

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 $\mathcal{P} = \{P_0, \dots, P_k\}$ 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:*

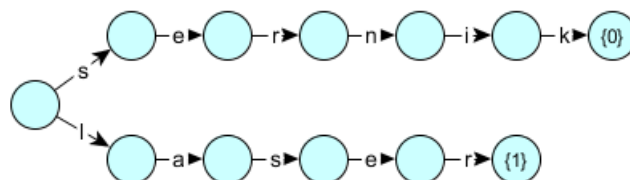
1. *Każda krawędź drzewa \mathcal{K} jest etykietowana jakimś znakiem*
2. *Każde dwie krawędzie wychodzące z jednego wierzchołka mają różne etykiety*

Definicja 2. *Etykietą węzła v nazywamy konkatencję 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 $P \in \mathcal{P}$*

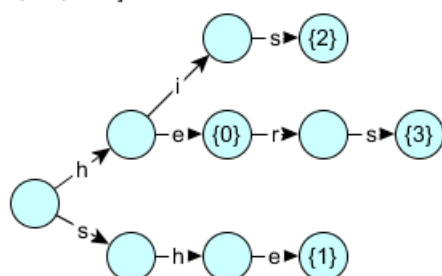
Przykładowe rysunki drzew trie znajdują się na następnej stronie. Numer w wierzchołku oznacza indeks słowa należącego do słownika, które jest etykietą tego wierzchołka.

$P=[\text{semik, laser}]$



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

$P=[\text{he, she, his, hers}]$



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

1.3 Konstrukcja drzewa trie

Jak budować drzewo trie dla $\mathcal{P} = \{P_0, \dots, P_k\}$? 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 $O(|P_0| + \dots + |P_k|) = O(n)$.

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 w 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 tekstu, n - suma długości wzorców.

By przyspieszyć wyszukiwanie wzorców, rozszerzamy drzewo trie do **automatu**.

1.5 Automat

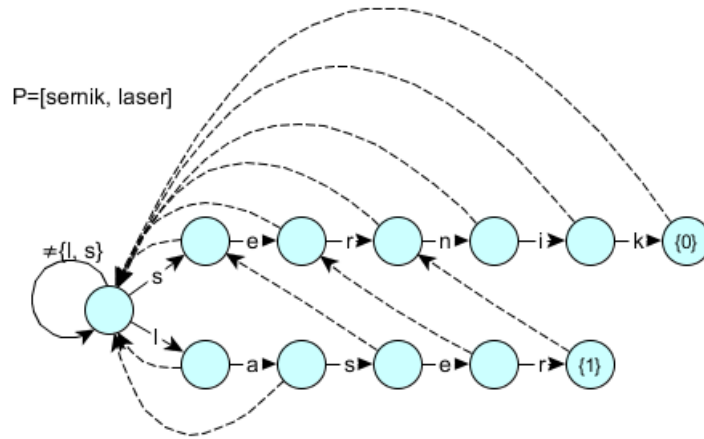
Definicja 3. *Automatem deterministycznym* nazywamy piątkę uporządkowaną $(\Sigma, Q, q, \delta, F)$, gdzie

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 pojęć, nie będzie ściśle deterministyczny. Automat będziemy budować na podstawie drzewa trie, zatem uściślam: zbiór stanów będą stanowiący 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 $\mathcal{P} = \{\textit{sernik}, \textit{laser}\}$. 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 $T[1 \dots]$, w którym szukamy wzorców. Procedura ta prezentuje się w następujący sposób:

Algorithm 1.1: SEARCH($T[0 \dots m-1]$)

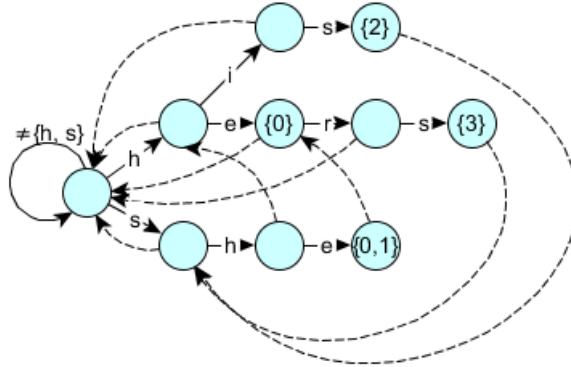
```

 $q \leftarrow 0$ 
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):

$P=[he, she, his, hers]$



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 wywołań 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 ← QUEUE()
for a ∈ Σ
  do { if q ← g(0, a) ≠ 0
        then { f(q) ← 0
              Q.ENQUEUE(q)
      while !Q.ISEMPTY()
        { r ← Q.DEQUEUE()
          for a ∈ Σ
            do if u ← g(r, a) ≠ ∅
                  { Q.ENQUEUE(u)
                    v ← f(r)
                    while g(v, a) = ∅
                      do v ← f(v) // (*)
                    f(u) ← g(v, a)
                    out(u) ← out(u) ∪ 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 sufix $\mathcal{L}(u)$. Z tego wynika, że powinniśmy spróbować dopasować $f(u) = g(f(r), a)$, bo $\mathcal{L}(f(u))$ może być sufiksem $\mathcal{L}(g(f(r), a))$, 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 rozmiaru drzewa - tj. $O(n)$. A jak określić, ile razy wykonamy linię (*)?

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

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

Kiedy wyliczamy położenie $f(u_{i+1})$, każde wywołanie linii (*) przybliży 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 (*) zostanie wykonana $\leq l$ 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 $\mathcal{L}(u)$. Jakielikolwiek 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 wszystkich stanów dla wszystkich 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*)

```

Q ← QUEUE()
for a ∈ Σ
  do { if q ← g(0, a) ≠ 0
        then Q.ENQUEUE(q)
        next(0, a) ← g(0, a)
  while !Q.ISEMPTY()
    do { r ← Q.DEQUEUE()
          for a ∈ Σ
            do if u ← g(r, a) = ∅
                  then { Q.ENQUEUE(u)
                          v ← f(r)
                          while g(v, a) = ∅
                            do v ← f(v)
                          next(r, a) ← g(v, a)
                        else { next(r, a) ← g(r, a)
                              Q.ENQUEUE(g(r, a))

```

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

Algorithm 1.4: SEARCHDETERMINISTIC($T[0 \dots m-1]$)

```

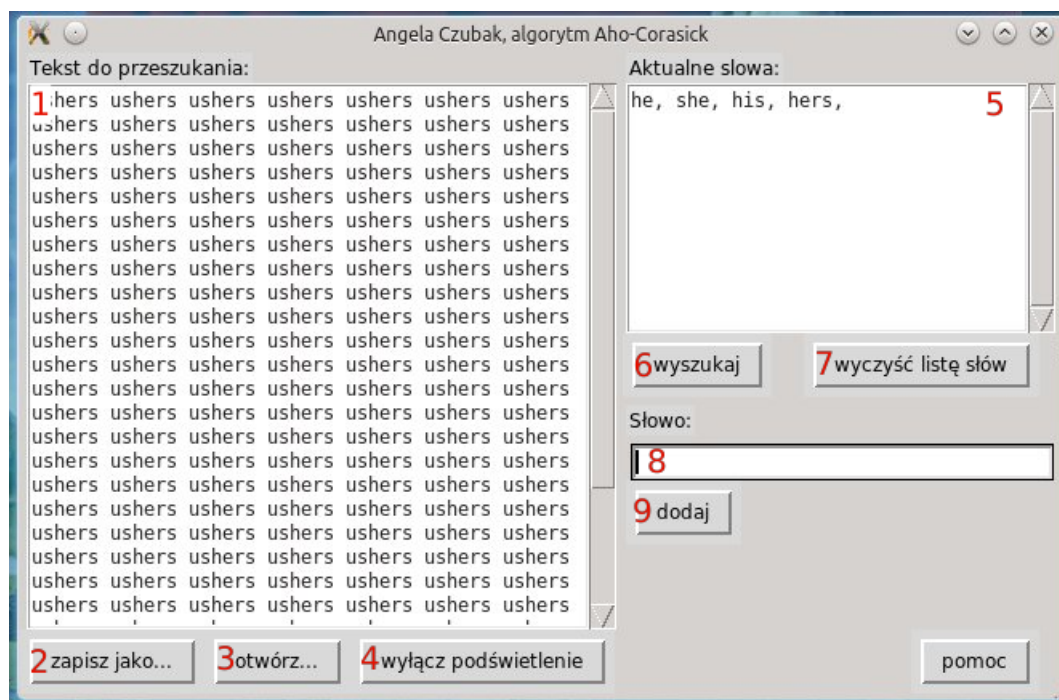
q ← 0
for i ← 0 to m-1
  do { q ← next(q, T[i])
        comment: przejście jest deterministyczne
        if out(q) ≠ ∅
          then output (i), out(q)

```

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

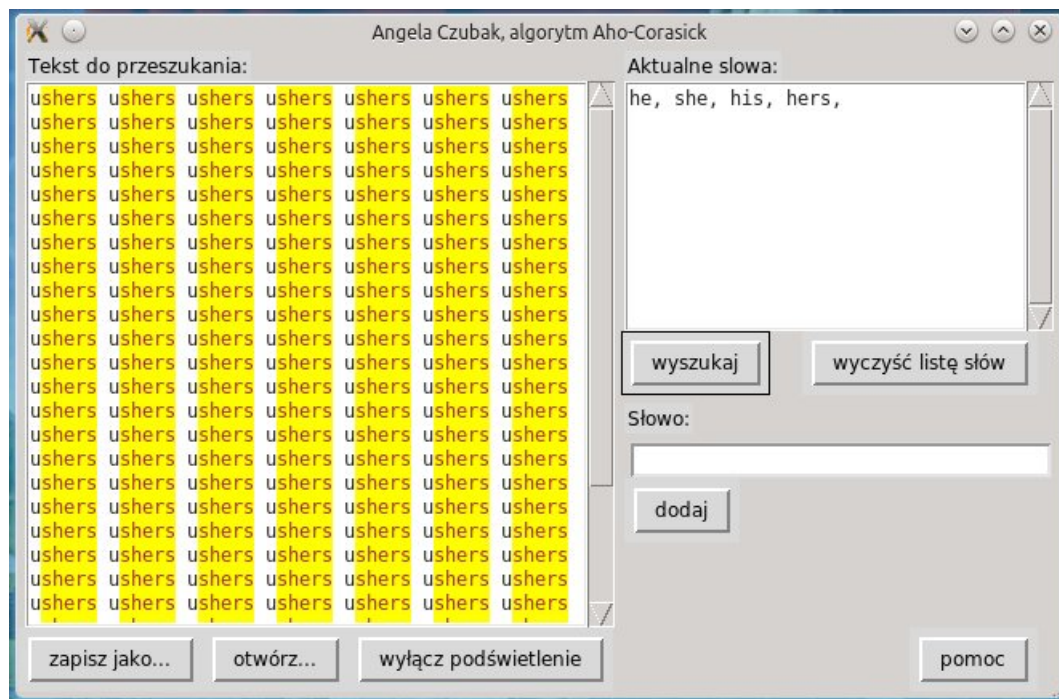
Jest to pole, w którym umieszczamy tekst, w którym będziemy wyszukiwać wzorce. Możemy tam wprowadzać tekst wprost z klawiatury lub wczytać tekst z pliku. W tym drugim przypadku należy kliknąć przycisk **otwórz...**, a następnie wybrać plik z użyciem okna dialogowego. Tekst, który wprowadzimy do tego 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 wyszukiwania 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

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/Podsłowo>
- <https://wiki.python.org/moin/TimeComplexity>