiennik $\Gamma : \gamma$ = najdłuższy suffix od początku tekstu do pozycji i, który jest jednocześnie prefiksem jakiegoś wzorca. Jeśli długość $\gamma \leqslant \frac{m}{2}$, 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.

Wynikiem tej części 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.

Algorithm 4 Fast practical multi-pattern matching, faza wyszukiwania

```
procedure MULTI-PATTERN-MATCHING(text, n, S)            ▷ tekst, długość, struktura
    i := 0                                               ▷ pozycja w tekście
    γ := S.acm.root                                      ▷ równoważnie γ := S.acm.root
    while True do                                        ▷ faza skanowania, zachowany niezmiennik Γ
        while γ.depth ⩾ m/2 do                           ▷ szukamy odpowiednio długiego prefiksu wzorca
            if γ is final state then
                yield (w, i) dla każdego w ∈ γ.out
            end if
            i = i + 1
            if i ⩾ n then
                return
            end if
            γ = NEXT1(γ, text[i])                        ▷ wykonujemy krok w AC
        end while
        crit_pos := i − γ.depth + 1                      ▷ pierwsza pozycja, której nie wykluczyliśmy
        SHIFT := m − γ.depth
        i = i + SHIFT                                    ▷ przeskakujemy fragment tekstu
        if i ⩾ n then
            return
        end if
        γ = NEXT2(crit_pos, i)                           ▷ szukamy najdłuższego sufiksu w DAWG
    end while
end procedure
```

5 Złożoność obliczeniowa algorytmu

Lemat 5.1. *Zakładając 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 5.2. *Algorytm w sumie porównuje co najwyżej* $2n$ *znaków.*

Dowód. *Zauważmy, że znak na danej pozycji może być przetworzony przez* $PROCESS1$ *co najwyżej raz, ponieważ indeks i jest sukcesywnie przesuwany do przodu oraz* $PROCESS2$ *nigdy nie cofa go do tyłu. Oczywistym jest, że w jednym przejściu* $PROCESS2$*, również wykona tylko jedno porównanie. Zauważmy również, że* $SHIFT > \frac{m}{2}$. *Aby dwa kolejne wykonania* $PROCESS2$ *odwiedziły ten sam znak, długość* γ *przed rozpoczęciem fazy musiała by być większa niż* $\frac{m}{2}$ *co jest sprzeczne z warunkiem pętli fazy pierwszej.* □

Niech σ to prawdopodobieństwo jednostajnego wystąpienia symbolu w tekście. Wtedy:

Lemat 5.3. $Pr[\Delta(i) > (3k+1) * log_\sigma m] \leqslant \frac{1}{m^{k+1}}$.

Twierdzenie 5.4. *Zakładając* $M \leqslant m^k$ *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* $K = (3k+1) * log_\sigma m$. *Zgodnie z powyższym lematem* $PROCESS2$ *zakończy się z dużym prawdopodobieństwem po K krokach (prawdopodobieństwo, że dopasujemy mniej niż K symboli jest wynosi co najmniej* $\frac{m^{k+1}-1}{m^{k+1}}$*). Jeśli nie, zrobi on* m^k *kroków z prawdopodobieństwem* $\frac{1}{m^{k+1}}$. *Dostajemy*

więc średnią złożoność $O(K)$ *co jest logarytmiczne.*
Skorzystajmy z podobnego argumentu dla $PROCESS1$. *Jeśli* $\Delta(i+K) < K$ *wtedy algorytm przegląda co najwyżej* K *znaków (od pozycji* i *do* $i+K$*). Prawdopodobieństwo przeciwnego zdarzenia ponownie jest bardzo małe* ($\leqslant \frac{1}{m^{k+1}}$). $\square$