Algorytmy i struktury danych z językiem Python
Algorytm Aho-Corasick

Angela Czubak

1 Teoria

1.1 Wprowadzenie

Algorytm Aho-Corasick został opracowany przez Alfreda V. Aho oraz Margaret J. Corasick. Jego celem jest znalezienie wzorców $\mathcal{P} = \{P_0, \ldots, P_k\}$ pochodzących z pewnego słownika w tekście.
Cechą charakterystyczną tego algorytmu jest to, że szukanie wystąpień zadanych słów następuje "na raz", dzięki czemu złożoność obliczeniowa tego algorytmu wynosi $O(m + z + n)$, gdzie m - długość tekstu, w którym wyszukujemy, z - liczba wystąpień wzorców w zadanym tekście oraz $n = \sum_{i=0}^{k} |P_i|$ - sumy długości tychże.
Algorytm ten jest stosowany np. w komendzie UNIX-a - `fgrep`.
Idea algorytmu opiera się na drzewach trie i automatach, których tworzenie omówię dalej.

1.2 Drzewo trie

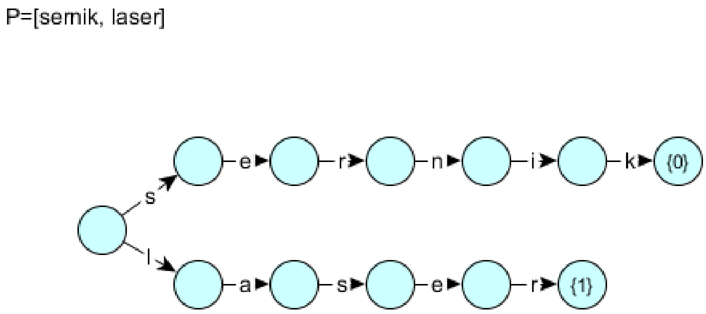
Definicja 1. ***Drzewem trie*** *dla zbioru wzorców $\mathcal{P}$ nazywamy takie ukorzenione drzewo $\mathcal{K}$, że:*

1. *Każda krawędź drzewa $\mathcal{K}$ jest etykietowana jakimś znakiem*
2. *Każde dwie krawędzie wychodzące z jednego wierzchołka mają różne etykiety*

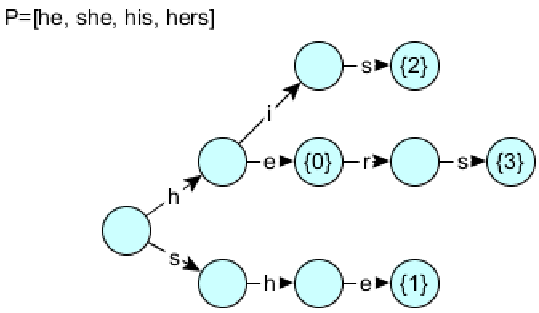
   Definicja 2. ***Etykietą węzła*** *v nazywamy konkatenację etykiet krawędzi znajdujących się na ścieżce z korzenia do v. Oznaczamy ją jako $\mathcal{L}(v)$.*

3. *Dla każdego $P \in \mathcal{P}$ istnieje wierzchołek v taki, że $\mathcal{L}(v) = P$, oraz*
4. *Etykieta $\mathcal{L}(v)$ jakiegokolwiek liścia v jest równa jakiemuś $P \in \mathcal{P}$*

Przykładowe rysunki drzew trie znajdują się na następnej stronie. Numer w wierzchołku oznacza indeks słowa należącego do słownika, które jest etykietą tego wierzchołka.



Rysunek 1: Drzewo trie dla słów $\mathcal{P} = \{sernik, laser\}$



Rysunek 2: Drzewo trie dla słów $\mathcal{P} = \{he, she, his, hers\}$

1.3 Konstrukcja drzewa trie

Jak budować drzewo trie dla $\mathcal{P} = \{P_0, \ldots, P_k\}$? Procedura jest następująca:

1. Rozpocznij od stworzenia korzenia
2. Umieszczaj kolejne wzorce jeden po drugim według poniższych kroków:
   (a) Poczynając od korzenia, podążaj ścieżką etykietowaną kolejnymi znakami wzorca P_i
   (b) Jeśli ścieżka kończy się przed P_i, to dodawaj nowe krawędzie i węzły dla pozostałych znaków P_i
   (c) Umieść identyfikator i wzorca P_i w ostatnim wierzchołku ścieżki

Jak łatwo zauważyć, konstrukcja drzewa zajmuje $O(|P_0| + \cdots + |P_k|) = O(n)$.

1.4 Wyszukiwanie wzorca w drzewie

Wyszukiwanie wzorca P odbywa się następująco:
Tak długo, jak to możliwe, podążaj ścieżką etykietowaną kolejnymi znakami P

1. Jeśli ścieżka prowadzi to wierzchołka z pewnym identyfikatorem, P jest słowem w naszym słowniku $\mathcal{P}$
2. Jeśli ścieżka kończy się przed P, to słowa nie ma słowniku
3. Jeśli ścieżka kończy się w wierzchołku bez identyfikatora, to słowa nie ma w słowniku

Wyszukiwanie zajmuje więc $O(|P|)$.
Naiwnie postępując, moglibyśmy chcieć wyszukiwać wzorce w tekście tak, by dla każdego znaku tekstu próbować iść wzdłuż krawędzi odpowiadającym kolejnym znakom - jeśli po drodze przejdziemy przez wierzchołki z identyfikatorami, to znaleźliśmy słowa im odpowiadające. Gdy już nie ma krawędzi, którą moglibyśmy przejść, zaczynamy wyszukiwanie dla kolejnego znaku tekstu. Jednak takie wyszukiwanie zajęłoby $O(nm)$ czasu, gdzie m - długość tesktu, n - suma długości wzorców.
By przyspieszyć wyszukiwanie wzorców, rozszerzamy drzewo trie do **automatu**.

1.5 Automat

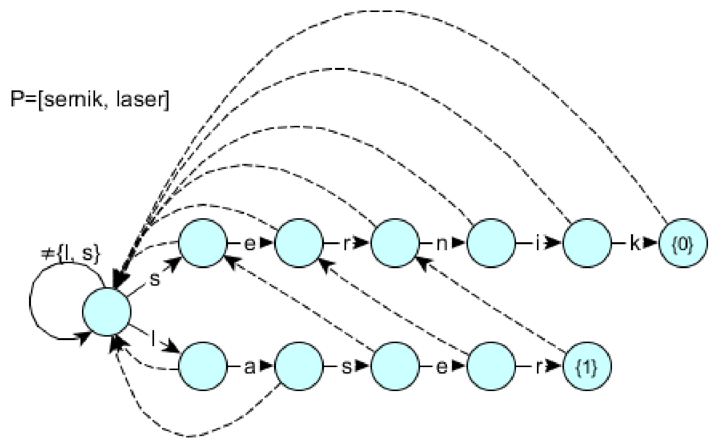
Definicja 3. ***Automatem deterministycznym*** *nazywamy piątkę uporządkowaną* $(\Sigma, Q, q, \delta, F)$*, gdzie*

1. Σ *- skończony alfabet*
2. Q *- skończony zbiór stanów*
3. $q \in Q$*- stan początkowy*
4. $\delta : Q \times \Sigma \rightarrow Q$ *- funkcja przejścia, przypisującą parze* (q, a) *nowy stan* p*, w którym znajdzie się automat po przeczytaniu symbolu* a *w stanie* q
5. $F \subset Q$ *- zbiór stanów końcowych*

W naszym przypadku automat, w związku z wprowadzeniem dodatkowych pojeć, nie będzie ściśle deterministyczny. Automat będziemy budować na podstawie drzewa trie, zatem uściślam: zbiór stanów będą stanowiłi węzły drzewa, a stanem początkowym będzie korzeń, do którego będę się czasami odnosiła jako 0. Wprowadzamy następujące funkcje:

1. Funkcja **goto**, oznaczana jako $g(q, a)$ - jest to odpowiednik funkcji przejścia δ, odpowiada ona krawędziom w drzewie, ponadto zachodzą jeszcze pewne własności; w skrócie:
   - jeśli krawędź (q, v) jest etykietowana przez a, to $g(q, a) = v$
   - $g(0, a) = 0$ dla każdego a nie będącego etykietą krawędzi wychodzącej z korzenia - automat ma pozostać w stanie początkowym, jeśli nie można znaleźć dopasowania
   - w przeciwnym przypadku $g(q, a) = \emptyset$ - brak przejścia w automacie
2. Funkcja **failure**, oznaczana jako $f(q)$, dla każdego stanu różnego od początkowego ($q \neq 0$) zwraca stan, do którego powinniśmy się udać w przypadku niemożności zastosowania funkcji $g(q, a)$ - nie istnieje krawędź wychodząca z q, etykietowana przez a. Stanem tym jest węzeł odpowiadający najdłuższemu właściwemu sufiksowi $L(q)$ (najdłuższemu podsłowu

y, krótszemu niż samo słowo s, takiemu, że istnieje słowo t o niezerowej długości, że $s = ty$). Chodzi o to, by nie przegapić żadnego potencjalnego dopasowania wzorca - np. biorąc słowa *laser, sernik* i szukając w tekście *lasernik*, zaczniemy od dopasowania do słowa *laser*, a powinniśmy mieć jeszcze możliwość przejścia do gałęzi odpowiadającej słowu *sernik*, by także je odnaleźć. Funkcje przejścia w tym przypadku przedstawiono na rysunku (3). Funkcja $f(q)$ jest zawsze dobrze zdefiniowana, gdyż $\mathcal{L}(0) = \epsilon$ jest sufiksem każdego słowa.



Rysunek 3: Automat dla słów $\mathcal{P} = \{sernik, laser\}$. Przerywaną linią oznaczono krawędzie odpowiadające funkcji $f(q)$

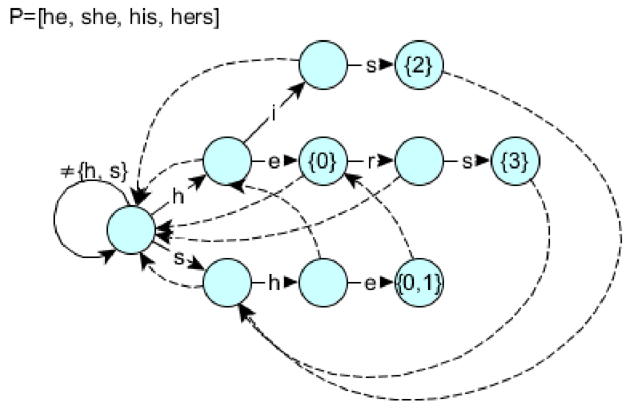
3. Funkcja **wyjścia**, oznaczana jako $out(v)$, zwraca indeksy wzorców, do których znajdujemy dopasowanie w stanie q.

1.6 Przeszukiwanie tekstu

Załóżmy, że mamy do dyspozycji gotowy automat oraz tekst $T[1 \cdots]$, w którym szukamy wzorców. Procedura ta prezentuje się w następujący sposób:
Algorithm 1.1: SEARCH($T[0 \dots m-1]$)

$q \leftarrow 0$
for $i \leftarrow 0$ **to** $m - 1$
do {
  while $g(q, T[i]) = \emptyset$
    do { $q \leftarrow f(q)$
    comment: podążaj za funkcją failure, aż znajdziesz dopasowanie }
  $q \leftarrow g(q, T[i])$
  comment: przejdź do dopasowanego stanu
  if $out(q) \neq \emptyset$
    then output $(i), out(q)$ }

Weźmy automat jak na rysunku (4):



Rysunek 4: Automat dla słów $\mathcal{P} = \{he, she, his, hers\}$. Przerywaną linią oznaczono krawędzie odpowiadające funkcji $f(q)$

Przeszukamy przy jego pomocy tekst *ushers*:

1. Czytamy znak u - zostajemy w korzeniu

2. Czytamy znak s - przechodzimy po odpowiedniej krawędzi, czyli idziemy do węzła etykietowanego s

3. Czytamy znak h - przechodzimy po odpowiedniej krawędzi, czyli idziemy do węzła etykietowanego sh

4. Czytamy znak e - przechodzimy po odpowiedniej krawędzi, czyli idziemy do węzła etykietowanego she; wypisujemy, że znaleźliśmy słowa o indeksach 0 i 1 na pozycji 3

5. Czytamy znak r - korzystamy z krawędzi **failure**, następnie przechodzimy po odpowiedniej krawędzi, czyli idziemy do węzła etykietowanego her

6. Czytamy znak s - przechodzimy po odpowiedniej krawędzi, czyli idziemy do węzła etykietowanego $hers$; wypisujemy, że znaleźliśmy słowo o indeksie 3 na pozycji 5

Złożoność czasowa takiej procedury jest $O(m + z)$, gdzie m - długość tekstu, w którym wyszukujemy, a z - liczba wystąpień wzorca w tekście.
Wynika to z faktu, że liczba wykonań funkcji $f(q)$ jest ograniczona z góry przez m - w danym momencie możemy wywołać funkcję $f(q)$ co najwyżej tyle razy, ile znaków zdążyliśmy przeczytać z T, a w sumie możemy ich przeczytać m (po wywołaniu tej funkcji "przesuwamy" początek sufiksu na pewną literę z ciągu T - przesunąć ten początek możemy maksymalnie m razy).
Podobnie, funkcja $g(q, a)$ jest wywoływana dokładnie raz dla każdego $a \in T$ - zostanie ona wywołana m razy. Wystąpienie wzorca możemy zgłaszać w czasie stałym, stąd zgłoszenie wszystkich zajmie $O(z)$.

1.7 Budowa automatu

W konstrukcji automatu możemy wyróżnić dwie fazy.

1.7.1 Faza I

1. Tworzymy drzewo trie dla słownika $\mathcal{P}$

   item Dla każdego $P_i \in \mathcal{P}$ ustawiamy $out(v) = i$ dla wierzchołka v etykietowanego przez P_i

2. Uzupełnij funkcje przejść dla korzenia

$$g(0, a) = 0$$

   dla każdego znaku a (należącego do alfabetu Σ) takiego, że nie etykietuje on żadnej krawędzi wychodzącej z korzenia

Złożoność czasowa takiej procedury, dla pewnego niezmiennego alfabetu, wynosi $O(n)$, gdzie $n = \sum_{i=0}^{k} |P_i|$.

1.7.2 Faza II

Przedstawię ją w postaci pseudokodu:
Algorithm 1.2: Phase2(*void*)

$Q \leftarrow$ Queue()
for $a \in \Sigma$
  do { **if** $q \leftarrow g(0, a) \neq 0$
    then { $f(q) \leftarrow 0$
      Q.enqueue(q)
while !Q.isEmpty()
  do { $r \leftarrow$ Q.dequeue()
    for $a \in \Sigma$
      do if $u \leftarrow g(r, a) \neq \emptyset$
        then { Q.enqueue(u)
          $v \leftarrow f(r)$
          while $g(v, a) = \emptyset$
            do $v \leftarrow f(v)//(*)$
          $f(u) \leftarrow g(v, a)$
          $out(u) \leftarrow out(u) \cup out(f(u))//(**)$

Jak widać funkcje f i out są wyliczane dla wierzchołków w kolejności BFS. Dzięki temu wierzchołki znajdujące się bliżej korzenia zostały już obsłużone, gdy zachodzi potrzeba skorzystania z odpowiednich funkcji na nich wykonywanych.
Rozważmy wierzchołki r i $u = g(r, a)$, w takim przypadku r jest rodzicem u. Co więcej, $\mathcal{L}(u) = \mathcal{L}(r)a$. Jakie więc powinno być $f(u)$? Przypomnijmy, że $f(u)$ powinno wskazywać na najdłuższy właściwy sufiks $\mathcal{L}(u)$. Z tego wynika, że powinniśmy spróbować dopasować $f(u) = g(f(r), a)$, bo $\mathcal{L}(f(u))$ może być sufiksem $L(g(f(r), a))$, o ile taka krawędź istnieje. Jeśli jej nie ma, to próbujemy

$f(u) = g(f(f(r)), a)$, itd., aż znajdziemy odpowiedni wierzchołek (pesymistycznie może to być korzeń). W linii oznaczonej (*) wykonujemy właśnie te czynności.

Czynności oznaczone (**) wykonujemy, gdyż wzorce rozpoznawane w stanie $f(u)$ (i jedynie te) są właściwymi sufiksami $\mathcal{L}(u)$ i dlatego powinny być rozpoznawane także w stanie u.

Jaka jest złożoność powyższej procedury? Zauważmy, że jest ona podobna do BFS-a - stąd przechodzenie po drzewie, pomijając linię oznaczoną (*), zajmie czas proporcjonalny do rozmiary drzewa - tj. $O(n)$. A jak określić, ile razy wykonamy linię (*)?

Rozważmy ścieżkę złożoną z wierzchołków $u_1, \ldots, u_l$, która jest tworzona podczas dodawania wzorca $a_1 \ldots a_l$. Oznaczmy dodatkowo $d\!f(u)$ jako głębokość w drzewie wierzchołka $f(u)$, zatem $d\!f(u_1), \ldots, fd(u_l)$ to ciąg głębokości dla wierzchołków z rozważanej ścieżki, wszystkie są $\geqslant 0$.

Zauważmy ponadto, że głębokość kolejnego wierzchołka może wzrosnąć co najwyżej o 1, czyli $d\!f(u_{i+1}) \leqslant d\!f(u_i) + 1$, zatem wartości $d\!f$ wzrastają sumarycznie co najwyżej o l podczas przechodzenia tej ścieżki.

Kiedy wyliczamy położenie $f(u_{i+1})$, każde wywołanie linii (*) przybliża v do korzenia, a stąd wartość $d\!f(u_{i+1})$ będzie mniejsza od $d\!f(u_i)$ co najmniej o jeden. W związku z ograniczeniem od dołu, możemy zmniejszać kolejne wartości $d\!f(u_i)$ co najwyżej tyle, ile razy zostały one zwiększone, czyli linia (*) zostnie wykonana $\leqslant l$ razy dla pewnego wzorca o długości l.

Sumaryczna długość wzorców wynosi n, dlatego podczas budowy automatu linia (*) zostanie wykonana co najwyżej n razy.

Zastanówmy się jeszcze, ile czasu zajmuje wykonanie linii (**). Zauważmy, że przed wykonaniem przypisania, $out(u) = \emptyset$ albo $out(u)$ jest równy indeksowi $\mathcal{L}(u)$. Jakiekolwiek wzorce znajdują się w $out(f(u))$, są one na pewno krótsze niż $\mathcal{L}(u)$, bo $f(u)$ jest bliżej korzenia - zatem zbiory te są rozłączne. Możemy więc przyjąć, że reprezentujemy je przez listy, do których da się dołączać drugą w stałym czasie.

Zatem złożoność czasowa drugiej fazy wynosi $O(n)$.

1.8 Determinizacja automatu

Ze względu na występowanie w automacie funkcji $f(u)$ jest on nie deterministyczny - nie znamy przejść ze wszytskich stanów dla wszytskich możliwych znaków (np. a), będziemy musieli więc czasem wykonać wiele przejść z użyciem funkcji $f(u)$, aż dojdziemy do stanu v , w którym istnieje dobrze określone przejście $g(v, a)$. Można jednak podejść inaczej o budowy automatu - mianowicie wprowadzić funkcję $next(u, a)$, którą wyliczamy w następujący sposób:

Algorithm 1.3: PHASE3(*void*)

$$
\begin{array}{l}
Q \leftarrow \text{QUEUE}() \\
\textbf{for } a \in \Sigma \\
\quad \textbf{do } \begin{cases} \textbf{if } q \leftarrow g(0,a) \neq 0 \\ \quad \textbf{then } \text{Q.ENQUEUE}(q) \\ next(0,a) \leftarrow g(0,a) \end{cases} \\
\textbf{while } !\text{Q.ISEMPTY}() \\
\quad \textbf{do } \begin{cases} r \leftarrow \text{Q.DEQUEUE}() \\ \textbf{for } a \in \Sigma \\ \quad \textbf{do if } u \leftarrow g(r,a) = \emptyset \\ \quad \textbf{then } \begin{cases} \text{Q.ENQUEUE}(u) \\ v \leftarrow f(r) \\ \textbf{while } g(v,a) = \emptyset \\ \quad \textbf{do } v \leftarrow f(v) \\ next(r,a) \leftarrow g(v,a) \end{cases} \\ \quad \textbf{else } \begin{cases} next(r,a) \leftarrow g(r,a) \\ \text{Q.ENQUEUE}(g(r,a)) \end{cases} \end{cases}
\end{array}
$$

Procedurę tę wykonujemy po wyliczeniu funkcji $f(u)$. Wtedy wyszukiwanie upraszcza się do:
Algorithm 1.4: SEARCHDETERMINISTIC($T[0 \dots m-1]$)

$$
\begin{array}{l}
q \leftarrow 0 \\
\textbf{for } i \leftarrow 0 \textbf{ to } m-1 \\
\quad \textbf{do } \begin{cases} q \leftarrow next(q, T[i]) \\ \textbf{comment: } \text{przejście jest deterministyczne} \\ \textbf{if } out(q) \neq \emptyset \\ \quad \textbf{then output } (i), out(q) \end{cases}
\end{array}
$$

Jednak wprowadzenie takich przejść wiąże się ze znacznym obciążeniem pamięciowym, dlatego ja w swojej implementacji pominę te kroki.

2 Opis interfejsu

3 Kod źródłowy

4 Testy

5 Podsumowanie

6 Bibliografia

- `http://pl.wikipedia.org/wiki/Algorytm_Aho-Corasick`
- `http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf`

- http://pl.wikipedia.org/wiki/Deterministyczny_automat_skończony
- http://pl.wikipedia.org/wiki/Pods\OT4\lowo
- https://wiki.python.org/moin/TimeComplexity