Sprawozdanie z laboratorium 2 z algorytmów tekstowych

Hubert Miklas 03-04-2025

Wstęp

Tematem laboratorium były wyrażenia regularne. Do każdego zadania przedstawiony jest szczegółowy opis rozwiązania.

Zadanie 1 - Parsowanie cytowań artykułów naukowych

Poniżej opisany kod służy do ekstrakcji informacji z cytowania publikacji naukowych w zadanym formacie. Korzstając z RegEx należało wydobyć określone dane - imiona i nazwiska autorów, rok publikacji, tytuł, nazwa czasopisma, tom, numer oraz zakres stron.

Opis działania kodu

1. Definicja wyrażeń regularnych

a) Wzorzec dla autorów i roku publikacji

```
authors_year_pattern = r"(?P<authors>(([A-ZŻŹĆĄŚĘŁÓŃ][a-zżźćńółąśę]+\s*,\s*[A-ZŻŹĆĄŚĘŁÓŃ]\.\s*),{0,1}\s*)+)\((?P<year>\d{4})\)"
```

Opis:

- `(?P<authors>(...)+)` : Grupa nazwanych autorów, zawierająca jeden lub więcej autorów.
- `([A-ZŻŹĆĄŚĘŁÓŃ][a-zżźćńółąśę]+\s*,\s*[A-ZŻŹĆĄŚĘŁÓŃ]\.)` : Dopasowanie nazwiska i inicjału autora.
- `,{0,1}\s*` : Opcjonalna dodatkowa przecinek i odstęp dla kolejnych autorów.
- `\((?P<year>\d{4})\)` : Rok w nawiasach jako liczba czterocyfrowa.

b) Wzorzec dla tytułu i nazwy czasopisma

```
title_journal_pattern = r"\.\s*(?P<title>.+?)\.\s*(?P<journal>.+?),"
```

Opis:

- `\.\s*` : Dopasowanie kropki i ewentualnych odstępów. Kropka w tym kotekście oznacza koniec listy z autorami.
- `(?P<title>.+?)` : Dopasowanie tytułu, ze względu na dowolne znaki dopasowanie jest przez wildcard (.).
- `\.\s*` : Kolejna kropka i odstępy.
- `(?P<journal>.+?),` : Dopasowanie nazwy czasopisma do przecinka.

c) Wzorzec dla numeru tomu, numeru czasopisma i stron

```
volume_issue_pages_pattern = r"\s*(?P<volume>\d+)(?:\((?P<issue>\d+)\))?,\s*(?P<start_page>\d+)-(?P<end_page>\d+)\."
```

Opis:

- `(?P<volume>\d+)` : Dopasowanie numeru tomu.
- `(?:\((?P<issue>\d+)\))?` : Opcjonalne dopasowanie numeru wydania.
- `(?P<start_page>\d+)-(?P<end_page>\d+)` : Dopasowanie zakresu stron.
- `.\` : Kolejna sekcja zakończona kropką.

2. Łączenie wzorców w jedno wyrażenie

```
full_pattern = authors_year_pattern + title_journal_pattern + volume_issue_pages_pattern
```

Złożenie wszystkich wzorców w jeden kompletny wzorzec dla całej referencji.

3. Dopasowanie wzorca do podanego tekstu

```
matches = re.match(full_pattern, reference)
if not matches:
    return None
```

Jeśli `reference` nie pasuje do wzorca, funkcja zwraca `None`.

4. Wydobycie listy autorów

```
authors_text = matches.group("authors")
author_pattern = r"(?P<last_name>[A-ZŻŹĆĄŚĘŁÓŃ][a-zżźćńółąśę]+),\s*(?P<initial>[A-ZŻŹĆĄŚĘŁÓŃ])\.,{0,1}\s"
authors_list = []
for author in re.finditer(author_pattern, authors_text):
    authors_list.append({"last_name": author.group('last_name'), "initial": author.group('initial')})
```

Opis:

- `authors_text` zawiera całą część referencji dotyczącą autorów.
- `author_pattern` dopasowuje pojedynczego autora.
- `re.finditer()` iteruje przez wszystkich autorów i tworzy listę słowników z nazwiskiem i inicjałem.

5. Tworzenie wyniku w formie słownika

```
result = {
    "authors": authors_list,
    "year": int(matches.group("year")),
    "title": matches.group("title"),
    "journal": matches.group("journal"),
    "volume": int(matches.group("volume")),
    "issue": int(matches.group("issue")) if matches.group("issue") else None,
    "pages": {
        "start": int(matches.group("start_page")),
        "end": int(matches.group("end_page")),
    },
}
```

Dane są wyodrębniane z dopasowań i konwertowane na odpowiednie typy:

- Lista słowników dla autorów.
- Liczba całkowita dla roku, tomu i numeru (jeśli istnieje).
- Słownik dla stron zawierający wartości liczbowe.

Funkcja zwraca kompletny słownik z wyodrębnionymi informacjami.

Kod został przetestowany i przeszedł wszystkie testy.

Zadanie 2 - Ekstrakcja linków z HTML

Poniżej opisany kod służy do analizy fragmentu kodu HTML i wyodrębniania wszystkich linków (`<a>` tagów). Wykorzystuje wyrażenia regularne (`RegEx`) do dopasowania oraz ekstrakcji kluczowych informacji, takich jak:

- `href` – adres URL,
- `title` – wartość atrybutu `title` (jeśli istnieje),
- `text` – tekst wyświetlany jako link.

Szczegółowy opis działania kodu

1. Definicja wyrażenia regularnego

```
pattern = r"<a\s+href=\"(?P<url>http(s){0,1}:\/\/[^\"]+)\"(\s+title\s*=\s*\"(?P<title>(.+))\"){0,1}>(?P<text>[A-ZŻŹĆĄŚĘŁÓŃa-zżźćńółąśę\s,.]+)<\/a>"
```

Opis:

- `<a\s+` – Dopasowuje tag `<a>` z ewentualnymi spacjami.
- `href=\"(?P<url>http(s){0,1}:\/\/[^\"]+)\"` – Grupuje wartość `href` jako `url`:
  - `http(s){0,1}:\/\/` – Obsługuje `http://` oraz `https://`.
  - `[^\"]+` – Dopasowuje dowolny ciąg znaków poza `"`.
- `(\s+title\s*=\s*\"(?P<title>(.+))\"){0,1}` – Opcjonalne dopasowanie atrybutu `title`:
  - `\s+title\s*=\s*\"(?P<title>(.+))\"` – Pobiera wartość `title`.
  - `{0,1}` – Wskazuje, że `title` może, ale nie musi występować.
- `>(?P<text>[A-ZŻŹĆĄŚĘŁÓŃa-zżźćńółąśę\s,.]+)<\/a>` – Dopasowuje tekst linku.

2. Iteracja po dopasowaniach i ekstrakcja danych

```
searched = ["title","url","text"]
links = []
for element in re.finditer(pattern,html):
    dic = {}
    for name in searched:
        dic[name] = element.groupdict().get(name)
    links.append(dic)
```

Proces:

1. `re.finditer(pattern, html)` – Znajduje wszystkie dopasowania w tekście.
2. Dla każdego dopasowania:
   - Tworzy pusty słownik `dic`.
   - Iteruje przez listę `searched` (`title`, `url`, `text`).
   - Pobiera wartości z `groupdict()` i dodaje je do słownika `dic`.
   - Dodaje słownik `dic` do listy `links`.

Funkcja zwraca listę słowników, gdzie każdy słownik zawiera kluczowe informacje o znalezionych linkach.

Kod został przetestowany i przeszedł wszystkie testy.

Zadanie 3 - Analiza publikacji w formacie piku tekstowego

Poniżej opisany kod analizuje plik tekstowy pod kątem statystyk i wzorców, wykorzystując RegEx do ekstrakcji dat i adresów email, a także do zliczenia słów, zdań i paragrafów. Poniżej znajduje się szczegółowy opis procesu parsowania pliku oraz zastosowanych wyrażeń regularnych.

Opis działania kodu

1. Wyodrębnienie słów

Zastosowane wyrażenie regularne wyszukuje wszystkie słowa składające się z małych i wielkich liter. Przetworzone słowa są konwertowane na małe litery dla ułatwienia dalszej analizy, w szczególności przy usuwaniu 'stop words'.

```
word_regex = r'(?P<word>([A-Z]|[a-z])[a-z]*)(\s|\,|\.|\!|\?)'
words = [word.group("word").lower() for word in re.finditer(word_regex, content)]
word_count = len(words)
```

Opis

1. `(?P<word>...)` – Grupę `word`, umożliwiając późniejszy dostęp do dopasowanego słowa.
2. `([A-Z]|[a-z])` – Słowo musi zaczynać się od litery (wielkiej lub małej).
3. `[a-z]*` – Po pierwszej literze mogą wystąpić dowolne małe litery (zero lub więcej razy).
4. `(\s|\,|\.|▯|\?)` – Słowo kończy się białym znakiem lub znakami interpunkcyjnymi: `, . ! ?`.

2. Podział tekstu na zdania

Funkcja wyszukuje zdania na podstawie znaków interpunkcyjnych takich jak `.`, `!`, `?`, a następnie liczy ich liczbę.

```
sentence_pattern = r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?|\!)\s'
sentences = [element for element in re.finditer(sentence_pattern, content)]
sentence_count = len([s for s in sentences if str(s).strip()])
```

Opis

1. `(?<!\w\.\w.)` – Negatywny lookbehind, ignorujący skróty takie jak `U.S.A.`.
2. `(?<![A-Z][a-z]\.)` – Negatywny lookbehind ignorujący skróty jak `Dr.` czy `Mr.`.
3. `(?<=\.|\?|\!)\s` – Zdanie kończy się `.`, `?` lub `!`, po którym następuje spacja.

3. Szukanie adresów e-mail

Adresy e-mail są wykrywane za pomocą wyrażenia regularnego, które obsługuje zarówno standardowe domeny, jak i złożone nazwy użytkowników.

```
email_pattern = r"([A-Za-z\-]+\.)*[A-Za-z\-]+@([A-Za-z\-]+\.)+[A-Za-z\-]+"
emails = [email.group() for email in re.finditer(email_pattern, content)]
```

Opis

1. `([A-Za-z\-]+\.)*` – Opcjonalna część identyfikatora użytkownika.
2. `[A-Za-z\-]+@` – Nazwa użytkownika zawierająca litery i `-`, zakończona `@`.
3. `([A-Za-z\-]+\.)+[A-Za-z\-]+` – Domena, np. `gmail.com`.

4. Liczenie częstości występowania słów

Lista słów jest filtrowana tak, aby usunąć często występujące angielskie słowa (stop words), a następnie zliczana jest ich częstość występowania.

```
stop_words = [...]  # Cały zbiór stop words
new_words_order = filter(lambda word: word not in stop_words, words)
frequent_words = {}
for word in new_words_order:
    if word not in frequent_words:
        frequent_words[word] = 1
    else:
        frequent_words[word] += 1
most_frequent_words = {key: value for key, value in list(filter(lambda x: x[1] > 1, sorted(frequent_words.items(), key=lambda x: x[1], reverse=True)))}
```

5. Ekstrakcja dat w różnych formatach

Funkcja wykrywa daty zapisane w różnych formatach, np. `YYYY-MM-DD`, `DD.MM.YYYY`, `MM/DD/YYYY`, itd.

```
date_patterns = [
    r"(\d{4})-(0[1-9]|1[0-2])-([0-2][0-9]|3[0-1])",  # YYYY-MM-DD
    r"([0-2][0-9]|3[0-1])-(0[1-9]|1[0-2])-(\d{4})",  # DD-MM-YYYY
    r"(0[1-9]|1[0-2])-([0-2][0-9]|3[0-1])-(\d{4})",  # MM-DD-YYYY
    r"([0-2][0-9]|3[0-1])\.(0[1-9]|1[0-2])\.(\d{4})",  # DD.MM.YYYY
    r"(0[1-9]|1[0-2])\/([0-2][0-9]|3[0-1])\/(\d{4})",  # MM/DD/YYYY
    r"(\d{4})\/(0[1-9]|1[0-2])\/([0-2][0-9]|3[0-1])",  # YYYY/MM/DD
    r"(\d{4})\.(0[1-9]|1[0-2])\.([0-2][0-9]|3[0-1])",  # YYYY.MM.DD
    r"(January|February|March|April|May|June|July|August|September|October|November|December) ([0-2][0-9]|3[0-1]), (\d{4})",  # Month DD, YYYY
    r"(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) ([0-2][0-9]|3[0-1]), (\d{4})"  # MMM DD, YYYY
]
dates = []
for pattern in date_patterns:
    dates += [date.group() for date in re.finditer(pattern, content)]
```

6. Podział tekstu na akapity

Akapity są wykrywane na podstawie pustych linii. Dla każdego akapitu liczona jest liczba słów.

```
paragraphs = re.finditer(r'.+\n+', content)
paragraph_sizes = {i: count_words(word_regex, paragraph.group()) for i, paragraph in enumerate(paragraphs)}
paragraph_sizes = {key: value for key, value in paragraph_sizes.items() if value != 0}
```

Opis

- `.+\n+` - Wyszukuje niepuste linie zakończone znakiem nowej linii (`\n`).
- `count_words()` - Oblicza liczbę słów w danym akapicie.

7. Zwrócenie wyników

Funkcja zwraca słownik z wszystkimi zgromadzonymi statystykami.

```
result = {
    "word_count": word_count,
    "sentence_count": sentence_count,
    "emails": emails,
    "frequent_words": most_frequent_words,
    "dates": dates,
    "paragraph_sizes": paragraph_sizes,
}
return result
```

Funkcja zwraca wymagany słownik, w którym znajduje się liczba słów, liczba zdań, wykryte adresy email, częste słowa z wykluczeniem 'stop words', wykryte datu i wielkości paragrafów.

Kod został przetestowany i przeszedł wszystkie testy.

Zadanie 4 - Implementacja uproszczonego parsera wyrażeń regularnych do DFA

Celem tego zadania jest zaimplementowanie algorytmu Brzozowskiego, który konwertuje wyrażenia regularne na deterministyczny automat skończony (DFA). Należało zaimplementować lub przeanalizować zadane fragmenty:

1. **Implementacje metody `nullable()`** dla różnych typów wyrażeń regularnych – określenie, czy dane wyrażenie akceptuje pusty ciąg.
2. **Obliczanie pochodnej wyrażenia względem symbolu (`derivative()`)**, co pozwala określić, jak wyrażenie reaguje na przetworzenie pierwszego symbolu.
3. **Uproszczenie (`simplify()`)** struktury wyrażenia w celu eliminacji zbędnych operacji i poprawy wydajności.
4. **Budowę automatu DFA (`build_dfa()`)** poprzez iteracyjne stosowanie pochodnych Brzozowskiego i przechowywanie stanów w postaci wyrażeń regularnych.

Struktura kodu

Kod składa się z kilku klas reprezentujących wyrażenia regularne oraz funkcji obsługujących ich właściwości:

- `RegEx` – klasa abstrakcyjna dla wszystkich wyrażeń regularnych.
- `Empty` – reprezentacja pustego języka.
- `Epsilon` – reprezentacja pustego ciągu (elementu neutralnego względem konkatenacji).
- `Symbol` – reprezentacja pojedynczego symbolu.
- `Concatenation` – reprezentacja konkatenacji dwóch wyrażeń.
- `Alternative` – reprezentacja alternatywy dwóch wyrażeń.
- `KleeneStar` – reprezentacja gwiazdki Kleene’a.
- `DFA` – klasa reprezentująca automat deterministyczny.
- `simplify()` – funkcja upraszczająca wyrażenia.
- `build_dfa()` – funkcja implementująca algorytm Brzozowskiego do budowy DFA.

Implementacja metod `nullable()`

Metoda `nullable()` określa, czy dane wyrażenie regularne akceptuje pusty ciąg (`ε`).

- **Empty**: `∅` nie akceptuje pustego ciągu -> `False`.
- **Epsilon**: `ε` akceptuje pusty ciąg -> `True`.
- **Symbol**: pojedynczy symbol nigdy nie akceptuje pustego ciągu -> `False`.
- **Concatenation**: konkatenacja `r · s` akceptuje pusty ciąg, jeśli oba składniki są nullable -> `r.nullable() and s.nullable()`.
- **Alternative**: alternatywa `r | s` akceptuje pusty ciąg, jeśli **co najmniej jedno** z wyrażeń jest nullable -> `r.nullable() or s.nullable()`.
- **KleeneStar**: `r*` zawsze akceptuje pusty ciąg -> `True`.

Implementacja metody `derivative()`

Metoda `derivative(symbol)` oblicza pochodną Brzozowskiego wyrażenia względem podanego symbolu. Idea polega na tym, by sprawdzić, jak wyrażenie zmienia się po przetworzeniu pierwszego symbolu.

- **Empty**: Pochodna pustego języka to nadal pusty język -> `∅`.

- **Epsilon**: `ε` nie ma żadnej treści do przetworzenia, więc pochodna to `∅`.

- **Symbol**: `D(a, a) = ε`, `D(a, b) = ∅`.

- **Concatenation**: Pochodna `rs` względem `a` to:

  `D(rs, a) = D(r, a) · s + δ(r) · D(s, a)`
  gdzie `δ(r) = ε` jeśli `r.nullable()`, w przeciwnym razie `∅`.

- **Alternative**: `D(r|s, a) = D(r, a) | D(s, a)`.

- **KleeneStar**: `D(r*, a) = D(r, a) · r*`.

Uproszczanie wyrażeń (`simplify()`)

Funkcja `simplify()` upraszcza wyrażenia według reguł:

- `r | ∅ = r`, `∅ | r = r`
- `r ∅ = ∅`, `∅ r = ∅`
- `r ε = r`, `ε r = r`
- `(r*)* = r*`, `ε* = ε`, `∅* = ε`

Budowa automatu DFA (`build_dfa()`)

Funkcja `build_dfa()` tworzy automat DFA na podstawie wyrażenia regularnego poprzez:

1. Rozpoczęcie od początkowego wyrażenia jako pierwszego stanu.
2. Iteracyjne obliczanie pochodnych dla każdego symbolu alfabetu.
3. Przypisanie unikalnych stanów do wynikowych wyrażeń.
4. Kontynuowanie, aż wszystkie stany zostaną przetworzone.
5. Ustalanie stanów akceptujących jako tych, które są `nullable()`.

Przykład użycia:

```
# Wyrażenie regularne: (a|b)*abb
regex = Concatenation(
    Concatenation(
        Concatenation(
            KleeneStar(Alternative(Symbol('a'), Symbol('b'))),
            Symbol('a')
        ),
        Symbol('b')
    ),
    Symbol('b')
)

# Sprawdzenie, czy łańcuch pasuje do wyrażenia
dfa = build_dfa(regex, {'a', 'b'})
assert dfa.accepts("abb") == True
assert dfa.accepts("aabb") == True
assert dfa.accepts("babb") == True
assert dfa.accepts("ab") == False
```

Kroki algorytmu:

1. **Inicjalizacja:** Rozpoczynamy od zainicjalizowania pierwszego stanu jako reprezentacji początkowego wyrażenia regularnego.
2. **Przetwarzanie stanów:**
   - Dla każdego stanu oraz każdego symbolu alfabetu:
     A. Obliczamy pochodną wyrażenia regularnego względem danego symbolu.
     B. Upraszczamy otrzymane wyrażenie regularne.
     C. Dodajemy przejście od aktualnego stanu do nowego stanu odpowiadającego uzyskanemu wyrażeniu regularnemu.
3. **Określenie stanów akceptujących:**
   - Stany są akceptujące, jeśli ich odpowiadające wyrażenie regularne jest puste (nullable).
4. **Powtarzanie procesu:**
   - Kontynuujemy przetwarzanie aż do momentu, gdy nie zostaną odkryte nowe stany.

Implementacja

Inicjalizacja struktur danych

```
states = set()  # Zbiór stanów (q0, q1, ...)
    state_to_regex = {}  # Mapa stanów do odpowiadających im wyrażeń regularnych
    accept_states = set()  # Zbiór stanów akceptujących
    transitions = {}  # Mapa przejść (stan, symbol) -> nowy stan
    regex_to_state = {}  # Mapa wyrażeń regularnych do stanów

    state_counter = 0  # Licznik stanów
```

- `states`: przechowuje unikalne identyfikatory stanów.
- `state_to_regex`: odwzorowuje stany na odpowiadające im wyrażenia regularne.
- `accept_states`: zawiera stany akceptujące.
- `transitions`: mapa przejść `(stan, symbol) -> nowy stan`.
- `regex_to_state`: odwzorowuje tekstową reprezentację wyrażenia regularnego na stan.
- `state_counter`: licznik unikalnych identyfikatorów stanów.

Funkcja pomocnicza do sprawdzania obecności elementu w liście

```
def is_in(element, array):
        for ael in array:
            if element == ael:
                return True
        return False
```

- Ta funkcja iteruje po tablicy `array` i sprawdza, czy dany `element` znajduje się w niej.
- Jest używana do sprawdzania, czy dany regex już został odwiedzony.

Przetwarzanie stanów za pomocą BFS

```
state_queue = deque()
    state_queue.append(regex)

    while len(state_queue) != 0:
        current_state: RegEx = state_queue.pop()
        states.add(state_counter)
```

- `state_queue`: kolejka stanów do przetworzenia.
- W pętli `while`, pobieramy stan z kolejki i dodajemy go do zbioru `states`.

Określanie stanów akceptujących

```
if current_state.nullable():
            accept_states.add(state_counter)
```

- Sprawdzamy, czy bieżący stan (`current_state`) akceptuje ciąg pusty (`nullable`). Jeśli tak, oznacza to, że stan ten jest akceptujący i dodajemy go do `accept_states`.

Tworzenie przejść

```
state_to_regex[state_counter] = str(current_state)
        regex_to_state[str(current_state)] = state_counter

        for symbol in alphabet:
            new_state = current_state.derivative(symbol)

            if not is_in(str(new_state), list(map(lambda x: x[0], regex_to_state.items()))):
                state_queue.append(new_state)

            transitions[(state_counter, symbol)] = new_state
```

- `current_state.derivative(symbol)`: obliczamy pochodną wyrażenia regularnego względem kolejnego symbolu z alfabetu.
- Jeśli nowe wyrażenie regularne nie jest jeszcze w `regex_to_state`, dodajemy je do kolejki.
- Rejestrujemy przejścia w `transitions`.

Tworzenie i zwracanie DFA

```
dfa = DFA(states, alphabet, transitions, regex, accept_states)
    return dfa
```

- Tworzymy obiekt DFA, przekazując do niego:
  - `states`: zbiór stanów,
  - `alphabet`: zbiór symboli alfabetu,
  - `transitions`: mapa przejść między stanami,
  - `regex`: jak założono w zadaniu - stan początkowy,
  - `accept_states`: zbiór stanów akceptujących.
- Zwracamy gotowy DFA.

Kod został przetestowany i przeszedł wszystkie testy.