

# Automat Aho-Corasik

— omówienie —

Piotr Mikołajczyk

## 1 Wprowadzenie

Niech  $\mathcal{K} = \{k_1, \dots, k_n\}$  będzie zbiorem wzorców (niepustych słów nad alfabetem  $\Sigma$ ). Otrzymawszy słowo  $T \in \Sigma^m$  chcemy wyznaczyć wszystkie indeksy z  $[m]$ , na których w  $T$  zaczyna się pewien wzorec z  $\mathcal{K}$ . Naturalnym rozwiązaniem jest  $n$ -krotne użycie jednego z wydajnych (liniowych) algorytmów do szukania pojedynczego wzorca. Podejście takie oczywiście jest poprawne, niestety skutkuje złożonością czasową  $\Theta(nm)$ . Okazuje się, że na podstawie  $\mathcal{K}$  jesteśmy w stanie skonstruować w czasie  $\Theta(\sum_{i \in [n]} |k_i|)$  odpowiednią strukturę, która będzie potrafiła znaleźć naraz wszystkie wystąpienia wzorców oglądając  $T$  wyłącznie raz. Przed nami: automat Aho-Corasick.

## 2 Idea

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  $\mathcal{K}$ , 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  $\mathcal{K}$ .

Zacniemy konstrukcję od utworzenia *drzewa trie* na podstawie  $\mathcal{K}$ . 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:

```
1 def find_occurrences(self, text, n):
2     state = self._root
3     for i in range(1, n + 1):
4         state = state.next(text[i])
5         for keyword, keyword_len in state.output():
6             yield i - keyword_len + 1, keyword
```

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

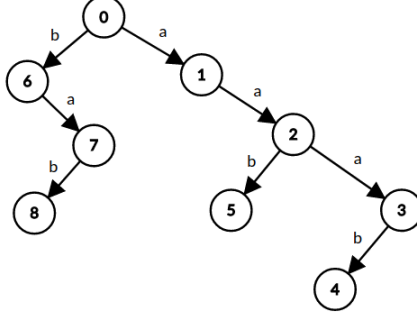
## 3 Konstrukcja

*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.*

### 3.1 Drzewo trie

Mając zbiór słów  $\mathcal{K}$  nad alfabetem  $\Sigma$  możemy w prosty sposób zbudować tzw. *drzewo trie*, ozn.  $\mathcal{T}_{\mathcal{K}}$ . Jest to skierowane, ukorzenione drzewo, którego krawędzie etykietowane są symbolami z  $\Sigma$ . Każde słowo  $k \in \mathcal{K}$

ma odpowiadającą mu ścieżkę z korzenia, etykietowaną kolejnymi literami  $k$ . Jeśli  $k \in \mathcal{K}$  nie jest prefiksem żadnego innego słowa z  $\mathcal{K}$ , to jego ścieżka kończy się w liściu. Inaczej ujmując, istnieje bijekcja pomiędzy ścieżkami w  $\mathcal{T}_{\mathcal{K}}$  a wszystkimi (różnymi) prefiksami słów z  $\mathcal{K}$ . Krawędź etykietowaną symbolem  $a$  pomiędzy  $s_1$  a  $s_2$  możemy kodować jako trójkę  $(s_1, a, s_2)$ .



Rysunek 1:  $\mathcal{T}_{\{aaab, aab, bab, ba\}}$

### 3.2 Plan

Zacznijmy temat konstrukcji od zdefiniowania funkcji, które będą nam potrzebne. Niech  $\mathcal{K}$  oznacza zbiór wzorców,  $\mathcal{T}$  oznacza drzewo trie dla  $\mathcal{K}$  (skrót dla  $\mathcal{T}_{\mathcal{K}}$ ),  $\mathbf{r}$  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  $\Sigma$ , który nie etykietuje żadnej krawędzi wychodzącej z  $\mathbf{r}$ , dodajemy pętlę w  $\mathbf{r}$ ; czyli:

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

- **output** ::  $\mathcal{S} \mapsto \wp(\mathcal{K})$  – jak zostało już wspomniane, chcemy w każdym stanie  $s$  przechowywać listę wzorców, których wystąpienie w  $T$  będzie poświadczane odwiedzeniem stanu  $s$ ; w sporym zakresie możemy to obliczyć już przy konstrukcji  $\mathcal{T}$  (np. wracając do rysunku 1, **output**(1) =  $\emptyset$ , **output**(5) =  $\{aab\}$ , **output**(7) =  $\{ba\}$ ); będziemy musieli jeszcze jednak obsłużyć przypadki, gdy jeden wzorec 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  $\mathbf{r}$  a  $s$  tworzą najdłuższy możliwy prefiks któregoś ze wzorców z  $\mathcal{K}$ , 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  $\mathbf{r} \rightsquigarrow s'$  było najdłuższym możliwym sufiksem słowa odpowiadającemu  $\mathbf{r} \rightsquigarrow s$ ; w dalszej części utożsamiać będziemy ścieżki ze słowami;

Chodzenie krawędziami **fail** opiera się na identycznym pomysłe co rekurencyjne używanie tablic *Border* w algorytmie Morrisa-Pratta. Warto również zauważyć, że **fail** nie ma sensu definiować dla  $\mathbf{r}$ .

- **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**

### 3.3 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  $\mathcal{T}_K$ :

```
1 def _construct_goto(self, keywords, alphabet):
2     for k, k_len in keywords:
3         self._enter(k, k_len)
4
5     for a in alphabet:
6         if self._root.goto(a) is None:
7             self._root.update_goto(a, self._root)
8
9 def _enter(self, keyword, keyword_len):
10     current_state = self._root
11     j = 1
12
13     while j < keyword_len and current_state.goto(keyword[j]) is not None:
14         current_state = current_state.goto(keyword[j])
15         j += 1
16
17     for a in keyword[j:keyword_len + 1]:
18         next_state = AhoCorasickAutomaton.Node()
19         current_state.update_goto(a, next_state)
20         current_state = next_state
21
22     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 `r` węzłów – podobnie jak w algorytmie Morrisa-Pratta długości najdłuższych borderów obliczaliśmy na podstawie wartości dla krótszych słów:

```
1 def _construct_fail(self, alphabet):
2     q = Queue()
3     for s in [self._root.goto(a) for a in alphabet]:
4         if s != self._root:
5             q.put(s)
6             s.update_fail(self._root)
7
8     while not q.empty():
9         current = q.get()
10        for a, child in [(a, current.goto(a)) for a in alphabet]:
11            if child is not None:
12                q.put(child)
13
14            fallback = current.fail()
15            while fallback.goto(a) is None:
16                fallback = fallback.fail()
17
18            child_fallback = fallback.goto(a)
19            child.update_fail(child_fallback)
20            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 wartość `fail(child)`. Szukamy więc najdłuż-

szego sufiksu słowa generowanego przez  $\tau \rightsquigarrow \text{child}$  obecnego w  $\mathcal{T}$ . Będzie to najdłuższy sufiks ścieżki  $\tau \rightsquigarrow \text{current}$ , który można przedłużyć o symbol  $a$ .

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

### 3.4 Obliczanie next

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

```

1  def _construct_next(self, alphabet):
2      q = Queue()
3      for a in alphabet:
4          a_child = self._root.goto(a)
5          self._root.update_next(a, a_child)
6          if a_child != self._root:
7              q.put(a_child)
8          self._root.use_only_next()
9
10     while not q.empty():
11         current = q.get()
12         for a, child in [(a, current.goto(a)) for a in alphabet]:
13             if child is not None:
14                 q.put(child)
15                 current.update_next(a, child)
16             else:
17                 fallback = current.fail()
18                 current.update_next(a, fallback.next(a))
19         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.

## 4 Analiza

### 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.** Niech  $s, t \in \mathcal{S}$  i niech  $u, v$  będą słowami generowanymi przez ścieżki  $\tau \rightsquigarrow s$  i  $\tau \rightsquigarrow t$ . Wówczas  $\text{fail}(s) = t \iff v$  jest najdłuższym właściwym sufiksem  $u$ , który jest prefiksem pewnego  $k_i \in \mathcal{K}$ .

**Lemat.** Słowo  $k$  należy do `output(s)`  $\iff k \in \mathcal{K}$  oraz  $k$  jest sufiksem słowa generowanego przez  $\tau \rightsquigarrow s$ .

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

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

### 4.2 Analiza złożoności

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

```

1 class AhoCorasickAutomaton:
2     def __init__(self, keywords, alphabet):
3         self._root = AhoCorasickAutomaton.Node()
4         self._construct_goto(keywords, alphabet)
5         self._construct_fail(alphabet)
6         self._construct_next(alphabet)

```

Bardzo łatwo zauważyć, że każdą z nich wykonać można w czasie  $\Theta(\sum_{i \in [n]} |k_i|)$  – 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 związane).

Pozostaje zastanowić się, ile czasu zajmie przeczytanie tekstu  $T$  ( $|T| = m$ ) i wypisanie wszystkich wystąpień wzorców. Moc  $\Sigma$  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  $2m$  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  $\mathbf{r}$ , a z  $\mathbf{r}$  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` przesuwają 'prawy' wskaźnik o 1 w prawo, przejście `fail` przesuwają 'lewy' wskaźnik o dodatnią liczbę pozycji w prawo, ale na tyle małą by nie przeskoczyć prawego;

- używając funkcji `next`, przy każdym przeczytanym symbolu z  $T$  wykonujemy zawsze jedno przejście –  $m$  ruchów

Problem może się pojawić, gdy będziemy chcieli wypisywać wszystkie wystąpienia wzorców. Pesymistycznym przykładem jest  $\mathcal{K} = \{a, a^2, a^3, \dots, a^n\}$  oraz  $T = a^m$ . Liczba wystąpień będzie  $\Theta(nm)$ . 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.

## 5 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  $m^2$ ,  $P$  - kwadratową tablicą rozmiaru  $n^2$  o różnych kolumnach,  $n < m$
2. niech  $\mathcal{K}$  będzie zbiorem kolumn  $P$
3. automatem Aho-Corasick szukamy  $\mathcal{K}$  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  $(1, 2, \dots, n)$

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