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

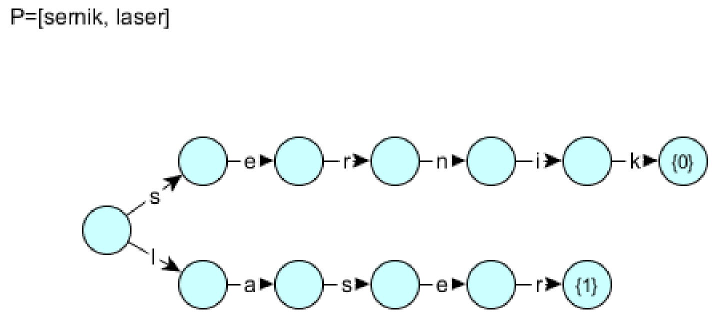
Definicja 1. ***Drzewem trie*** *dla zbioru wzorców* $\mathcal{P}$ *nazywamy takie ukorzenione drzewo* $\mathcal{K}$*, że:*

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

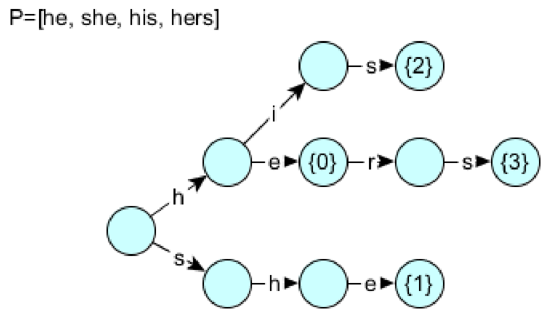
   Definicja 2. ***Etykietą węzła*** v *nazywamy konkatenację etykiet krawędzi znajdujących się na ścieżce z korzenia do* v*. Oznaczamy ją jako* $\mathcal{L}(v)$.

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

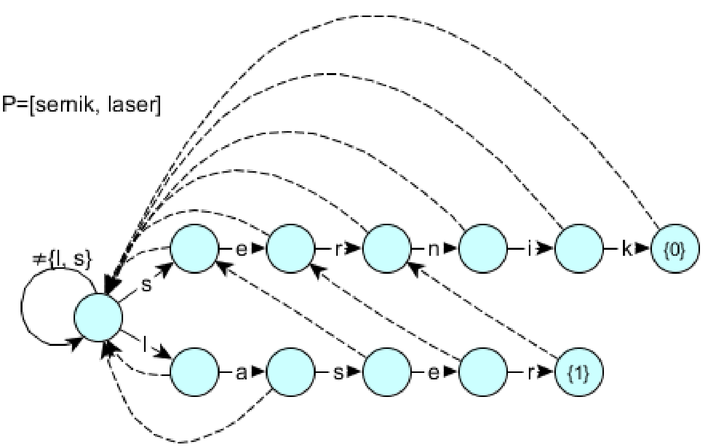
Definicja 3. ***Automatem deterministycznym*** *nazywamy piątkę uporządkowaną* $(\Sigma, Q, q, \delta, F)$, *gdzie*

1. Σ *- skończony alfabet*
2. Q *- skończony zbiór stanów*
3. $q \in Q$*- stan początkowy*
4. $\delta : Q \times \Sigma \to Q$ *- 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 $f(q)$

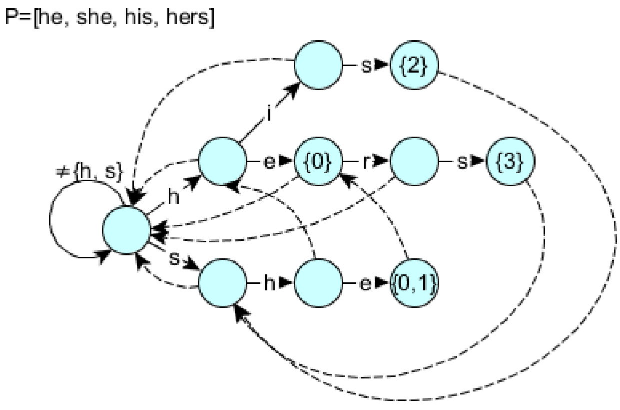
3. Funkcja **wyjścia**, oznaczana jako $out(v)$, zwraca indeksy wzorców, do których znajdujemy dopasowanie w stanie q.

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 $df(u)$ jako głębokość w drzewie wierzchołka $f(u)$, zatem $df(u_1), \ldots, fd(u_l)$ 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 $df(u_{i+1}) \leqslant df(u_i) + 1$, zatem wartości df 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 $df(u_{i+1})$ będzie mniejsza od $df(u_i)$ co najmniej o jeden. W związku z ograniczeniem od dołu, możemy zmniejszać kolejne wartości $df(u_i)$ 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: $\textsc{Phase3}(void)$

$Q \leftarrow \textsc{Queue}()$
for $a \in \Sigma$
  do $\begin{cases} \textbf{if } q \leftarrow g(0,a) \neq 0 \\ \quad \textbf{then } \text{Q.}\textsc{enqueue}(q) \\ next(0,a) \leftarrow g(0,a) \end{cases}$
while $!\text{Q.}\textsc{isEmpty}()$
  do $\begin{cases} r \leftarrow \text{Q.}\textsc{dequeue}() \\ \textbf{for } a \in \Sigma \\ \quad \textbf{do if } u \leftarrow g(r,a) = \emptyset \\ \quad \textbf{then } \begin{cases} \text{Q.}\textsc{enqueue}(u) \\ v \leftarrow f(r) \\ \textbf{while } g(v,a) = \emptyset \\ \quad \textbf{do } v \leftarrow f(v) \\ next(r,a) \leftarrow g(v,a) \end{cases} \\ \quad \textbf{else } \begin{cases} next(r,a) \leftarrow g(r,a) \\ \text{Q.}\textsc{enqueue}(g(r,a)) \end{cases} \end{cases}$

Procedurę tę wykonujemy po wyliczeniu funkcji $f(u)$. Wtedy wyszukiwanie upraszcza się do:

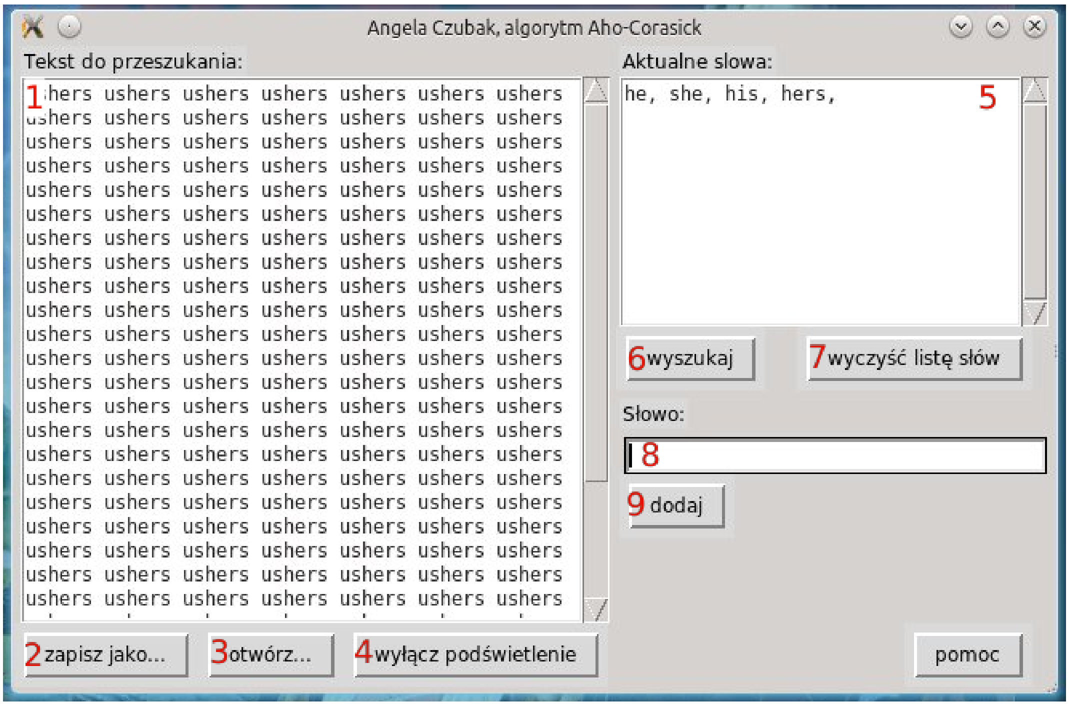
Algorithm 1.4: $\textsc{SearchDeterministic}(T[0 \ldots m-1])$

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

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

2 Opis interfejsu

Zaimplementowano interfejs graficzny ułatwiający korzystanie z napisanego kodu. Został on przedstawiony na rysunku (5).



Rysunek 5: Interfejs graficzny programu zaliczeniowego

Poniżej znajduje się opis poszczególnych elementów interfejsu:

1. TEKST DO PRZESZUKANIA

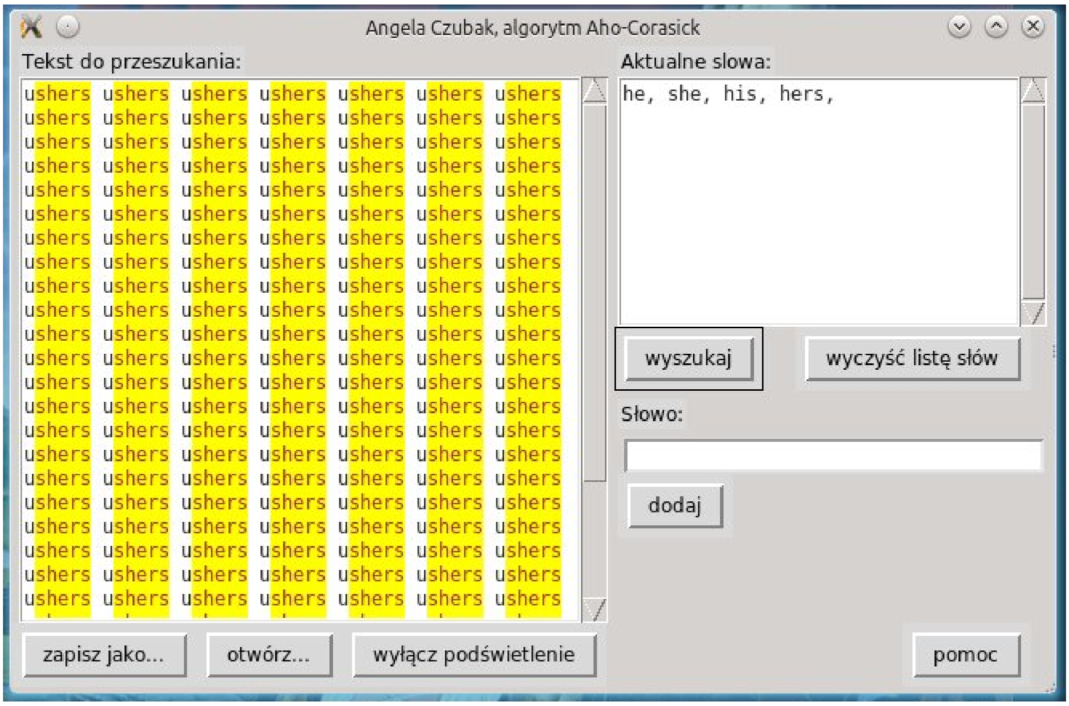
   Jest to pole, w którym umieszczamy tekst, w którym będziemy wyszukiwać wzorce. Możemy tam wprowadzać tekst wprost z klawiatury lub wczytać tekst z pliku. W tym drugim przypadku należy kliknąć przycisk `otwórz...`, a następnie wybrać plik z użyciem okna dialogowego. Tekst, który wprowadzimy do tego pola możemy zapisać. By to zrobić, należy kliknąć przycisk `zapisz jako...`, a następnie wybrać odpowiednią nazwę pliku. Po wykonaniu wyszukania znalezione wzorce zostaną podświetlone na żółto (patrz rysunek (6)). By wyłączyć to podświetlenie, należy kliknąć na przycisk `wyłącz podświetlenie`.

2. ZAPISZ JAKO...

   Tekst, który wprowadzimy do tego pola możemy zapisać. By to zrobić, należy kliknąć przycisk `zapisz jako...`, a następnie wybrać odpowiednią nazwę pliku.

3. OTWÓRZ...

   Zamiast wpisywać tekst, możemy otworzyć gotowy plik tekstowy. W tym celu należy kliknąć przycisk `otwórz...`, a następnie wybrać plik z użyciem okna dialogowego.



Rysunek 6: Interfejs graficzny programu zaliczeniowego, działanie podświetlenia

4. WYŁĄCZ PODŚWIETLENIE

   Po wykonaniu wyszukania znalezione wzorce zostaną podświetlone na żółto (patrz rysunek (6)). By wyłączyć to podświetlenie, należy kliknąć na przycisk `wyłącz podświetlenie`.

5. AKTUALNE SŁOWA

   W tym polu znajdują się słowa (wzorce), które będą wyszukiwane w tekście przy użyciu algorytmu Aho-Corasick. By wyszukać wzorce, należy kliknąć przycisk `wyszukaj`, wtedy znalezione wystąpienia z pola `Tekst do przeszukania` zostaną podświetlone na żółto (patrz rysunek (6)). By wyczyścić listę słów, należy kliknąć przycisk `wyczyść listę słów`. By dodać słowo należy umieścić wymyślony przez nas wzorzec w polu `Słowo`, a następnie wcisnąć `ENTER` na klawiaturze lub przycisk `dodaj`.

6. WYSZUKAJ

   By wyszukać wzorce, należy kliknąć przycisk `wyszukaj`, wtedy znalezione wystąpienia z pola `Tekst` do przeszukania zostaną podświetlone na żółto (patrz rysunek (6)).

7. WYCZYŚĆ LISTĘ SŁÓW

   By wyczyścić listę słów, należy kliknąć przycisk `wyczyść listę słów`.

8. SŁOWO

   W tym polu wpisujemy wzorzec, który chcemy wyszukiwać w tekście znajdującym się w polu `Tekst do wyszukania`. Następnie należy nacisnąć `ENTER` na klawiaturze lub przycisk `dodaj`.

9. DODAJ

   Po wpisaniu słowa w polu `Słowo`, które chcemy wyszukać w tekście znajdującym się w polu `Tekst do wyszukania`, można nacisnąć przycisk `dodaj`, by dodać słowo do listy wzorców.

3 Kod źródłowy

Cały kod oraz opis zmian można podejrzeć na `https://github.com/cosmia/pythonProject`.

3.1 Klasa MyList

Ponieważ złożoność dołączania jednej listy do drugiej w *Pythonie* jest zależna od długości tej drugiej, postanowiono napisać własną implementację listy tak, aby łącznie następowało w czasie stałym.

3.1.1 MyListException

Wyjątek związany z operacjami na obiektach klasy MyList

3.1.2 Element

Klasa opisująca element listy.
Posiada ona następujące **pola**:

1. *arg* - zawartość tego elementu listy
2. *follow* - następny element na liście

Metody:

1. konstruktor *__init__(self, arg=None, follow=None)*- arg - element znajdujący się w liście, $follow$ - następny element na liście
2. $setData(self, arg)$ - ustawienie zawartości elementu listy na arg
3. $setNext(self, follow)$ -ustawienie następnego elementu na liście na $follow$, rzuca wyjątkiem $MyListException$, jeśli $follow$ nie jest klasy *Element*
4. $getData(self)$ - zwraca zawartość elementu listy
5. $getNext(self)$ - zwraca następny element na liście

3.1.3 MyList

Klasa opisująca moją wersję listy.
Posiada ona cztery **pola**:

1. *first* - pierwszy element listy, jeśli lista jest pusta, jest to *None*
2. *last* - ostatni element listy, jeśli jest to lista pusta, jest to *None*
3. *current* - jest to zmienna pomocnicza, używana przy iterowaniu listy, początkowo ustawiona na None
4. *length* - długość listy

Metody:

1. konstruktor bezargumentowy
2. *add(self, argument)* - dodaje element argument do listy na ostatniej pozycji
3. *__add(self, other)__* - dodaje do siebie dwa obiekty *MyList*, zmienia pierwszy obiekt, zwraca wskaźnik na pierwszy obiekt; rzuca *MyListException*, jeśli drugi argument nie jest obiektem *MyList*
4. __iter(self)__ - zwraca iterator dla listy *MyList*
5. *next(self)* - zwraca następny element na liście, metoda dla iteratora
6. *__len(self)__* - zwraca długość listy
7. *__eq(self, other)__* - metoda porownujaca listy, zwraca *True*, jeśli listy równe, *False* wpp
8. *__ne(self, other)__* - metoda sprawdzająca, czy listy są różne, zwraca *True*, jeśli tak; *False* wpp
9. *__contains(self, other)__* - metoda sprawdzająca, czy lista zawiera *other*, zwraca *True*, jeśli tak; *False* wpp

3.1.4 Kod

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

class MyListException(Exception):
    '''wyjatek dla klasy MyList'''
    def __init__(self, mes):
  '''konstruktor, argumentem tresc przy rzucaniu wyjatku'''
  self.value = mes
    def __str__(self):
  '''podaje tresc wyjatku'''
  return self.value

class Element:
    '''klasa opisujaca element MyList'''
    def __init__(self, arg=None, follow=None):
  '''konstruktor; arg - element znajdujacy sie w liscie,
```

```
       follow - nastepny element na liscie'''
  if follow is not None and not isinstance(follow, Element):
      raise MyListException("argument is not an Element")
  self.arg = arg
  self.follow = follow
    def setData(self, arg):
  '''ustawienie zawartosci elementu listy na arg'''
  self.arg = arg
    def setNext(self, follow):
  '''ustawienie nastepnego elementu na liscie na follow'''
  if not isinstance(follow, Element):
      raise MyListException("argument is not an Element")
  self.follow = follow
    def getData(self):
  '''zwraca zawartosc elementu listy'''
  return self.arg
    def getNext(self):
  '''zwraca nastepny element na liscie'''
  return self.follow

class MyList:
    '''lista, ktora bedzie mozna laczyc z druga w czasie stalym
   jest to uproszczona lista, nie zawiera np. usuwania elementow,
   gdyz nie wydaje sie to potrzebne'''
    def __init__(self):
  '''konstruktor, tworzy pusta liste'''
  self.first = None
  self.last = None
  self.current = None
  self.length = 0
    def add(self, argument):
  '''dodaje argument do listy na ostatniej pozycji'''
  if self.first is None:
      self.first = Element(argument, None)
      self.last = self.first
  else:
      tmp = Element(argument, None)
      self.last.setNext(tmp)
      self.last = tmp
  self.length += 1
    def __add__(self, other):
  '''dodaje do siebie dwa obiekty MyList
     zmienia pierwszy obiekt, zwraca wskaznik na pierwszy obiekt'''
  if other is None or other.first is None:
      return self
  if not isinstance(other, MyList):
      raise MyListException("the other argument is not a MyList")
  if self.first is None:
      self.first = other.first
      self.length = other.length
      self.last = other.last
      return self
  #print other.first
  self.last.setNext(other.first)
  self.length += other.length
  return self
    def __iter__(self):
  '''metoda zwracajaca iterator'''
  self.current = self.first
  return self
    def next(self):
  '''zwraca nastepny element na liscie'''
```

```
  if self.current is None:
      raise StopIteration
  else:
      tmp = self.current.getData()
      self.current = self.current.getNext()
      return tmp
    def __len__(self):
  '''metoda zwracajaca dlugosc listy'''
  return self.length
    def __eq__(self, other):
  '''metoda porownujaca listy
     zwraca True, jesli listy rowne, False wpp'''
  if not isinstance(other, MyList):
      return False
  dl = len(other)
  if dl != len(self):
      return False
  iter1 = iter(self)
  iter2 = iter(other)
  for i in range(dl):
      e1 = iter1.next()
      e2 = iter2.next()
      if e2 != e1:
    return False
  return True
    def __ne__(self, other):
  '''metoda sprawdzajaca, czy listy sa rozne
     zwraca True, jesli tak; False wpp'''
  return not self == other
    def __contains__(self, other):
  '''metoda sprawdzajaca, czy lista zawiera other
     zwraca True, jesli tak; False wpp'''
  for i in self:
      if other == i:
    return True
  return False
```

myList.py

3.2 Klasa Node

3.2.1 Kod

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from myList import *

class NodeException(Exception):
    '''wyjatek dla klasy Node'''
    def __init__(self, value):
  '''konstruktor, argumentem tresc przy rzucaniu wyjatku'''
  self.napis = value
    def __str__(self):
  '''podaje powod wyjatku'''
  return repr(self.napis)
    __repr__ = __str__

class Node:
    '''klasa opisujaca wezel/stan w automacie/drzewie Trie'''
    def __init__(self):
```

```
    '''konstruktor bezargumentowy
        accept = MyList() - pusta lista,
        fail = None
        edges = {}'''
    self.accept = MyList()
    self.edges = {}
    self.fail = None
      def labelCorrect(self, label):
    '''sprawdza, czy label jest poprawna etykieta krawedzi
        jesli nie, rzuca NodeException'''
    if not isinstance(label, (str, unicode)):
         raise NodeException("label must be a character")
    if len(label) != 1:
         raise NodeException("label must be exactly one character long
      ")
      def nodeCorrect(self, node, n=1):
    '''sprawdza, czy node jest obiektem klasy Node
        jesli nie, to rzuca NodeException
       n to numer argumentu, ktorym jest node w jakiejs funkcji
          sluzy uszczegolowieniu, ktory argument jest bledny'''
    if not isinstance(node, Node):
         lan = "the "
         if n != 1: lan += "second "
         lan += "argument must be a node"
         raise NodeException(lan)
      def getAccept(self):
    '''zwraca liste indeksow slow, ktore akceptuje ten wezel,
        lista jest pusta, jesli ten wezel niczego nie akceptuje'''
    return self.accept
      def getLabels(self):
    '''zwraca liste etykiet dla krawedzi wychodzacych z tego wezla'''
    return self.edges.keys()
      def getAim(self, label):
    '''zwraca wezel, do ktorego prowadzi krawedz z etykieta label
        jesli brak takiej krawedzi, zwraca None'''
    self.labelCorrect(label)
    if label not in self.edges:
         if self.fail is None:
      return self
         else:
      return None
    else:
         return self.edges[label]
      def getFail(self):
    '''zwraca wezel odpowiadajacy najdluzszemu wlasciwemu sufiksowi
       slowa, ktore do ktorego probowalismy znalezc dopasowanie'''
    return self.fail
      def setAccept(self, number):
    '''ustalamy, ze ten wezel akceptuje slowo o indeksie number lub
      slowa o
       indeksach z MyList number
       rzuca NodeException, jesli number nie jest calkowita liczba
      nieujemna
       albo obiektem MyList calkowitych liczb nieujemnych'''
    if isinstance(number, MyList):
         for i in number:
      if not isinstance(i, (long, int)) or i < 0:
           mes = "argument should be a non-negative integer or long or
       a set of those"
           raise NodeException(mes)
        #print number
         self.accept + number
```

```
      return
  if not isinstance(number, (long, int)):
      raise NodeException("argument should be an integer or long or
    a set of those")
  if number < 0:
      raise NodeException("argument must be non-negative")
  self.accept.add(number)
    def setAim(self, label, node):
  '''ustalamy, ze z tego wezla bedzie wychodzic krawedz
     etykietowana label i bedzie ona prowadzic do node
     rzuca NodeException, jesli label niepoprawna lub node nie jest
     wezlem'''
  self.labelCorrect(label)
  self.nodeCorrect(node,2)
  self.edges[label] = node
    def setFail(self, node):
  '''ustalamy, ze najdluzszy sufiks slowa, do ktorego probowalismy
    dopsowac w tym wezle odpowiada wezlowi node
    rzuca wyjatkiem, jesli node nie jest wezlem'''
  self.nodeCorrect(node)
  self.fail = node
```

node.py

3.3 Klasa AhoCorasick

3.3.1 Kod

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from node import *
from Queue import *

class AhoCorasickException(Exception):
     '''wyjatek dla klasy Node'''
    def __init__(self, value):
  '''konstruktor, argumentem tresc przy rzucaniu wyjatku'''
  self.napis = value
    def __str__(self):
  '''podaj powod wyjatku'''
  return repr(self.napis)

class AhoCorasick:
     '''klasa opisujaca drzewo Trie / automat, sluzacy wyszukiwaniu
    wzorcow'''
    def __init__(self):
  '''konstruktor bezargumentowy
     n - korzen drzewa, pusty
     words - pusta liczba slow zakodowanych w drzewie'''
  self.n = Node()
  self.words = []
  self.built = False
    def addWord(self, word):
  '''dodaje slowo word do automatu/drzewa
     rzuca AhoCorasickException, jesli word nie jest stringiem
        lub zbudowano juz automat'''
  if self.built:
      raise AhoCorasickException("automaton has been built already"
    )
  if not isinstance(word, (str, unicode)):
```

```
        raise AhoCorasickException("argument is not a string")
    dl = len(word)
    if dl == 0: return #nie dodajemy pustego slowa
    wezel = self.n
    i = 0
    #idziemy dopoki mozemy po istniejacych wezlach
    while i < dl:
        litera = word[i]
        labels = wezel.getLabels()
        if litera in labels:
      wezel = wezel.getAim(litera)
        else:
      break
        i += 1
    #a teraz tworzymy nowe, jesli taka potrzeba
    while i < dl:
        litera = word[i]
        wezel.setAim(litera, Node())
        wezel = wezel.getAim(litera)
        #wezel.setFail(self.n) #na poczatku najdluzszy wlasciwy
      sufiks to slowo puste
        #mozna to w sumie robic przy budowaniu automatu...
        i += 1
    #jesli jeszcze nie dodalismy tego slowa
    if wezel.getAccept() == MyList():
        ktore = len(self.words)
        wezel.setAccept(ktore)
        self.words.append(word)
      def lookUp(self, word):
    '''sprawdza, czy dane slowo wystepuje w drzewie
     zwraca True, jesli tak; False wpp
     rzuca AhoCorasickException, jesli word nie jest strigiem'''
    if not isinstance(word, str):
        raise AhoCorasickException("argument is not a string")
    if word == "": return False
    i = 0
    dl = len(word)
    wezel = self.n
    while i < dl:
        litera = word[i]
        labels = wezel.getLabels()
        if litera not in labels:
      return False
        wezel = wezel.getAim(litera)
        i += 1
    if wezel.getAccept() != MyList():
        return True
    return False
      def build(self):
    '''konstruuje automat skonczony na podstawie drzewa, ktore
     powstaje podczas dodawania slow metoda addWord'''
    q = Queue()
    for i in self.n.getLabels():
        s = self.n.getAim(i)
        s.setFail(self.n)
        q.put(s)
    while not q.empty():
        r = q.get()
        for a in r.getLabels():
      u = r.getAim(a)
      q.put(u)
      v = r.getFail()
```

```
    while v.getAim(a) is None: #jesli stad nie ma tego przejscia
        v = v.getFail()       #to szukaj krotszego dopasowania
    u.setFail(v.getAim(a))
    u.setAccept(u.getFail().getAccept()) #dodaj nowe akceptowane
    slowa
  self.built = True
    def makeTree(self, wordList):
  '''konstruuje drzewo i automat na podstawie listy slow wordList
     takze dodaje do istniejacego automatu wzorce z wordList
     rzuca AhoCorasickException, jesli wordList nie jest lista
    stringow
       lub zbudowano juz automat'''
  if self.built:
      raise AhoCorasickException("automaton has been built already"
    )
  if not isinstance(wordList, list):
      raise AhoCorasickException("argument is not a list")
  for i in wordList:
      if not isinstance(i, (str, unicode)):
    raise AhoCorasickException("element of the list is not a string
    ")
  for i in wordList:
      self.addWord(i)
  self.build()
    def clear(self):
  '''czysci automat i drzewo'''
  self.words = []
  self.n = Node()
  self.built = False
    def search(self, tekst, returnSet=False):
  '''wyszukuje wzorce w zmiennej tekst, zwraca string z wiadomoscia
     o wynikach
     domyslny argument returnSet mowi o formacie zwracanej wartosci
     jesli returnSet jest False, to zwracamy string z informacjami
     jesli returnSet jest True, to zwracamy zbior krotek o dlugosci
    dwa, krotka
        zawiera pozycje, na ktorej znalazla slowo, oraz indeks
    slowa
     jesli tekst nie jest zmienna string, to rzuca
    AhoCorasickException'''
  if not isinstance(tekst, (str, unicode)):
      raise AhoCorasickException("argument is not a string")
  node = self.n
  if not returnSet: message = ""
  else: message = set()
  dl = len(tekst)
  for i in range(dl):
      while not node.getAim(tekst[i]):
    node = node.getFail()
      node = node.getAim(tekst[i])
      if node.getAccept() != set():
    zbior = node.getAccept()
    for j in zbior:
        if returnSet:
      message.add((i,j))
        else:
      message += "Found \""+self.words[j]+"\" in position "+str(i)+
    "\n"
  if not returnSet and message == "":
      message = "Nothing found\n"
  if not returnSet: message = message[:len(message)-1]
  return message
```

ahoCorasick.py

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