

Uniwersytet im. Adama Mickiewicza  
w Poznaniu  
Wydział Matematyki i Informatyki

mgr Rafał Jaworski

Algorytmy przeszukiwania  
i przetwarzania pamięci tłumaczeń

Rozprawa doktorska  
napisana pod kierunkiem  
prof. UAM dra hab.  
Krzysztofa Jassem

Poznań, 2013

*Składam serdeczne podziękowania  
Panu Profesorowi  
Krzysztofowi Jassemowi  
za wszelką naukową pomoc  
oraz okazaną życzliwość.*

# Spis treści

<b>Wstęp</b> . . . . .	5
<b>Rozdział 1. Preliminaria</b> . . . . .	7
1.1. Podstawowe pojęcia . . . . .	7
<b>Rozdział 2. Tłumaczenie maszynowe</b> . . . . .	11
2.1. Historia . . . . .	11
2.1.1. Pionierzy tłumaczenia maszynowego . . . . .	12
2.1.2. Raport ALPAC . . . . .	14
2.1.3. Ożywienie w latach 1970-tych . . . . .	15
2.1.4. Pierwsze systemy komercyjne . . . . .	15
2.1.5. Badania na korpusach tekstów . . . . .	16
2.2. Paradygmaty tłumaczenia maszynowego . . . . .	17
2.2.1. Tłumaczenie oparte na regułach . . . . .	17
2.2.2. Tłumaczenie statystyczne . . . . .	22
2.2.3. Tłumaczenie oparte na przykładzie . . . . .	23
<b>Rozdział 3. Wspomaganie tłumaczenia ludzkiego</b> . . . . .	26
3.1. Idea wspomagania tłumaczenia . . . . .	26
3.1.1. Potrzeby tłumacza ludzkiego . . . . .	26
3.1.2. Zarys mechanizmu wykorzystania pamięci tłumaczeń . . . . .	27
3.1.3. Inne mechanizmy CAT . . . . .	27
3.2. Historia systemów CAT . . . . .	28
3.2.1. Wczesne początki i specjalizacja . . . . .	28
3.2.2. Pionierski system ALPS . . . . .	29
3.2.3. Rozwój systemów CAT . . . . .	29
3.3. Studium użyteczności systemów CAT . . . . .	30
3.3.1. Systemy CAT na rynku . . . . .	30
3.3.2. Indywidualne studium użyteczności . . . . .	31
3.3.3. Wnioski dotyczące użyteczności systemów CAT . . . . .	32
3.4. Współczesny system CAT - memoQ . . . . .	33
3.4.1. Opis ogólny . . . . .	33
3.4.2. Pamięć tłumaczeń w systemie memoQ . . . . .	33

<b>Rozdział 4. Przeszukiwanie pamięci tłumaczeń</b>	35
4.1. Motywacja badań	35
4.2. Problem wyszukiwania przybliżonego	36
4.2.1. Sformułowanie problemu	36
4.2.2. Funkcje dystansu	36
4.3. Znane rozwiązania problemu wyszukiwania przybliżonego	39
4.3.1. Rozwiązanie naiwne	39
4.3.2. Algorytm Sellersa	40
4.3.3. Metody oparte na tablicy sufiksów	42
4.4. Autorski algorytm przeszukiwania pamięci tłumaczeń	44
4.4.1. Problem przeszukiwania pamięci tłumaczeń	44
4.4.2. Wymagania odnośnie algorytmu	45
4.4.3. Skrót zdania	45
4.4.4. Kodowana tablica sufiksów	45
4.4.5. Dodawanie zdania do indeksu	46
4.4.6. Procedura getLongestCommonPrefixes	47
4.4.7. Obiekt OverlayMatch	47
4.4.8. Funkcja przeszukująca	49
4.4.9. Obliczanie oceny dopasowania	49
4.5. Analiza algorytmu przeszukiwania pamięci tłumaczeń	50
4.5.1. Ograniczenie oceny dopasowania	50
4.5.2. Własność przykładu doskonałego	51
4.5.3. Ocena dopasowania jako dystans, metryka, podobieństwo	51
4.5.4. Wnioski	54
4.6. Złożoność obliczeniowa algorytmu	55
4.6.1. Złożoność czasowa algorytmu	55
4.6.2. Złożoność pamięciowa algorytmu	56
4.7. Porównanie z konkurencyjnymi algorytmami	56
4.7.1. Algorytm Koehna i Senellarta	56
4.7.2. Algorytm Huerty	60
4.7.3. Algorytm Ghodsi'ego	61
<b>Rozdział 5. Przetwarzanie pamięci tłumaczeń</b>	63
5.1. Motywacja i cel badań	63
5.2. Tradycyjna metoda tworzenia pamięci tłumaczeń	64
5.2.1. Segmentacja tekstu	65
5.2.2. Urównoleglanie	68
5.3. Metoda autorska - przetwarzanie	71
5.3.1. Problemy podejścia tradycyjnego	71
5.3.2. Zarys rozwiązania	73

5.4.	Analiza skupień . . . . .	73
5.4.1.	Opis problemu . . . . .	73
5.4.2.	Algorytmy hierarchiczne . . . . .	74
5.4.3.	Algorytm K-średnich . . . . .	74
5.4.4.	Algorytm QT . . . . .	75
5.5.	Autorski algorytm przetwarzania pamięci tłumaczeń . . . . .	77
5.5.1.	Idea algorytmu . . . . .	77
5.5.2.	“Tania” funkcja dystansu zdań . . . . .	78
5.5.3.	“Droga” funkcja dystansu zdań . . . . .	79
5.5.4.	Procedura wyodrębniania skupień . . . . .	81
5.6.	Analiza algorytmu . . . . .	82
5.6.1.	Złożoność czasowa algorytmu QT . . . . .	82
5.6.2.	Złożoność czasowa funkcji $d_c$ . . . . .	83
5.6.3.	Złożoność czasowa funkcji $d_e$ . . . . .	83
5.6.4.	Złożoność czasowa autorskiego algorytmu . . . . .	84
<b>Rozdział 6. Ewaluacja algorytmu przeszukiwania pamięci tłumaczeń</b>		<b>86</b>
6.1.	Kompletność i dokładność wyszukiwania . . . . .	86
6.1.1.	Testowa pamięć tłumaczeń . . . . .	86
6.1.2.	Procedura ewaluacji . . . . .	86
6.1.3.	Kompletność przeszukiwań . . . . .	88
6.1.4.	Automatyczna analiza wyników . . . . .	88
6.1.5.	Dokładność przeszukiwań . . . . .	93
6.1.6.	Podsumowanie eksperymentu . . . . .	95
6.2.	Szybkość wyszukiwania . . . . .	95
6.2.1.	Porównanie z memoQ . . . . .	95
6.2.2.	Porównanie z Lucene . . . . .	96
6.2.3.	Analiza szybkości wyszukiwania - podsumowanie . . . . .	98
<b>Rozdział 7. Ewaluacja algorytmu przetwarzania pamięci tłumaczeń</b>		<b>99</b>
7.1.	Warunki eksperymentu . . . . .	99
7.1.1.	System wspomagania tłumaczenia . . . . .	99
7.1.2.	Scenariusze eksperymentu . . . . .	99
7.1.3.	Testowe pamięci tłumaczeń i zbiór zdań wejściowych . . . . .	100
7.2.	Kompletność przeszukiwania . . . . .	101
7.3.	Przydatność przetworzonej pamięci tłumaczeń . . . . .	101
7.3.1.	Miara ilości pracy - z sugestią . . . . .	101
7.3.2.	Miara ilości pracy – bez sugestii . . . . .	102
7.3.3.	Wyniki badania przydatności . . . . .	103
7.4.	Wnioski z eksperymentu . . . . .	103
<b>Podsumowanie</b> . . . . .		<b>105</b>

<b>Bibliografia</b> . . . . .	107
<b>Spis tablic</b> . . . . .	112
<b>Spis rysunków</b> . . . . .	113
<b>Dodatek A. Wykaz publikacji z udziałem autora rozprawy</b> . . . . .	114
A.1. Opublikowane . . . . .	114
A.2. Przyjęte do druku . . . . .	114
<b>Dodatek B. Reguły rozpoznawania jednostek nazwanych</b> . . . . .	115
B.1. Budowa reguły rozpoznawania . . . . .	115
B.2. Lista reguł . . . . .	115

# Wstęp

Problematyka niniejszej pracy jest związana ze wspomaganie pracy osoby tłumaczącej teksty z jednego języka naturalnego na inny. W obecnych czasach, dzięki wysokiemu poziomowi zaawansowania technologicznego, skuteczne wspomaganie tłumaczenia może być realizowane za pomocą komputera.

W idealnym przypadku, komputer powinien realizować tłumaczenie między językami naturalnymi w całości automatycznie. Idea ta, zwana tłumaczeniem maszynowym, narodziła się niedługo po zakończeniu II Wojny Światowej. Algorytmy tłumaczenia maszynowego są rozwijane do dzisiaj. Niestety, mimo uzyskiwanych postępów, jakość tłumaczenia wykonywanego przez komputer wciąż jest daleka od tego wykonanego przez człowieka.

W związku z powyższym, na początku lat 1980-tych zaczęto pracować nad algorytmami, których celem nie jest wykonanie całego tłumaczenia automatycznie, ale jedynie pomoc człowiekowi wykonującemu tłumaczenie. Najpopularniejszą techniką wspomaganie tłumaczenia ludzkiego jest wykorzystanie tzw. pamięci tłumaczeń. Jest to baza wcześniej wykonanych tłumaczeń, które służą jako podpowiedź przy wykonywaniu nowych. Zostało udowodnione, że pamięć tłumaczeń pozwala tłumaczowi ludzkiemu na znaczne zwiększenie wydajności pracy.

W niniejszej pracy opisane są dwa autorskie algorytmy, operujące na pamięci tłumaczeń. Pierwszym z nich jest algorytm jej przeszukiwania. W obliczu faktu, iż tworzone są coraz większe pamięci, konieczne jest opracowanie wydajnych technik ich przeszukiwania. Istniejące rozwiązania nie zawsze są dostosowane do znacznych rozmiarów danych. Proponowany przez autora algorytm opiera się na osiągnięciach w dziedzinie wyszukiwania przybliżonego oraz przetwarzania języka naturalnego. Pożądane cechy algorytmu są zagwarantowane dzięki użyciu opracowanej przez autora funkcji dystansu zdań, zachowującej dobre własności matematyczne. Dzięki temu algorytm posiada niską złożoność obliczeniową czasową oraz pamięciową.

Drugi autorski algorytm, przedstawiony w niniejszej pracy, służy do przetwarzania pamięci tłumaczeń. Celem tego przetwarzania jest utworzenie nowej, specjalistycznej pamięci, zawierającej tłumaczenia szczególnie przydatne tłumaczowi. Algorytm jest oparty na zdobyczach teorii analizy skupień. Jego niska złożoność obliczeniowa pozwala na przetwarzanie nawet znacznej wielkości pamięci tłumaczeń.

Pierwszy rozdział niniejszej pracy wprowadza definicje pojęć, używanych w dalszych rozdziałach. Rozdział drugi opisuje dziedzinę tłumaczenia maszynowego oraz jej wpływ na rozwój technik przetwarzania języka naturalnego. W rozdziale trzecim przedstawiona jest idea wspomagania tłumaczenia ludzkiego przy pomocy komputera.

Dokładny opis autorskiego algorytmu przeszukiwania pamięci tłumaczeń jest zamieszczony w rozdziale czwartym. W tym samym rozdziale przedstawiona jest także jego analiza oraz porównanie z konkurencyjnymi algorytmami przeszukującymi.

Tematem rozdziału piątego jest przetwarzanie pamięci tłumaczeń. W rozdziale tym znajduje się opis autorskiego algorytmu przetwarzającego pamięć tłumaczeń oraz jego analiza.

W rozdziale szóstym przedstawione są procedury i wyniki ewaluacji autorskiego algorytmu przeszukiwania pamięci tłumaczeń. Wyniki ewaluacji algorytmu przetwarzania pamięci tłumaczeń są natomiast zamieszczone w rozdziale siódmym.



## Rozdział 1

# Preliminaria

### 1.1. Podstawowe pojęcia

W podrozdziale tym zdefiniowane są pojęcia pomocne w opisywaniu i analizowaniu algorytmów, będących przedmiotem niniejszej pracy. Są to na ogół pojęcia powszechnie znane w dziedzinie przetwarzania tekstu, doprecyzowane na potrzeby spójności opisu.

Kluczowe znaczenie z punktu widzenia projektowania algorytmów operujących na tekstach odgrywa pojęcie alfabetu. Aby określić zbiór znaków nazywany alfabetem, posługujemy się standardem kodowania znaków UTF, opisanym w dokumencie [Con11].

**Definicja 1** (Alfabet). *Alfabetem* nazywamy zbiór znaków możliwych do zakodowania w standardzie UTF. Alfabet będziemy oznaczać symbolem:  $\Sigma$ .

#### Przykłady znaków:

'a', 'b', 'c', 'ą', 'ś', 'ł', '0', '1', '2'  $\in \Sigma$ ;

'abc', 'str1'  $\notin \Sigma$

W zbiorze  $\Sigma$  wyróżniamy podzbiór znaków specjalnych, tzw. białych znaków.

**Definicja 2** (Biały znak). *Znaki: spacja (kod UTF: U+0020), tabulacja (U+0009), powrót karetki (U+000D) oraz nowa linia (U+000A) to białe znaki.* Przyjmujemy następujące oznaczenie zbioru białych znaków:  $B$ .

Pojęcie tekstu definiujemy klasycznie:


**Definicja 3** (Tekst). *Teskstem* nazywamy niepusty ciąg znaków alfabetu.

#### Przykłady tekstów:

'abc', 'Dzisiaj świeci słońce. Wczoraj padał deszcz.'

Tablica 1.1. Przykładowe stemy słów

Słowo	Stem
prowadzone	prowadz
badania	badan
poszlibyśmy	poszl
zielonego	zielon

**Definicja 4** (Słowo). *Słowem* nazywamy niepusty ciąg znaków alfabetu, niezawierający białych znaków. Zbiór wszystkich możliwych słów oznaczamy przez  $\Theta$ .  $\Theta = (\Sigma \setminus B)^* \setminus \{\emptyset\}$ , li do zbioru tego należą wszystkie możliwe ciągi utworzone ze znaków niebędących białymi, za wyjątkiem ciągu pustego.


**Przykłady słów:**

'abc', 'str1'  $\in \Theta$

'ab c', "  $\notin \Theta$

Pojęciem szerszym wobec słowa jest jednostka nazwana. Pojęcie to jest wykorzystywane w opisie algorytmu przetwarzania pamięci tłumaczeń.

**Definicja 5** (Jednostka nazwana). *Jednostką nazwaną* nazywamy niepusty ciąg słów, któremu przypisane jest jedno wspólne znaczenie.

W praktyce, algorytmy wyszukujące w tekście jednostek nazwanych są w stanie identyfikować określone ich typy, np. nazwisko, data, lokalizacja geograficzna. To, czy dany ciąg słów zostanie rozpoznany jako jednostka nazwana, zależy wyłącznie od algorytmu rozpoznawania jednostek nazwanych. Przykładowe reguły rozpoznawania takich jednostek, oparte na wyrażeniach regularnych, są przedstawione w Dodatku B gorytmy rozpoznawania jednostek nazwanych nie są jednak przedmiotem niniejszej pracy.

**Przykłady jednostek nazwanych:**

'Jan Nowak', '27.10.2001', '12 września 1994 roku', '52°24'N 16°55'E'.

Algorytmy opisane w tej pracy będą wykorzystywały wyniki procesu stemowania. Według [SP05], stem jest podstawową częścią słowa, do której mogą być dołączone końcówki fleksyjne. Przykładowe stemy dla słów w języku polskim są przedstawione w Tabeli 1.1.

**Definicja 6** (Stemowanie). *Stemowaniem* nazywamy funkcję  $stem : \Theta \mapsto \Theta$ , która przypisuje dowolnemu słowu jego stem.

Dokładna implementacja funkcji stemowania nie jest tematem niniejszej pracy. Opracowanie poprawnie działającego mechanizmu stemowania stanowi odrębne, nietrywialne zadanie badawcze.

Zaprojektowane przeze mnie algorytmy operują na zdaniach języka naturalnego. Definicja pojęcia zdania używana w językoznawstwie odnosi się do jego struktury gramatycznej. W ramach rozważań będących przedmiotem niniejszej pracy nie prowadzi się jednak żadnej analizy składniowej zdań. Wobec tego, przyjmujemy następującą, uproszczoną definicję zdania:

**Definicja 7 (Zdanie).** *Zdaniem*  $z$  nazywamy niepusty ciąg słów. Kolejne słowa zdania  $z$  oznaczamy:  $z[0], z[1], z[2]$  itd. Przez długość zdania rozumiemy liczbę słów w zdaniu (oznaczenie:  $length(z)$ ).

#### Przykłady zdań:

['aaa', 'bbb', 'ccc'], ['pogoda', 'jest', 'ładna'].

Zwyczajowo pojęcie sufiksu łączy się ściśle z łańcuchem, tj. skończonym ciągiem znaków. Sufiksem ciągu znaków  $t$  nazywany jest każdy jego podciąg  $s$  taki, że istnieje ciąg znaków  $p$ , taki że  $t = p \cdot s$  (gdzie  $\cdot$  oznacza operację konkatenacji ciągów). Definicja ta jest zaczerpnięta z książki [Lin11] w niniejszej pracy posługujemy się nieco inaczej zdefiniowanym pojęciem sufiksu. Zdefiniowany przeze mnie sufiks nie jest ciągiem znaków, ale słów.

**Definicja 8 (Sufiks).** *Sufiksem* zdania  $z$  nazywamy takie zdanie  $s$ , że  $\exists n_0 \in \mathbb{N} \forall i \in [0, length(s)-1], i \in \mathbb{N} s[i] = z[n_0 + i]$ . Parametr  $n_0$  nazywamy *offsetem* sufiksu  $s$ .

#### Przykłady:

- Zdanie ['bbb', 'ccc'] jest sufiksem zdania ['aaa', 'bbb', 'ccc'], przy czym jego offset wynosi 1.
- Zdanie ['Witaj', 'świecie'] jest sufiksem zdania ['Witaj', 'świecie'], przy czym jego offset wynosi 0.
- Zdanie ['Witaj', 'świecie'] nie jest sufiksem zdania ['Witaj', 'piękny', 'świecie']

W zagadnieniach związanych z tłumaczeniem często używa się pojęcia kierunku tłumaczenia.

**Definicja 9 (Kierunek tłumaczenia).** *Kierunkiem tłumaczenia* nazywamy parę języków naturalnych. Pierwszy język kierunku nazywamy *źródłowym*, a drugi *docelowym*. Kierunek tłumaczenia oznaczamy następująco:

$(l_1, l_2)$ , gdzie  $l_1$  jest dwuliterowym symbolem języka źródłowego, a  $l_2$  jest analogicznym symbolem języka docelowego.

**Przykłady kierunków tłumaczenia:**

- kierunek polsko-angielski:  $(pl, en)$
- kierunek niemiecko-rosyjski:  $(de, ru)$
- kierunek hiszpańsko-rosyjski:  $(es, ru)$

**Definicja 10** (Przykład). *Przykładem* w określonym kierunku tłumaczenia nazywamy parę zdań, w której pierwsze jest w języku źródłowym, a drugie w języku docelowym. Pierwsze zdanie przykładu nazywamy **źródłowym**, natomiast drugie - **docelowym**.


**Przykład**



- **pl** : ['Na', 'stole', 'leży', 'książka']  
**en** : ['There', 'is', 'a', 'book', 'on', 'the', 'table']
- **de** : ['Es', 'gibt', 'ein', 'Buch', 'auf', 'den', 'Tisch']  
**es** : ['Hay', 'un', 'libro', 'sobre', 'la', 'mesa']

**Definicja 11** (Pamięć tłumaczeń). *Pamięcią tłumaczeń* w określonym kierunku tłumaczenia nazywamy dowolny niepusty zbiór przykładów w tym kierunku.

## Rozdział 2

# Tłumaczenie maszynowe

Tematyka niniejszej rozprawy jest związana z zagadnieniem tłumaczenia maszynowego, zwanego także tłumaczeniem automatycznym lub komputerowym. Koncepcją  polega na wykonywaniu tłumaczenia w całości przez maszynę, bez udziału człowieka. Badania prowadzone w tej dziedzinie w sposób znaczący wpłynęły na opracowanie różnorodnych technik przetwarzania języka naturalnego oraz wspomaganie tłumaczenia, opisanego w Rozdziale 3.

W niniejszym rozdziale przedstawiona jest historia tłumaczenia maszynowego. Daje ona pogląd na to  jak przebiegała ewolucja systemów tłumaczących. Analizując kolejne rozwiązania można odkryć, które z przyjętych założeń i kierunków rozwoju okazały się słuszne, a które ni 

W dalszej części, przedstawione są różne paradygmaty tłumaczenia maszynowego. W tym kontekście, paradygmat rozumiany jest jako zbiór technik, opierających się na tej samej hipotezie naukowej, dotyczącej języka i tłumaczenia.

### 2.1. Historia


O tłumaczeniu automatycznym, które miało rozwiązać problem komunikacji wielojęzycznej, marzono już od XVII w. Marzenia te zostały częściowo zrealizowane dopiero pod koniec wieku XX.

Obecne algorytmy nie oferują doskonałej jakości tłumaczenia. W żaden sposób nie aspirują do zastąpienia człowieka w zadaniu tłumaczenia. W określonych warunkach pozwalają jednak wygenerować tłumaczenie zrozumiałe dla osoby posługującej się językiem docelowym, a nie znającej źródłowego języka tłumaczenia.

Droga od pierwszych prób opracowania algorytmów tłumaczenia maszynowego do stanu dzisiejszego może być podzielona na etapy. Autor artykułu [Hut95] proponuje następujący podział:

1. pionierzy tłumaczenia maszynowego (lata 1950-te i 1960-te);

2. wpływ raportu ALPAC (lata 1960-te);
3. ożywienie badawcze (lata 1970-te);
4. pojawienie się systemów komercyjnych (lata 1980-te);
5. nowe odkrycia (lata 1990-te);
6. rosnące wykorzystanie systemów tłumaczących (do dnia dzisiejszego).



Informacje dotyczące historii tłumaczenia maszynowego, przedstawione w tym rozdziale, pochodzą z artykułu [Hut95] 

### 2.1.1. Pionierzy tłumaczenia maszynowego


#### Pierwsze kroki

Wczesne podejścia do problemu tłumaczenia maszynowego polegały na stworzeniu słowników, przechowywanych w pamięci maszyny. W 1933 roku, Francuz George Artsrouni opatentował słownik, zapisany na taśmie magnetycznej. Pozwalał on na szybkie odnajdywanie tłumaczeń pojedynczych wyrazów na język obcy. Niezależnie od Astrouniego, w tym samym roku, Rosjanin Piotr Smirnow-Trojański zaprezentował podobny słownik, lecz podał inny sposób jego użycia. Według niego, proces tłumaczenia miał być podzielony na trzy fazy:

1. analiza słów i sprowadzenie ich do formy podstawowej przez człowieka znającego język źródłowy;
2. automatyczne odnajdywanie słów w słowniku;
3. analiza i wygładzenie wyników przez człowieka znającego język docelowy.


Koncepcja ta silnie wyprzedzała czasy Trojańskiego  zakładała współpracę człowieka i maszyny przy wykonywaniu tłumaczenia. Idea ta została wykorzystana m.in. do opracowania techniki wspomagania tłumaczenia (opisanej w Rozdziale 3), która pochodzi z lat 1980-tych. Sam jej autor był jednak przekonany, że w przyszłości nie będzie konieczna ingerencja człowieka w proces tłumaczenia  maszynowego.

#### Memorandum Warrena Weavera


Ważnym krokiem milowym w rozwoju idei tłumaczenia maszynowego  było memorandum, opublikowane w lipcu 1949 roku przez Warrena Weavera. Memorandum zawierało następujące tezy:

1. Wszystkie języki świata mają wspólną "bazę", tj. wyrażają tę samą treść, choć na różne sposoby.
2. Tłumaczenie automatyczne jest możliwe dzięki technikom kryptograficznym.

3. Problem wieloznaczności słów można rozwiązać metodami statystycznymi.

Pierwsza teza memorandum nawiązuje do koncepcji tzw. “interlingua”  języka pośredniego, zdolnego wyrazić treść, napisaną lub wymówioną w dowolnym języku. Znane są liczne próby opracowania takiego języka, w tym: — język o nazwie Interlingua, opracowany przez stowarzyszenie *International Auxiliary Language Association*, w latach 1937-1951; — Lojban, rozwijany przez *The Logical Language Group* od roku 1987.

Koncepcja języka pośredniego nie zrewolucjonizowała jednak podejścia do tłumaczenia maszynowego. Zarówno opracowanie dobrego języka typu interlingua, jak i tłumaczenie z i na język naturalny, okazały się problemami istotnie trudniejszymi, niż tłumaczenie pomiędzy parą języków naturalnych.

Druga teza memorandum Weavera mówi o wykorzystaniu technik kryptograficznych w tłumaczeniu automatycznym. Język naturalny miał być traktowany jak pewnego rodzaju szyfr, kodujący wiedzę logiczną. Techniki kryptograficzne zyskały po II Wojnie Światowej dużą popularność, dlatego nie dziwi pomysł ich wykorzystania. Niestety  koncepcja ta nie została później wykorzystana.


Ostatnia teza, dotycząca metod statystycznych, może być uznana za wizjonerską. Metody te faktycznie w znaczący sposób wpłynęły na rozwój tłumaczenia maszynowego. Okres ich popularności przypada jednak dopiero na początek XXI wieku, kiedy rozwój technologiczny umożliwił dobre ich wykorzystanie.

### **Eksperyment na Georgetown University**

Memorandum Warrena Weavera zainspirowało badaczy do wyteżonej pracy nad zagadnieniem tłumaczenia maszynowego. Pierwszą osobą zatrudnioną w jednostce naukowej na stanowisku specjalisty od tłumaczenia automatycznego był Yehoshua Bar-Hillel, na uczelni Massachusetts Institute of Technology w 1951 roku. Rok później zorganizował on pierwszą konferencję dotyczącą tej dziedziny. Jednym z wniosków tej konferencji był pomysł zwrócenia uwagi szerszej publiczności na potencjalne korzyści, płynące z tłumaczenia automatycznego.

Zadania tego podjął się Leon Dostert z Georgetown University. Współpracował on z firmą IBM w celu opracowania systemu tłumaczącego z języka rosyjskiego na angielski. W obliczu zimnej wojny, właśnie ten kierunek tłumaczenia wydawał się najbardziej interesujący. W styczniu 1954 roku odbyła się pierwsza publiczna prezentacja systemu tłumaczącego.

System działał na niewielkim zbiorze 49 starannie wyselekcjonowanych zdań rosyjskich. Zdania te zostały zbudowane na zbiorze zaledwie 250 słów. Przy użyciu tylko 6 reguł gramatycznych, nastąpiło tłumaczenie zdań rosyjskich na język angielski. Eksperyment miał więc bardzo niewielką wartość naukową.


Nie mniej dnak, eksperyment zyskał ogromny rozgłos. Gazety rozpisywały się na temat tajemniczej technologii, która pozwoliła operatorowi maszyny, nie znającemu języka rosyjskiego, uzyskać tłumaczenie zdań rosyjskich na język angielski.

### 2.1.2. Raport ALPAC

Eksperyment na Georgetown University przyczynił się do dalszego wzrostu zainteresowania dziedziną tłumaczenia maszynowego. W latach 1956-1966 opracowano wiele systemów tłumaczących, opartych na różnych założeniach i wykorzystujących różne algorytmy. Powstawały systemy statystyczne (takie jak opracowany w firmie *RAND Corporation* tłumacz rosyjskich tekstów z dziedziny fizyki), analizatory gramatyczne oraz znacznej wielkości słowniki wielojęzyczne. Trudno mówić o jednym kierunku rozwoju tłumaczenia maszynowego w tym okresie.

Dekada 1956-1966 to jednak również czas zawodów i niepowodzeń. Szybko zorientowano się, że tłumaczenie automatyczne, takie jak zaprezentowane w Georgetown, jest niemal nieosiągalne. Systemy, które powstawały, rozwiązywały często tylko pewien konkretny przypadek tłumaczenia automatycznego i daleko im było do ideału. Sytuację dodatkowo utrudnił rząd Stanów Zjednoczonych.

W roku 1964, organizacje rządowe Stanów Zjednoczonych, finansujące badania nad tłumaczeniem maszynowym, powołały do życia komitet *Automatic Language Processing Advisory Committee*, w skrócie *ALPAC*. Zadaniem komitetu było zbadanie postępów w dziedzinie tłumaczenia automatycznego oraz określenie perspektyw na przyszłość. Od raportu tego miało zależeć dalsze finansowanie badań.

- Raport ALPAC został opublikowany w roku 1966. Głosił między innymi:
- Tłumaczenie automatyczne blniejsze, mniej dokładne i dwukrotnie droższe, niż tłumaczenie wykonywane przez człowieka.
  - Nie istnieje prawdopodobieństwo, że w najbliższej lub dalszej przyszłości znajdzie się użyteczne zastosowanie tłumaczenia automatycznego.
  - Nie jest zasadne dalsze finansowanie badań w tej dziedzinie.

Przez badaczy tłumaczenia maszynowego raport ten został przyjęty jako



powierzchnowy, nieobiektywny i wysoce krzywdzący. Jednak skutki jego działania były bardzo rozległe. Raport przyczynił się do niemal całkowitego wygaszenia badań nad tłumaczeniem maszynowym w Stanach Zjednoczonych i wielu innych częściach świata. Powszechna stała się opinia, że próby opracowania tłumaczenia maszynowego zakończyły się klęską.

### 2.1.3. Ożywienie w latach 1970-tych

W związku z raportem ALPAC, głównym ośrodkiem prac nad tłumaczeniem maszynowym w latach 1970-tych przestały być Stany Zjednoczone. Badania były prowadzone w Europie oraz Kanadzie, gdzie występowało duże zapotrzebowanie na tłumaczenie. W Europie zapotrzebowanie to generowała Wspólnota Europejska, w obrębie której zachodziła wymiana dokumentów prawnych, administracyjnych i technicznych pomiędzy krajami członkowskimi. W Kanadzie zachodziła natomiast szczególna potrzeba tłumaczenia tekstów pomiędzy dwoma oficjalnymi językami tego kraju, angielskim i francuskim.

Sporym osiągnięciem badaczy kanadyjskich były wyniki uzyskane w ramach projektu TAUM (fr. *Traduction Automatique de l'Université de Montréal*). Projekt był prowadzony na Uniwersytecie w Montrealu od 1970 do 1981 roku. Jednym z głównych jego rezultatów było opracowanie formalizmu Q-system, służącego do opisu operacji na łańcuchach znaków oraz drzew składniowych (opisanych w Podrozdziale 2.2). Innym osiągnięciem było stworzenie podstaw języka Prolog, który znalazł szerokie zastosowanie w przetwarzaniu języka naturalnego. W ramach TAUM powstał także system *Météo*, służący do tłumaczenia komunikatów meteorologicznych. System oferował wysoką jakość i szybkość tłumaczenia. Swoją sukces zawdzięczał skupieniu się na ograniczonym, specjalistycznym słownictwie i zawężonej gramatyce tłumaczonych tekstów. Idea ta jest często wykorzystywana w nowoczesnych systemach tłumaczących.

### 2.1.4. Pierwsze systemy komercyjne

Okolo dekady po opublikowaniu raportu ALPAC można było zaobserwować znaczący wzrost zainteresowania systemami tłumaczącymi. Był on spowodowany licznymi sukcesami we wdrażaniu tłumaczenia maszynowego do zastosowań praktycznych. Wśród tych sukcesów wymienić można:

- opisany wcześniej system *Météo*,
- system TITUS (*Institut Textile de France*), służący do tłumaczenia abstraktów tekstów naukowych,

— system CULT (*Chinese University of Hong Kong*), tłumaczący teksty matematyczne,

Jednak największym sukcesem tamtego okresu i jednym z najbardziej znaczących systemów tłumaczenia maszynowego do dnia dzisiejszego jest Systran. System ten został opracowany przez Petra Tomę, który wcześniej pracował przy projekcie na Uniwersytecie w Georgetown. Najważniejsze wdrożenia tego systemu to instalacja automatycznego tłumacza rosyjsko-angielskiego dla Marynarki Stanów Zjednoczonych (1970) oraz instalacja systemu tłumaczącego w kierunku angielsko-francuskim dla Wspólnoty Europejskiej (1976). Wdrożenie dla Wspólnoty Europejskiej zostało szybko rozszerzone o możliwość tłumaczenia tekstów w kierunkach francusko-angielski i angielsko-włoskim. W kolejnych latach włączono obsługę wielu innych par językowych w obrębie Wspólnoty.

Systran zawdzięczał swój sukces między innymi dobrej architekturze rozwiązania. Funkcje systemu, związane z przetwarzaniem języka naturalnego i tłumaczeniem, zostały pogrupowane w moduły, które łatwo było wymieniać, co pozwalało to na znaczącą redukcję kosztów dodawania obsługi nowego kierunku tłumaczenia. Dzięki temu, Systran nie był pojedynczym systemem tłumaczącym, ale platformą do tworzenia tłumaczy maszynowych.

Kolejne udane wdrożenia ugruntowywały pozycję Systrana na rynku. O marketingowym sukcesie systemu świadczy fakt, że w odświeżonej wersji jest on dostępny w sprzedaży do dziś. Co więcej, jeszcze do 2007 roku mechanizmy popularnego tłumacza maszynowego Google Translate opierały się na Systranie.

### 2.1.5. Badania na korpusach tekstów

Przed rokiem 1990 badania nad tłumaczeniem maszynowym koncentrowały się na opracowaniu metod analizy lingwistycznej tekstu. Jednak z początkiem lat 1990-tych, możliwości technologiczne pozwoliły na opracowywanie tłumaczy automatycznych opartych na korpusach tekstów.

Formalnie, dwujęzyczny korpus tekstów jest tym samym, co pamięć tłumacza. Tłumaczenie maszynowe na podstawie korpusu dwujęzycznego polega na analizie statystycznej danych z korpusu (proces ten jest dokładniej opisany w Podrozdziale 2.2.2). Jakość tłumacza automatycznego, opartego na korpusie, silnie zależy od jakości przykładów w tym korpusie oraz od ich liczby.

W roku 1988 firma IBM opublikowała wyniki prac nad tłumaczem maszynowym, opartym wyłącznie na metodach statystycznych, zaskakująco wy-

soka jakość tłumaczeń systemu IBM zainspirowała badaczy na całym świecie do podjęcia prób opracowywania systemów opartych na tych metodach.

Najpopularniejszymi statystycznymi systemami tłumaczenia automatycznego są Google Translate (opisany w [Doo]) oraz Moses (opisany w artykule [KHB<sup>+</sup>07]). Ze względu na stały wzrost dostępności zasobów tekstowych i korpusów dwujęzycznych, tłumaczenie statystyczne znajduje się w centrum zainteresowania współczesnych badaczy.

## 2.2. Paradygmaty tłumaczenia maszynowego

W niniejszym podrozdziale zostaną przedstawione najważniejsze paradygmaty tłumaczenia maszynowego. Paradygmat należy tu rozumieć jako zbiór pojęć, założeń oraz teorii, dotyczących sposobu wykonywania tłumaczenia przez maszynę. Paradygmaty tłumaczenia maszynowego różnią się między sobą przede wszystkim przyjętą hipotezą, dotyczącą konstrukcji języków naturalnych oraz istoty procesu tłumaczenia.

### 2.2.1. Tłumaczenie oparte na regułach

Tłumaczenie oparte na regułach opiera się na założeniu, że języki naturalne określone są przez skończony zbiór słownictwa oraz reguł gramatycznych. Zgodnie z tym założeniem można wprowadzić do pamięci maszyny wszystkie słowa i reguły, uzyskując w ten sposób urządzenie obdarzone wiedzą lingwistyczną identyczną lub zbliżoną do wiedzy człowieka.

#### Tłumaczenie słownikowe

Najmniej skomplikowaną odmianą tłumaczenia opartego na regułach jest tłumaczenie słownikowe (ang. *Dictionary-Based Machine Translation*, zwane także tłumaczeniem bezpośrednim (ang. *Direct Machine Translation*). W podejściu tym tłumaczenie odbywa się słowo po słowie, na podstawie słownika. Koncepcja systemu tłumaczącego bezpośrednio została przedstawiona w Podrozdziale 2.1.1.


Podczas tłumaczenia słownikowego, każde słowo zdania źródłowego jest odnajdywane w słowniku i zastępowane odnalezionym odpowiednikiem. Niekiedy stosuje się analizę morfologiczną słowa, w celu ustalenia takich jego cech, jak przypadek, osoba lub rodzaj gramatyczny. Mimo to tłumaczenie słownikowe nie rozwiązuje takich problemów, jak wieloznaczność wyrazów lub różnice szyku zdań pomiędzy językiem źródłowym i docelowym.

Pomimo swoich wad, tłumaczenie bezpośrednio znajduje zastosowania praktyczne. Jest przydatne na przykład do tłumaczenia katalogów produktów lub spisów inwentaryzacyjnych. Zastosowanie to jest opisane w artykule z roku 2006: [Mue06].

## Interlingua

Idea języka typu *interlingua*, zdolnego wyrazić znaczenie tekstu w dowolnym języku naturalnym, została już wspomniana w Podrozdziale 2.1.1, przy okazji omawiania też memorandum Warrena Weavera. Tłumaczenie przez język pośredni jest jednym z klasycznych podejść do problemu tłumaczenia maszynowego.

Tłumaczenie tekstu źródłowego przez język typu *interlingua* na język docelowy przebiega w następujących etapach:

1. pełna analiza składniowa i semantyczna tekstu źródłowego;
2. tłumaczenie na język pośredni 
3. wygenerowanie poprawnego zdania w języku docelowym na podstawie tekstu w języku pośrednim.

Powodzenie tego procesu zależy od przygotowania następujących zasobów:

- słowniki języka źródłowego i docelowego;
- leksykony języka źródłowego i docelowego, zawierające informacje semantyczne;
- zbiór reguł transferu z języka źródłowego na pośredni;
- zbiór reguł generowania języka docelowego z pośredniego.

Najważniejszym zasobem jest jednak sam język pośredni. Musi on posiadać następujące cechy:

- zasób słownictwa pokrywający tłumaczone języki;
- zdolność wyrażania tego samego zbioru informacji semantycznych, co tłumaczone języki;
- jednoznaczność słów;
- sztywność reguł gramatycznych (brak wyjątków).

Niewątpliwą zaletą tłumaczenia maszynowego poprzez język typu *interlingua* jest łatwość opracowywania obsługi kolejnych języków. W przypadku technik tłumaczenia bezpośredniego pomiędzy parą języków, skonstruowanie systemu tłumaczącego pomiędzy  $n$  językami wymaga oddzielnego opracowania  $n(n - 1)$  par. Dla każdej z tych par należy opracować osobne słowniki i reguły tłumaczenia. Natomiast w przypadku używania języka pośredniego, obsługa  $n$  języków wymaga opracowania  $2n$  zasobów - dla każdego języka

musi istnieć mechanizm analizy (tłumaczenia na język pośredni) oraz generowania (tłumaczenia z języka pośredniego).

Niestety opracowanie dobrego języka pośredniego, jak również mechanizmów analizy języków naturalnych, okazało się problemem zbyt trudnym

## Tłumaczenie przez transfer

Tłumaczenie przez transfer jest kolejną odmianą tłumaczenia regułowego. Podobnie jak w przypadku tłumaczenia pośredniego, technika transferu opiera się na stworzeniu pośredniej reprezentacji tłumaczonego zdania. Zasadniczą różnicą są jednak własności tej reprezentacji. W tłumaczeniu przez język pośredni, reprezentacja pośrednia jest niezależna od języka źródłowego i docelowego. Natomiast w tłumaczeniu przez transfer dopuszcza się tę zależność.

Działanie systemów tłumaczenia przez transfer jest bardzo zróżnicowane. Można jednak wyróżnić następujące etapy ich pracy:

1. analiza morfologiczna
2. kategoryzacja leksykalna
3. parsing płytki - analiza syntaktyczna
4. (w niektórych systemach) parsing głęboki - analiza semantyczna
5. transfer struktury informacji syntaktycznych (i ew. semantycznych)
6. generowanie morfologiczne

Podczas pierwszego etapu pracy, słowa zdania źródłowego poddawane są analizie morfologicznej (podobnie, jak w przypadku niektórych systemów tłumaczenia bezpośredniego). Każdemu słowu przypisywany jest co najmniej jeden wektor cech, zawierających informacje morfologiczne. W praktyce, ze względu na wieloznaczność wyrazów, jedno słowo często posiada kilka lub kilkanaście wektorów, odpowiadających poszczególnym jego interpretacjom. Na przykład, słowo “albo” może mieć co najmniej dwie interpretacje:

**forma podstawowa** : { albo }

**część mowy** : { spójnik }

**typ** : { spójnik wykluczający }

oraz:

**forma podstawowa** : { alba }

**część mowy** : { rzeczownik }

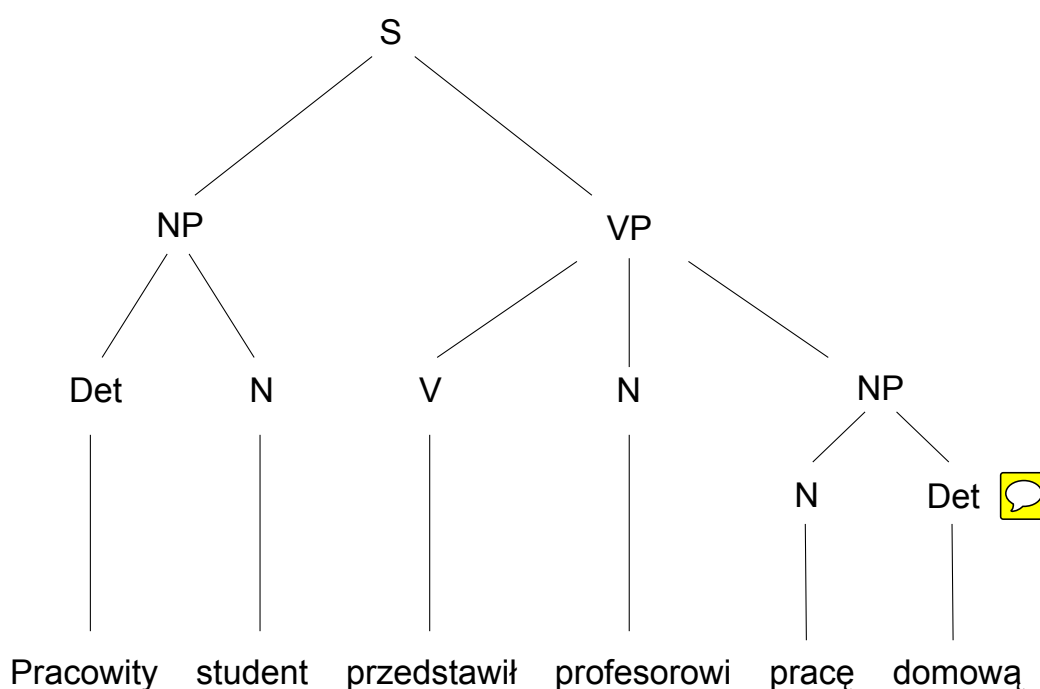
**typ** : { wołacz }

Na etapie analizy morfologicznej bierze się pod uwagę wszystkie możliwe interpretacje.

W drugim etapie pracy dokonuje się kategoryzacji leksykalnej słów. Ope-

racja ta ma na celu ujednoznaczenie interpretacji zdania źródłowego. Ujednoznaczenie wyrazów (ang. *Word-sense disambiguation*) jest możliwe dzięki analizie kontekstu, w którym wystąpiły w zdaniu. Problem ten znajduje się obecnie w centrum zainteresowania badaczy. W niektórych systemach, podczas kategoryzacji leksykalnej przeprowadza się również inne operacje, takie jak rozpoznawanie jednostek nazwane

Trzeci etap pracy systemu tłumaczenia przez transfer polega na przeprowadzeniu analizy składniowej (syntaktycznej) zdania. W wyniku tej operacji powstaje tzw. drzewo parsingu. Na Rysunku 2.1 przedstawione jest przykładowe drzewo parsingu dla zdania w języku polskim: “Pracowity student przedstawił profesorowi pracę domową.” Symbole widniejące w drzewie ozna-



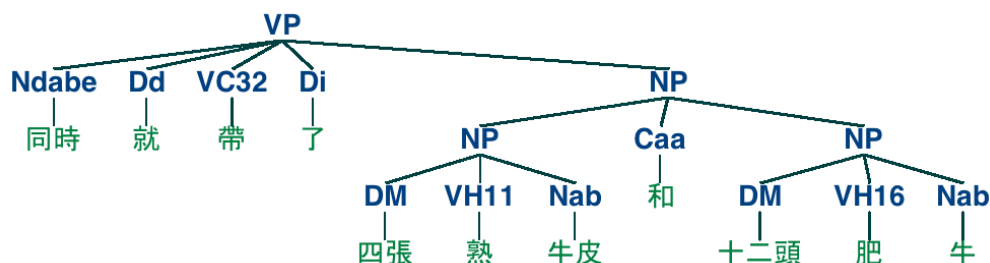
Rysunek 2.1. Proste drzewo parsingu

czają odpowiednio:


- **Det**: słowo określające
- **N**: rzeczownik
- **V**: czasownik
- **NP**: fraza rzeczownikowa
- **VP**: fraza czasownikowa
- **S**: zdanie


Zbiór symboli, mogących się pojawić w drzewie, jest ustalany w zależności od przeznaczenia systemu tłumaczącego oraz języków, które obsługuje. Na Rysunku 2.2 przedstawione jest drzewo parsingu zdania w języku chińskim,

które wykorzystuje inny zestaw symboli (drzewo jest zaczerpnięte z opracowania [Bir]).




Rysunek 2.2. Drzewo parsingu zdania w języku chińskim

Po analizie składniowej (oraz, w niektórych przypadkach, semantycznej) następuje proces transferu. Jest on sterowany zdefiniowanymi regułami przekształcania drzew języka źródłowego na drzewa języka docelowego. Reguły te uwzględniają różnice w składni obu języków .

Ostatnim krokiem do uzyskania tłumaczenia zdania źródłowego na język docelowy jest generowanie morfologiczne. Polega ono na przetłumaczeniu słów, znajdujących się w przekształconym drzewie, na język docelowy. W procesie tym wykorzystywane są informacje morfologiczne, uzyskane we wcześniejszych etapach analiz .

Tłumaczenie przez transfer jest jednym z najskuteczniejszych paradygmatów tłumaczenia maszynowego. Systemy tłumaczące w ten sposób opracowywane są najczęściej dla konkretnej pary języków, dzięki czemu są zazwyczaj dobrze dostosowane do występujących w nich zjawisk językowych. Jakość tłumaczeń generowanych przez te systemy jest szczególnie wysoka, kiedy język źródłowy i docelowy są do siebie podobne (np. polski-rosyjski, hiszpański-włoski). Poza tym, systemy te są zazwyczaj zbudowane modułowo. Każdy z modułów może być rozwijany niezależnie przez różnych badaczy, specjalizujących się w wąskich dziedzinach wiedzy. Opracowane przez nich rozwiązania, połączone w jeden system tłumaczący, pozwalają uzyskać wysokiej jakości tłumaczenie maszynowe.

Największą wadą tłumaczenia przez transfer jest duży nakład pracy podczas opracowywania obsługi kolejnych kierunków tłumaczenia. W praktyce często powoduje to, że koszt opracowania systemu tłumaczenia maszynowego jest zbyt duży .

### 2.2.2. Tłumaczenie statystyczne

Tłumaczenie statystyczne, wspomniane w Podrozdziale 2.1.5, jest techniką tłumaczenia maszynowego, opartego na korpusie tekstów. Proces tłumaczenia statystycznego jest możliwy dzięki analizie korpusu metodami statystycznym. Dzięki temu łatwiej jest uzyskać informacje na temat tego, w jaki sposób dane słowa lub frazy w języku źródłowym są tłumaczone na język docelowy. Proces tłumaczenia zdania źródłowego na język docelowy polega na zastąpieniu jego słów i fraz najbardziej prawdopodobnymi odpowiednikami w języku docelowym.

Metody statystyczne, wykorzystywane do analizy korpusu, są zaczerpnięte z dziedziny teorii informacji. Ich opis, zamieszczony poniżej, jest oparty na opracowaniu [BPPM93].

Rozważmy problem tłumaczenia tekstu pomiędzy językiem francuskim, a angielskim. Oznaczmy przez  $e$  łańcuch znaków z tekstu angielskiego, a przez  $f$  - łańcuch znaków z tekstu francuskiego. Łańcuch  $e$  może być przetłumaczony na język francuski na wiele różnych sposobów. Wybór sposobu tłumaczenia zależy w dużej mierze od kontekstu, w którym wystąpił  $e$ . Ogólnie, w tłumaczeniu statystycznym przyjmujemy, że każdy łańcuch może być tłumaczeniem łańcucha  $e$ .

Do każdej pary łańcuchów znaków  $(e, f)$  oznaczamy przez  $P(f|e)$  prawdopodobieństwo, że łańcuch  $e$  został przetłumaczony na  $f$ . Informacja taka jest interpretowana jako istnienie w korpusie przykładu, którego zdanie źródłowe zawiera łańcuch  $f$ , a zdanie docelowe zawiera łańcuch  $e$ . W przypadku tłumaczenia łańcucha  $f$  z języka francuskiego na angielski, szukamy takiego łańcucha  $\hat{e}$ , dla którego  $P(e|f)$  jest maksymalne.

Na podstawie Twierdzenia Bayes'a, otrzymujemy:

$$P(e|f) = \frac{P(e)P(f|e)}{P(f)} \quad (2.1)$$

gdzie:

- $P(e)$  jest prawdopodobieństwem, że losowo wybrany łańcuch znaków z części źródłowej korpusu jest równy  $e$ ;
- $P(f)$  jest prawdopodobieństwem, że losowo wybrany łańcuch znaków z części docelowej korpusu jest równy  $f$ .

Ponieważ mianownik ułamka po prawej stronie wzoru jest niezależny od  $e$ , otrzymujemy:

$$\hat{e} = \arg \max_{e \in e^*} P(e)P(f|e) \quad (2.2)$$

gdzie  $e^*$  jest zbiorem wszystkich łańcuchów znaków w angielskiej części korpusu.



Na podstawie wzoru 2.2 można zbudować system tłumaczenia statystycznego, który przeszukuje wszystkie możliwe łańcuchy języku docelowym, szukając tego o największym prawdopodobieństwie. System taki nie byłby jednak efektywny, dlatego w praktyce stosuje się heurystyki, pozwalające ograniczyć zbiór potencjalnych tłumaczeń.

Wśród zalet tłumaczenia statystycznego można wymienić łatwość opracowywania nowych kierunków tłumaczenia. Przy założeniu, że dysponujemy gotowym tłumaczem statystycznym, do obsługi nowego kierunku konieczne jest wyłącznie pozyskanie odpowiedniego korpusu tekstów. Oprócz tego, łatwość trenowania systemu tłumaczenia statystycznego sprawia, że bez dużego nakładu pracy możliwe jest wykonanie tłumacza specjalizowanego w określonym typie tekstów. W tej sytuacji potrzebny jest wyłącznie odpowiedniej wielkości korpus tekstów specjalistycznych.

Najczęściej wymieniane wady tłumaczenia statystycznego są następujące:

- Choć tekst wyjściowy wydaje się być płynnym, często zawiera kolokacje słów i fraz, które nie mają sensu.
- Metody statystyczne mają ograniczoną skuteczność w przypadku tłumaczenia języków o silnie różniącej się składni (np. język japoński i języki europejskie).
- W tekście wyjściowym występują problemy z morfologią słów.
- Występują problemy z tzw. słowami OOV (ang. *out of vocabulary*), tzn. takimi, które nie znajdują się w korpusie.

### 2.2.3. Tłumaczenie oparte na przykładzie

Tłumaczenie oparte na przykładzie (ang. *Example-Based Machine Translation*, w skrócie EBMT) jest techniką tłumaczenia automatycznego, opartego na pamięci tłumaczeń. Idea EBMT została zaproponowana przez Makoto Nagao w artykule [Nag84] z 1984 roku. Polega ona na automatycznym tłumaczeniu zdania wejściowego na język docelowy jedynie w oparciu o przykład pamięci tłumaczeń. W odróżnieniu od tłumaczenia statystycznego, przykłady nie są rozbijane na pojedyncze słowa lub frazy i analizowane metodami statystycznymi. Do tłumaczenia wykorzystywane są przykłady w całości.

Jak wskazuje autor artykułu [Som99], z powodu wielości i różnorodności realizacji EBMT, trudno jest nakreślić wzorzec systemu tłumaczenia opartego na przykładzie. Jednak wspólne dla większości systemów tej klasy są dwie fazy tłumaczenia.

1. Wyszukiwanie w pamięci tłumaczeń przykładów, których zdanie źródłowe jest podobne do wejściowego.
2. Rekombinacja - dostosowanie przykładów, w celu wygenerowania tłumaczenia jak najwyższej jakości.

W fazie pierwszej EBMT może być wykorzystany opisany w niniejszej pracy autorski algorytm przeszukiwania pamięci tłumaczeń, którego szczegółowy opis znajduje się w Podrozdziale 4.4. Jednak głównym przeznaczeniem tego algorytmu jest technika wspomaganie tłumaczenia ludzkiego, opisana w Rozdziale 3.

Faza rekombinacji polega na takim zmodyfikowaniu odnalezionych w pamięci tłumaczeń przykładów, aby były maksymalnie podobne do zdania wejściowego. Po modyfikacjach, zdaniom źródłowym przykładów są wykorzystywane do wygenerowania tłumaczenia.

Przykładowy system klasy EBMT może działać według algorytmu opisanego na Rysunku 2.3. Niech  $T$  oznacza pamięć tłumaczeń,  $w$  zdanie wejściowe do tłumaczenia,  $sim$  funkcję podobieństwa zdań.

---

#### Algorytm: Przykładowa realizacja EBMT

---

1. Znajdź taki przykład  $p \in T$ , że  $sim(w, p.source)$  jest maksymalne.
  2. Przetwórz  $p.source$  tak, aby  $p.source = w$ , zapisując listę  $D$  operacji do tego koniecznych.
  3. Dla każdej operacji w  $D$ :
    - a) Wykonaj operację na zdaniu  $p.target$ , o ile to możliwe.
  4. Zwróć zmodyfikowane  $p.target$  jako wynik tłumaczenia.
- 

Rysunek 2.3. Przykładowy algorytm EBMT

Działanie algorytmu zostanie zilustrowane poniższy przykład. Załóżmy, że tłumaczone jest zdanie “Dzisiaj jest 12.03.2005” z języka polskiego na angielski. W pamięci tłumaczeń został odnaleziony przykład:


**pl** : Dzisiaj jest 03.11.1999.

**en** : Today is 1999/11/03.

Aby upodobnić zdanie źródłowe przykładu do zdania wejściowego, należy w nim podmienić datę. Lista  $D$  ma wobec tego postać:

$$D = \{date(03.11.1999) \rightarrow date(12.03.2005)\}.$$

Operacja podmiany daty jest wykonywana na zdaniu docelowym przykładu. Ponieważ zdanie jest w języku angielskim, data zostaje sformatowana zgodnie ze standardem tego języka. Zmodyfikowane zdanie docelowe ma więc postać: “Today is 2005/03/12” i zostaje zwrócone jako wynik tłumaczenia.

Największą zaletą tłumaczenia opartego na przykładzie jest wysoka jakość tłumaczeń w sytuacji, w której dostępny jest odpowiedni przykład. Podstawową wadą jest jednak niska skuteczność w sytuacji, kiedy pamięć tłumaczeń nie zawiera przykładu dostatecznie podobnego do zdania wejściowego. Jakość tłumaczenia EBMT jest więc ściśle związana z jakością używanej w systemie pamięci tłumaczeń.  niniejszej pracy proponuję autorską metodę przygotowywania specjalizowanej pamięci wysokiej jakości. Jest ona opisana w Rozdziale 5.

## Rozdział 3

# Wspomaganie tłumaczenia ludzkiego

Autorskie algorytmy przeszukiwania i przetwarzania pamięci tłumaczeń, przedstawione w niniejszej pracy, powstały na potrzeby techniki komputerowego wspomaganie tłumaczenia. W niniejszym rozdziale technika ta zostanie szczegółowo opisana.

### 3.1. Idea wspomaganie tłumaczenia

Technika komputerowego wspomaganie tłumaczenia (ang. *Computer-Aided Translation*, w skrócie CAT) została zaprojektowana w celu ułatwienia pracy tłumacza ludzkiego. W istocie, pod terminem CAT kryje się wiele różnorodnych mechanizmów. Najważniejszym z nich jest generowanie podpowieździ tłumaczenia zdań. Sugestie te są przeglądane przez tłumacza ludzkiego i pomagają mu opracować końcową, wygładzoną wersję tłumaczenia.

#### 3.1.1. Potrzeby tłumacza ludzkiego

Można wyróżnić następujące potrzeby tłumacza ludzkiego, zajmującego się profesjonalnym tłumaczeniem tekstów:

- dostęp do bazy uprzednio opracowanych tłumaczeń;
- automatyczne wyszukiwanie w bazie aktualnie tłumaczonego zdania oraz zdań do niego podobnych;
- otrzymywanie oceny podobieństwa aktualnie tłumaczonego zdania do wyszukanych w bazie zdań podobnych.

Przed opracowaniem systemów CAT, tłumacze chcący uzyskać dostęp do wcześniej przetłumaczonych tekstów stosowali notatki w formie papierowej. Choć czas odszukania odpowiednich treści w notatkach był znacznie dłuższy, niż w przypadku systemu komputerowego, był on wciąż krótszy od czasu tłumaczenia zdania bez żadnej pomocy. Co więcej, notatki pozwalały na uzyskanie spójności tłumaczeń. Pozwalały zagwarantować, że dane zdanie bę-

dzie tłumaczone zawsze w ten sam sposób. Spójność ma szczególne znaczenie w przypadku tłumaczenia tekstów, zawierających słownictwo specjalistyczne.

Na przeci<sup>o</sup>pisanym wyżej potrzebo<sup>o</sup>wychodzi mechanizm tworzenia i przeszukiwania pamięci tłumaczeń.

### 3.1.2. Zarys mechanizmu wykorzystania pamięci tłumaczeń

Z technicznego punktu widzenia, główna trudność leży w opracowaniu sposobu efektywnego przeszukiwania bazy wcześniej wykonanych tłumaczeń. W celu rozwiązania tego problemu, większość systemów CAT używa pamięci tłumacze<sup>o</sup> podczas tłumaczenia zdania wejściowego, system przeszukuje pamięć tłumaczeń wszystkich przykładów, których zdanie źródłowe jest podobne do wejściowego. Zdania docelowe tych przykładów służą tłumaczowi<sup>o</sup> jako sugestie podczas tłumaczenia zdania wejściowego. W przypadku, kiedy pamięć tłumaczeń nie zawiera przykładów podobnych, zdanie wejściowe jest tłumaczone ręcznie. Następnie, nowo powstały przykład jest dodawany do pamięci, w celu jej wzbogacenia.

### 3.1.3. Inne mechanizmy CAT

Oprócz przeszukiwania pamięci tłumaczeń, systemy klasy CAT wykorzystują także inne mechanizmy, mające ułatwić pracę ludzkiego tłumacza.

#### Zarządzanie terminologią

Zarządzanie terminologią jest mechanizmem automatycznego przeszukiwania słowników podczas tłumaczenia zdania. Jest to mechanizm implementowany w większości systemów klasy CAT. W trakcie tłumaczenia danego zdania wejściowego, wyszukiwane są słownikowe dopasowania słów i fraz, które w tym zdaniu wystąpiły.

Zazwyczaj przeszukiwanych jest wiele słowników, podzielonych na dwie kategorie: wbudowane słowniki systemowe oraz tzw. glosariusze - słowniki tworzone przez użytkownika. Słowniki systemowe są zwykle znacznej wielkości zbiorami terminów, zawierającymi szeroki zakres słownictwa. Natomiast glosariusze użytkownika zbierają tłumaczenia słów, nad którymi tłumacz najczęściej pracuje.

Słowniki pełnią podobną funkcję do pamięci tłumaczeń. Pozwalają na odszukanie wcześniej wykonanych tłumaczeń i zapewnienie ich spójności. Dobre zarządzanie terminologią jest szczególnie przydatne w sytuacji, w której mechanizm przeszukiwania pamięci tłumaczeń nie jest w stanie wygenerować użytecznych podpowiedzi.

## Tłumaczenie maszynowe

Tłumaczenie maszynowe, opisane szczegółowo w Rozdziale 2, może stanowić mechanizm wspomaganie tłumaczenia. W niektórych systemach CAT, jeśli przeszukiwanie pamięci tłumaczeń nie zwraca zadowalających wyników, zdanie wejściowe jest tłumaczone na język docelowy automatycznie. Takie tłumaczenie, mogące zawierać błędy, jest następnie wygładzane przez tłumacza.

Niestety, jak wskazują tłumacze, poprawienie tłumaczenia maszynowego jest często bardziej pracochłonne, niż przetłumaczenie zdania bez żadnej pomocy.

## Narzędzia wspomagające przetwarzanie dokumentów

Pod pojęciem CAT kryją się także czysto techniczne mechanizmy wspomagające przetwarzanie dokumentów. Należą do nich na przykład:

- mechanizmy dzielenia tekstu na zdania (takie jak opisane w Podrozdziale 5.2.1);
- konwertery elektronicznych formatów dokumentów (np. doc, docx, odt, pdf);
- narzędzia typu OCR, odczytujące tekst z zeskanowanych dokumentów papierowych;
- korektor pisowni;
- środowisko pracy tłumacza - ergonomiczna aplikacja komputerowa, zapewniająca dostęp do mechanizmów CAT.

Twórcy systemów klasy CAT prześcigają się w opracowywaniu nowych mechanizmów, mających ułatwić tłumaczenie ludzkie.

## 3.2. Historia systemów CAT

### 3.2.1. Wczesne początki i specjalizacja

Wczesne początki technik zwanych dzisiaj wspomaganie tłumaczenia ludzkiego sięgają lat 80-tych XX wieku (patrz [Hut07]), kiedy to systemy tej klasy zostały opracowane w Japonii. Japońskie firmy komputerowe (Fujitsu, Hitach, NEC, Sharp i Toshiba) pracowały nad oprogramowaniem, mającym ułatwić proces tłumaczenia szczególnie w kierunkach japońsko-angielskim oraz angielsko-japońskim. Inne kierunki tłumaczenia również były brane pod uwagę. Systemy te opierały się na tłumaczeniach automatycznych, które były poprawiane przez ludzkich tłumaczy. Tłumaczenie maszynowe, wykonywane

przez systemy, opierało się albo na transferze bezpośrednim, albo na bardzo powierzchownej analizie morfologicznej słów.

Co ciekawe, systemy te skupiały się na wąskich dziedzinach tekstów. Korzyścią z tego ograniczenia było zmniejszenie kosztów opracowywania zasobów lingwistycznych (ze względu na mniejsze słowniki), szybsze tłumaczenie (z tego samego powodu) oraz lepsza jakość tłumaczeń automatycznych. Większość opracowanych w tym czasie systemów była skupiona na tekstach z dziedziny informatyki i technologii informacyjnej.

### 3.2.2. Pionierski system ALPS

Ważnym kamieniem milowym w rozwoju systemów klasy CAT było opracowanie systemu ALPS w roku 1981 (jak podano w artykule [Hut07]). ALPS był pierwszym systemem wspomaganie tłumaczenia, opracowanym w całości o komputerowo wspomaganych i wypuszczonym na rynek. Oferował następujące funkcjonalności:

- wielojęzyczne przetwarzanie tekstu,
- automatyczne przeszukiwanie słownika,
- zarządzanie terminologią,
- tłumaczenie interaktywne,
- ekstrakcja powtórzeń.


Zwłaszcza ta ostatnia funkcjonalność jest godna wzmianki, gdyż stanowiła ona wczesną wersję pamięci tłumaczeń. Wszystkie tłumaczenia opracowane przez tłumacza były zapisywane w tzw. “pliku powtórzeń”. Podczas pracy nad tłumaczeniem nowego zdania, tłumacz mógł porównać je ze zdaniami w pliku powtórzeń. Proces ten był ułatwiony dzięki automatycznemu wyszukiwaniu fragmentów zdań.

Niestety, ALPS nie odniósł sukcesu w sprzedaży. Nie mniej jednak, wiele nowych systemów wspomaganie tłumaczenia zostało opracowanych krótko po jego wycofaniu z rynku.


### 3.2.3. Rozwój systemów CAT

Pod koniec lat 1980-tych tłumacze odkryli korzyści, płynące ze stosowania komputerowych narzędzi wspomaganie tłumaczenia. Narzędzia klasy CAT nie tylko ułatwiały sam proces tłumaczenia dokumentu, ale także pomagały w organizacji zleceń tłumaczenia. Systemy takie są dziś znane jako “Środowiska pracy tłumacza” (ang. *Translator’s workstation*). Najwcześniejszymi sprzedawcami środowisk pracy tłumacza, jak wskazuje autor artykułu [Hut07], byli:

- Trados (Translator's Workbench, system rozwijany do dziś)
- STAR AG (Transit)
- IBM (TranslationManager, nieobecny już na rynku)
- Eurolang Optimizer (nieobecny na rynku).

W ostatnich czasach  latach 1990-tych i w pierwszej dekadzie XXI wieku, na rynku pojawiło się znacznie więcej wytwórców oprogramowania typu *Translator's workstation*:


- Atril (Déjà Vu)
- SDL (system SDLX)
- Xerox (XMS)
- Terminotix (LogiTerm)
- MultiCorpora (MultiTrans)
- Champollion (WordFast)
- MetaTaxis
- ProMemoria
- Kilgray Translation Technologies (system memoQ) 

W dzisiejszych czasach systemy typu CAT są bardzo popularne zarówno wśród tłumaczy niezależnych, jak i zrzeczonych w biurach tłumaczeniowych 

### 3.3. Studium użyteczności systemów CAT

#### 3.3.1. Systemy CAT na rynku

W ostatnich latach, systemy klasy CAT zrewolucjonizowały rynek tłumaczeń (patrz [Twi06] oraz [CGSSO04]). Dzięki temu, że systemy te pozwalają zredukować koszty tłumaczenia, biura tłumaczeń, które ich używają, są w stanie oferować swoje usługi w niższych cenach. Ponadto, spójność tłumaczeń, zapewniana przez systemy CAT stała się nie tylko pożądaną, ale wymaganą cechą. W rezultacie, komputerowe wspomaganie tłumaczenia pozwala biurom tłumaczeń na oferowanie lepszych tłumaczeń za niższą cenę i uzyskać tym samym znaczącą przewagę nad tymi, które pracują w sposób tradycyjny.

Jednakże, w dalszym ciągu wielu tłumaczy stroni od używania systemów CAT. Niektóre biura wymuszają na swoich pracownikach używanie tego typu systemów  podobnie, nie jest prawdą, że wszyscy tłumacze niezrzeszeni korzystają z pomocy narzędzi CAT. Przeciwnicy CAT argumentują, że systemy te generują tłumaczowi dodatkową pracę. Wygładzenie tłumaczenia zdania z jednoczesnym sprawdzeniem terminologii i innymi procedurami może, ich zdaniem, zająć więcej czasu, niż przetłumaczenie zdania bez żadnej pomocy.




Zostały przeprowadzone liczne badania, mające ustalić, czy narzędzia typu CAT faktycznie przyczyniają się do zmniejszenia ilości pracy tłumacza. W kolejnym podrozdziale przedstawione są wyniki jednego z tych badań.

### 3.3.2. Indywidualne studium użyteczności

W przypadku tłumaczy niezrzeszonych, wyniki i wnioski z analizy opisanej w artykule [Val05] mogą dostarczyć odpowiedzi na pytanie, czy systemy CAT przyczyniają się do zmniejszenia ilości pracy nad tłumaczeniem. Autorka tego artykułu jest profesjonalną tłumaczką, która pracowała w swoim zawodzie jeszcze przed popularyzacją narzędzi wspomagających tłumaczenie. Podczas rewolucji spowodowanej wprowadzeniem tych narzędzi na rynek, autorka tłumaczyła przy użyciu wielu różnych systemów CAT, gdyż klienci często wymagali pracy na konkretnych narzędziach.

Autorka zgadza się z tezą, że używanie narzędzi wspomagających tłumaczenie podnosi wydajność pracy. Postanowiła jednak możliwie dokładnie zmierzyć te korzyści. Kalkulacja taka ma szczególne znaczenie, gdyż większość klientów wymaga obniżenia ceny tłumaczenia w sytuacji, gdy dostępna jest pamięć tłumaczeń. Konieczna jest wiedza, czy zysk z używania narzędzia CAT rekompensuje zmniejszone wynagrodzenie za wykonane tłumaczenie.

Podczas eksperymentu, autorka artykułu [Val05] używała trzech różnych narzędzi CAT: DejaVu (wersja X), Trados (wersja 6) oraz niekomercyjnego narzędzia, opracowanego przez jednego z jej klientów. W DejaVu i Tradosie autorka używała własnych pamięci tłumaczeń, zebranych na przestrzeni lat pracy, liczących po około 150 000 przykładów każda. Zlecenia wykonane przez autorkę zostały podzielone na dwie kategorie: pełnopłatne oraz objęte obniżką ceny. W pierwszym przypadku klient nie wymagał obniżki ceny w związku z używaniem systemu CAT, a tłumaczka używała swoich własnych pamięci tłumaczeń. W drugim przypadku klient dostarczył tłumacze wyspecjalizowaną pamięć tłumaczeń, nałożył wymóg jej użycia oraz zarząda  obniżki ceny tłumaczenia zdań w niej odnalezionych. W związku z tym, ze względu na narzędzie CAT oraz politykę obniżki cen, zlecenia tłumaczenia zostały podzielone na następujące kategorie:

1. Trados, pełnopłatne
2. Trados, z obniżką
3. DejaVu, pełnopłatne
4. DejaVu, z obniżką
5. Narzędzie niekomercyjne, pełnopłatne
6. Narzędzie niekomercyjne, z obniżką

Tablica 3.1. Ilości pracy w analizie produktywności

CAT	L. projektów	L. słów	Całkowity czas (h)
Trados	36	158940	192.8
DejaVu	25	42525	74.35
niekomercyjny	26	155023	271.2
brak	3	2617	6.5

Tablica 3.2. Produktywność tłumaczenia dla różnych narzędzi CAT

Narzędzie CAT	Produktywność
Trados (pełnopłatne i z obniżką)	824.3
Trados (pełnopłatne)	424.5
Trados (z obniżką)	1104.3
DejaVu (pełnopłatne i z obniżką)	571.9
niekomercyjny (tylko z obniżką)	571.6
brak	250

## 7. Bez użycia CAT


Ilość pracy nad tłumaczeniem dokumentów, wzięta pod uwagę w tej analizie, jest przedstawiona w Tabeli 3.1.

Produktywność tłumaczenia została zmierzona w jednostce słów na godzinę. Punktem odniesienia była produktywność uzyskana bez użycia narzędzi wspomagających, wynosząca **250** słów na godzinę. Wyliczona produktywność tłumaczenia przy użyciu danych narzędzi CAT jest przedstawiona w Tabeli 3.2.

### 3.3.3. Wnioski dotyczące użyteczności systemów CAT

Uzyskane przez autorkę wyniki w bardzo znaczący sposób wskazują na zysk z używania narzędzi typu CAT. Zrozumiałym jest, że produktywność jest wyższa w sytuacji, w której klient dostarcza specjalizowaną pamięć tłumaczeń. Jakość pamięci jest tak samo ważna, jak użyteczność samego narzędzia CAT. Specjalizowane pamięci tłumaczeń, dedykowane na potrzeby konkretnego zlecenia tłumaczenia, zapewniają wysokie prawdopodobieństwo generowania użytecznych sugestii tłumaczenia. W świetle powyższych faktów, wymaganie klienta odnośnie obniżenia ceny tłumaczenia jest uzasadnione.


Powyższa analiza wykazała również, że używanie narzędzia CAT z pamięcią tłumaczeń, powstałą na przestrzeni lat pracy, pozwala na uzyskanie wyższej produktywności, niż w przypadku braku narzędzi wspomagających.

Z przeprowadzonej analizy wynika, że komputerowe wspomaganie tłumaczenia może być praktyczną, użyteczną techniką, którą warto udoskonalać 

### 3.4. Współczesny system CAT - memoQ

W niniejszym podrozdziale opisany zostanie system memoQ (opisany na [mula]), opracowany przez firmę Kilgray Translation Technologies. Jego moduł przeszukiwania pamięci tłumaczeń posłuży jako punkt odniesienia w testach autorskiego algorytmu przeszukującego.

#### 3.4.1. Opis ogólny

MemoQ jest nowoczesnym narzędziem klasy CAT, które może być uznane  pełni funkcjonalne środowisko pracy tłumacza. Premiera systemu na rynku miała miejsce w roku 2006. Dzięki jego funkcjonalności i rozszerzalności, memoQ zyskał dużą popularność i jest w dalszym ciągu rozwijany.

Do funkcji memoQ należą:

- pamięć tłumaczeń,
- baza terminologii,
- automatyczne sprawdzanie poprawności tłumaczenia (na podstawie pamięci tłumaczeń, bazy terminologii, korektorów pisowni i wielu innych),
- ergonomiczny interfejs edycji tłumaczenia,
- edytor pamięci tłumaczeń,
- analizator tagów formatujących,
- zrównoleglacz tekstów,
- podgląd dokumentów w formatach .doc, .docx, .ppt, .pptx, .html oraz XML,
- obsługa plików wymiany pamięci tłumaczeń w formacie TMX, XLIFF, oraz dwujęzycznym DOC/RTF,
- kompatybilność z systemami Trados, WordFast i STAR Transit,
- elastyczna architektura, ułatwiająca dołączanie dodatków.

Jednym z najważniejszych czynników, budujących siłę systemu memoQ jest integracja z innymi popularnymi narzędziami typu CAT. Cecha ta ułatwia tłumaczom przejście z innych narzędzi na memoQ.


#### 3.4.2. Pamięć tłumaczeń w systemie memoQ

Podobnie jak w przypadku większości narzędzi klasy CAT, dostępnych na rynku, główną funkcjonalnością systemu memoQ jest obsługa pamięci tłumaczeń. Pamięć jest przeszukiwana pod kątem tzw. “dopasowań 100%”

(przykładów, których zdanie źródłowe jest identyczne z wejściowym), jak również pod kątem tzw. “dopasowań nieostrych” (przykładów, których zdanie źródłowe jest tylko podobne do wejściowego w sensie określonej nieostrej miary podobieństwa). System memoQ podaje procentową ocenę dopasowania dla każdego znalezionej w pamięci tłumaczeń wyniku. Użytkownik ma możliwość ustalenia procentowego progu dla dopasowań z pamięci. Dopasowania, które otrzymują ocenę podobieństwa poniżej tego progu nie są zwracane jako sugestie tłumaczenia.

Wyróżniającą cechą pamięci tłumaczeń memoQ jest podawanie tzw. “dopasowań 101%”. Sztuczna ocena 101% jest przypisywana takim dopasowaniom 100%, które znalazły się w tym samym kontekście w pamięci tłumaczeń i w tłumaczonym dokumencie. W przypadku tych dopasowań tłumacz ma gwarancję, że wymagają one minimalnego nakładu pracy na wygładzenie tłumaczenia. Dopasowanie na podstawie kontekstu jest określane w nomenklaturze systemu jako “dopasowanie ICE”.

Kolejną interesującą techniką, używaną w memoQ, jest tzw. “dopasowywanie SPICE”. Dostarcza ona innej możliwości uzyskania przez przykład z pamięci tłumaczeń oceny 101%. Dopasowywanie SPICE jest możliwe, jeśli tłumaczony dokument jest w formacie XML. Ocena 101% przyznawana jest tym dopasowaniom 100%, które odnoszą się do tej samej informacji w sensie atrybutów XML i ich wartości. Taka sytuacja jest powszechna na przykład w dokumentach lokalizacyjnych.


Pamięć tłumaczeń systemu memoQ ma jednak wadę - nie jest dobrze zoptymalizowana pod kątem szybkości przeszukiwania. Jest to w dużej mierze spowodowane używaniem nieostrej miary podobieństwa zdań, która nie jest najszybszym znanym algorytmem wyszukiwania zdań 

W praktyce, szybkość narzędzia CAT jest jednym z głównych czynników, decydujących o jego użyteczności. Tłumacz musi otrzymać sugestie tłumaczenia zdania i wygładzić tłumaczenie w czasie krótszym, niż tłumaczenie zdania bez pomocy. W związku z tym, szybkość przeszukiwania i generowania sugestii jest kluczowa, zwłaszcza w przypadku obsługiwanego znacznej wielkości pamięci tłumaczeń.

## Rozdział 4

# Przeszukiwanie pamięci tłumaczeń

### 4.1. Motywacja badań

Główną motywacją prowadzenia badań nad przeszukiwaniem pamięci tłumaczeń jest dążenie do opracowania efektywnych algorytmów, zdolnych sprostać wymaganiom obecnych systemów wspomaganie tłumaczenia ludzkiego, opisanych w Rozdziale 3. Pamięci tłumaczeń w dzisiejszych zastosowaniach mają coraz większy rozmiar. Najczęściej waha się on od setek tysięcy do kilku milionów przykładów. W przygotowaniu są projekty mające na celu zebranie nawet większych pamięci tłumaczeń, takie jak Wordfast VLTm (opisane na [mulb]). W tej sytuacji konieczne jest zastosowanie algorytmów przeszukiwania o niskiej złożoności obliczeniowej zarówno czasowej, jak i pamięciowej. Akceptowalne rozwiązania powinny mieć logarytmiczną złożoność czasową względem wielkości pamięci tłumaczeń oraz wielomianową względem długości zdania wyszukiwanego .

Inną wymaganą cechą algorytmów przeszukiwania pamięci tłumaczeń jest podobieństwo wyszukiwanych zdań do zdania wzorcowego. Prowadzone są badania nad opracowaniem takiej miary podobieństwa zdań, która dobrze odzwierciedlałaby ludzką intuicję. Ma to duże znaczenie zarówno teoretyczne, jak i praktyczne. Z punktu widzenia teorii, problem ten polega na znalezieniu odpowiedniej funkcji dystansu pomiędzy dwoma obiektami - zdaniami w języku naturalnym. Zastosowania praktyczne - systemy wspomaganie tłumaczenia - wymagają, aby algorytm wyszukujący zwracał zdania podobne i tylko takie zdania przy każdym wyszukiwaniu. Wyszukiwanie nie może pominąć zdań z pamięci tłumaczeń, które są podobne do wyszukiwanego, jak i nie może zwrócić zbyt szerokiego zbioru wyników, który zawierałby zdania niepodobne do wzorca.


## 4.2. Problem wyszukiwania przybliżonego


### 4.2.1. Sformułowanie problemu

Problem wyszukiwania zdań podobnych w pamięci tłumaczeń jest związany z problemem przybliżonego wyszukiwania łańcuchów znaków. Przybliżone wyszukiwanie łańcuchów znaków (ang. *approximate string matching*) polega na wyszukaniu w długim tekście wszystkich podłańcuchów, które różnią się od wyszukiwanego wzorca najwyżej nieznacznie. Różnica ta (nazywana błędem) jest obliczana precyzyjnie przy użyciu wybranej funkcji dystansu.

Bardziej formalnie, problem przybliżonego wyszukiwania łańcuchów znaków definiuje się następująco (za [Nav01]).

**Definicja 12** (Problem wyszukiwania przybliżonego). *Niech będą dane:*

- skończony alfabet  $\Sigma$ , o rozmiarze  $\sigma$ ,  $|\Sigma| = \sigma$
- tekst  $T$  o długości  $n$ ,  $T \in \Sigma^*$ ,  $|T| = n$
- wzorzec  $P$  o długości  $m$ ,  $P \in \Sigma^*$ ,  $|P| = m$
- liczba  $k \in \mathbb{R}$ , reprezentująca dozwolony błąd 
- funkcja dystansu  $d : \Sigma^* \times \Sigma^* \mapsto \mathbb{R}$

**Problem wyszukiwania przybliżonego** polega na odnalezieniu wszystkich takich pozycji  $j$  w tekście  $T$ , że  $\exists_i d(P, T[i..j]) \leq k$ , gdzie  $T[i..j]$  oznacza podłańcuch tekstu  $T$  od pozycji  $i$  do pozycji  $j$ . 

### 4.2.2. Funkcje dystansu

Poniżej przedstawione zostaną niektóre funkcje dystansu wykorzystywane przy rozważaniach nad problemem wyszukiwania przybliżonego.

Najstarszą i najczęściej wykorzystywaną funkcją dystansu jest odległość Levenshteina ([Lev65]).

**Definicja 13** (Odległość Levenshteina). *Odległość Levenshteina (oznaczana  $d_L$ ) pomiędzy dwoma łańcuchami znaków jest liczbą naturalną, która odpowiada minimalnej liczbie operacji podstawowych koniecznych do przekształcenia jednego łańcucha znaków w drugi. Operacjami podstawowymi są:*

- *insercja* - wstawienie dowolnego znaku do łańcucha na dowolnej pozycji,
- *delecja* - usunięcie dowolnego znaku z łańcucha,
- *substytucja* - zamiana dowolnego znaku łańcucha innym znakiem.

#### Przykład:

Dla łańcuchów znaków 'orczyk' i 'oracz' odległość Levenshteina wynosi 3, gdyż do przekształcenia pierwszego łańcucha w drugi konieczne są operacje:

1. Delecja znaku **y** ('orczk')
2. Delecja znaku **k** ('orcZ')
3. Insercja znaku **a** ('oracz')

Odległość Levenshteina jest symetryczna (gdyż operacja odwrotna do każdej operacji podstawowej jest operacją podstawową). Dla łańcuchów znaków  $x$  i  $y$  takich, że  $|x| = |y|$  odległość ta spełnia dodatkowo warunki  $d_L(x, y) < \infty$  oraz  $0 < d_L(x, y) < |x|$ . Co więcej, funkcja ta jest metryką w przestrzeni łańcuchów znaków.

Do obliczenia odległości Levenshteina pomiędzy dwoma ciągami znaków wykorzystywany jest algorytm Needlemana-Wunscha (opisany w [NW70a]), który działa w czasie  $O(|x| \cdot |y|)$ .

Pokrewną, często stosowaną funkcją dystansu łańcuchów znaków jest odległość Hamminga ([Ham50]).

**Definicja 14** (Odległość Hamminga). *Odległość Hamminga (oznaczana  $d_H$ ) jest określona na łańcuchach znaków równej długości jako minimalna liczba substytucji koniecznych do przekształcenia jednego łańcucha w drugi. Innymi słowy, odległość Hamminga jest liczbą pozycji, na których dwa łańcuchy znaków się różnią.*

**Przykład:**

Odległość Hamminga pomiędzy łańcuchami:

*worek*

*totem*

wynosi 3.

Odległość Levenshteina stanowi uogólnienie odległości Hamminga. Istnieją jednak pary łańcuchów znaków, dla których odległości te mają różne wartości. Na przykład, dla łańcuchów:

$x = '0123456789'$

$y = '1234567890'$

$d_H(x, y) = 10$ , gdyż łańcuchy te różnią się na wszystkich 10 pozycjach, ale  $d_L(x, y) = 2$ , gdyż do przekształcenia łańcucha  $x$  w  $y$  wystarczą 2 operacje podstawowe:

1. Delecja znaku **0** z pierwszej pozycji łańcucha
2. Insercja znaku **0** na ostatnią pozycję łańcucha

Podobnie jak odległość Levenshteina, odległość Hamminga spełnia warunki symetryczności, skończoności oraz  $\forall_{x,y} 0 \leq d_H(x, y) \leq |x|$ .

Inną funkcją odległości, opartą na idei operacji podstawowych, jest odległość epizodyczna (ang. *episode distance*), zaproponowana w [DFG<sup>+</sup>97].

**Definicja 15** (Odległość epizodyczna). *Odległość epizodyczna (oznaczana  $d_{EP}$ ) pomiędzy dwoma łańcuchami znaków jest liczbą naturalną, która odpowiada minimalnej liczbie insercji koniecznych do przekształcenia jednego łańcucha znaków w drugi.*

W praktyce, funkcja odległości epizodycznej służy do obliczania podobieństwa ciągów, których elementami są nie znaki, ale zdarzenia. W tej sytuacji ważne jest, aby za podobne zostały uznane wyłącznie te ciągi, które zachowują kolejność elementów.

**Przykłady obliczenia odległości epizodycznej pomiędzy łańcuchami znaków:**

- $d_{EP}('abcd', 'a12bc345d') = 5$ , gdyż do przekształcenia konieczne było wykonanie 5 insercji.
- $d_{EP}('abcd', 'abef') = \infty$ , gdyż nie ma możliwości przekształcenia pierwszego ciągu w drugi wyłącznie za pomocą insercji.

Jak widać z powyższych przykładów, wartość odległości epizodycznej wynosi albo  $|y| - |x|$ , albo  $\infty$ .

Inne podejście stanowi funkcja największego wspólnego podciągu (ang. *longest common subsequence distance*), opisana w [NW70b].


**Definicja 16** (Odległość największego wspólnego podciągu). *Odległość największego wspólnego podciągu (oznaczana  $d_{LCS}$ ) pomiędzy dwoma łańcuchami znaków jest liczbą naturalną, która odpowiada minimalnej liczbie insercji i delecji koniecznych do przekształcenia jednego łańcucha znaków w drugi.*

**Przykłady:**


1.  $d_{LCS}('abcd', 'abde') = 2$ , gdyż konieczna jest delecja znaku **c** oraz insercja znaku **e**.
2.  $d_{LCS}('abcd', 'a1b2c3') = 4$ , gdyż konieczna jest delecja znaku **d** oraz insercja znaków **1, 2, 3**.


Nazwa funkcji nawiązuje do faktu, że w istocie oblicza ona długość największego wspólnego podciągu jednego i drugiego ciągu. Wartość dystansu jest liczbą znaków z obu ciągów wejściowych, które nie należą do znalezionej najdłuższego podciągu. W powyższym przykładzie 1, największym wspólnym podciągiem obu ciągów jest *'abd'*. Nie należą do niego znaki **c** oraz **e**, stąd wartość odległości wynosi 2. W przykładzie 2, największym wspólnym podciągiem jest *'ab'*. Nie należą do tego podciągu znaki **d**, **1**, **2** oraz **3**, więc wartość odległości w tym przypadku wynosi 4.



Odległość największego wspólnego podciągu spełnia warunek symetryczności oraz  $\forall_{x,y} 0 \leq d_{LCS}(x,y) \leq |x| + |y|$ . 


### 4.3. Znane rozwiązania problemu wyszukiwania przybliżonego

Kluczową cechą algorytmów wyszukiwania przybliżonego jest niska złożoność czasowa. Jest ona konieczna przy przeszukiwaniu znacznej wielkości zbiorów danych. 

Ze względu na dostępność danych, algorytmy wyszukiwania dzielimy na dwie klasy: on-line i off-line. W algorytmach typu on-line tekst  $T$  nie może być przetwarzany przed rozpoczęciem wyszukiwania. Natomiast algorytmy off-line korzystają z możliwości wcześniejszego przetworzenia tekstu, tworząc indeksy na potrzeby wyszukiwania.  Indeksy te przyczyniają się do znacznego obniżenia złożoności obliczeniowej wyszukiwania. Ich utworzenie jest obciążone kosztem, ale jest to koszt jednorazowy. Raz utworzony indeks można wykorzystywać do wielu wyszukiwań.

W prezentowanych poniżej opisach algorytmów wyszukiwania przybliżonego przyjmuje się, że funkcją dystansu jest odległość Levenshteina.

#### 4.3.1. Rozwiązanie naiwne

Naiwne rozwiązanie problemu polega na obliczeniu odległości Levenshteina  każdego podciągu  $T$ , a następnie wybranie tych podciągów, które mają najmniejszą wartość dystansu. Pseudokod tego rozwiązania jest zaprezentowany na Rysunku 4.1 (oznaczenia danych wejściowych oraz specyfikacja wyjścia są takie, jak w Definicji 12, wprowadzającej problem wyszukiwania przybliżonego).

---

#### Algorytm: Naiwne wyszukiwanie przybliżone

---

```
function search_naive(T,P,k)
  for windowSize:=1 to n
    for index:=1 to n-windowSize+1
      fragmentEnd:=index+windowSize-1
      if (d(T[index..fragmentEnd],P)<=k)
        resultSet.add(fragmentEnd)

  return resultSet
end function
```


---

Rysunek 4.1. Naiwny algorytm wyszukiwania przybliżonego

Algorytm naiwny jest algorytmem typu on-line. Oszacujmy jego złożoność czasową. Zarówno pierwsza, jak i druga pętla *for* wykona się  $O(n)$  razy. Funkcja dystansu zostanie więc obliczona  $O(n^2)$  razy. Każde jej obliczenie, przy wykorzystaniu algorytmu Needlemana-Wunscha pochłonie  $O(nm)$  obliczeń, gdyż porcja tekstu ma długość rzędu  $n$ , a długość wzorca wynosi  $m$ . Oznacza to, że złożoność obliczeniowa algorytmu naiwnego wynosi  $O(n^3m)$ , co czyni go szczególnie mało przydatnym dla długich tekstów, a więc dużych wartości  $n$ .


### 4.3.2. Algorytm Sellersa

Znacznie lepsze rozwiązanie zaproponował Sellers w artykule [Sel80]. Jego algorytm klasy on-line wykorzystuje technikę programowania dynamicznego.

Algorytm posługuje się alternatywnym sformułowaniem problemu wyszukiwania przybliżonego: dla każdej pozycji  $j$  w tekście  $T$  i każdej pozycji  $i$  we wzorcu  $P$  oblicz najmniejszy dystans pomiędzy  $P[1..i]$ , a  $T[j'..j]$  dla jakiegokolwiek  $j'$ . Dystans ten oznacz przez  $E[i, j]$ . Wtedy rozwiązaniem oryginalnego problemu jest taki podłańcuch  dla którego  $E[m, j]$  jest minimalne.

Wypełnienie tablicy  $E$  jest wykonywane dzięki algorytmowi podobnemu do wspomnianego wcześniej algorytmu Needlemana-Wunscha, służącego do obliczenia odległości Levenshteina pomiędzy dwoma ciągami. Procedura tego wypełnienia jest przedstawiona na Rysunku 4.2.

Po wypełnieniu tablicy, odczytuje się najmniejszą z wartości z rzędu numer  $m$ . Przyjmijmy, że jest to  $E[m, j_k]$ . Następnie, wykorzystując informacje o pochodzeniu wartości tablicy  $E$ , wraca się od elementu  $E[m, j_k]$  do rzędu nr 0. Przyjmijmy, że powrót nastąpił do elementu  $E[0, j_p]$ . Wtedy wynikiem wyszukiwania wzorca  $P$  w tekście  $T$  jest podłańcuch  $T[j_p + 1..j_k]$ . Odległość Levenshteina wyszukanego podłańcucha od wzorca wynosi  $E(m, j_k)$ .

Rozpatrzmy następujący przykład  Niech tekst do wyszukiwania będzie postaci  $T = 'abcdefg'$ , a wzorec do wyszukania:  $P = 'bzdff'$ . Wypełniona tablica w algorytmie Sellersa dla powyższych danych jest przedstawiona w Tabeli 4.1.

W ostatnim wierszu wypełnionej tabeli znajdujemy 2 minimalne wartości. Ścieżki powrotu i rozwiązania dla każdej z nich przedstawione są kolejno w Tabelach 4.2 i 4.3. Pierwszym rozwiązaniem jest podłańcuch  $'bcd'$ , jego odległość od wzorca  $'bzdff'$  wynosi 2 (jedna substytucja, jedna insercja). Drugim rozwiązaniem jest  $'bcde'$ , jego odległość od wzorca również wynosi 2 (2 substytucje).

Złożoność czasowa wypełniania tablicy w algorytmie Sellersa jest rzędu

Tablica 4.1. Wypełniona tablica w algorytmie Sellersa.

		a	b	c	d	e	f	g
	0	0	0	0	0	0	0	0
<b>b</b>	↑1	↖1	↖0	↖1	↖1	↖1	↖1	↖1
<b>z</b>	↑2	↖2	↑1	↖1	↖2	↖2	↖2	↖2
<b>d</b>	↑3	↖3	↑2	↖2	↖1	↖3	↖3	↖3
<b>f</b>	↑4	↑4	↑3	↖3	↑2	↖2	↖3	↖4

Tablica 4.2. Algorytm Sellersa: rozwiązanie pierwsze

		a	b	c	d	e	f	g
	0							
<b>b</b>			↖0					
<b>z</b>				↖1				
<b>d</b>					↖1			
<b>f</b>					↑2			

Tablica 4.3. Algorytm Sellersa: rozwiązanie drugie

		a	b	c	d	e	f	g
	0							
<b>b</b>			↖0					
<b>z</b>				↖1				
<b>d</b>					↖1			
<b>f</b>						↖2		

**Algorytm Sellersa: wypełnienie tablicy**

```

function fill_sellers_table(T,P)

    //zainicjuj rząd zerowy wartościami 0
    for j:=0 to n
        E[0,j] := 0

    for i:=1 to m      //rzędy odpowiadają znakom wzorca
        for j:=0 to n  //kolumny odpowiadają znakom tekstu
            topValue := E[i-1,j]

            leftValue := null
            if (j>0)
                leftValue := E[i,j-1]

            diagonalCost := 1
            if (P[i] == T[j])
                diagonalCost := 0
            diagonalValue := null
            if (j>0)
                diagonalValue := E[i-1,j-1] + diagonalCost

            minValue = min(diagonalValue,leftValue,topValue)
            E[i,j] := minValue

            //zapamiętaj informację, skąd pochodzi wartość tablicy
            if (minValue == diagonalValue)
                E[i,j].from := 'd'
            else if (minValue == topValue)
                E[i,j].from := 't'
            else if (minValue == leftValue)
                E[i,j].from := 'l'

end function

```

Rysunek 4.2. Wypełnienie tablicy w algorytmie Sellersa

$O(mn)$ , gdyż takie są wymiary tej tablicy. Przejście ścieżki powrotu zajmuje  $O(m+n)$  czasu, wobec czego całkowita złożoność czasowa algorytmu wyszukiwania przybliżonego Sellersa wynosi  $O(mn+m+n)$ . Jest to wynik znacznie lepszy od złożoności algorytmu naiwnego, wynoszącej  $O(n^3m)$ .

**4.3.3. Metody oparte na tablicy sufiksów**

Rozwiązania problemu wyszukiwania przybliżonego znalazły zastosowanie w przeszukiwaniu sekwencji DNA. Miało to wpływ na ogromne ożywienie badań nad tym problemem w latach 2000-2010. Najnowsze rozwiązania w tej dziedzinie są algorytmami klasy off-line, bazującymi na indeksach. Opracowanie [NByST00] przedstawia różne metody indeksowania w wyszukiwaniu przybliżonym, w tym metody oparte na tak zwanej tablicy sufiksów.

Tablica 4.4. Przykładowa tablica sufiksów

Sufiks	Nr zdania	Offset
badania, są, prowadzone	1	0
są, prowadzone	1	1
prowadzone	1	2
to, nie, jest, wszystko	2	0
nie, jest, wszystko	2	1
jest, wszystko	2	2
wszystko	2	3
stąd, widać, całe, miasto	3	0
widać, całe, miasto	3	1
całe, miasto	3	2
miasto	3	3

Tablica sufiksów jest strukturą danych, przechowującą informacje o przeszukiwanym tekście. Aby zobrazować konstrukcję tablicy sufiksów, rozważmy następujący przykład. Załóżmy, że dysponujemy trzema zdaniami:

1. badania są prowadzone
2. to nie jest wszystko
3. stąd widać całe miasto

Z każdego zdania tworzone są wszystkie możliwe sufiksy. Pierwszym sufiksem zdania jest całe to zdanie, drugim - całe zdanie bez pierwszego słowa, trzecim - całe zdanie bez dwóch pierwszych słów itd. Ostatni sufiks składa się z pojedynczego ostatniego słowa w zdaniu. W tablicy zostaje przechowany każdy sufiks z każdego zdania. Razem z sufiksem zapisany jest jego offset (patrz Podrozdział 1.1) oraz identyfikator zdania, z którego pochodzi sufiks. Przykładowa tablica sufiksów jest przedstawiona w Tabeli 4.4. Na koniec, tablica zostaje posortowana względem sufiksów (stosując porządek leksyko-graficzny). Wynikowa tablica jest przedstawiona w Tabeli 4.5.

Artykuł [Gho06] opisuje przykładowy algorytm wykorzystujący tę strukturę danych. Autor artykułu [KS10] proponuje nawet użycie pewnego algorytmu bazującego na tablicy sufiksów do przeszukiwania pamięci tłumaczeń (algorytm ten jest dokładniej omówiony w Podrozdziale 4.7.1).

Tablica 4.5. Posortowana tablica sufiksów

Sufiks	Nr zdania	Offset
badania, są, prowadzone	1	0
całe, miasto	3	2
jest, wszystko	2	2
miasto	3	3
nie, jest, wszystko	2	1
prowadzone	1	2
są, prowadzone	1	1
stąd, widać, całe, miasto	3	0
to, nie, jest, wszystko	2	0
widać, całe, miasto	3	1
wszystko	2	3


## 4.4. Autorski algorytm przeszukiwania pamięci tłumaczeń

### 4.4.1. Problem przeszukiwania pamięci tłumaczeń

Problemem pokrewnym wobec wyszukiwania przybliżonego jest problem przeszukiwania pamięci tłumaczeń. Polega on na odnalezieniu w pamięci takich przykładów, których zdania źródłowe są podobne do danego zdania wejściowego w sensie danej funkcji dystansu zdań. Bardziej formalnie, problem przeszukiwania pamięci tłumaczeń można zdefiniować następująco:

**Definicja 17** (Przeszukiwanie pamięci tłumaczeń). *Niech będą dane:*

- pamięć tłumaczeń  $T$
- zdanie wejściowe  $w$
- funkcja dystansu  $d : Z \times Z \mapsto [0, 1]$ , gdzie  $Z$  oznacza zbiór wszystkich możliwych zdań

**Problem przeszukiwania pamięci tłumaczeń** polega na odnalezieniu wszystkich takich przykładów  $p$  w pamięci tłumaczeń  $T$ , że  $d(w, p.source) > 0$   gdzie  $p.source$  oznacza zdanie źródłowe przykładu  $p$ .

Kluczowe dla opracowania skutecznego algorytmu przeszukiwania pamięci tłumaczeń jest znalezienie takiej funkcji podobieństwa zdań  $d$ , która dobrze odzwierciedla ludzką intuicję podobieństwa zdań i ma niską złożoność czasową. Ponadto, sam algorytm tej klasy powinien być zoptymalizowany pod kątem złożoności czasowej i pamięciowej.

W kolejnych podrozdziałach przedstawię autorski algorytm przeszukiwania pamięci tłumaczeń, jego analizę oraz porównanie z najnowszymi osiągnięciami w dziedzinie.

#### 4.4.2. Wymagania odnośnie algorytmu

Autorski algorytm przeszukiwania pamięci tłumaczeń wychodzi naprzeciw stawianym przed nim wymaganiom. Do wymagań funkcjonalnych, wynikających z potrzeb przedstawionych w Podrozdziale 3.1.1, należą:

1. Znalezienie wszystkich przykładów z pamięci tłumaczeń, których zdanie wejściowe jest podobne do wejściowego.
2. Znalezienie w szczególności przykładu, którego zdanie źródłowe jest identyczne z wejściowym, o ile taki przykład znajduje się w pamięci tłumaczeń.
3. Podanie oceny dopasowania (liczby rzeczywistej z przedziału  $[0, 1]$ ) dla każdego odnalezionego przykładu.
4. Podanie w szczególności oceny 1 dla przykładu, którego zdanie źródłowe jest identyczne z wejściowym.


Przed algorytmem stawia się także następujące wymagania нефункционалне:

1. Szybkość działania.
2. Niskie wykorzystanie pamięci.

Opis algorytmu jest podzielony ze względu na procedury przez niego wykorzystywane.

#### 4.4.3. Skrót zdania

Algorytm przeszukiwania pamięci tłumaczeń bazuje na tzw. skrótach zdań. Algorytm generowania skrótu zdania  $z$  jest przedstawiony na Rysunku 4.3.

Funkcja *has*amienia każde słowo w zdaniu stemem tego słowa. Jeśli stem jest zbyt długi, zostaje skrócony do pierwszych 5 znaków. Co więcej, usuwane są z niego wszystkie cyfry. Operacje te mają znaczenie wydajnościowe.

#### 4.4.4. Kodowana tablica sufiksów

Opracowania [CT10] oraz [HkHwLkS04] proponują modyfikacje standardowej tablicy sufiksów, opisanej w Podrozdziale 4.3.3. W szczególności, drugie z wymienionych opracowań przedstawia metodę jej kompresji. Autorski al-

**Algorytm: Generowanie skrótu zdania**


---

```

function hash(z)
  for all (Słowo s in z)
    if (s contains [0..9])
      z.remove(s)
    else
      h := stem(s)
      if (length(h) > 5)
        h := h.substring(0,5)
  return z
end function

```

---

Rysunek 4.3. Algorytm generowania skrótu zdania

gorytm wyszukiwania zdań przybliżonych podąża za ideą zmniejszania rozmiaru tablicy sufiksów. Używa do tego celu własnej metody. Metoda jest oparta na ogólnej idei kodowania algorytmem Huffmana ([Huf52]). Wykorzystywana jest bijekcja  $code : S \mapsto \mathbb{N}$ , gdzie  $S \subset \Theta$  jest zbiorem wszystkich możliwych stemów. Każdemu stemowi, który ma być przechowywany w pamięci, jest przypisywana unikatowa liczba, która zajmuje w pamięci mniej miejsca.

Technika jest skuteczna, o ile w pamięci tłumaczeń jest stosunkowo mało unikatowych stemów. W przeciwnym wypadku, stemy otrzymywałyby kody będące dużymi liczbami, które zajmowałyby zbyt dużo miejsca w pamięci. Wyniki eksperymentalne wskazują, że w średnim przypadku, stosunek liczby unikatowych słów w pamięci tłumaczeń do całkowitej liczby słów wynosi ok. 5% (por. [TI06]). Sytuacja jest jeszcze bardziej korzystna, kiedy rozpatruje się stosunek liczby unikatowych stemów do całkowitej liczby słów. W ramach niniejszej pracy zbadano przykładową pamięć tłumaczeń, liczącą 3 593 227 słów. Liczba unikatowych stemów w tej pamięci wyniosła zaledwie 17 001, co stanowi ok. 0.5%.

**4.4.5. Dodawanie zdania do indeksu**

Zanim możliwe będzie wyszukiwanie zdań, należy utworzyć indeks poprzez dodanie do niego zdań do przeszukiwania. Algorytm jest więc klasy off-line. Do dodawania zdań do indeksu służy procedura *indexAdd*. Przyjmuje ona dwa argumenty: zdanie do dodania oraz jego unikatowy numer. Procedura wykorzystuje słownik stemów *dictionary*, który stanowi tabelkę funkcji `de`. Możliwe operacje na obiekcie *dictionary* są następujące:

- odczytanie liczby będącej kodem stemu (*get*)
- przydzielenie nowego kodu dla nowego stemu (*createNewCode*)



Indeks, oparty na tablicy sufiksów, jest reprezentowany przez obiekt *array*. Obiekt ten posiada metodę *add*, która służy do dodania do indeksu sufiksu wraz z numerem zdania, z którego pochodzi oraz jego offsetem. Dodanie sufiksu do tablicy zachowuje jej porządek. Procedura dodawania jest zoptymalizowana dzięki użyciu algorytmu wyszukiwania binarnego (opisanego w [CSRL01]). Algorytm dodawania zdania do indeksu jest przedstawiony na Rysunku 4.4.

---

**Algorytm: Dodawanie zdania do indeksu**


---

```

procedure indexAdd(z,id)
  h := hash(z)
  for all (Stem s in h)
    code := dictionary.get(s)
    if (code == null)
      code := dictionary.createNewCode(s)
    s := code //zamień stem jego kodem
  for (i := 0 to length(h))
    array.addSuffix(h.subsequence(i,length(h)),id, i)
end procedure

```

---

Rysunek 4.4. Algorytm dodawania zdania do indeksu

Procedura ta najpierw zamienia każdy stem sufiksu przez jego kod. Następnie dodaje zakodowany sufiks do tablicy sufiksów. W związku z tym, w indeksie przechowywane są nie stemy, ale odpowiadające im liczbowe kody.

#### 4.4.6. Procedura *getLongestCommonPrefixes*

Algorytm przeszukiwania indeksu używa funkcji *getLongestCommonPrefixes* oraz obiektu *OverlayMatch*.

Funkcja *getLongestCommonPrefixes* przyjmuje jeden parametr - ciąg stemów. Zwraca zbiór sufiksów z tablicy, które mają najdłuższy wspólny prefiks z wejściowym ciągiem stemów. Funkcja wykorzystuje metodę tablicy *subArray*, która zwraca zbiór sufiksów z tablicy, zaczynających się danym ciągiem stemów. Metoda *subArray* jest zoptymalizowana dzięki technice wyszukiwania binarnego (opisanej w [CSRL01]). Algorytm funkcji *getLongestCommonPrefixes* jest przedstawiony na Rysunku 4.5.

#### 4.4.7. Obiekt *OverlayMatch*

Obiekt *OverlayMatch* przechowuje informacje o wzajemnym pokrywaniu się zdania wyszukiwanego z jednym ze zdań indeksu. Na podstawie tych informacji można obliczyć stopień podobieństwa tych zdań. Każde znalezione

---

**Algorytm: funkcja `getLongestCommonPrefixes`**

---

```

function getLongestCommonPrefixes(h)
  longestPrefixesSet := empty set
  pos := 0
  currentScope := array
  while(not empty(currentScope) and pos < length(h))
    currentScope := currentScope.subArray(h.subSequence(0, pos))
    if (size(currentScope) > 0)
      longestPrefixesSet := currentScope
    pos := pos + 1
  return longestPrefixesSet
end function

```

---

Rysunek 4.5. Funkcja `getLongestCommonPrefixes`

w indeksie zdanie ma przypisany swój własny obiekt *OverlayMatch*. Rysunek 4.6 przedstawia definicję obiektu *OverlayMatch*.

---

**Definicja obiektu *OverlayMatch***

---

```

object OverlayMatch
{
  patternMatches - lista rozłącznych przedziałów, reprezentujących
                  pokrycie zdania wyszukiwanego (pattern) fragmentami
                  zdania z indeksu
  exampleMatches - lista rozłącznych przedziałów, reprezentujących
                  pokrycie zdania z indeksu (example) fragmentami
                  zdania wyszukiwanego
}

```

---

Rysunek 4.6. Obiekt *OverlayMatch*

Na tym etapie, zdania są reprezentowane w indeksie poprzez ciągi liczb. Każda z tych liczb jest kodem, który odpowiada jakiemuś stemowi. Kody w zdaniu są ponumerowane od 0 i odpowiadają kolejno stemom utworzonym ze słów zdania.

Przedział  $[a, b]$  na liście *patternMatches* oznacza, że ciąg kodów zdania *pattern* o numerach od  $a$  do  $b$  (włącznie) jest podciągiem ciągu kodów zdania *example*. Podobnie, przedział  $[a, b]$  na liście *exampleMatches* oznacza, że ciąg kodów zdania *example* o numerach od  $a$  do  $b$  jest podciągiem ciągu kodów zdania *pattern*.

Na przykład, przyjmijmy, że zdanie wyszukiwane (*pattern*) brzmi: “Stąd widać prawie całe miasto”, a zdanie wyszukane w indeksie (*example*): “Stąd widać całe miasto”. Po stemowaniu przykładową funkcją stemowania, odpowiadające tym zdaniom ciągi stemów będą następujące: [’stąd’, ’wid’, ’prawi’, ’cał’, ’miast’] oraz [’stąd’, ’wid’, ’cał’, ’miast’]. Stemy te zostaną zakodowane kolejnymi liczbami naturalnymi, przez co wynikowe ciągi kodów będą nastę-

pujące: [0,1,2,3,4] oraz [0,1,3,4]. Obiekt *OverlayMatch* dla tych dwóch zdań (reprezentowanych przez ciągi kodów) będzie miał postać:

```
patternMatches : { [0,1]; [3,4] }
exampleMatches : { [0,1]; [2,3] }
```

#### 4.4.8. Funkcja przeszukująca

Centralna część algorytmu przeszukującego - funkcja *search* - przyjmuje jeden parametr: ciąg stemów. Funkcja zwraca tablicę asocjacyjną, zawierającą pary klucz-wartość. W każdej parze, kluczem jest identyfikator wyszukanego zdania, a wartością obiekt *OverlayMatch*, reprezentujący pokrycie tego zdania ze wzorcem. Algorytm funkcji *search* jest przedstawiony na Rysunku 4.7.

---

#### Algorytm: Centralna funkcja wyszukująca

---

```
function search(h)
  for( i := 0 to length(h))
    longestPrefixes :=
      getLongestCommonPrefixes(h.subSequence(i,length(h)))
    for all (Suffix suffix in longestPrefixes)
      prefixLength := longestPrefixes.getPrefixLength()
      currentMatch := matchesMap.get(suffix.id)
      currentMatch.addExampleMatch(suffix.offset,
                                   suffix.offset+prefixLength)
      currentMatch.addPatternMatch(i, i+prefixLength)
end function
```

---

Rysunek 4.7. Funkcja *search*

#### 4.4.9. Obliczanie oceny dopasowania

Autorski algorytm obliczania oceny dopasowania jest częściowo inspirowany odległością największego wspólnego podciągu, opisaną w Podrozdziale 4.2.2. Stosując podobne nazewnictwo, opracowaną na potrzeby niniejszej pracy funkcję dystansu można nazwać odległością wszystkich wspólnych podciągów.

Dla danego zdania wejściowego (*pattern*) i zdania wyszukanego w indeksie (*example*), posiadającego obiekt *OverlayMatch*, ocena dopasowania jest obliczana przy użyciu następującego wzoru:

$$score = \frac{\sum_{i=0}^n patternMatches[i].length + \sum_{i=0}^m exampleMatches[i].length}{length(pattern) + length(example)} \quad (4.1)$$

gdzie:

- $patternMatches[k].length$  jest długością (w sensie liczby stemów)  $k$ -tego przedziału
- $length(pattern)$  jest długością zdania wyszukiwanego
- $length(example)$  jest długością zdania wyszukanego

Należy pamiętać, że zdania te są reprezentowane w indeksie w postaci ciągów kodów, odpowiadających stemom. Ponieważ istnieje bijekcja pomiędzy zbiorami kodów i stemów, bez utraty ogólności można definiować ocenę dopasowania dla stemów.

Dla przykładu użytego do demonstracji obiektu *OverlayMatch* w Podrozdziale 4.4.7, wartość oceny dopasowania byłaby następująca:

$$score = \frac{(2 + 2) + (2 + 2)}{5 + 4} \approx 88.9\%$$

## 4.5. Analiza algorytmu przeszukiwania pamięci tłumaczeń

W niniejszym podrozdziale zostaną przedstawione najważniejsze własności opracowanego przeze mnie algorytmu przeszukiwania pamięci tłumaczeń.

### 4.5.1. Ograniczenie oceny dopasowania

Ocena dopasowania, zwracana przez algorytm, ma odzwierciedlać ludzką intuicję podobieństwa zdań w taki sposób, aby wartość oceny 100% odpowiadała identyczności zdań, a wartość 0% - całkowitemu braku podobieństwa. Jest zatem konieczne, aby dla dowolnych danych wejściowych, ocena dopasowania zwracana przez algorytm, mieściła się w przedziale  $[0, 1]$ .

**Twierdzenie 1** (Ograniczenie oceny dopasowania). *Ocena dopasowania  $s$ , zwracana przez algorytm, spełnia warunek  $s \in [0, 1]$*

**Dowód 1.** Z równania 4.1, ocena dopasowania jest ilorazem liczby nieujemnej przez liczbę dodatnią (*pattern* i *example* jako zdania są ciągami niepustymi), a więc  $s \geq 0$ . Co więcej,

$$\sum_{i=0}^n patternMatches[i].length \leq length(pattern)$$

gdyż z założenia, przedziały *patternMatches* są rozłączne. Analogicznie:

$$\sum_{i=0}^m \text{exampleMatches}[i].\text{length} \leq \text{length}(\text{example}).$$

Stąd  $s \leq 1$ .

#### 4.5.2. Własność przykładu doskonałego

W algorytmie wyszukiwania przybliżonego istotne jest, aby potrafił on wyszukać w indeksie w szczególności zdanie, które jest identyczne z wyszukiwanym. Od funkcji oceny dopasowania wymagamy dodatkowo, aby dla takiego dopasowania zwróciła wartość 1. Poniżej pokażę, że algorytm ma obie te własności.

**Twierdzenie 2** (Własność przykładu doskonałego). *Jeśli indeks algorytmu zawiera przykład  $p$ , którego zdaniem źródłowym jest zdanie  $z_p$ , to wyszukiwanie zdania  $z_p$  w tym indeksie zwróci (między innymi) wynik  $p$  z wartością funkcji oceny dopasowania 1.*

**Dowód 2.** W takiej sytuacji indeks systemu musi zawierać sufix  $s_{z_p}$ , będący całym zdaniem  $z_p$ . Procedura *search* uruchomi procedurę *getLongestCommonPrefixes*. Ta druga zwróci zbiór  $S$  sufixów, posiadających najdłuższy wspólny prefiks ze zdaniem  $z_p$ . Ponieważ żaden sufix nie może mieć ze zdaniem  $z_p$  wspólnego prefiksu dłuższego niż  $\text{length}(z_p)$ , a  $s_{z_p}$  ma z tym zdaniem wspólny prefiks długości dokładnie  $\text{length}(z_p)$ ,  $s_{z_p} \in S$  dowodzi, że doskonały przykład zostanie zwrócony przez algorytm.

Ocena doskonałego przykładu zostanie obliczona na podstawie jego obiektu *OverlayMatches*. Obiekt ten został określony w liniach 6 i 7 procedury *search*. W linii 6 został dodany przedział pokrycia przykładu  $[0, \text{length}(z_p) - 1]$  (gdyż offset sufixu  $s_{z_p}$  jest równy 0). W linii 7 został dodany przedział pokrycia wzorca  $[0, \text{length}(z_p) - 1]$  (gdyż zmienna  $i$  w momencie odnalezienia  $s_{z_p}$  była równa 0. Odpowiadało to temu, że był wtedy wyszukiwany cały wzorec  $z_p$ , a nie jego fragment). Ponieważ przedziały pokrycia wzorca i przykładu muszą być między sobą rozłączne, nie mógł zostać dodany żaden inny przedział. W związku z tym, długości pokrycia przykładu i wzorca są dokładnie równe długościom odpowiednio przykładu i wzorca. Ocena dopasowania, na mocy definicji (równanie 4.1), wynosi więc 1.

#### 4.5.3. Ocena dopasowania jako dystans, metryka, podobieństwo

Zdefiniujmy funkcję dopasowania zdań  $s : Z \times Z \mapsto [0, 1]$ , w taki sposób, że  $s(x, y)$  jest oceną dopasowania zdań  $x$  i  $y$ , zwróconą przez algorytm podczas wyszukiwania zdania  $y$  w indeksie zawierającym zdanie  $x$ .

Zdefiniujmy również funkcję dystansu pomiędzy zdaniem  $d_s : Z \times Z \mapsto [0, 1]$ , gdzie  $Z$  jest zbiorem wszystkich możliwych zdań, jako:

$$d_s(x, y) = 1 - s(x, y) \quad (4.2)$$

**Twierdzenie 3** (Dystans pomiędzy zdaniem). *Funkcja  $d_s$  spełnia własności dystansu, tj.  $\forall x, y \in Z$ :*

1.  $d_s(x, y) \geq 0$  (nieujemność)
2.  $d_s(x, y) = d_s(y, x)$  (symetryczność)
3.  $d_s(x, x) = 0$  (zwrotność)

Definicja dystansu według [DD09].

**Dowód 3.** Własność nieujemności wynika bezpośrednio z Twierdzenia 1 oraz definicji funkcji  $d_s$ . Symetryczność funkcji  $d_s$  wynika z symetryczności obliczania oceny dopasowania:  $s(x, y) = s(y, x)$ . Własność zwrotności wynika natomiast z faktu, iż

$$\forall_{x \in Z} s(x, x) = 1 \quad (4.3)$$

Warunek 4.3 wynika z tego, że jeśli algorytm szuka zdania  $x$  w indeksie zawierającym zdanie  $x$ , to obiekt *OverlayMatches* dla dopasowania  $x$  do  $x$  będzie zdefiniowany następująco:

**patternMatches** : { [0,length(x)-1] }

**exampleMatches** : { [0,length(x)-1] }

Zarówno *patternMatches*, jak i *exampleMatches*, będą zawierały informację o dopasowaniu całego zdania  $x$  do siebie samego. Obliczenie oceny podobieństwa w tym przypadku będzie następujące:

$$s(x, x) = \frac{\text{length}(x) + \text{length}(x)}{\text{length}(x) + \text{length}(x)} = 1$$

Stąd:  $d_s(x, x) = 1 - s(x, x) = 0$ .

Funkcja  $d_s$  nie jest jednak metryką w zbiorze  $Z$  (w rozumieniu definicji podanej w [DD09]).

**Twierdzenie 4** (Niespełnianie własności metryki w  $Z$ ). *Funkcja  $d_s$  nie jest metryką w zbiorze  $Z$ .*

**Dowód 4.** Gdyby  $d_s$  była metryką, spełniałaby warunek identyczności elementów nierozróżnialnych:

$$\forall_{x, y \in Z} (d_s(x, y) = 0 \Leftrightarrow x = y) \quad (4.4)$$

Zauważmy, że podczas działania algorytmu, zarówno zdanie  $x$ , jak i zdanie  $y$  jest najpierw poddane procesowi generowania skrótu, opisanemu w Podrozdziale 4.4.3. Zdefiniujmy funkcję skrótu  $h : Z \mapsto Z$ , przyporządkowującą zdaniu jego skrót, który powstał w wyniku algorytmu generowania skrótu. O funkcji  $h$  nie zakładamy że jest różnowartościowa. Mogą zatem istnieć  $x_1$  i  $x_2$  takie, że  $x_1 \neq x_2$  i  $h(x_1) = h(x_2)$ . Ponieważ algorytm operuje tylko na skrótach zdań,  $d_s(x_1, x_2) = 0$ , co przeczy warunkowi 4.4.

Rozpatrzmy teraz, czy  $d_s$  jest metryką w zbiorze  $H$  - zbiorze wszystkich możliwych skrótów zdań (zbiorze wartości funkcji  $h$ ). W tym przypadku odpowiedź jest również negatywna.

**Twierdzenie 5** (Niespełnianie własności metryki w  $H$ ). *Funkcja  $d_s$  nie jest metryką w zbiorze  $H$ .*

**Dowód 5.** Gdyby  $d_s$  była metryką, spełniałaby warunek 4.4. Rozpatrzmy jednak zdania:

$z_1 : \{a, a\}$

$z_2 : \{a\}$

Naturalnie  $z_1 \neq z_2$ , ale  $s(z_1, z_2) = 1$ , gdyż obiekt *OverlayMatches* dla tych zdań jest postaci:

**patternMatches** : { [0,0] }

**exampleMatches** : { [0,1] }

(zdanie  $z_1$  występuje w roli *example*, a  $z_2$  w roli *pattern*), a wyliczenie oceny dopasowania jest następujące:

$$s(z_1, z_2) = \frac{1 + 2}{1 + 2} = 1$$

Zauważmy, że ponieważ  $H \subset Z$ , to dowód Twierdzenia 5 jest alternatywnym dowodem Twierdzenia 4.

Rozpatrzmy jeszcze, czy funkcja  $s$  jest funkcją podobieństwa w sensie definicji podanej w [DD09]. Funkcja  $s$  jest podobieństwem, o ile:

1.  $\forall x, y \in Z \ s(x, y) \geq 0$
2.  $\forall x, y \in Z \ s(x, y) = s(y, x)$
3.  $\forall x, y \in Z \ s(x, y) \leq s(x, x)$
4.  $s(x, y) = 1 \Leftrightarrow x = y$

**Twierdzenie 6** (Niespełnianie własności podobieństwa w  $Z$ ). *Funkcja  $s$  nie jest podobieństwem w zbiorze  $Z$ .*

**Dowód 6.** Funkcja  $s$  spełnia warunki 1, 2, 3, jak zostało wykazane w dowodach poprzednich twierdzeń. Nie spełnia jednak warunku 4, co wynika z dowodu Twierdzenia 4.

#### 4.5.4. Wnioski

Funkcja podobieństwa  $s$  nie spełnia własności metryki (Twierdzenia 4 i 5) oraz podobieństwa (Twierdzenie 6). Niespełnianie to wynika z braku zachowania przez funkcję  $s$  warunku identyczności elementów nierozróżnialnych. Zwróćmy jednak uwagę, że algorytm oparty o funkcję  $s$  zachowuje wymagane własności ograniczenia oceny podobieństwa (Twierdzenie 1) oraz przykładu doskonałego (Twierdzenie 2). Ma to związek z faktem, iż funkcja  $s$  jest dystansem (Twierdzenie 3). Prawdziwe jest następujące twierdzenie:

**Twierdzenie 7** (Wpływ własności dystansu). *Niech będzie dany zbiór zdań  $S$  i funkcja  $d : Z \times Z \mapsto \mathbb{R}$ . Skonstruujmy algorytm  $A$  wyszukiwania zdań w zbiorze  $S$ , który dla zdania wejściowego  $x$  zwraca wszystkie elementy  $y \in S$ , każdemu przypisując ocenę dopasowania  $score$ , zdefiniowaną następująco:*

$$score = \begin{cases} 1 - d(x, y) & \text{gdy } d(x, y) < 1 \\ 0 & \text{gdy } d(x, y) \geq 1 \end{cases}$$

*Algorytm  $A$  spełnia własności ograniczenia oceny dopasowania oraz przykładu doskonałego wtedy i tylko wtedy, gdy funkcja  $d$  spełnia warunki 1 i 3 dystansu, tj.  $\forall x, y \in Z$ :*

- $d(x, y) \geq 0$  (nieujemność)
- $d(x, x) = 0$  (zwrotność)

**Dowód 7.** Załóżmy najpierw, że funkcja  $d$  spełnia powyższe warunki. Wtedy  $\forall x, y \in Z$   $d(x, y) \geq 0$ . Ocena dopasowania  $score$  jest wtedy ograniczona z góry przez 1. Z definicji, ocena dopasowania jest ograniczona z góry przez 0. Algorytm  $A$  posiada więc cechę ograniczenia oceny dopasowania. Zauważmy również, że jeśli  $x \in S$ , to zostanie on zwrócony z oceną dopasowania  $1 - d(x, x)$ . Ponieważ funkcja  $d$  spełnia warunek zwrotności, ocena dopasowania przykładu doskonałego będzie równa 1. Dowodzi to spełniania przez algorytm  $A$  własności przykładu doskonałego.

Z drugiej strony, załóżmy że algorytm  $A$  spełnia własności ograniczenia oceny dopasowania oraz przykładu doskonałego. Z faktu, że  $score \leq 1$ , wynika  $\forall x, y \in Z$   $1 - d(x, y) \leq 1$ , a więc  $\forall x, y \in Z$   $d(x, y) \geq 0$ . Funkcja  $d$  spełnia warunek nieujemności. Ponadto, algorytm  $A$  zwraca zdanie  $x$  z oceną dopasowania 1, o ile  $x \in S$ . Oznacza to, że  $\forall x \in Z$   $1 - d(x, x) = 1$ , a więc



$\forall_{x \in Z} d(x, x) = 0$ , co dowodzi spełniania przez funkcję  $d$  warunku zwrotności i kończy dowód twierdzenia.

Oznacza to, że konstruując algorytm przybliżonego wyszukiwania zdań należy wybrać funkcję podobieństwa, która spełnia warunki dystansu. Decyduje to o spełnianiu przez algorytm istotnych i pożądanых własności. Nie jest natomiast w tym kontekście istotne, czy funkcja ta spełnia matematyczne warunki podobieństwa lub metryki.


## 4.6. Złożoność obliczeniowa algorytmu

### 4.6.1. Złożoność czasowa algorytmu

Ponieważ autorski algorytm wyszukiwania przybliżonego został zaprojektowany w celu przetwarzania dużego zbioru danych, jedną z najistotniejszych jego własności jest złożoność obliczeniowa. Poniżej zostanie przedstawiona złożoność czasowa i pamięciowa algorytmu wyszukiwania przybliżonego.

**Twierdzenie 8** (Złożoność czasowa algorytmu). *Niech  $n$  oznacza liczbę zdań w pamięci tłumaczeń,  $d$  oznacza średnią długość tych zdań (w sensie liczby słów), a  $m$  - długość zdania wyszukiwanego. Złożoność czasowa algorytmu wyszukującego jest rzędu:*

- $O(m \cdot \log(nd))$  w optymistycznym przypadku,
- $O(m^2 \log(nd))$  w pesymistycznym i średnim przypadku.

**Dowód 8.** W indeksie jest  $n$  zdań o średniej długości  $d$ . W tablicy sufiksów znajduje się więc  $n \cdot d$  sufiksów. Procedura przeszukująca  $m$  razy wywołuje procedurę *getLongestCommonPrefixes*, która przeszukuje zbiór sufiksów. Przeszukiwanie to odbywa się w  $m$  krokach. W  $i$ -tym kroku, zbiór przeszukiwanych sufiksów jest zawężany do tych, które mają ze wzorcem wspólny prefix długości  $i - 1$ . W pesymistycznym  przypadku (jeśli w indeksie są wyłącznie doskonale przykłady), procedura *getLongestCommonPrefixes* ma złożoność  $O(m \cdot \log(nd))$ , gdyż  $m$  razy zostanie wywołana procedura wyszukiwana binarnego, która ma złożoność logarytmiczną (za [CSRL01]). W optymistycznym przypadku (brak jakichkolwiek dopasowań wzorca w pamięci tłumaczeń), złożoność tej procedury wynosi  $O(\log(nd))$ . W średnim przypadku, zdanie wzorcowe ma wspólny prefix z jakimś sufiksem w indeksie długości około  $\frac{m}{2}$ , tj. rzędu  $m$ . Oznacza to, że średnia złożoność procedury *getLongestCommonPrefixes* wynosi  $O(m \cdot \log(nd))$ . Ponieważ procedura ta

wywoływana jest  $m$  razy, optymistyczna złożoność czasowa algorytmu wyszukiwania wynosi  $O(m \cdot \log(nd))$ , natomiast złożoność pesymistyczna i średnia wynoszą  $O(m^2 \log(nd))$ .

#### 4.6.2. Złożoność pamięciowa algorytmu


Najistotniejszą własnością algorytmu jest jego niewielka złożoność pamięciowa. Cecha ta ma duże znaczenie zarówno teoretyczne, jak i praktyczne (gdyż pozwala na przechowywanie indeksu w pamięci operacyjnej komputera i uzyskanie dużej szybkości wyszukiwania).

**Twierdzenie 9** (Złożoność pamięciowa algorytmu). *Niech  $n$  oznacza liczbę zdań w pamięci tłumaczeń,  $d$  oznacza średnią długość tych zdań. Złożoność pamięciowa algorytmu w optymistycznym, pesymistycznym i średnim przypadku, niezależnie od długości wyszukiwanego zdania, jest rzędu:  $O(nd)$ .*

**Dowód 9.** Jedyną strukturą danych o znacznych rozmiarach, którą wykorzystuje algorytm, jest tablica sufiksów. W tablicy sufiksów znajduje się  $nd$  sufiksów. Doliczyć należy również słownik tokenów, którego rozmiar jest w najgorszym rzędu  $nd$ , gdyż tyle jest ogółem tokenów, a w najgorszej sytuacji wszystkie mogą być różne. Stąd złożoność pamięciowa algorytmu jest rzędu  $O(nd)$ .

### 4.7. Porównanie z konkurencyjnymi algorytmami

Złożoność czasowa autorskiego algorytmu wyszukiwania wynosi  $O(m \cdot \log(nd))$  w optymistycznym oraz  $O(m^2 \log(nd))$  w pesymistycznym i średnim przypadku. Jego złożoność pamięciowa wynosi  $O(nd)$ . Ponadto, cechą wyróżniającą autorski algorytm na tle innych algorytmów tej klasy jest wykorzystywanie własnej oceny dopasowania zdań, która nie jest oparta na odległości Levenshteina.

W kolejnych podrozdziałach  przedstawię porównanie teoretycznych cech autorskiego algorytmu z najnowszymi osiągnięciami w dziedzinie. Wyniki praktycznych testów algorytmu są przedstawione w Rozdziale 6.

#### 4.7.1. Algorytm Koehna i Senellarta

##### Ogólna charakterystyka i złożoność obliczeniowa

Philipp Koehn i Jean Senellart deklarują złożoność czasową swojego algorytmu przeszukiwania pamięci tłumaczeń na poziomie  $O(m^3 \log(nd))$  w pe-

symistycznym oraz  $O(m \cdot \log(nd))$  w optymistycznym i średnim przypadku ([KS10]). Ich algorytm, podobnie jak przedstawiony w niniejszej pracy, jest oparty na tablicy sufiksów. Stosuje on mechanizmy filtrowania potencjalnych wyników wyszukiwania, pochodzących z tablicy. Algorytm ten, w przeciwieństwie do opisanego w niniejszej pracy, nie zwraca oceny dopasowania dla znalezionych zdań.


Złożoność pamięciowa algorytmu Koehna i Senellarta jest rzędu  $O(nd)$ . Autorzy algorytmu nie dążyli jednak do maksymalnego zmniejszenia rozmiaru tablicy sufiksów (np. dzięki wykorzystaniu stemów). Planowali przechowywanie indeksu do wyszukiwania na dysku twardym komputera.

## Opis algorytmu

Algorytm wyszukiwania przybliżonego autorstwa Koehna i Senellarta podzielony jest na następujące fazy:

1. Wyszukiwanie w tablicy sufiksów
2. Filtrowanie dopasowań
3. Filtrowanie na podstawie długości
4. Weryfikacja dopasowań

W fazie pierwszej, w tablicy sufiksów wyszukiwane są wszystkie możliwe dopasowania do zdania wzorcowego. W wyniku tej operacji powstaje zbiór wszystkich par  $(pattern\_ngram, example\_ngram)$ , gdzie  $pattern\_ngram$  jest podciągiem słów zdania wzorcowego, a  $example\_ngram$  jest podciągiem słów zdania z przykładu. Pary te nazywane będą **dopasowaniami**. Znalezione dopasowania zostają pogrupowane ze względu na przykłady z pamięci tłumaczeń, do których się odnoszą.


Druga faza polega na wstępnym odrzuceniu niektórych dopasowań. W ramach jednego przykładu w pamięci tłumaczeń, odrzucane są między innymi te dopasowania  $(p_1, e_1)$ , dla których istnieje dopasowanie  $(p_2, e_2)$ , że  $p_1$  jest podciągiem  $p_2$  lub  $e_1$  jest podciągiem  $e_2$ . Identyczny efekt uzyskuje autorski algorytm, ale nie wymaga on dodatkowego filtrowania. W rozwiązaniu autorskim zwracane są tylko najdłuższe możliwe dopasowania z tablicy sufiksów. Dzieje się tak ze względu na sposób działania funkcji *getLongestCommonPrefixes*, we współpracy z obiektem *OverlayMatches* (patrz Podrozdział 4.4.6).  Ilustruje to następujący przykład. Załóżmy, że w pamięci tłumaczeń znajduje się przykład o zdaniu źródłowym: “Wynik badań potwierdził początkowe przypuszczenia.” Załóżmy również, że wyszukiwanym zdaniem jest: “Wynik badań potwierdził hipotezę profesora N.” W tej sytuacji, algorytm Koehna i Senellarta odnajdzie w pamięci tłumaczeń dopasowania:

- wynik
- wynik, badań
- wynik, badań, potwierdził

Następnie, algorytm ten przeprowadzi analizę dopasowań i pozostawi trzecie, najdłuższe dopasowanie. Natomiast algorytm autorski, podczas przeszukiwania tablicy sufiksów, natrafi na sufiksy:

- wynik, badań, potwierdził, początkowe, przypuszczenia
- badań, potwierdził, początkowe, przypuszczenia
- potwierdził, początkowe, przypuszczenia

Na podstawie pierwszego ze znalezionych sufiksów otrzymane zostanie dopasowanie “wynik, badań, potwierdził”. Zostanie ono wpisane do obiektu *OverlayMatches*. Ponieważ dopasowania uzyskane z kolejnych sufiksów (“badań, potwierdził” oraz “potwierdził”) musiałyby się znaleźć w miejscu zajmowanym przez pierwsze dopasowanie w obiekcie *OverlayMatches*, zostają natychmiast odrzucone.

Trzecia faza algorytmu Koehna i Senellarta polega na odrzuceniu tych dopasowań, które nie spełniają kryterium długości. Kryterium to polega na sprawdzeniu, czy dysproporcja długości pomiędzy zdaniem wzorcowym, a znalezionym przykładem, nie jest większa od ustalonej z góry wartości. Dysproporcja długości zdań jest brana pod uwagę również w autorskim algorytmie, jednak nie jest ona nigdy powodem odrzucenia dopasowania, a jedynie wpływa negatywnie na jego ocenę. Odrzucanie na przykład dopasowań dłuższych od wzorca powoduje utratę potencjalnie cennych wyników wyszukiwania.  zważmy następującą sytuację. Niech w pamięci tłumaczeń znajduje się zdanie “Wynik badań potwierdził początkowe przypuszczenia, dotyczące ciężaru cząsteczkowego niektórych związków siarki, azotu, tlenu i manganu”. Niech zdaniem wyszukiwanym będzie “Wynik badań potwierdził początkowe przypuszczenia”. Podczas wyszukiwania, autorski algorytm zwróci przykład z pamięci tłumaczeń z relatywnie wysoką oceną (na podstawie wzoru 4.1):

$$\frac{5 + 5}{5 + 15} = 50\%$$

Bez wątpienia odnaleziony przykład może posłużyć tłumaczowi jako odpowiedź przy tłumaczeniu wyszukiwanego zdania. Wystarczy, że w uzyskanej odpowiedzi pominie on drugą część, a zostanie tylko tłumaczenie, które go interesuje. Stąd zasadne jest, aby podobne przykłady nie tylko nie były odrzucane podczas wyszukiwania, ale także otrzymywały relatywnie wysokie oceny.


Ostatnią fazą algorytmu Koehna i Senellarta jest tzw. pełna walidacja znalezionych dopasowań, która dąży do obliczenia odległości Levensh-

teina pomiędzy zdaniem wyszukiwanym, a odnalezionym przykładem. Faza ta sprowadza się do łączenia mniejszych dopasowań w większe i znalezienia optymalnego dopasowania zdania wyszukanego do wzorca. Technika łączenia dopasowań opiera się na idei algorytmu A\*. Podobny krok nie jest potrzebny w autorskim algorytmie, gdyż tam za łączenie dopasowań odpowiada obiekt *OverlayMatches*.

Wśród zalet algorytmu Koehna i Senellarta wymienić można dużą szybkość działania i dostosowanie do przeszukiwania nawet znacznej wielkości pamięci tłumaczeń. Wadą jest natomiast precyzja wyszukiwania. Algorytm Koehna i Senellarta odrzuca potencjalnie dobre wyniki wyszukiwania podczas fazy filtrowania.

Po fazie filtrowania, podczas weryfikacji algorytm ten używa do określenia podobieństwa zdań odległości Levenshteina na poziomie słów, która nie jest dobrze dostosowana do tego problemu. Na przykład, odległość Levenshteina pomiędzy zdaniami:

— nie widzę tu jabłek

— nie widzę tu jabłek nie widzę tu grusze 

wynosi 4. Jeśli odnieść ją do sumy długości zdań (12), można wywnioskować, że zdania te są w  $\frac{1}{3}$  niepodobne, czyli ich stopień podobieństwa wynosi około 66.7%. Natomiast w autorskim algorytmie, podobieństwo tych zdań wyniosłoby (na podstawie wzoru 4.1):

$$\frac{4 + 7}{4 + 8} \approx 91.7\%$$

Ponadto, z powodu braku wykorzystania narzędzi przetwarzania języka naturalnego (takich jak np. funkcja stemowania), algorytm ten nie uzna za podobne niektórych par zdań podobnych w sensie ludzkim. Na przykład zdania:

— Mam szybki, czerwony samochód

— Nie mam szybkiego, czerwonego samochodu

zostaną uznane przez algorytm Koehna i Senellarta za bardzo mało podobne. Wykorzystanie funkcji stemowania w autorskim algorytmie uodparnia go na tego typu problemy, a co za tym idzie, czyni go szczególnie przydatnym do przeszukiwania pamięci tłumaczeń, w których językiem źródłowym jest język polski.

## Pseudokod algorytmu

Na Rysunkach 4.8 i 4.9 przedstawiony jest pełny pseudokod algorytmu Koehna i Senellarta. Jest on zaczerpnięty z artykułu [KS10].

---

**Algorytm Koehna i Senellarta: main, find-matches, find-in-suffix-array**

---

```

procedure main(Pattern  $p = p_1..p_n$ )
  ceiling-cost =  $\lceil 0.3 \cdot p.length \rceil$ 
  M := find-matches(p)
  S := find-segments(p,M)
  return S

procedure find-matches(Pattern  $p = p_1..p_n$ )
  for (start := 1 to p.length)
    for (end := start to p.length)
      remain := p.length - end
       $M_{start,end}$  := find-in-suffix-array( $p_{start}, p_{end}$ )
      break if  $M_{start,end} == \emptyset$ 
      for all (m in M)
        m.leftmin := |m.start - start|
        if (m.leftmin == 0 and start > 0)
          m.leftmin := 1
        m.rightmin := |m.remain - remain|
        if (m.rightmin == 0 and remain > 0)
          m.rightmin := 1
        min-cost := m.leftmin+m.rightmin
        break if min-cost > ceiling-cost
        m.leftmax := max(m.start, start)
        m.rightmax := max(m.remain, remain)
        m.pstart := start
        m.pend := end
         $M := M \cup \{m\}$ 
      return M

procedure find-in-suffix-array(String string)
  first-match := find first occurrence of string in array
  last-match := find last occurrence of string in array
  for (Match i:=first_match to last_match)
    m := new match()
    m.start := i.segment - start
    m.end := i.segment - end
    m.length := i.segment.length
    m.remain := m.length - m.end
    m.segment-id := i.segment.id
     $N := N \cup \{m\}$ 
  return N

```

---

Rysunek 4.8. Algorytm Koehna i Senellarta - część pierwsza**4.7.2. Algorytm Huerty**

Jak zostało nakreślone w Rozdziale 4.2, autorski algorytm przeszukiwania pamięci tłumaczeń bazuje na rozwiązaniach problemu wyszukiwania przybli-

zonego. Zasadne jest więc porównywanie go z najnowszymi osiągnięciami w tej dziedzinie.

Jednym z najnowszych algorytmów wyszukiwania przybliżonego został opublikowany w 2010 roku przez Juana Manuela Huertę ([Hue10]). Wykorzystuje on strukturę stosu, która służy kodowaniu indeksu na potrzeby przybliżonego wyszukiwania łańcuchów znaków. Istotną różnicą w stosunku do autorskiego algorytmu jest operowanie na ciągach znaków, a nie na ciągach słów. Ponadto, algorytm Huerty tylko aproksymuje odległość edycyjną, przez co zwrócone przez niego wyniki wyszukiwania mogą nie być właściwe (autor algorytmu deklaruje margines błędu rzędu 2,5%). Dzięki tym zabiegom udało się mu osiągnąć złożoność czasową algorytmu wyszukiwania rzędu  $O(m \cdot s \cdot \log(s))$ , gdzie  $s$  jest rozmiarem stosu kodującego. Zwykle rozmiar ten jest znacznie mniejszy, niż  $n$ . Warto jednak zwrócić uwagę, że złożoność czasowa autorskiego algorytmu również jest liniowa lub kwadratowa względem długości obiektu wyszukiwanego i logarytmiczna względem rozmiaru zbioru do przeszukania.

### 4.7.3. Algorytm Ghodsi'ego

Mohammadreza Ghodsi opublikował w 2006 roku własny algorytm wyszukiwania przybliżonego ([Gho06]). Algorytm ten służy do wyszukiwania podobnych łańcuchów znaków i podobnie jak wiele algorytmów tej klasy znalazł zastosowanie w badaniu sekwencji DNA.

Algorytm Ghodsi'ego bazuje na odległości Hamminga ([Ham50]) i zakłada wyszukiwanie łańcuchów znaków odległych od wzorca o  $k$  w sensie tej odległości. Algorytm pracuje z tekstami zbudowanymi na alfabecie o rozmiarze  $\sigma$ . Deklarowana przez autora algorytmu złożoność czasowa operacji wyszukiwania jest rzędu  $O(m^{k+3}\sigma^k \cdot \log_2 n)$ . Można tu więc dostrzec wielomianową zależność od długości słowa wyszukiwanego oraz złożoność logarytmiczną względem wielkości zbioru przeszukiwanego.

**Algorytm Koehna i Senellarta: find-segments, parse-validate, combinable**

```

procedure find-segments(Pattern p, Matches M)
  for all (Segment s:  $\exists m \in M: m.\text{segment-id} == s.\text{id}$ )
    a := new agenda-item()
    a.M :=  $\{m \in M: m.\text{segment-id} == s.\text{id}\}$ 
    a.sumlength :=  $\sum m \in a.M m.\text{length}$ 
    a.priority := -a.sumlength; a.s := s
    A := A  $\cup$  {a}
  while (a := pop(A))
    break if a.s.length - p.length > ceiling-cost
    break if max(a.s.length, p.length) - a.sumlength > ceiling-cost
    cost := parse-validate(a.s, a.M)
    if (cost < ceiling-cost)
      ceiling-cost := cost
      S :=  $\emptyset$ 
      S := S  $\cup$  {a, s} if cost == ceiling-cost
  return S

procedure parse-validate(String string, Matches M)
  for all ( $m_1 \in M, m_2 \in M$ )
    A := A  $\cup$  {a} if a == combinable( $m_1, m_2$ )
  while (a := pop(A))
    break if a.mincost > ceiling-cost
    mm := new match()
    mm.leftmin := a.m1.leftmin; mm.leftmax := a.m1.leftmax
    mm.rightmin := a.m2.rightmin; mm.rightmax := a.m2.rightmax
    mm.start := a.m1.start; mm.end := a.m2.end
    mm.pstart := a.m1.pstart; mm.pend := a.m2.pend
    mm.internal := a.m1.internal + a.m2.internal + a.internal
    cost := min(cost, mm.leftmax + mm.rightmax + mm.internal)
    for all (m in M)
      A := A  $\cup$  {a} if a == combinable(mm, m)
  return cost

procedure combinable(Matches  $m_1, m_2$ )
  return null unless  $m_1.\text{end} < m_2.\text{start}$ 
  return null unless  $m_1.\text{pend} < m_2.\text{pstart}$ 
  a.m1 := m1; a.m2 := m2
  delete :=  $m_2.\text{start} - m_1.\text{end} - 1$ 
  insert :=  $m_2.\text{pstart} - m_1.\text{pend} - 1$ 
  internal := max(insert, delete)
  a.internal := internal
  a.mincost :=  $m_1.\text{leftmin} + m_2.\text{rightmin} + \text{internal}$ 
  a.priority := a.mincost
  return a

```






## Rozdział 5

# Przetwarzanie pamięci tłumaczeń

### 5.1. Motywacja i cel badań

Technika wspomaganie pamięci tłumaczeń, opisana w Rozdziale 3, silnie  opiera się na jakości wykorzystywanej pamięci tłumaczeń. Pamięć tłumaczeń dobrej jakości zawiera przykłady użyteczne z punktu widzenia tłumacza. W związku z tym, tłumacze dążą do zbudowania na swój użytek jak największej i przede wszystkim wysokiej jakości pamięci tłumaczeń.


Jakość pamięci tłumaczeń może być mierzona dwoma kryteriami. Po pierwsze, przykłady mają być odnajdywane w pamięci możliwie często. Częstość tego odnajdywania bezpośrednio przekłada się na zmniejszenie czasu tłumaczenia dokumentu przez tłumacza. Do mierzenia tej częstości służy zaproponowany przeze mnie w Definicji 18 parametr kompletności. Pojęcie to pochodzi z dziedziny pozyskiwania informacji, w języku angielskim znane jest pod nazwą *recall*. Jest ono opisane m.in. w [MKS99]. W niniejszej pracy podaję własną definicję kompletności, dostosowaną do zagania przeszukiwania pamięci tłumaczeń, inspirowaną opracowaniem [WS99].

**Definicja 18** (Kompletność). *Kompletnością* wyszukiwania zdań przez algorytm  $A$  przeszukiwania pamięci tłumaczeń dla danego zbioru zdań  $Z$  i danej pamięci tłumaczeń  $T$  nazywamy stosunek liczby zdań w zbiorze  $Z$ , dla których algorytm  $A$  zwróci niepusty zbiór wyników wyszukiwania w pamięci  $T$  do całkowitej liczby zdań w zbiorze  $Z$ .

Drugim kryterium jakości pamięci tłumaczeń jest użyteczność przykładów znalezionych w pamięci tłumaczeń. Użyteczność przykładu jest rozumiana jako jego przydatność przy tłumaczeniu zdania. Do jej mierzenia służy pojęcie dokładności (ang. *precision*). Na podstawie [WS99] definiuję to pojęcie następująco:

**Definicja 19** (Dokładność). Niech będzie dany algorytm  $A$  przeszukiwania pamięci tłumaczeń, zbiór zdań wejściowych  $Z$  i pamięć tłumaczeń  $T$ . Niech zbiór przykładów  $S$  będzie dany następująco:  $S = \{p : p = A(z), \text{ dla } z \in Z\}$ , gdzie  $A(z)$  jest przykładem zwróconym przez algorytm  $A$  dla zdania  $z$ , który otrzymał najwyższą ocenę. **Dokładnością** wyszukiwania zdań przez algorytm  $A$ , wyposażony w pamięć tłumaczeń  $T$ , dla zbioru zdań  $Z$ , nazywamy stosunek liczby przykładów w zbiorze  $S$ , które są uznane za przydatne przez tłumacza, do całkowitej liczby przykładów w zbiorze  $S$ .

Zauważmy, że definicja dokładności ma charakter wyłącznie praktyczny, gdyż jest ściśle uzależniona od subiektywnego osądu tłumacza.

Kompletność i dokładność, według podanych definicji, zależą od wybranego algorytmu wyszukiwania oraz od wykorzystywanej pamięci tłumaczeń. W Rozdziale 4 zaproponowałem skuteczny algorytm wyszukiwania. Jego skuteczność tj. jego kompletność i dokładność, są przedstawione w Rozdziale 6 . W niniejszym rozdziale skupię się na zagadnieniu przygotowania dobrej pamięci tłumaczeń.

Obecnie środowisko tłumaczy zdaje sobie sprawę z potencjału techniki komputerowego wspomaganie tłumaczenia i docenia wartość dobrej pamięci tłumaczeń (patrz [CGSSO04]). Zostało udowodnione, że użyteczna pamięć może w znaczący sposób zmniejszyć nakład pracy na tłumaczenie (patrz [Val05]).

## 5.2. Tradycyjna metoda tworzenia pamięci tłumaczeń

Najczęściej pamięć tłumaczeń jest kolekcjonowana przez tłumaczy w trakcie procesu tłumaczenia. Proces ten jest wspierany przez narzędzia klasy CAT, opisane w Rozdziale 3. Jeden tłumacz jest w stanie samodzielnie zebrać pamięć tłumaczeń liczącą kilkadziesiąt lub kilkaset tysięcy przykładów. Tłumacze pracujący w większych organizacjach, na przykład biurach tłumaczeń, często budują wspólną pamięć na użytek wszystkich zrzeszonych w danej organizacji. Im większa pamięć, tym większe prawdopodobieństwo, że będzie ona pomocna.

W związku z powyższym, tłumacze nie poprzestają na rozszerzaniu pamięci tłumaczeń dzięki bieżącym zleceniom, ale sięgają po dokumenty przetłumaczone przed erą programów CAT. Dokumenty te najczęściej są przechowywane w formie elektronicznej (format .doc) i są połączone w pary: dokument źródłowy - dokument docelowy. Aby stworzyć z nich pamięć tłumaczeń, należy najpierw podzielić tekst tych dokumentów na zdania. Opera-

cja ta nazywa się segmentacją. Następnie należy dopasować zdania źródłowe do docelowych w taki sposób, aby dopasowane zdania były swoimi tłumaczeniami. Proces ten nazywa się urównoleganiem.

Zarówno segmentacja, jak i urównoleganie są operacjami, które mogą zostać wykonane automatycznie. Ze względu na to, że mają one duże znaczenie w zagadnieniu przygotowywania pamięci tłumaczeń, opiszę je dokładniej.

### 5.2.1. Segmentacja tekstu

Segmentacja, czyli podział tekstu na zdania, nie jest, wbrew pozorom, operacją trywialną. Naiwny algorytm dzieli tekst w miejscach kropek, wykrzykników i znaków zapytania. Rozważmy jednak następujące zdanie:

*“Sz. Pan mgr inż. Jan Kowalski, zamieszkały przy ul. Boh. Westerplatte (??), zgłosi się do W.K.U. w Poznaniu, celem m.in. przedstawienia zaświadczenia o odbyciu P.W. w m.st. Warszawie.”*

W oparciu o naiwny algorytm segmentacji, zostanie dokonany podział:

1. *Sz.*
2. *Pan mgr inż.*
3. *Jan Kowalski, zamieszkały przy ul.*
4. *Boh.*
5. *Westerplatte (?)*
6. *?*
7. *), zgłosi się do W.*
8. *K.*
9. *U.*
10. *w Poznaniu, celem m.*
11. *in.*
12. *przedstawienia zaświadczenia o odbyciu P.*
13. *W.*
14. *w m.*
15. *st.*
16. *Warszawie.*

Algorytm naiwny błędnie podzielił jedno zdanie na aż 16 segmentów. W praktyce, do segmentacji stosuje się algorytm oparty o zbiór reguł segmentacji (taki jak opisany w pracy [Lip07]). Reguły te są najczęściej zapisane w formacie SRX (opisanym w [LIS04]). Format ten dopuszcza dwa rodzaje reguł: reguły podziału, oraz reguły zabraniające podziału (wyjątki). Dla każdej reguły określa się, jaki tekst ma wystąpić przed miejscem podziału (lub

braku podziału), a jaki po tym miejscu. Na Rysunku 5.1 przedstawiona jest definicja reguły, która ustawia punkt podziału po znaku interpunkcyjnym, o ile zaraz za tym znakiem znajduje się biały znak.

---

#### Reguła podziału SRX

---

```
<rule break="yes">
  <beforebreak>[\.\?!]</beforebreak>
  <afterbreak>\s</afterbreak>
</rule>
```

---

Rysunek 5.1. Prosta reguła podziału SRX

Napis *break=yes* oznacza, że jest to reguła podziału. Element *<beforebreak>* zawiera wyrażenie regularne, które musi być dopasowane do tekstu bezpośrednio przed miejscem podziału. Analogicznie, element *<afterbreak>* zawiera wyrażenie regularne, dopasowywane do tekstu za miejscem podziału.

Na Rysunku 5.2 jest zaprezentowana reguła, która zabrania dzielić zdania po skrócie 'ul.', jeśli zaraz za nim występuje biały znak.

---

#### Wyjątek SRX

---

```
<rule break="no">
  <beforebreak>ul\.</beforebreak>
  <afterbreak>\s</afterbreak>
</rule>
```

---

Rysunek 5.2. Prosty wyjątek SRX

Rysunek 5.3 przedstawia bardziej skomplikowaną regułę SRX, obsługującą niektóre typy wypunktowań. Według tej reguły, kolejne punktory mają stanowić odrębne segmenty.

---

#### Reguła podziału SRX

---

```
<rule break="yes">
  <beforebreak>[:;\n]</beforebreak>
  <afterbreak>\s+([0-9]+|\w|[iIvVxXmMlLdD]+)\.?\)\s</afterbreak>
</rule>
```

---

Rysunek 5.3. Reguła SRX, obsługująca wypunktowanie

Dla zdefiniowanego zbioru reguł, do podziału na zdania tekstu stosuje się algorytm podziału, na przykład taki, jak opisany w pracy [Lip07]. Algorytm ten przyjmuje na wejściu zbiór  $R$  reguł SRX oraz tekst  $T$ . Zwraca zbiór pozycji tekstu, w których ma nastąpić podział zdania. Pseudokod tego algorytmu jest przedstawiony na Rysunku 5.4.

---

**Algorytm: Podział tekstu na zdania**

---

```
procedure split(R,T)
  splitPositions := empty set

  for (textIndex := 0 to length(T)-1)

    activeRule := null
    ruleIndex := 0

    while (activeRule == null and ruleIndex < size(R) - 1)
      currentRule := R[ruleIndex]
      if (currentRule.beforeBreak.matches(T[0..textIndex])
          and currentRule.afterBreak.
              matches(T[textIndex+1..length(T)-1]))

        activeRule := currentRule
      else
        ruleIndex := ruleIndex + 1

    if (activeRule != null and activeRule.break = 'yes')
      splitPositions.add(textIndex)

  return splitPositions
end procedure
```

---

Rysunek 5.4. Algorytm podziału tekstu na zdania

Działanie algorytmu dzielenia na zdania jest następujące. Dla każdej pozycji w tekście dopasowywane są po kolei reguły SRX. Pierwsza dopasowana reguła decyduje o tym, czy w danym miejscu powinien nastąpić podział zdania. Jeśli regułą tą jest reguła podziału, następuje podział zdania w rozpatrywanym miejscu w tekście i algorytm przechodzi do analizy kolejnej pozycji w tekście. Jeśli natomiast znaleziona reguła jest wyjątkiem, w miejscu tym nie następuje podział zdania i algorytm przechodzi do analizy kolejnej pozycji w tekście. W przypadku nieznaledzenia pasującej reguły, nie następuje podział zdania w tym miejscu, a algorytm przechodzi do następnej pozycji.

W związku z powyższym, ważna jest kolejność reguł. Na przykład, aby zapewnić, że tekst będzie dzielony na zdania po kropce, ale nie po skrócie 'prof.', należy najpierw podać wyjątek od reguły podziału, a dopiero potem właściwą regułę. Zestaw takich dwóch reguł jest przedstawiony na Rysunku 5.5.

Powodzenie segmentacji zależy więc od przygotowania właściwych reguł podziału oraz wyjątków od nich. W praktyce, warto stosować różne zestawy reguł dla tekstów w różnych językach. Pozwala to uniknąć problemów z wzajemnym wykluczaniem się reguł podziału. Co więcej, w niektórych przypadkach uzasadnione jest nawet stosowanie kilku zestawów reguł w obrębie jednego języka. Właściwy zestaw wybiera się na podstawie klasy tekstu, który

**Zestaw reguł SRX**


---

```

<rule break="no">
  <beforebreak>prof.</beforebreak>
  <afterbreak>\s</afterbreak>
</rule>

<rule break="yes">
  <beforebreak>\.</beforebreak>
  <afterbreak>\s</afterbreak>
</rule>

```

---

Rysunek 5.5. Zestaw reguły SRX i wyjątku od niej

chcemy podzielić na zdania. Jest to uzasadnione na przykład dla tekstów prawniczych i medycznych. Skróty stosowane w tekstach prawniczych mogą być zupełnie różne od tych stosowanych w tekstach medycznych. Na przykład, jeśli ze względu na teksty prawnicze, określimy globalną regułę zabraniającą dzielenia zdania po skrócie *ust.* (od *ustęp*), w kontekście medycznym nastąpi błąd podziału dla tekstu: *“Należy dbać o higienę ust. Pomaga w tym szczotkowanie zębów.”*

**5.2.2. Urównoleglanie**

Po podziale dokumentów na zdania, kolejnym krokiem jest ich urównoleglanie. Najogólniej, operacja ta polega na przyporządkowaniu do siebie tych zdań z dwóch tekstów w dwóch różnych językach, które są swoimi tłumaczeniami.

Operację dopasowywania można bardziej formalnie zdefiniować następująco.

**Definicja 20** (Urównoleglanie, dopasowania). *Niech  $S$  będzie tekstem źródłowym, a  $T$  - docelowym. Niech  $s = (s_1, s_2, \dots, s_n)$  będzie ciągiem zdań tekstu źródłowego, a  $t = (t_1, t_2, \dots, t_n)$  ciągiem zdań tekstu docelowego. Urównoleglaniem tekstu  $S$  do tekstu  $T$  nazywamy ciąg  $d$  par  $(c_s, c_t)$ , takich że:*

- $c_s$  i  $c_t$  są podciągami odpowiednio ciągów  $s$  i  $t$
- przynajmniej jeden z ciągów  $c_s$  i  $c_t$  jest niepusty

*Pary  $(c_s, c_t)$  nazywa się dopasowaniami. Dopasowaniem typu  $(i - j)$  nazywamy parę  $(c_s, c_t)$  taką, że długość ciągu  $c_s$  wynosi  $i$ , a długość ciągu  $c_t$  wynosi  $j$ .*

*Ciąg  $d$  musi spełniać następujące własności:*

- $\bigcup_{(c_s, c_t) \in d} c_s = s$
- $\bigcup_{(c_s, c_t) \in d} c_t = t$

gdzie  $\cup$  oznacza operację konkatencji ciągów.

Przy spełnieniu powyższych warunków, podczas operacji urównoleglania, żadne zdania nie mogą być pominięte lub zamienione w kolejności ani w ciągu  $s$ , ani w  $t$ . Dzięki temu, dysponując tylko urównolegleniem tekstów (ciągiem  $d$ ), można te teksty odtworzyć w takiej postaci, w jakiej były przed operacją urównoleglania.

Podczas operacji urównoleglania dąży się do tego, aby w dopasowaniach  $(c_s, c_t)$  zdania z ciągu  $c_t$  były tłumaczeniem zdań z ciągu  $c_s$ . Najprostszym algorytmem urównoleglania jest algorytm, którego ciąg  $d$  ma postać:

$$d = ((s_1, t_1), (s_2, t_2), (s_3, t_3), \dots)$$

Polega on więc na dopasowywaniu kolejno po jednym zdaniu źródłowym do jednego zdania docelowego. Taki algorytm nie jest wystarczający, gdyż nie uwzględnia rozbieżności, jakie zwykle występują pomiędzy tekstami. Wpływ na powstanie tych rozbieżności mają na przykład (za [Lip07]):

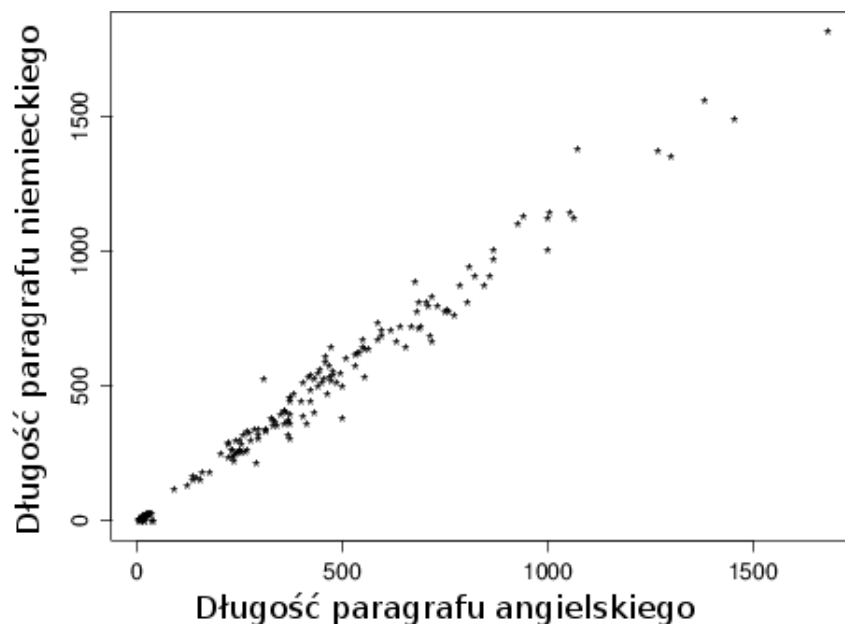
- różne liczby zdań użyte do wyrażenia tego samego w obu językach
- błędy spowodowane złą segmentacją tekstu
- pomyłkowe przeoczenie zdania przez tłumacza
- różnice w treści tekstów (zwykle na skutek aktualizacji tekstu  $S$  po przetłumaczeniu).

Szacuje się, że podobne problemy dotyczą około 10% wszystkich zdań w tekstach równoległych (za [Lip07]). Gdy najprostszy algorytm napotka choćby jeden tego typu problem, dopasowania utworzone za nim będą błędne. Z tego powodu opracowano algorytmy mniej wrażliwe na wyżej wymienione problemy.

### Algorytm Gale’a Church’a

Jest to algorytm bazujący na długości zdań, opisany w [GC91]. Oparty jest na założeniu, że zdania w różnych językach, które są swoimi tłumaczeniami, mają podobną długość. Według badań przeprowadzonych przez autorów algorytmu, założenie to jest poprawne. Z opracowania [GC91] pochodzi wykres korelacji długości paragrafów przykładowego tekstu angielskiego oraz jego niemieckiego tłumaczenia. Wykres ten jest przedstawiony na Rysunku 5.6. Pozioma oś przedstawia długość paragrafu angielskiego, a pionowa - długość odpowiadającego mu paragrafu niemieckiego.

Algorytm bierze pod uwagę wszystkie możliwe dopasowania. Dla każdego z nich obliczane jest prawdopodobieństwo w oparciu o stosunek długości



Rysunek 5.6. Korelacja długości paragrafów tekstów w dwóch językach

zdań ciągu  $c_s$  do długości zdań ciągu  $c_t$ . Następnie ze wszystkich dopasowań wybiera się ciąg dopasowań o największym prawdopodobieństwie.

Wyróżnia się następujące zalety algorytmów dopasowania, bazujących na długości zdań (za [Lip07]):

- niezależność od języków tekstów
- niska złożoność czasowa
- niska złożoność pamięciowa.

Wadą jest jednak niska jakość urównoległeń dla trudnych tekstów, tj. takich, w których występuje wiele dopasowań innego typu niż (1-1).

### Algorytm Moore'a

Algorytm ten, opisany w [Moo02], działa analogicznie do algorytmu Gale'a Church'a, z tą jednak różnicą, że prawdopodobieństwo dopasowania obliczane jest nie w oparciu o długość, ale o treść zdań. Algorytm próbuje dopasować te zdania, które zawierają słowa, będące swoimi odpowiednikami. Aby taki algorytm mógł funkcjonować, konieczne jest wykorzystanie słowników do sprawdzenia, które słowa są swoimi odpowiednikami. Prawdopodobieństwo dopasowania zdań jest wtedy obliczane dzięki metodom tłumaczenia statystycznego.

Podstawową zaletą algorytmów dopasowania bazujących na treści zdań jest wysoka jakość urównoległeń. Według danych zebranych przez autora pracy [Lip07], algorytm Moore'a pozwala uzyskać około 95% prawidłowych



dopasowań typu (1-1). Wadą algorytmu Moore'a (jak i wszystkich algorytmów bazujących na treści zdań) jest zależność od języków. Powoduje ona następującą niedogodność - gdy zachodzi potrzeba dopasowania tekstów w  $n$  różnych językach (dopasowując zawsze tylko 2 teksty na raz), należy na potrzeby algorytmu przygotować aż  $\binom{n}{2}$  słowników dwujęzycznych (jeden słownik dla każdego dwuelementowego zbioru języków). Opracowanie takich słowników może być bardzo czasochłonne.

Obliczanie podobieństwa dopasowania w algorytmie Moore'a jest znacznie bardziej skomplikowane, niż w algorytmie Gale'a-Church'a. Jednak dzięki optymalizacjom, zastosowanym w algorytmie Moore'a, nie występuje duża utrata szybkości działania w stosunku do algorytmu Gale'a-Church'a.

## Hunalign

Jednym z najbardziej rozpowszechnionych algorytmów urownowleglania jest hunalign (opisany w [VNH<sup>+</sup>05]). Jest on wykorzystywany w darmowym oprogramowaniu, opracowanym na potrzeby większego projektu. Projekt ten miał na celu zaimplementowanie tłumaczenia statystycznego pomiędzy językiem węgierskim, a angielskim. Mimo to, hunalign nadaje się do urownowleglania tekstów w dowolnych językach. Wejściem do algorytmu jest tekst podzielony na zdania oraz na tokeny. Wyjściem - ciąg par zdań w dwóch językach (pary te są nazywane *sentences*).

Algorytm hunalign łączy cechy algorytmu Gale'a-Church'a oraz Moore'a. Kiedy jest dostępny słownik, do obliczania prawdopodobieństwa dopasowania wykorzystywana jest zarówno informacja o długości dopasowywanych zdań, jak i informacje słownikowe. W przypadku braku słownika, algorytm działa dwufazowo. Najpierw dopasowuje teksty, wykorzystując jedynie informację o długości zdań. Na podstawie tego urownowleglenia buduje słownik. Następnie dokonuje dopasowania tekstów ponownie, tym razem korzystając nie tylko z informacji o długości zdań, ale także z nowo opracowanego słownika.

O popularności algorytmu zdecydowały następujące jego cechy: dostępność, wydajność i dokładność urownowlegleń.

## 5.3. Metoda autorska - przetwarzanie

### 5.3.1. Problemy podejścia tradycyjnego

Tradycyjna metoda tworzenia pamięci tłumaczeń niesie ze sobą pewne trudności. Powstałe w jej wyniku pamięci mają istotne wady, pierwszą z nich


jest występowanie błędów, wynikających z niedoskonałości algorytmów segmentacji i urównoleglania. W typowych zastosowaniach, odsetek błędnych przykładów w tradycyjnie przygotowanych pamięciach sięga nawet 10%-15%. Podawanie tłumaczowi błędnych przykładów podczas pracy jest nie tylko bezużyteczne, ale dodatkowo powoduje jego frustrację i ograniczenie zaufania do mechanizmu wspomagania tłumaczenia.

Drugą, znacznie istotniejszą wadą pamięci tłumaczeń pochodzących z różnorodnych źródeł jest duże rozdrobnienie informacji. Oznacza to, że pamięci te zawierają dużą ilość zdań, które mocno różnią się między sobą. Takie rozdrobnienie wynika z faktu, że zdania pochodzą na ogół z dokumentów słabo ze sobą powiązanych. Skutkuje to niskim prawdopodobieństwem znalezienia w takiej pamięci tłumaczeń przykładów przydatnych podczas tłumaczenia.

Pierwszym pomysłem na przezwycięzenie problemu rozdrobnienia jest skupienie się na wąskiej domenie tekstów (pomysł zainspirowany jest m.in. wnioskami z artykułu [GW03]). Pamięci tłumaczeń zawierające teksty z jednej dziedziny są przydatne podczas tłumaczenia nowych tekstów z tej dziedziny. Przykładowymi dziedzinami pamięci tłumaczeń są:

- prawo Unii Europejskiej
- ustawodawstwo konkretnego państwa
- medycyna
- instrukcje techniczne
- napisy filmowe


Wszystkie te dziedziny cechuje duża powtarzalność tekstów, która zwiększa przydatność pamięci tłumaczeń podczas tłumaczenia.

Samo skupienie się na wąskiej domenie tekstów nie gwarantuje jednak uzyskania dobrej jakości pamięci tłumaczeń. Występują następujące problemy 


- tłumaczenia pozyskiwane z dwujęzycznych dokumentów mogą być niskiej jakości,
- nawet po zawężeniu do jednej dziedziny, dane wciąż cechują się rozdrobnieniem.

Żeby dalej przeciwdziałać efektowi rozdrobnienia, klienci biur tłumaczeń często dysponują własnymi pamięciami, zawierającymi teksty ze szczególnie wąskiej dziedziny, takiej jak instrukcje samochodów konkretnej marki. Pamięci takie często są znacznej wielkości, przez co są trudne w przechowywaniu i przeszukiwaniu. Ponadto, ze względu na różne pochodzenie tłumaczeń, nie ma gwarancji ich jakości.

### 5.3.2. Zarys rozwiązania

Powyższym problemom wychodzi na przeciw autorska metoda tworzenia wyspecjalizowanej pamięci tłumaczeń wysokiej jakości. Metoda polega na przetwarzaniu gotowych pamięci tłumaczeń 

W pierwszym kroku zbierana jest dziedzinowa pamięć tłumaczeń. Nie ma przy tym znaczenia jakość tłumaczeń w niej występujących. W szczególnym przypadku może zostać użyta zdegenerowana pamięć tłumaczeń, której przykłady mają puste zdania docelowe.

W drugim kroku analizuje się wyłącznie zbiór zdań źródłowych wejściowej pamięci tłumaczeń. Spośród tych zdań wybiera się te najczęściej występujące, korzystając z algorytmu analizy skupień. Dla najczęstszych zdań źródłowych pobiera się ich tłumaczenia z wejściowej pamięci tłumaczeń (o ile pamięć ta nie była zdegenerowana). Tłumaczenia te są weryfikowane (lub opracowywane od nowa) przez tłumaczy. W wyniku tych prac powstaje wysokiej jakości pamięć, zawierająca tylko takie przykłady, które mają największą szansę kazać się przydatne podczas pracy tłumacza.

## 5.4. Analiza skupień

Centralnym punktem autorskiego algorytmu przetwarzania pamięci tłumaczeń jest analiza skupień zdań. W niniejszym rozdziale przybliżę zatem problem analizy skupień danych.


### 5.4.1. Opis problemu

W książce [KWGS09] podaje się następującą definicję.

**Definicja 21** (Analiza skupień). *Analiza skupień jest narzędziem analizy danych, służącym do grupowania  $n$  obiektów, opisanych za pomocą wektora  $p$  cech, w  $K$  niepustych, rozłącznych i możliwie “jednorodnych” grup - skupień.*

W obrębie skupień obiekty mają być maksymalnie podobne do siebie (w sensie przyjętej funkcji podobieństwa), natomiast pomiędzy skupieniami - maksymalnie niepodobne. Naiwny algorytm analizy skupień, dla ustalonego  $K$  i danego kryterium optymalnego podziału, sprawdza wszystkie możliwe podziały zbioru wejściowego. Podziałów tych jest (za [KWGS09]):

$$\frac{1}{K!} \sum_{k=1}^K (-1)^{K-k} \binom{K}{k} k^n$$

Dla  $n=100$  i  $K=4$  jest to liczba około  $10^{58}$ . Wykracza ona znacznie poza możliwości dzisiejszych komputerów. Poza tym, problemem jest stalenie

z góry parametru  $K$ , czyli liczby skupień. Tak jest między innymi w przypadku zbioru zdań.

### 5.4.2. Algorytmy hierarchiczne

Podstawową klasą algorytmów analizy skupień są rozwiązania hierarchiczne. Polegają one na krokowym budowaniu coraz większych skupień poprzez łączenie mniejszych zbiorów (metody aglomeracyjne). W innej wersji rozwiązania hierarchicznego, za pierwsze skupienie przyjmuje się cały zbiór, który jest później stopniowo dzielony.

Na Rysunku 5.7 przedstawiony jest słowny opis hierarchicznego algorytmu aglomeracyjnego (polegającego na łączeniu skupień). Opis jest zaczerpnięty z książki [KWGS09].

---

#### Algorytm: Aglomeracyjna analiza skupień

---

1. Utwórz jednoelementowe skupienia dla każdego z  $n$  elementów zbioru.
  2. Połącz dwa skupienia, zawierające najbardziej podobne obiekty.
  3. Jeśli liczba skupień jest większa od  $K$ , idź do 2.
- 

Rysunek 5.7. Aglomeracyjny algorytm analizy skupień

Algorytm ten wymaga zdefiniowania miary niepodobieństwa między dwoma obiektami oraz dwoma skupieniami. Wybranie dobrej miary niepodobieństwa jest kluczowe z punktu widzenia jakości skupień, powstałych w wyniku jego działania. Z książki [KWGS09] pochodzi następująca definicja miary niepodobieństwa:

**Definicja 22** (Miara niepodobieństwa). *Funkcję  $\rho : X \times X \mapsto \mathbb{R}$  nazywamy miarą niepodobieństwa, jeśli  $\forall x_r, x_s \in X$ :*

1.  $\rho(x_r, x_s) \geq 0$
2.  $\rho(x_r, x_s) = 0 \Leftrightarrow x_r = x_s$
3.  $\rho(x_r, x_s) = \rho(x_s, x_r)$

Z powyższej definicji wynika, iż określanie miary niepodobieństwa obiektów jest czynnością mającą dużo wspólnego z określeniem funkcji dystansu lub podobieństwa między nimi.

### 5.4.3. Algorytm K-średnich

Najczęściej wykorzystywanym algorytmem analizy skupień jest metoda K-średnich. Jest to algorytm niehierarchiczny. Do zadanej liczby skupień  $K$  przyporządkowuje się  $n$  obiektów, przy czym obliczenia nie bazują na wyznaczonych wcześniej mniejszych lub większych skupieniach. Opis metody K-średnich zaczerpnięty jest z książki [KWGS09].

Niech  $C_K$  oznacza funkcję, która przyporządkowuje każdemu obiektowi numer skupienia, do którego ten obiekt należy (przy podziale na  $K$  skupień). Załóżmy, że cechy obiektu są ilościowe i mogą być opisane liczbami rzeczywistymi (wektor cech należy do przestrzeni  $\mathbb{R}^p$ ). Główną ideą metody K-średnich jest takie rozmieszczenie obiektów w skupieniach, które minimalizuje zmienność wewnątrz skupień (tj. obiekty należące do jednego skupienia są podobne), a co za tym idzie maksymalizuje zmienność między skupieniami. Algorytm K-średnich jest przedstawiony na Rysunku 5.8.

---

**Algorytm: Analiza skupień metodą K-średnich**


---

1. Rozmieść w losowy sposób  $n$  obiektów w  $K$  skupieniach. Niech rozmieszczenie to będzie opisane przez funkcję  $C_K^{(1)}$ .
2. Dla każdego z  $K$  skupień oblicz wektor średnich  $\bar{x}_k$ ,  $k = (1, 2, \dots, K)$ .
3. Rozmieść ponownie obiekty w  $K$  skupieniach w taki sposób, że:

$$C_K^{(l)}(i) = \arg \min_{1 \leq k \leq K} \rho(x_i, \bar{x}_k).$$

4. Powtarzaj kroki 2 i 3 aż do momentu, kiedy przyporządkowanie obiektów pozostanie niezmiennym, czyli  $C_K^{(l)} = C_K^{(l-1)}$ .
- 

Rysunek 5.8. Algorytm K-średnich

Z punktu widzenia analizy skupień zdań, algorytm K-średnich ma dwie istotne wady. Po pierwsze, wymaga określenia z góry liczby skupień  $K$ . Po drugie, obiekty poddawane analizie muszą być opisane ilościowym wektorem cech, składającym się z liczb rzeczywistych.

Choć można stworzyć opis zdania w języku naturalnym w postaci ciągu liczb, taka reprezentacja nie byłaby naturalna. Potrafimy natomiast określić funkcję dystansu pomiędzy dwoma zdaniami. Poszukiwany algorytm analizy skupień dla obiektów będących zdaniami musiałby zatem korzystać z funkcji dystansu, a nie z wektorów cech.

#### 5.4.4. Algorytm QT

Algorytm Quality Threshold (QT), opisany w artykule [HKY99], powstał na potrzeby bioinformatyki. Posłużył do przeprowadzania analizy skupień genów, celem wyróżnienia skupień sekwencji genetycznych zależnych od siebie.

Algorytm QT jest odpowiedzią na wady algorytmów hierarchicznych oraz metody K-średnich. Według badań przeprowadzonych przez autorów artykułu [HKY99], algorytmy hierarchiczne mają tendencję do tworzenia albo małej liczby zbyt ogólnych (tzw. wydłużonych) skupień, albo dużej liczby zbyt zwartych zbiorów obiektów, które bardzo mało różnią się od siebie. Jest to spowodowane faktem, iż decyzje podejmowane przez algorytm hierar-

chiczny mają charakter lokalny. Co więcej, raz utworzone skupienie nie jest już modyfikowane. Zasada ta powoduje, że algorytm pozostaje przy pierwszym znalezionym przez siebie rozwiązaniu, które niekoniecznie jest najlepszym.

QT nie posiada wyżej wymienionych wad. Co więcej, nie wymaga określenia z góry liczby skupień  $K$ . Ponadto, opiera się wyłącznie na funkcji dystansu obiektów poddawanych analizie, nie na ich cechach ilościowych. Czyni go to zatem znacznie bardziej przydatnym w analizie skupień zdań, niż metoda K-średnich.

Opis algorytmu QT wykorzystuje pojęcie średnicy skupienia, które jest zdefiniowane poniżej.

**Definicja 23** (Średnica skupienia). *Średnicą skupienia  $S$  (ozn.  $sr(S)$ ) nazywamy największy dystans pomiędzy dwoma obiektami należącymi do tego skupienia, tj.  $sr(S) = \max_{a,b \in S} d(a,b)$*

Wejściami do algorytmu Quality Threshold są:

- zbiór  $U$  obiektów do analizy,
- funkcja  $d : U \times U \mapsto \mathbb{R}$ , spełniająca warunki dystansu (jak w Twierdzeniu 3),
- liczba  $d_{max}$ , oznaczająca maksymalną średnicę skupienia,
- liczba  $s_{min}$ , oznaczająca minimalną liczbę obiektów w skupieniu.

Działanie algorytmu jest następujące. Pierwszym kandydatem na skupienie jest zbiór składający się z pierwszego obiektu zbioru  $U$  oraz z obiektu znajdującego się od niego w najmniejszym dystansie. Następnie, pozostałe obiekty są poddawane analizie. Jeśli dodanie danego obiektu nie powoduje zwiększenia średnicy skupienia ponad  $d_{max}$ , obiekt jest dodawany do skupienia. Jeśli natomiast dodanie go spowodowałoby nadmierne zwiększenie średnicy, nie jest on dodawany. Proces ten jest kontynuowany do czasu, kiedy nie ma można już dodać do skupienia żadnych obiektów. W ten sposób powstaje kandydat na skupienie oparty o pierwszy obiekt z  $U$ . Podobną analizę przeprowadza się dla drugiego i kolejnych obiektów zbioru  $U$ . Należy tu zaznaczyć, że kandydaci na skupienia oparci na różnych obiektach mogą się przecinać. Spośród wszystkich kandydatów na skupienia wybiera się tego, który jest najliczniejszy. Staje się on pierwszym wyznaczonym skupieniem. Obiekty do niego należące są usuwane ze zbioru  $U$  i cała procedura jest powtarzana na pozostałych obiektach tego zbioru. Warunkiem stopu jest spadek liczności największego kandydata na skupienie poniżej  $s_{min}$ .

Pseudokod algorytmu Quality Threshold (na podstawie [HKY99]), jest przedstawiony na Rysunku 5.9.

---

**Algorytm: Analiza skupień metodą Quality Threshold**

---

```

procedure QTclust( $U, d_{max}, s_{min}$ )
  if ( $|G| \leq s_{min}$ )
    wypisz G
  else
    for all (Object  $i$  in  $G$ )
      flag := true
       $A_i := i$ 
      while (flag = true and  $A_i \neq G$ )
        znajdź  $j \in (G \setminus A_i)$  takie, że  $sr(A_i \cup \{j\})$  jest najmniejsza
        if ( $sr(A_i \cup \{j\}) > d_{max}$ )
          flag := false
        else
           $A_i := A_i \cup \{j\}$ 
      wybierz zbiór  $C \in \{A_1, A_2, \dots, A_{|G|}\}$  z największą liczbą elementów
      wypisz C
      QTclust( $G \setminus C, d_{max}$ )

```

---

Rysunek 5.9. Algorytm Quality Threshold

Istotną korzystną cechą algorytmu Quality Threshold jest możliwość podania parametru  $d_{max}$ . Gwarantuje to, że odległość obiektów należących do jednego skupienia nie będzie większa, niż wartość tego parametru. Poza tym, nie ma niebezpieczeństwa, że wyniki działania algorytmu QT będą silnie zależne od początkowych skupień, gdyż podczas jego działania sprawdzanych jest bardzo wiele kandydatów na skupienia. Cecha ta jest jednak okupiona dość wysoką złożonością czasową algorytmu, omówioną dokładnie w Podrozdziale 5.6.1.

## 5.5. Autorski algorytm przetwarzania pamięci tłumaczeń

Ogólna idea opracowanej przeze mnie metody przetwarzania pamięci tłumaczeń została przedstawiona w Podrozdziale 5.3.2. W niniejszym rozdziale zostanie opisany algorytm analizy skupień w zbiorze zdań.

### 5.5.1. Idea algorytmu


Algorytm działa na zbiorze zdań w jednym języku. Jego zadaniem jest wyodrębnienie skupień zdań podobnych w sensie ludzkim. Ponieważ wejściowy zbiór zdań może być i zazwyczaj jest znacznej wielkości, a dobra funkcja dystansu między zdaniem oraz algorytm QT są kosztowne obliczeniowo, opra-

cowalem metodę dwustopniowej analizy skupień zdań. Jest ona częściowo inspirowana ideą tzw. pseudoskupienia, opisanego w [Kar73].

Autorski algorytm wykorzystuje dwie funkcje dystansu między zdaniami: “tanią” i “drogą” (nazwy te odnoszą się do kosztowności obliczeniowej). Zbiór obiektów jest najpierw poddawany analizie skupień przy użyciu “taniej” funkcji dystansu. Następnie, wewnątrz prowizorycznych skupień wyodrębnionych w pierwszym kroku, szuka się ścisłych skupień przy użyciu “drogiej” funkcji dystansu. Pomysł dwustopniowej analizy skupień został zaczerpnięty z artykułu [MNU00]. Opisana tam metoda wykorzystywała jednak inny algorytm analizy skupień podczas każdego ze stopni. Ponadto, nie była ona dostosowana do analizy zbioru zdań.

Opisany poniżej autorski algorytm analizy skupień zdań został zoptymalizowany pod kątem szybkości i dokładności obliczeń.

### 5.5.2. “Tania” funkcja dystansu zdań

“Tania” funkcja dystansu zdań, oznaczona przez  $d_c$ , musi być obliczana szybko. Ma za zadanie wyznaczenie jedynie wstępnych skupień. Ważne, żeby obiekty trafiające do innych skupień na tym etapie były istotnie niepodobne do siebie. Nie ma jednak dużego znaczenia, czy obiekty, które znajdują się w jednym skupieniu, są podobne. O tym rozstrzygnie ugi etap analizy.


Wobec powyższych założeń, funkcję  $d_c : Z \times Z \mapsto [0, 1]$ , gdzie  $Z$  jest zbiorem wszystkich możliwych zdań, definiuję jak w równaniu 5.1:

$$\forall_{z_1, z_2 \in Z} d_c(z_1, z_2) = 1 - 2^{-\frac{|L_{z_1} - L_{z_2}|}{10}} \quad (5.1)$$

gdzie:

- $L_{z_1}$  jest długością zdania  $z_1$  w znakach,
- $L_{z_2}$  jest długością zdania  $z_2$  w znakach.

Funkcja  $d_c$  jest zatem oparta wyłącznie na długości zdań. Pozwala to na szybkie obliczanie dystansu między zdaniami. Do pierwszych, ogólnych skupień trafiają zdania o podobnej długości.

Różnica długości zdań,  $|L_{z_1} - L_{z_2}|$  jest regulowana parametrem  funkcja  $2^{-x}$  służy do mapowania różnicy długości zdań z przedziału  $[0, \infty]$  na przedział  $[0, 1]$ . Gdyby jednak przyjąć wzór  $d_c = 2^{-\frac{|L_{z_1} - L_{z_2}|}{10}}$ , różnica długości zdań wynosząca 0 odpowiadałaby dystansowi 1, a przy różnicy długości dążącej do  $\infty$ , dystans dążyłby do 0. W związku z tym, zastosowana została funkcja  $1 - x$ .



### 5.5.3. “Droga” funkcja dystansu zdań

Zakłada się, że “droga” funkcja dystansu zdań dobrze odzwierciedla ludzką intuicję podobieństwa zdań. Ma więc takie samo zadanie, jak funkcja  $d_s$ , opisana w Podrozdziale 4.5.3, używana przy wyszukiwaniu w pamięci tłumaczeń zdań podobnych. Funkcja  $d_s$  powstała jednak na inne potrzeby. Jest ona łatwo obliczalna, o ile dysponuje się odpowiednio skonstruowanym indeksem zdań. Na użytek algorytmu analizy skupień, opracowałem inną funkcję dystansu, oznaczaną przez  $d_e$ .

Aby funkcja dystansu dobrze odzwierciedlała ludzką intuicję, musi bazować nie na długości, ale na treści zdań. Pierwszym krokiem przy obliczaniu wartości funkcji  $d_e$  dla dwóch zdań jest oznaczenie w tych zdaniach jednostek nazwanych. Operacja ta wykonywana jest przy użyciu algorytmu rozpoznawania jednostek nazwanych, który nie jest przedmiotem niniejszej pracy. W wyniku oznaczenia tych jednostek, zdania stają się ciągami złożonymi ze słów oraz jednostek nazwanych.

Do obliczenia dystansu pomiędzy takimi ciągami wykorzystywane są pojęcia identyczności oraz odpowiadania słów i jednostek nazwanych. Są one zdefiniowane poniżej.

**Definicja 24** (Identyczność słów). *Dwa słowa nazywamy identycznymi, o ile ciągi znaków, z których są zbudowane, są identyczne.*

**Przykłady:** 'abc' = 'abc', 'aa'  $\neq$  'aaa'.


**Definicja 25** (Identyczność jednostek nazwanych). *Dwie jednostki nazwane nazywamy identycznymi, o ile składają się z identycznych słów.*

**Przykłady:** 'Jan Nowak' = 'Jan Nowak', '12 lipca 2001'  $\neq$  '12.07.2001'.

**Definicja 26** (Odpowiadanie słów). *Dwa słowa  $s_1$  i  $s_2$  nazywamy odpowiadającymi w sensie danego stemowania **stem**, o ile  $stem(s_1) = stem(s_2)$ .*

**Przykłady:** Słowo 'gramy' odpowiada słowu 'graliście' (wspólny stem to 'gra'), słowo 'przyjmujący' odpowiada słowu 'przyjmowany' (wspólny stem - 'przyjm'). Natomiast słowo 'widzę' nie odpowiada słowu 'wiem'.

Odpowiadanie słów zależy od przyjętego algorytmu stemowania. Odpowiadające sobie słowa są podobne w sensie ludzkim, gdyż najczęściej różnią się wyłącznie fleksją.

**Definicja 27** (Odpowiadanie jednostek nazwanych). *Dwie jednostki nazwane nazywamy odpowiadającymi, o ile mają ten sam typ.* 

Tablica 5.1. Wartości kar za różnice pomiędzy zdaniem  $z_1$  i  $z_2$ .

Różnica	Wartość kary
odpowiadanie (nie identyczność) słów	0.5
odpowiadanie (nie identyczność) jednostek nazwanych	0.5
słowo w $z_1$ , które nie należy do $z_2$	1.0
słowo w $z_2$ , które nie należy do $z_1$	1.0
jednostka nazwana w $z_1$ , która nie należy do $z_2$	1.5
jednostka nazwana w $z_2$ , która nie należy do $z_1$	1.5

**Przykłady:** Jednostka '12 lipca 2001' odpowiada jednostce '3.03.1995', gdyż obie są datami. Jednostka 'Jan Nowak' odpowiada jednostce 'Anna Kowalska', gdyż obie są nazwiskami.

Odpowiadanie jednostek nazwanych zależy od algorytmu ich oznaczania, którego zadaniem jest przypisanie typów znalezionych jednostek.

Po oznaczeniu jednostek nazwanych w dwóch zdaniach, obliczenie wartości funkcji  $d_e$  dla tych zdań jest oparte na odszukaniu różnic między nimi. Każdej różnicy przypisywana jest punktowa "kara", która zwiększa wartość dystansu pomiędzy zdaniem. Wartości kar za różnice pomiędzy zdaniem zostały wyznaczone na podstawie konsultacji z tłumaczami, którzy odwoływali się do własnej intuicji podobieństwa zdań. Z intuicji tej wynika na przykład, że jeśli w jednym zdaniu występuje jakaś jednostka nazwana, a w drugim nie, zdania te są mało podobne. Jednostka nazwana przenosi bowiem największą ilość informacji w zdaniu i w istotny sposób wpływa na jego treść. Skądinąd, jeśli jednostki nazwane występujące w dwóch zdaniach odpowiadają sobie, wtedy istnieje duże prawdopodobieństwo, że zdania te są podobne, gdyż różnią się tylko treścią informacji przesyłanej przez te jednostki. Na przykład zdania 'Umowę zawarto w dniu 7.06.2001 roku.' oraz 'Umowę zawarto w dniu 3.04.2006 roku.' będą uznane przez tłumaczy za podobne. Pełna tabela wartości kar przedstawiona jest w Tabeli 5.1.

Niech  $p$  będzie sumą kar naliczonych za różnice pomiędzy  $z_1$  i  $z_2$ . Określenie funkcji  $d_e : Z \times Z \mapsto [0, 1]$  jest przedstawione w równaniu 5.2.

$$\forall_{z_1, z_2 \in Z} d_e(z_1, z_2) = \begin{cases} \frac{2p}{\text{length}(z_1) + \text{length}(z_2)} & \text{gdy } 2p \leq \text{length}(z_1) + \text{length}(z_2) \\ 1 & \text{w przeciwnym przypadku} \end{cases} \quad (5.2)$$

Wartość funkcji  $d_e$  dla dwóch zdań odpowiada proporcji różnic między tymi zdaniem do ich długości. Dla par zdań, w których stosunek sumy kar do długości przekracza 1, przyjmuje się maksymalną wartość dystansu 1.

Opis słowny algorytmu naliczania kar dla zdań  $z_1$  i  $z_2$  poddanych stemowaniu i oznaczeniu jednostek nazwanaych jest przedstawiony na Rysunku 5.10.

---

**Algorytm: Naliczanie kar za różnice między zdaniem**

---

1. Utwórz tablicę o wymiarach  $length(z_1) \times length(z_2)$ .
  2. Wypełnij tablicę informacjami o odpowiedniości. W komórkę  $(i, j)$  wpisz jedną z informacji:
    - a)  $i$ -te słowo/jednostka nazwana  $z_1$  jest identyczna z  $j$ -tym słowem/jednostką nazwaną  $z_2$
    - b)  $i$ -te słowo/jednostka nazwana  $z_1$  odpowiada  $j$ -temu słowu/jednostce nazwanej  $z_2$
    - c)  $i$ -te słowo/jednostka nazwana  $z_1$  jest różne od  $j$ -tego słowa/jednostki nazwanej  $z_2$
  3. Nalicz karę 0.5 za każdy wiersz i każdą kolumnę tablicy, które nie zawierają informacji a). Są to kary za odpowiadanie.
  4. Nalicz karę 1.0 dla każdego wiersza i każdej kolumny, która została utworzona przez słowo i zawiera wyłącznie informacje c). Są to kary za braki słów.
  5. Nalicz karę 1.5 dla każdego wiersza i każdej kolumny, która została utworzona przez jednostkę nazwaną i zawiera wyłącznie informacje c). Są to kary za braki jednostek nazwanaych.
- 

Rysunek 5.10. Algorytm naliczania kar dla zdań

#### 5.5.4. Procedura wyodrębniania skupień

Dla zdefiniowanych funkcji  $d_c$  i  $d_e$ , danego zbioru zdań  $U$ , ustalonych parametrów  $d_{c-max}$ ,  $d_{e-max}$  oraz  $s_{min}$ , słowny opis opracowanego przeze mnie algorytmu analizy skupień zdań jest przedstawiony na Rysunku 5.11.

---

**Algorytm: Analiza skupień zdań**

---

1. Wyodrębnij skupienia w zbiorze  $U$  przy użyciu algorytmu QT z funkcją dystansu  $d_c$ , maksymalną średnicą skupienia  $d_{c-max}$  i minimalną liczbą obiektów w skupieniu  $s_{min}$ .
  2. Posortuj skupienia powstałe w kroku 1 malejąco względem liczby obiektów i wpisz te skupienia na listę  $C$
  3. Dla każdego skupienia  $c_L$  z listy  $C$ :
    - a) Wyodrębnij skupienia w zbiorze  $c_L$  przy użyciu algorytmu QT z funkcją dystansu  $d_e$ , maksymalną średnicą skupienia  $d_{e-max}$  i minimalną liczbą obiektów w skupieniu  $s_{min}$ .
    - b) Posortuj skupienia wyodrębnione w zbiorze  $c_L$  malejąco względem liczby obiektów.
    - c) Przekopiuj skupienia powstałe w kroku 3b) do wynikowej listy  $R$ .
- 

Rysunek 5.11. Autorski algorytm analizy skupień zdań

W wyniku działania algorytmu powstaje lista  $R$  skupień podobnych zdań, wyodrębnionych w zbiorze  $U$ . Z każdego zbioru wybierane jest pierwsze zdanie, które jest reprezentantem skupienia. Tak wybrane zdania stanowią zbiór zdań najczęściej występujących w zbiorze  $U$ . Po przetłumaczeniu lub zweryfikowaniu tłumaczeń tych zdań na język docelowy, powstanie pamięć tłumaczeń o wysokiej jakości i wyjątkowej przydatności dla tłumaczy.

## 5.6. Analiza algorytmu

### 5.6.1. Złożoność czasowa algorytmu QT

Ponieważ autorzy algorytmu QT nie podali jego złożoności czasowej, przeprowadzę tę analizę samodzielnie. Poniżej przedstawię złożoność czasową QT w przypadku optymistycznym, pesymistycznym i średnim. Za operację dominującą przyjmuję obliczenie dystansu pomiędzy dwoma obiektami.

**Twierdzenie 10** (Złożoność czasowa algorytmu QT). *Niech  $n$  oznacza liczbę obiektów poddawanych analizie. Złożoność czasowa algorytmu QT jest rzędu:*

- $O(n^2)$  w optymistycznym przypadku,
- $O(n^3)$  w pesymistycznym i średnim przypadku.

**Dowód 10.** W optymistycznym przypadku, wszystkie obiekty poddawane analizie skupień są identyczne. Wówczas, w pierwszym przebiegu algorytmu QT, każdy obiekt zostanie porównany z każdym obiektem, celem wyodrębnienia kandydata na skupienie. Skutkuje to obliczeniem wartości funkcji dystansu dokładnie  $n^2$  razy. Największym i jedynym kandydatem na skupienie będzie cały zbiór obiektów. Wobec tego, nie będzie konieczności przeprowadzania dalszych przebiegów algorytmu i obliczenia zostaną zakończone. To dowodzi, że złożoność obliczeniowa algorytmu QT w optymistycznym przypadku jest rzędu  $O(n^2)$ .

W przypadku pesymistycznym, otrzymujemy na wejściu  $s_{min} = 1$ , pewne ustalone  $d_{max}$ , oraz zbiór obiektów  $U$ , taki że:

$$\forall_{x_1, x_2 \in U} d(x_1, x_2) > d_{max}$$

Warunek ten można łatwo spełnić, wybierając  $d_{max} = 0$  lub bliskie 0. Wtedy, w pierwszym przebiegu algorytmu, który pochłonie  $n^2$  obliczeń dystansu, zostanie wyodrębnione 1-elementowe skupienie, złożone z pierwszego elementu zbioru  $U$ . Skupienie to zostanie odjęte ze zbioru  $U$  i w drugim przebiegu analizie będzie poddanych  $n - 1$  obiektów. Analiza ta będzie wiązała się z przeprowadzeniem  $(n - 1)^2$  obliczeń dystansu między obiektami. W jej wyniku ponownie zostanie wyodrębnione 1-elementowe skupienie. Proces ten będzie powtarzał się, aż w zbiorze  $U$  zostanie tylko jeden obiekt. On już nie zostanie poddany analizie, ale natychmiast zwrócony jako ostatnie skupienie. W sumie obliczeń dystansu pomiędzy obiektami będzie:

$$n^2 + (n - 1)^2 + (n - 2)^2 + \dots + 2^2$$

Suma ta ma  $(n - 1)$  wyrazów, z których każdy jest rzędu  $n^2$ . Stąd pesymistyczna złożoność czasowa algorytmu QT jest rzędu  $O(n^3)$ .

W przypadku średnim założmy, że w każdym przebiegu algorytmu zostanie wyodrębnione skupienie o mocy  $c_{avg}$ . Wtedy obliczeń dystansu między obiektami będzie w sumie:

$$n^2 + (n - c_{avg})^2 + (n - 2c_{avg})^2 + \dots + 2^2$$

Wyrazy powyższej sumy, podobnie jak w przypadku pesymistycznym, są rzędu  $n^2$ . Liczba tych wyrazów jest rzędu  $\lceil \frac{n}{c_{avg}} \rceil$ . Stąd złożoność czasowa algorytmu QT w średnim przypadku jest rzędu  $O(n^3)$ .

Dokładne wyznaczenie złożoności czasowej algorytmu QT potwierdziło przypuszczenia, że jest ona wysoka. Złożoność rzędu  $n^3$  jest zbyt wysoka, aby móc pracować z dużymi zbiorami zdań przy wykorzystaniu kosztownej funkcji dystansu pomiędzy zdaniami. W tej sytuacji bardziej korzystne jest stosowanie przedstawionego w niniejszej pracy algorytmu analizy skupień zdań, którego złożoność zostanie przedstawiona w kolejnych podrozdziałach.

### 5.6.2. Złożoność czasowa funkcji $d_c$

Aby określić złożoność czasową autorskiego algorytmu analizy skupień zdań, należy najpierw oszacować złożoność funkcji dystansu  $d_c$ . W przypadku tej funkcji operacją dominującą będzie odczytanie znaku w zdaniu, a rozmiarem danych sumaryczna długość zdań.

**Twierdzenie 11** (Złożoność czasowa funkcji  $d_c$ ). *Złożoność czasowa obliczania wartości funkcji  $d_c$  jest rzędu  $O(1)$ .*

**Dowód 11.** Obliczenie wartości funkcji  $d_c$  na podstawie wzoru 5.1 wymaga odczytania długości dwóch zdań wejściowych. Łańcuchy znaków są przechowywane w pamięci komputera wraz z informacją o ich długości (najczęściej długość łańcucha jest zapisana w jego zerowym znaku). Złożonością czasową funkcji  $d_c$  jest zatem funkcja  $f(n) = 2$ , a więc funkcja stała.

### 5.6.3. Złożoność czasowa funkcji $d_e$

Wyznaczenie złożoności czasowej funkcji  $d_e$  będzie opierało się na tych samych założeniach, co w przypadku funkcji  $d_c$ . Operacją dominującą będzie odczytanie znaku zdania, a rozmiarem danych sumaryczna długość zdań wejściowych, liczona w znakach. W rozważaniach tych pominięte jednak zostaną obliczenia wykonywane przez algorytmy stemowania oraz rozpoznawania jednostek nazwanych.

**Twierdzenie 12** (Złożoność czasowa funkcji  $d_e$ ). *Niech  $l$  oznacza sumaryczną liczbę znaków w zdaniach wejściowych funkcji  $d_e$ . Złożoność czasowa obliczania wartości funkcji  $d_e$  jest rzędu  $O(l^2)$ .*

**Dowód 12.** Jedyną operacją algorytmu obliczania wartości funkcji  $d_e$  (przedstawionego na Rysunku 5.10), w której odczytywane są znaki zdań, jest tworzenie tablicy odpowiedniości. Polega ono na odczytywaniu słów i jednostek nazwanych zdaniami, które z kolei sprowadza się do odczytywania znaków tych zdań. Jeśli przez  $m$  oznaczymy sumaryczną liczbę słów i jednostek nazwanych w zdaniach wejściowych, utworzenie tablicy odpowiedniości będzie wymagało  $O(m^2)$  kroków. Odczytanie słowa lub jednostki nazwanej wiąże się z odczytaniem stałej liczby znaków wchodzących w jego lub jej skład. W związku z tym, złożoność czasowa funkcji  $d_e$  jest rzędu  $O(l^2)$ .

Pamiętać jednak należy, że obliczenie wartości funkcji  $d_e$  w praktyce zajmuje jeszcze więcej czasu, z uwagi na konieczność użycia algorytmów stemowania i oznaczania jednostek nazwanych.

#### 5.6.4. Złożoność czasowa autorskiego algorytmu

Po wyznaczeniu złożoności czasowych algorytmu QT oraz funkcji  $d_c$  i  $d_e$ , można przystąpić do oszacowania złożoności czasowej autorskiego algorytmu analizy skupień zdań.

Przyjmijmy za podstawową operację odczytanie znaku zdania. Oznaczmy przez  $U$  zbiór zdań, poddawanych analizie. Moc tego zbioru oznaczmy przez  $n$ . Średnią długość zdania (w znakach) oznaczmy przez  $l$ .

**Twierdzenie 13** (Złożoność czasowa autorskiego algorytmu analizy skupień). *Złożoność czasowa autorskiego algorytmu analizy skupień jest rzędu:*

- $O(n^3)$  w optymistycznym przypadku,
- $O(n^3 + \frac{n^3}{\log^2 n} \cdot l^2)$  w pesymistycznym i średnim przypadku.

**Dowód 13.** Na podstawie Twierdzenia 10, złożoność czasowa algorytmu QT wynosi  $O(n^3)$ . Algorytm analizy skupień w przypadku optymistycznym po pierwszym stopniu analizy wyodrębni skupienia 1-elementowe. Będzie do tego potrzebował  $O(n^3)$  obliczeń, gdyż wykorzystywana w tym kroku funkcja  $d_c$  działa w czasie  $O(1)$ . Drugi stopień analizy nie będzie w tej sytuacji potrzebny. Stąd, optymistyczna złożoność czasowa autorskiego algorytmu analizy skupień zdań jest rzędu  $O(n^3)$ .

W przypadku pesymistycznym i średnim, po pierwszym stopniu analizy, pochłaniającym  $O(n^3)$  obliczeń, pozostają jeszcze skupienia do analizy w drugim stopniu. Zarówno liczba, jak i liczność tych skupień są proporcjonalne

do  $\log n$ . Każde z  $\log n$  skupień w drugiej fazie algorytmu jest poddane analizie przy użyciu algorytmu QT z funkcją  $d_e$ , o złożoności  $O(l^2)$ . Drugi etap analizy skupień ma zatem złożoność  $O(\log n \cdot (\frac{n}{\log n})^3 \cdot l^2)$ , co jest równoważne  $O(\frac{n^3}{\log^2 n} \cdot l^2)$ . W sumie, cały autorski algorytm analizy skupień zdań ma złożoność  $O(n^3 + \frac{n^3}{\log^2 n} \cdot l^2)$ .

W przypadku stosowania tylko algorytmu QT (jednostopniowo) z funkcją  $d_e$ , złożoność czasowa byłaby wyższa i wynosiłaby  $O(n^3 \cdot l^2)$  (na podstawie Twierdzeń 10, 12). Oznaczałoby to w praktyce znacznie większą ilość obliczeń, szczególnie w obliczu faktu, iż funkcja  $d_e$  jest obliczana dłużej, niż wskazywałaby na to jej teoretyczna złożoność (patrz Podrozdział 5.6.3).

Niska złożoność czasowa autorskiego algorytmu analizy skupień zdań czyni go przydatnym do przetwarzania nawet znacznej wielkości pamięci tłumaczeń.

## Rozdział 6

# Ewaluacja algorytmu przeszukiwania pamięci tłumaczeń

### 6.1. Kompletność i dokładność wyszukiwania

Ewaluacja kompletności i dokładności przeszukiwania pamięci tłumaczeń przy użyciu autorskiego algorytmu została wykonana jako analiza porównawcza z uznanym systemem klasy CAT - memoQ, opisanym w Podrozdziale 3.4. Analiza miała wykazać, czy algorytm dorównuje pod względem kompletności i dokładności mechanizmowi wykorzystywanemu w memoQ. Do mierzenia dokładności wyszukiwania, określonej w Definicji 19, konieczna była pomoc profesjonalnych tłumaczy. Ich zadaniem było sprawdzenie, czy wyniki przeszukiwania pamięci tłumaczeń, zwrócone przez oba algorytmy, mogą być wykorzystane jako podpowiedzi przy tłumaczeniu.

#### 6.1.1. Testowa pamięć tłumaczeń

Eksperyment został przeprowadzony na autentycznej pamięci tłumaczeń, zbudowanej przez tłumacza podczas jego pracy. Używanie takiej pamięci tłumaczeń uwiarygodnia wyniki testów. Często do podobnych testów używa się pamięci tłumaczeń pozyskanych z różnych źródeł, powstałych dzięki automatycznemu dzieleniu na zdania i urównoległaniu (proces ten jest opisany w Podrozdziale 5.2). Takie pamięci różnią się od autentycznych, wykorzystywanych przez tłumaczy. Wykorzystywana w niniejszej ewaluacji pamięć jest ograniczona do wąskiej dziedziny tekstów tłumaczonych przez jej autora i zawiera przykłady, które rzeczywiście są mu przydatne. Statystyki pamięci tłumaczeń wykorzystywanej w ewaluacji są przedstawione w Tabeli 6.1.

#### 6.1.2. Procedura ewaluacji

Procedura ewaluacji została podzielona na trzy fazy:

1. przygotowanie
2. przeszukiwanie



Tablica 6.1. Statystyki testowej pamięci tłumaczeń

Cecha	Wartość
Język źródłowy	polski
Język docelowy	angielski
Liczba przykładów	215 904
Liczba słów polskich	3 194 713
Liczba słów angielskich	3 571 598
Całkowita liczba słów	6 766 311

### 3. oznaczanie

Faza przygotowania polegała na utworzeniu testowego zbioru zdań oraz utworzenia indeksów do wyszukiwania. Zbiór testowy składał się z 1500 polskich zdań, wybranych losowo z pamięci tłumaczeń (oznaczymy ten zbiór przez *TEST*). Następnie, cała pamięć tłumaczeń została dodana do indeksów algorytmu autorskiego oraz memoQ.

Podczas fazy przeszukiwania, zarówno algorytm autorski, jak i memoQ wyszukiwały w pamięci tłumaczeń przykłady, służące jako podpowiedzi tłumaczenia dla każdego zdania ze zbioru *TEST*. Dokładny opis tej procedury jest przedstawiony na Rysunku 6.1.

---

#### Procedura: Przeszukiwanie testowej pamięci

---

```

for all sentence in TEST
  suggestions = getTranslationSuggestions(sentence)
  if (size(suggestions) == 0)
    report('error!')
  else if (size(suggestions) == 1)
    report('no suggestion found')
  else
    suggestions.remove(0)
  print(sentence, suggestions[0])

```

---

Rysunek 6.1. Faza przeszukiwania

W procedurze przeszukiwania, każde zdanie zostało wyszukane w pamięci tłumaczeń, która zawierała to zdanie. W związku z tym, oczekiwanym wynikiem przeszukiwania było zwrócenie między innymi przykładowego zdania wyszukiwane. Jeśli któryś z algorytmów zwróciłby pusty zbiór przykładów, sygnalizowałoby to niepożądaną sytuację, w której stuprocentowy przykład nie zostałby znaleziony. Jeśli zwrócony zbiór wyników zawierałby tylko jeden element, stuprocentowy przykład, zostałoby to zinterpretowane jako sytuacja, w której algorytm nie mógł znaleźć żadnego dobrego przykładu w swoim indeksie. Jeśli jednak zbiór wyników, oznaczany przez *suggestions*, zawierałby więcej niż jeden przykład, byłyby one (poza pierwszym, stupro-

Tablica 6.2. Kompletność przeszukiwania pamięci tłumaczeń

	memoQ	algorytm autorski
Przeanalizowanych zdań	1500	1500
Zgłoszone błędy	0	0
Niepuste wyniki przeszukiwania	1156	962
Kompletność	77,1%	64,1%

centowym przykładem) właściwymi przykładami. Pod uwagę brany był zawsze najlepszy właściwy przykład, tj. ten z największą wartością oceny dopasowania.

### 6.1.3. Kompletność przeszukiwań

Procedura przeszukiwania została wykonana zarówno przez algorytm autorski, jak i przez memoQ. Wyniki kompletności przeszukiwania są zaprezentowane w Tabeli 6.2.


“Zgłoszone błędy” odnoszą się do liczby błędów, polegających na nieodnalezieniu w pamięci tłumaczeń stuprocentowego przykładu. Oba algorytmy nie popełniły błędów na testowych danych. W przypadku algorytmu autorskiego wynika to bezpośrednio z Twierdzenia 2, natomiast w przypadku memoQ można tylko przypuszczać, że również posiada on własność przykładu doskonałego. “Niepuste wyniki przeszukiwania” to liczba zdań ze zbioru *TEST*, dla których algorytm znalazł przynajmniej jeden przykład właściwy (tj. zbiór *suggestions* zawierał więcej niż 1 element). Wyniki kompletności były porównywalne, choć algorytm memoQ przetłumaczył ok. 20% więcej zdań. Było to jednak spowodowane faktem, iż implementacja algorytmu autorskiego była wyposażona w mechanizm, który usuwał ze zbioru wyników przykłady o ocenie niższej niż 50%. Taki zabieg ma duże znaczenie praktyczne, gdyż przykłady o niskich ocenach bardzo rzadko są przydatne tłumaczowi podczas pracy. Algorytm memoQ pracował bez tego ograniczenia, dlatego osiągnął lepszy wynik kompletności.

### 6.1.4. Automatyczna analiza wyników

#### Analiza wstępna

Podczas fazy przeszukiwania, automatycznej analizie porównawczej zostały poddane wyniki przeszukiwania, podane przez algorytm memoQ oraz autorski. Tabela 6.3 przedstawia statystyki tej analizy.

Tablica 6.3. Statystyki przeszukiwania

	memoQ	Anubi 
Wspólnych niepustych wyników wyszukiwania	871	
w tym:		
- identycznych wyników	444	
- różnych wyników	427	

Dla 871 zdań, które były wyszukiwane, oba algorytmy znalazły jakieś dopasowania z pamięci tłumaczeń. Zaskakujące jest, że w przypadku aż 427, czyli ok. 50% z nich, wyniki te były różne. Zostały one poddane wnikliwej analizie pod kątem dokładności, której wyniki są przedstawione w Podrozdziale 6.1.5.

### Analiza ocen dopasowania - idea

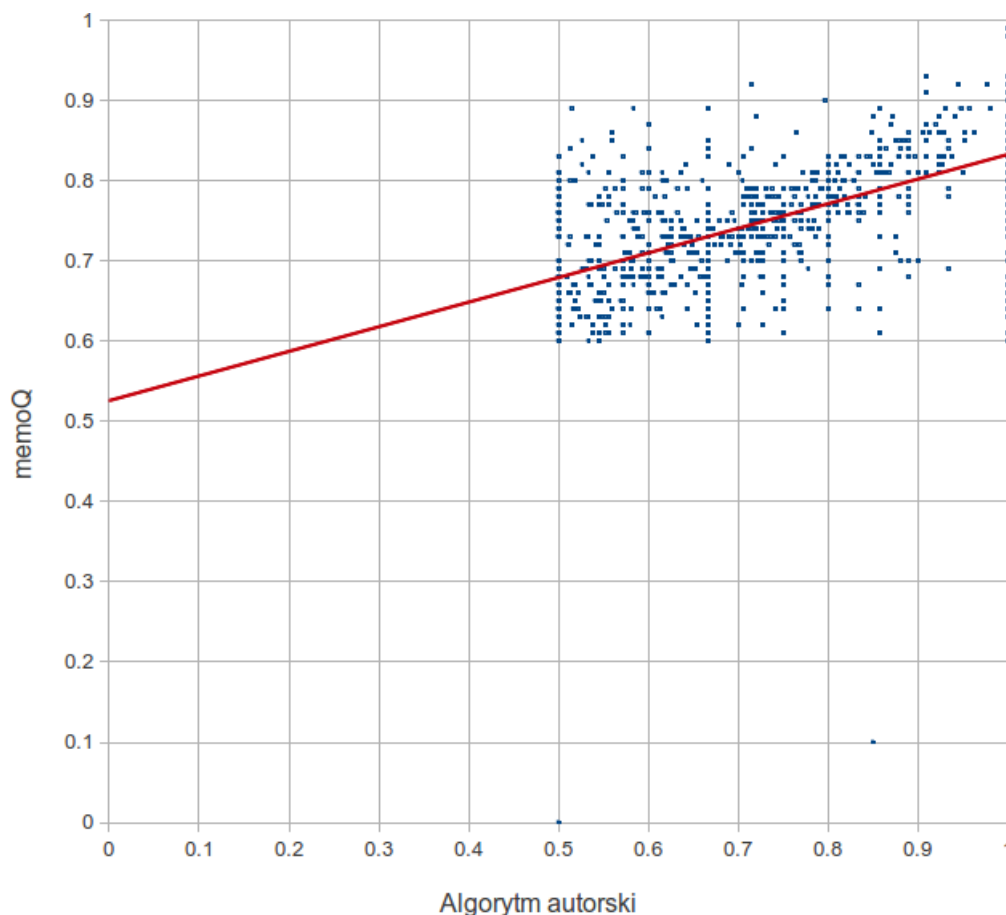
Innym parametrem, który został przeanalizowany automatycznie, były oceny dopasowań, zwrócone przez oba algorytmy dla tych samych wyników wyszukiwania. We wstępnej analizie udało się ustalić, że algorytm memoQ przyznawał na ogół lepsze oceny dopasowań, niż autorski. Oceny tych samych wyników podawane przez algorytm memoQ były wyższe w 491 przypadkach (spośród 871), podczas gdy oceny autorskiego algorytmu były lepsze tylko w 374 przypadkach. W pozostałych 6 przypadkach oba algorytmy podały tę samą ocenę.

Podczas bardziej wnikliwej analizy, zbadana została korelacja ocen dopasowania, podawanych przez oba algorytmy. Badania te miały na celu sprawdzenie, czy ocena w algorytmie autorskim jest w istocie tym samym, co ocena w algorytmie memoQ, czy też parametrem niezależnym.

### Regresja liniowa

Oceny podawane przez algorytmy zostały potraktowane jako zmienne losowe. Oceny algorytmu autorskiego zostały oznaczone jako rozkład zmiennej losowej  $X$ , natomiast oceny memoQ jako rozkład zmiennej  $Y$ . Rysunek 6.2 przedstawia wykres, obrazujący korelację zmiennych  $X$  i  $Y$ . Każdy punkt odpowiada parze ocen dopasowania, zwróconych przez oba algorytmy dla jednego wyniku wyszukiwania.

Na wykresie trudno dopatrzeć się ścisłej zależności pomiędzy ocenami jednego i drugiego algorytmu. Czerwonym kolorem zaznaczona jest prosta regresji (prosta najlepszego dopasowania do punktów wykresu), wyznaczona



Rysunek 6.2. Korelacja ocen dopasowania

dzięki metodzie najmniejszych kwadratów (opisanej w [RT99]). Jej równanie jest postaci:

$$f(x) = 0.307x + 0.526$$

Obliczony został także współczynnik korelacji Pearsona dla zmiennych  $X$  i  $Y$  (na podstawie opracowania [Sti89]). Jeśli przez  $x_i, y_i$  oznaczymy wartości prób losowych tych zmiennych (gdzie  $i = 1, 2, \dots, n$ ), przez  $\bar{x}, \bar{y}$  wartości średnie tych prób (tj.  $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i, \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$ ) wtedy estymator współczynnika korelacji liniowej Pearsona dany jest wzorem:

$$r_{XY} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \quad (6.1)$$

Innymi słowy, współczynnik ten jest ilorazem kowariancji i iloczynu odchyłeń standardowych zmiennych  $X$  i  $Y$ :

$$r_{XY} = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y}$$

Dla zmiennych losowych, reprezentujących oceny dopasowania, podawane przez algorytm autorski i memoQ, wartość współczynnika korelacji liniowej Pearsona wynosi:

$$r_{XY} = 0.5410990466 \quad (6.2)$$

Taka wartość współczynnika korelacji oznacza słabą dodatnią korelację badanych zmiennych. Dodatnia korelacja oznacza, że na ogół jeśli rośnie ocena dopasowania z algorytmu autorskiego, rośnie również ocena memoQ.

### Korelacja rangowa Spearmana

Ponieważ jednak współczynnik korelacji Pearsona jest wrażliwy na obserwacje skrajne (za [Spe04]), przeprowadzone zostały badania korelacji rangowej zmiennych  $X$  i  $Y$ . Rangowanie zmiennej losowej polega na zastąpieniu jej wartości rangami. Rangą nazywamy numer kolejny wartości zmiennej, po uporządkowaniu wszystkich wartości. W eksperymencie zastosowano rangi będące liczbami naturalnymi, rozpoczynającymi się od 1. Rysunek 6.3 przedstawia wykres rang zmiennych  $X$  i  $Y$ .

Dla zmiennych  $X$  i  $Y$  obliczony został współczynnik korelacji rang Spearmana, nazwany także  $\rho$  Spearmana (opisany w [Spe04]). Współczynnik ten pozwala zbadać zależność monotoniczną zmiennych losowych. Jest on zdefiniowany jako współczynnik korelacji Pearsona (dany wzorem 6.1) zmiennych losowych po operacji rangowania. Obliczona wartość współczynnika korelacji Spearmana dla zmiennych  $X$  i  $Y$  wynosi:

$$\rho_{XY} = 0.5531498628 \quad (6.3)$$

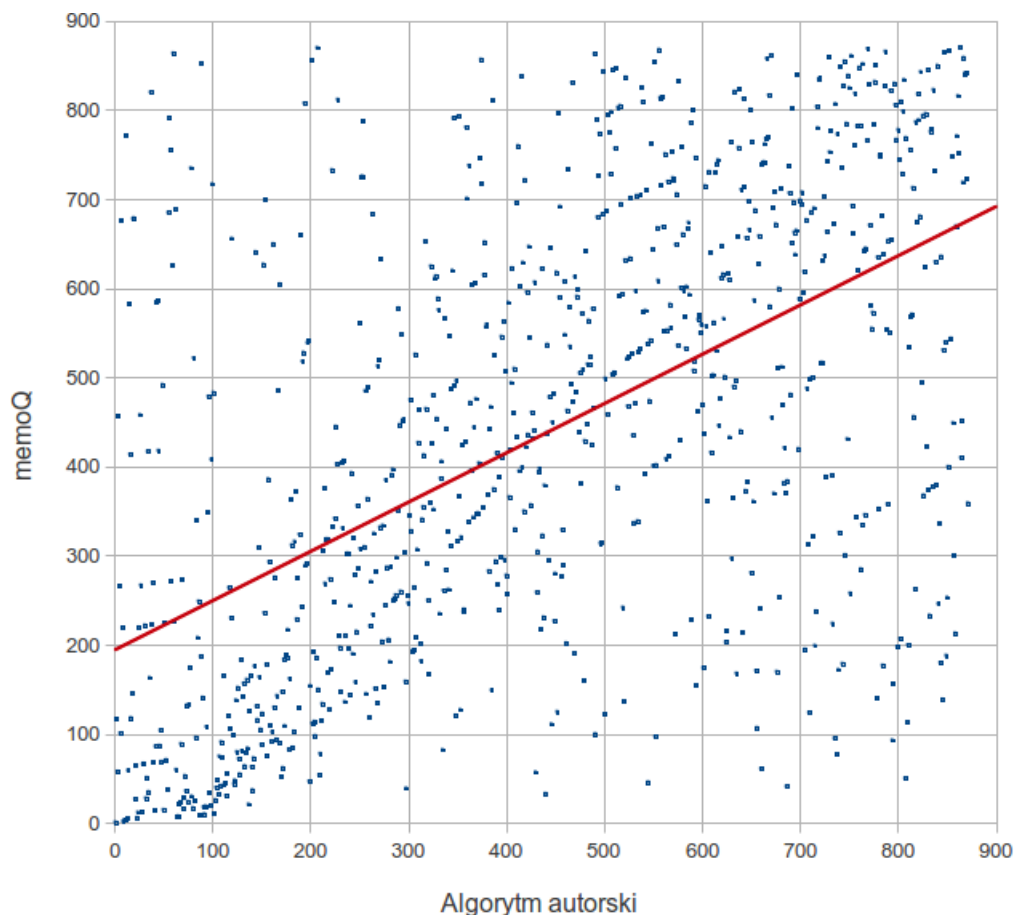
Natomiast prosta regresji rang zmiennych  $X$  i  $Y$ , zaznaczona na wykresie kolorem czerwonym, jest dana wzorem:

$$f(x) = 0.553x + 194.827$$

Wartość współczynnika Spearmana (wzór 6.3, bardzo zbliżona do wartości współczynnika Pearsona (wzór 6.2), oznacza w dalszym ciągu słabą dodatnią korelację ocen dopasowania algorytmu autorskiego i memoQ.

### Korelacja rangowa Kendalla

Dla pełności wiedzy na temat korelacji badanych zmiennych, został obliczony jeszcze jeden współczynnik:  $\tau$  Kendalla (opisany w [Ken38]). Podob-



Rysunek 6.3. Korelacja rang ocen dopasowania

nie jak  $\rho$  Spearmana, pozwala on badać zależność monotoniczną zmiennych losowych, operując na rangach tych zmiennych. Aby obliczyć wartość tego współczynnika, należy najpierw utworzyć wszystkie możliwe pary obserwacji  $\{(x_i, y_i), (x_j, y_j)\}$ . Następnie pary te dzieli się na trzy kategorie:

- **pary zgodne**, w których  $x_i > x_j$  oraz  $y_i > y_j$  lub  $x_i < x_j$  oraz  $y_i < y_j$ . Liczbę par zgodnych oznaczmy przez  $P$ .
- **pary niezgodne**, w których  $x_i > x_j$ , a  $y_i < y_j$  lub  $x_i < x_j$ , a  $y_i > y_j$ . Liczbę par niezgodnych oznaczmy przez  $Q$ .
- **pary wiązane**, w których  $x_i = x_j$  lub  $y_i = y_j$ . Liczbę par wiązanych oznaczmy przez  $T$ .

$\tau$  Kendalla definiuje się według wzoru:

$$\tau = \frac{P - Q}{P + Q + T} \quad (6.4)$$


Obliczona wartość  $\tau$  Kendalla dla zmiennych  $X$  i  $Y$  wynosi:

$$\tau = 0.4126186046$$

Wartość ta jest interpretowana jako różnica prawdopodobieństwa, że dwie

oceny będą się układały w tym samym porządku dla dwóch obserwacji oraz prawdopodobieństwa, że ułożą się w porządku odwrotnym. Wartość rzędu 40% wskazuje na słabą korelację zmiennych.

### Korelacja ocen dopasowania - podsumowanie

Badane oceny dopasowania wykazały średnią  relację między sobą. Można na tej podstawie wysnuć wniosek, że z dużym prawdopodobieństwem oceny te są od siebie niezależne i obliczane istotnie różnymi metodami. Wystąpienie jakiegokolwiek korelacji między nimi jest spowodowane faktem, że obie oceny zostały opracowane na potrzeby mierzenia tego samego parametru - podobieństwa wyniku przeszukiwania pamięci tłumaczeń do zdania wyszukiwanego.

Wniosek ten może zostać uznany za pozytywny, gdyż jednym z celów badań nad przeszukiwaniem pamięci tłumaczeń było opracowanie nowej oceny dopasowania. Ocena ta miała opierać się na innych założeniach, niż rozwiązania znane do tej pory, które najczęściej oparte były algorytmach opisanych w Podrozdziale 4.2.2. Rozwiązania te, zaprojektowane na potrzeby przybliżonego wyszukiwania łańcuchów znaków, nie były dobrze dostosowane do wyszukiwania zdań. Algorytmy przeszukiwania pamięci tłumaczeń, wyposażone w takie mechanizmy obliczania oceny dopasowania, nie mogły zawsze zwracać wszystkich zdań z pamięci, przydatnych tłumaczowi.

#### 6.1.5. Dokładność przeszukiwań

Algorytm autorski oraz memoQ podały identyczne wyniki wyszukiwania w 444 na 871 przypadków, w których oba podały jakikolwiek wynik. Pozostałe 427 przypadków zostało przeanalizowane przez dwoje tłumaczy. Stanowiło to trzecią fazę eksperymentu - oznaczanie.

Tłumacze otrzymali zbiór uporządkowanych trójek:  $(S, R_A, R_B)$ , gdzie:

- $S$  - zdanie źródłowe
- $R_A$  - wynik wyszukiwania algorytmu A
- $R_B$  - wynik wyszukiwania algorytmu B

Aby uzyskać bardziej zbilansowane wyniki eksperymentu, tłumaczom nie zostało ujawnione, który algorytm został nazwany  $A$ , a który  $B$ . Co więcej, kolejność elementów  $R_A$  i  $R_B$  w części losowo wybranych trójek była zamieniona.

Każda trójka miała zostać oznaczona przez obu tłumaczy jedną z następujących ocen:

- wynik  $R_A$  jest lepszy

Tablica 6.4. Porównawcza dokładność przeszukiwania pamięci tłumaczeń

	<b>Tłumacz 1</b>	<b>Tłumacz 2</b>
Razem trójek	427	
Zwycięstw algorytmu autorskiego	<b>156</b>	<b>103</b>
Zwycięstw algorytmu memoQ	150	96
Remisów	121	228

Tablica 6.5. Zgodność tłumaczy podczas oznaczania

<b>Cecha</b>	<b>Wartość</b>
Ogółem oznaczeń	427
Zgodnych oznaczeń	262
Ogółem niezgodnych oznaczeń	165
Silnie niezgodnych oznaczeń	24

- wynik  $R_B$  jest lepszy
- wyniki  $R_A$  i  $R_B$  są takiej samej wartości

Kryterium wartości wyniku wyszukiwania było następujące: “Na ile pomocny jest wynik przeszukiwania pamięci tłumaczeń przy tłumaczeniu zdania źródłowego na język docelowy?”. Jest ono zgodne z definicją pojęcia dokładności (Definicja 19).

Takie podejście do sprawdzania dokładności przeszukiwania pamięci tłumaczeń w istocie nie mierzy dokładności każdego algorytmu z osobna, ale analizuje ten parametr porównawczo. Metoda ta, nazywana porównywaniem parami (ang. Paired Comparison Analysis), została po raz pierwszy opisana w [Thu27] i w dalszym ciągu znajduje szerokie zastosowanie. Porównywanie parami ma w opisywanym eksperymencie większe uzasadnienie, niż badanie z osobna dokładności wyszukiwania obu algorytmów. W tym drugim przypadku, uzyskane obiektywne wartości dokładności byłyby silnie zależne od użytej pamięci tłumaczeń i tym samym nie dawałyby janszego poglądu na przydatność algorytmów. Wyniki fazy oznaczania przedstawione są w Tabeli 6.4.

Zmierzona została także zgodność tłumaczy. Jej wyniki są przedstawione w Tabeli 6.5. Przez silnie niezgodne oznaczenie trójki  $(S, R_A, R_B)$  rozumiana jest sytuacja, w której jeden z tłumaczy przyznał jej ocenę: “zwycięstwo systemu  $A$ ”, a drugi: “zwycięstwo systemu  $B$ ”. Na podstawie wyników przedstawionych w Tabeli 6.5, uznać można, że zgodność osób oznaczających była wysoka.



Tablica 6.6. Parametry techniczne maszyny testowej

Parametr	Wartość
Procesor	Intel Core 2 Duo
Liczba rdzeni	2
Częstotliwość taktowania	2.0 GHz
Obsługa przetwarzania 64-bitowego	tak
Pamięć cache	L2, 1.0MB
Typ pamięci RAM	DDR3 SDRAM
Prędkość pamięci RAM	1066.0 MHz / PC3-8500
Rozmiar pamięci RAM	3GB
Dysk twardy	Optyczny, prędkość 5400 RPM

### 6.1.6. Podsumowanie eksperymentu

Wyniki porównawczej analizy dokładności algorytmów ukazują nieznaczną przewagę algorytmu autorskiego. W istocie jednak można zaobserwować, iż dokładność obu algorytmów jest porównywalna. Wniosek ten może być rozpatrywany w kategoriach sukcesu, gdyż oznacza, że nowo opracowany algorytm autorski potrafi konkurować z algorytmem dobrze ugruntowanym w dziedzinie przeszukiwania pamięci tłumaczeń.

Przed autorskim algorytmem przeszukiwania pamięci tłumaczeń zostało postawione także inne, równie istotne wymaganie: szybkość działania. Wyniki teoretycznej analizy złożoności czasowej i pamięciowej, przedstawione w Podrozdziale 4.5 sugerują, że algorytm rzeczywiście jest w stanie przeszukiwać pamięć tłumaczeń szybko. W Podrozdziale 6.2 przedstawione są wyniki praktycznych eksperymentów, mających zweryfikować te przypuszczenia.

## 6.2. Szybkość wyszukiwania

### 6.2.1. Porównanie z memoQ

Podczas fazy wyszukiwania eksperymentu opisanego w Podrozdziale 6.1, zmierzona została szybkość wyszukiwania, osiągnięta przez algorytm autorski oraz memoQ. Testy zostały przeprowadzone na maszynie o parametrach przedstawionych w Tabeli 6.6.

Wyniki testu szybkości są przedstawione w Tabeli 6.7. Wyniki te ujawniają znaczącą przewagę algorytmu autorskiego.

Tablica 6.7. Szybkość wyszukiwania algorytmu autorskiego i memoQ

	memoQ	algorytm autorski
Razem wyszukiwań	1500	
Czas wyszukiwania [s]	414.6	<b>94.7</b>
Średnia szybkość [zdań/s]	3.618	<b>15.846</b>
Średni czas wyszukiwania zdania [s]	0.2764	<b>0.0632</b>

### 6.2.2. Porównanie z Lucene

#### Procedura ewaluacji

W celu ewaluacji szybkości, przeprowadzono jeszcze jeden eksperyment. Jego zadaniem było porównanie szybkości przeszukiwania pamięci tłumaczeń z innym algorytmem wyszukiwania przybliżonego: Apache Lucene (opisanym na [luc]). Algorytm ten, podobnie jak autorski, wykorzystuje indeks wyszukiwania, który przechowuje w pamięci operacyjnej.

Do eksperymentu wybrano pamięć tłumaczeń, złożoną z przykładów wybranych losowo z korpusu Komisji Europejskiej, JRC-Acquis (opisanego na [aut]). Oznaczmy ten zbiór przez  $JRC-TM$ . Moc zbioru  $JRC-TM$  wynosiła 193 827. Z całego korpusu JRC-Acquis wybrano zbiór zdań testowych  $JRC-TEST$  w taki sposób, aby  $JRC-TM \cap JRC-TEST = \emptyset$ . Moc zbioru  $JRC-TEST$  wynosiła 452.

Procedura testowa algorytmu Lucene jest przedstawiona na Rysunku 6.4. Uwagi do procedury testowej Lucene 

- Należy zwrócić uwagę na konstrukcję kwerendy wyszukiwania Lucene. Po każdym słowie zdania wejściowego jest dodawany znak tyldy. Kwerenda skonstruowana w taki sposób szuka każdego zdania indeksu, które zawiera jedno ze słów kwerendy. Co więcej, jeśli słowa nie pasują do siebie idealnie (np. “zadania” i “zagania”), będą i tak traktowane jako pasujące (na podstawie dokumentacji Lucene).
- Zarówno  $JRC-TM$ , jak i  $JRC-TEST$  są ładowane do pamięci operacyjnej, żeby uniknąć opóźnień spowodowanych czytaniem z dysku twardego.
- Algorytm Lucene wykorzystuje procedurę *optimizeIndex*, która jest wywoływana po dodaniu wszystkich przykładów z  $JRC-TM$  do indeksu.
- Przed mierzonymi wyszukiwaniami, uruchamiana jest tzw. “kwerenda rozgrzewająca”. Ma ona zapewnić, że algorytm wykona wszystkie procedury inicjalizacyjne przed przystąpieniem do wyszukiwania.

Procedura testowa algorytmu autorskiego została skonstruowana podobnie, jak w przypadku Lucene. Jest ona przedstawiona na Rysunku 6.5.

---

**Procedura testowa: Lucene**

---

```
procedure testLucene()

    tmSet := loadFrom(TM)
    lucene.addToIndex(tmSet)
    lucene.optimizeIndex()

    lucene.search("warm up query")

    testSet := loadFrom(TEST)
    startTimer()
    for sentence s in testSet do
        sentenceWords := s.split("\s")
        query := ""

        for word w in sentenceWords do
            query := query + w + "~ "
        lucene.search(query)
    stopTimer()

end procedure
```

---

Rysunek 6.4. Procedura testowa algorytmu Lucene

---

**Procedura testowa: algorytm autorski**

---

```
procedure testAlgorithm()

    tmSet := loadFrom(TM)
    algorithm.addToIndex(tmSet)

    algorithm.search("warm up query")

    testSet := loadFrom(TEST)
    startTimer()
    for sentence s in testSet do
        algorithm.search(s)
    stopTimer()

end procedure
```

---

Rysunek 6.5. Procedura testowa algorytmu autorskiego

Różnica w stosunku do procedury testowej Lucene polega na braku wywoływania procedury optymalizującej indeks. Procedura ta nie jest wykorzystywana przez algorytm autorski, gdyż jego indeks jest optymalny w momencie utworzenia. Inną różnicą jest sposób konstrukcji kwerendy wyszukującej.

## Wyniki eksperymentu

Obie procedury testowe zostały uruchomione 3 razy, aby uzyskać zbalansowane wyniki. Rezultaty testów są przedstawione w Tabeli 6.8. Liczby wska-

Tablica 6.8. Szybkość wyszukiwania - uruchomienia procedury testowej

Algorytm	Test 1 [s]	Test 2 [s]	Test 3 [s]
Lucene	169.369	168.562	168.223
Autorski	<b>16.109</b>	<b>15.172</b>	<b>16.486</b>

Tablica 6.9. Szybkość wyszukiwania - wyniki testów

Algorytm	Czas [s]	Szybkość [zdań/s]	Średni czas [s]
Lucene	168.718	2.68	0.373
Autorski	<b>15.922</b>	<b>28.39</b>	<b>0.035</b>

zują czas (w sekundach) każdego z trzech uruchomień procedury testowej. W Tabeli 6.9 są natomiast zaprezentowane uśrednione wyniki 3 uruchomień procedury testowej.

Podobnie jak w przypadku testów porównawczych z algorytmem memoQ, również w przypadku Lucene widoczna jest znacząca przewaga algorytmu autorskiego.

### 6.2.3. Analiza szybkości wyszukiwania - podsumowanie

Autorski algorytm przeszukiwania pamięci tłumaczeń charakteryzuje się dużą szybkością wyszukiwania. Parametr ten może być kluczową cechą systemu klasy CAT w kontekście używania serwerów pamięci tłumaczeń. Wraz ze wzrostem objętości zbieranych danych, biura tłumaczeń starają się wdrożyć ideę scentralizowanych serwerów pamięci tłumaczeń. Zadaniem takich serwerów jest udostępnianie przydatnych przykładów tłumaczenia wszystkim zrzeszonym tłumaczom. Z drugiej strony, tłumacze samodzielni (określani angielskim mianem *freelance translators*) często tworzą społeczności, w których dzielą się swoimi pamięciami tłumaczeń. Przykładem takiego projektu jest Wordfast VLTM - Very Large Translation Memory, który jest opisany na [mulb]).

Tendencje te prowadzą do tworzenia znacznej wielkości pamięci tłumaczeń, dzielonych pomiędzy wieloma użytkownikami. W tej architekturze, elementem najwyższej wagi jest skuteczny i szybki algorytm przeszukiwania pamięci tłumaczeń. Algorytm ten musi być w stanie przeszukać znaczne zbiory danych w krótkim czasie, gdyż wielu użytkowników pracujących jednocześnie na jednej pamięci będzie przeszukiwać ją często. W związku z tym, idea zwiększenia szybkości przeszukiwania pamięci tłumaczeń jest kluczowym warunkiem rozwoju nowoczesnych systemów klasy CAT.

## Rozdział 7

# Ewaluacja algorytmu przetwarzania pamięci tłumaczeń

Pamięć tłumaczeń przygotowana przy użyciu autorskiej metody, wykorzystującej analizę skupień zdań, powinna umożliwiać zwiększenie kompletności i dokładności przeszukiwań. Jeśli wyniki tych przeszukiwań byłyby zwracane przez system wspomagania tłumaczenia, pozwoliłyby one ograniczyć ilość ludzkiej pracy nad tłumaczeniem.

Niniejszy rozdział opisuje eksperyment, który miał na celu zweryfikować w jakim stopniu specjalizowana pamięć tłumaczeń jest w stanie zwiększyć jakość sugestii, generowanych przez system wspomagania tłumaczenia.

### 7.1. Warunki eksperymentu

#### 7.1.1. System wspomagania tłumaczenia

Eksperyment został przeprowadzony przy udziale osób, będących zawodowymi tłumaczami. Użyto systemu wspomagania tłumaczenia, który był na co dzień wykorzystywany przez tych tłumaczy. Był on wyposażony w algorytm przeszukiwania pamięci tłumaczeń, oparty na bibliotece Apache Lucene ([luc]). Nie użyto autorskiego algorytmu przeszukującego, gdyż celem eksperymentu było zbadanie wpływu samej pamięci tłumaczeń na jakość generowanych podpowiedzi tłumaczenia.

#### 7.1.2. Scenariusze eksperymentu

W eksperymencie brało udział czworo tłumaczy. Dwoje z nich wykonywało zadania w ramach scenariusza A, a dwoje pozostałych w ramach scenariusza B. Zadaniem wszystkich tłumaczy było przetłumaczenie na język angielski polskich zdań ze zbioru testowego, przy użyciu systemu wspomagania tłumaczenia. Tłumacze pracowali wyłącznie na tych zdaniach, dla których system wygenerował podpowiedź tłumaczenia.

W scenariuszu A, tłumacze pracowali na pamięci tłumaczeń nieprzetworzonej. Natomiast w wersji B, tłumacze otrzymali specjalistyczną pamięć tłumaczeń, powstałą dzięki przetworzeniu odpowiedniego zbioru zdań. Wszyscy tłumacze pracowali niezależnie.

### 7.1.3. Testowe pamięci tłumaczeń i zbiór zdań wejściowych

Testowa pamięć tłumaczeń dla scenariusza A została utworzona na podstawie tłumaczeń polskich tekstów prawniczych na język angielski, nad którymi pracowali tłumacze. W skład tej pamięci weszły:

- Biuletyn na temat ustawodawstwa Unii Europejskiej
- Kodeks Karny
- Kodeks Postępowania Karnego
- Pamięć zebrana podczas pracy z Systemem Informacji Prawnej LEX ([Pol])
- Ustawa o prawie autorskim
- Ustawa o systemie oświaty

Oznaczmy tę pamięć przez  $M$ . Całkowita liczba przykładów w pamięci  $M$  wyniosła **20301**.

Do utworzenia zbioru testowego posłużyły 3 umowy, pochodzące z systemu LEX. Oznaczmy zbiór zdań testowych przez  $T_c$ . Moc zbioru  $T_c$  wyniosła **371**. Teksty te nie weszły w skład pamięci  $M$ , gdyż były dostępne wyłącznie w języku polskim.

Na potrzeby scenariusza B, przygotowano pamięć tłumaczeń  $M_{500}$ , składającą się z:

- Pamięci tłumaczeń  $M$ ,
- Specjalistycznej pamięci tłumaczeń umów.

Specjalistyczna pamięć tłumaczeń umów powstała w oparciu o zbiór tekstów polskich umów, pochodzący z systemu LEX. Zbiór ten nie zawierał zdań testowych  $T_c$ . Został on przetworzony zgodnie z ideą opisaną w Podrozdziale 5.3.2. Użyte zostały w tym celu reguły rozpoznawania jednostek nazwanych, dostosowane do tekstów prawniczych. Pełna lista tych reguł znajduje się w Dodatku B. W wyniku tego procesu powstała specjalistyczna pamięć tłumaczeń, licząca 464 przykłady.

Całkowita liczba przykładów pamięci  $M_{500}$  wyniosła: **20765**.

Tablica 7.1. Kompletność przeszukiwania w scenariuszach A i B

	Scenariusz A	Scenariusz B
Liczba zdań	379	
Niepustych wyników przeszukiwania	35	<b>118</b>
Kompletność	9.2%	<b>31.1%</b>

## 7.2. Kompletność przeszukiwania

W Tabeli 7.1 przedstawione są wyniki badania kompletności przeszukiwania pamięci tłumaczeń w scenariuszach A oraz B. Wyraźnie widoczna jest przewaga pamięci tłumaczeń  $M_{500}$ , używanej w scenariuszu B. Kompletność przeszukiwania jest w jej przypadku aż trzykrotnie wyższa.

## 7.3. Przydatność przetworzonej pamięci tłumaczeń

Po wykonaniu tłumaczeń przez wszystkich tłumaczy biorących udział w eksperymencie, została oszacowana ilość pracy, którą poświęcili na ten proces. Czas pracy nie jest miarodajnym wskaźnikiem jej ilości, gdyż różni tłumacze mają różne indywidualne możliwości. W związku z tym, opracowano miary szacowania ilości pracy tłumacza, które nie są oparte na czasie.

Pierwsza miara jest używana w sytuacji, gdy tłumacz tłumaczy zdanie, na podstawie podpowiedzi wygenerowanej przez system wspomagający. Druga miara jest stosowana, gdy sugestia nie jest dostępna i całe zdanie musi być przetłumaczone ręcznie od początku.

### 7.3.1. Miara ilości pracy - z sugestią

Miara ilości pracy, potrzebnej do przetłumaczenia zdania w sytuacji, gdy jest dostępna sugestia, jest obliczana poprzez porównanie sugestii z końcowym tłumaczeniem, wykonanym przez człowieka. Porównywanie polega na obliczeniu dystansu Levenshteina (opisanego w Podrozdziale 13) na poziomie słów. Operacjami podstawowymi są wtedy:

- Usunięcie słowa z sugestii
- Wstawienie słowa do sugestii
- Podstawienie słowa w sugestii

Taki sposób porównywania jest znany jako Word Error Rate (w skrócie WER, opisane w [Hun90]). Na potrzeby niniejszego eksperymentu, zmodyfikowano jednak tę miarę. Przed zaaplikowaniem WER, sugestia oraz tłumaczenie końcowe poddawane są następującym operacjom:

1. Poddaj każde słowo zdania operacji stemowania.
2. Usuń ze zdania słowa krótsze niż 3 znaki.
3. Usuń powtórzenia słów w zdaniu.

Dzięki stemowaniu, zmiana formy słowa nie będzie traktowana jako pracochłonna czynność. Jest to zgodne z intuicją tłumaczy, którzy przekonują, że najbardziej pracochłonne podczas tłumaczenia jest odnalezienie dobrego tłumaczenia słowa na język docelowy. Zmiana formy słowa jest czynnością techniczną, która nie nastęrcza trudności osobie dobrze władającej językiem docelowym. To samo dotyczy tłumaczenia krótkich słów, które najczęściej są zaimkami lub przyimkami, doskonale znanymi tłumaczowi. Poza tym, miara ta nie liczy pracy poświęconej na odnalezienie tłumaczenia tego samego słowa więcej niż jeden raz.

Rozważmy następujący przykład, zaczerpnięty z eksperymentu:

Sugestia: “the object of the partnership’s activity”

Tłumaczenie: “The objects of the Company shall be”

Po kroku 1:

Sugestia: “the object of the partnership activity”

Tłumaczenie: “the object of the company shall be”

Po kroku 2:

Sugestia: “the object the partnership activity”

Tłumaczenie: “the object the company shall”

Po kroku 3:

Sugestia: “the object partnership activity”

Tłumaczenie: “the object company shall”

Obliczona miara ilości pracy w tym przypadku wynosi **2**, z powodu dwóch podstawień słów.

### 7.3.2. Miara ilości pracy – bez sugestii

Kiedy nie jest dostępna sugestia tłumaczenia, stosowana jest inna miara ilości pracy, poświęconej na tłumaczenie. Miara ta jest oparta na liczbie unikatowych słów w zdaniu źródłowym. Jednakże, zdanie źródłowe jest poddawane modyfikacjom, podobnym do tych opisanych w Podrozdziale 7.3.1. Operacjami tymi są:

1. Usuń ze zdania słowa krótsze niż 3 znaki.



2. Usuń powtórzenia słów w zdaniu.

Stemowanie nie jest w tym przypadku konieczne.

Rozważmy następujący przykład (zaczerpnięty z Kodeksu Spółek Handlowych):

Zdanie źródłowe: “Jeżeli zbycie uzależnione jest od zgody spółki, stosuje się przepisy 3 - 5, chyba że umowa spółki stanowi inaczej.”

Po kroku 1:

Zdanie źródłowe: “Jeżeli zbycie uzależnione jest zgody spółki stosuje się przepisy chyba umowa spółki stanowi inaczej”

Po kroku 2:

Zdanie źródłowe: “Jeżeli zbycie uzależnione jest zgody spółki stosuje się przepisy chyba umowa stanowi inaczej”

W tym przypadku, miara ilości pracy wynosi **13**.

### 7.3.3. Wyniki badania przydatności

W Tabeli 7.2 przedstawione są wyniki badania przydatności przetworzonej pamięci tłumaczeń. Poniżej znajdują się objaśnienia wpisów w tabeli:

- $z$  – liczba zdań ze zbioru  $T_c$ , dla których system wspomagania tłumaczenia podał sugestie i które zostały przetłumaczone przez tłumacza,
- $W_z$  – obliczona ilość pracy nad tłumaczeniem zdań przy dostępnej sugestii tłumaczenia (na podstawie metody opisanej w Podrozdziale 7.3.1),
- $W_{avg}$  – średnia ilość pracy nad tłumaczeniem zdania ( $W_{avg} = \frac{W_z}{z}$ ),
- $avg(W_{avg})$  – średnia parametru  $W_{avg}$  dla obu tłumaczy,
- $W_c$  – obliczona całkowita ilość pracy nad tłumaczeniem, wliczając zdania, dla których nie była dostępna sugestia (obliczenie dla tych zdań odbyło się na podstawie metody opisanej w Podrozdziale 7.3.2),
- $avg(W_c)$  – średnia parametru  $W_c$  dla obu tłumaczy.

W Tabeli 7.3 przedstawione są statystyki pamięci tłumaczeń, wraz z wynikami uzyskanymi przez nie w eksperymencie.

## 7.4. Wnioski z eksperymentu

Wyniki eksperymentu, jak również komentarze uzyskane od biorących w nim udział tłumaczy, wskazują na znaczne zwiększenie jakości podpowiedzi tłumaczenia podczas stosowania pamięci  $M_{500}$ , w stosunku do pamięci  $M$ .

Tablica 7.2. Przydatność przetworzonej pamięci tłumaczeń

	Scenariusz A		Scenariusz B	
	Tłumacz 1	Tłumacz 2	Tłumacz 3	Tłumacz 4
$z$	35	35	118	118
$W_z$	210	214	362	343
$W_{avg}$	6.0	6.11	3.07	2.91
$avg(W_{avg})$	<b>6.06</b>		<b>2.99</b>	
$W_c$	3849	3853	3376	3357
$avg(W_c)$	<b>3851</b>		<b>3366.5</b>	

Tablica 7.3. Statystyki pamięci i wyniki eksperymentu

	$M$ (Scenariusz A)	$M_{500}$ (Scenariusz B)
Całkowita liczba przykładów	20301	20765 (+2.3%)
Kompletność przeszukiwania	9.2%	31.1% (+237.1%)
Całkowita ilość pracy	3851	3366.5 (-12.6%)
Średnia ilość pracy na zdanie	6.06	2.99 (-50.7%)

Średnia ilość pracy nad jednym zdaniem zmniejszyła się z wartości 6.06 do 2.99. Wartość ta odpowiada około dwukrotnemu zwiększeniu jakości odpowiedzi generowanych przez system wyposażony w pamięć  $M_{500}$ . Poza tym, znacząco zwiększyła się liczba zdań, dla których system był w stanie wygenerować odpowiedź. Wartość kompletności zwiększyła się ponad trzykrotnie (z 9.2% do 31.1%). Oba te czynniki doprowadziły do zmniejszenia całkowitej ilości pracy nad tłumaczeniem zdań ze zbioru testowego  $T_c$ . Miara tej pracy spadła z poziomu 3851 do 3366.5. Wszystkie te usprawnienia zostały uzyskane poprzez zwiększenie pamięci  $M$  o zaledwie 2.3%.

Eksperyment potwierdził zasadność stosowania autorskiej metody przygotowywania specjalistycznej pamięci tłumaczeń na potrzeby tłumaczenia tekstów z konkretnej dziedziny.

## Podsumowanie

W niniejszej pracy zaprezentowane zostały autorskie algorytmy przeszukiwania i przetwarzania pamięci tłumaczeń. Podane zostały dowody twierdzeń, mówiących o ich złożoności obliczeniowej. We wszystkich przypadkach złożoność ta była korzystniejsza, niż w znanych do tej pory rozwiązaniach.

Innym ważnym wynikiem teoretycznym pracy jest twierdzenie mówiące o wpływie funkcji dystansu zdań, użytej w algorytmie przeszukiwania pamięci tłumaczeń, na własności tego algorytmu. Według niego, własność przykładu doskonałego oraz ograniczenia oceny dopasowania jest zachowana wtedy i tylko wtedy, gdy funkcja dystansu zdań zachowuje własności nieujemności oraz zwrotności.

Dobre własności teoretyczne algorytmów znalazły potwierdzenie w eksperymentach praktycznych. Algorytm przeszukiwania pamięci tłumaczeń uzyskał podobną kompletność i dokładność przeszukiwania w porównaniu z powszechnie stosowanym algorytmem memoQ. Szybkość autorskiego algorytmu była jednak kilkukrotnie wyższa zarówno od memoQ, jak i od innego testowanego algorytmu: Lucene.

Wyniki ewaluacji algorytmu przetwarzania pamięci tłumaczeń potwierdziły zasadność jego używania. Procedura tworzenia specjalistycznej pamięci tłumaczeń, której sercem jest ten algorytm, prowadzi do utworzenia wartościowego zasobu danych. Testy potwierdziły, iż specjalistyczna pamięć przyczynia się do zwiększenia kompletności i dokładności przeszukiwania w systemie wspomagania tłumaczenia.

Opracowane w ramach niniejszej pracy algorytmy są gotowe do zastosowania w nowoczesnych systemach wspomagania tłumaczenia. Co więcej, mogą być wykorzystane jako podzespoły systemu tłumaczenia automatycznego na podstawie przykładu. Oprócz tego, idee opracowane na potrzeby tych algorytmów, takie jak przedstawiona wersja kodowanej tablicy sufiksów, funkcji podobieństwa zdań w języku naturalnym oraz dwustopniowego algorytmu

analizy skupień, mogą posłużyć jako inspiracje do tworzenia nowych algorytmów w różnych dziedzinach informatyki.

## Bibliografia

- [aut] Multiple authors. The JRC-Acquis Multilingual Parallel Corpus. <http://langtech.jrc.it/JRC-Acquis.html>.
- [Bir] Steven Bird. Natural language toolkit. <https://sites.google.com/site/naturallanguagetoolkit>.
- [BPPM93] Peter Brown, Vincent Della Pietra, Stephen Della Pietra, and Robert Mercer. The mathematics of statistical machine translation: parameter estimation. *Journal Computational Linguistics - Special issue on using large corpora: II archive Volume: 19 Issue: 2*, pp. 263-311, 1993.
- [CGSSO04] Olivia Craciunescu, Constanza Gerding-Salas, and Susan Stringer-O’Keeffe. Machine translation and computer-assisted translation: a new way of translating? *The Translation Journal Volume: 8 Issue: 3*, 2004.
- [Con11] The Unicode Consortium. The unicode standard, version 6.0. <http://www.unicode.org/versions/Unicode6.0.0/ch03.pdf>, 2011.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2 edition, 2001.
- [CT10] Maxime Crochemore and German Tischler. The gapped suffix array: A new index structure for fast approximate matching. *Proceedings of the String Processing and Information Retrieval Conference, 2010*, 2010.
- [DD09] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009.
- [DFG+97] Gautam Das, Rudolf Fleischer, Leszek Gasieniec, Dimitris Gunopulos, and Juha Karkkainen. Episode matching. *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM ’97)*. LNCS, vol. 1264, Springer-Verlag, Berlin, pp. 12–27., 1997.
- [GC91] William Gale and Kenneth Church. A program for aligning sentences in bilingual corpora. *ACL ’91 Proceedings of the 29th an-*

- nual meeting on Association for Computational Linguistics*, pp. 177-184, 1991.
- [Gho06] Mohammadreza Ghodsi. Approximate string matching using backtracking over suffix arrays, 2006.
- [Goo] Google. Tłumacz google - informacje. <http://translate.google.pl/about>.
- [GW03] Nano Gough and Andy Way. Controlled generation in example-based machine translation. *Proceedings of the Ninth Machine Translation Summit (MT Summit IX)* pp. 133 - 140, 2003.
- [Ham50] Richard Hamming. Error detecting and error correcting codes. *Bell System Technical Journal* 29 (2), pp. 147-160, 1950.
- [HkHwLkS04] Trinh N. D. Huynh, Wing kai Hon, Tak wah Lam, and Wing kin Sung. Approximate string matching using compressed suffix arrays. *Proceedings of Symposium on Combinatorial Pattern Matching*, 2004.
- [HKY99] Laurie Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring expression data: Identification and analysis of coexpressed genes. *Genome Research* 9, pp. 1106-1115, 1999.
- [Hue10] Juan Huerta. A stack decoder approach to approximate string matching. *Proceedings of the SIGIR conference*, 2010.
- [Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, pp. 1098-1102, 1952.
- [Hun90] Melvyn Hunt. Figures of merit for assessing connected word recognisers. *Speech Communication* 9, pp. 239-336, 1990.
- [Hut95] John Hutchins. Machine translation: a brief history. *Concise history of the language sciences: from the Sumerians to the cognitivists*, pp. 431-445, 1995.
- [Hut07] John Hutchins. Machine translation: a concise history. *Computer aided translation: Theory and practice*, ed. Chan Sin Wai. Chinese University of Hong Kong, 2007.
- [Kar73] Michał Karoński. On a definition of cluster and pseudocluster for multivariate normal population. *Bull. of the ISI. Proc. of the 39th Session*, pp. 590-598, 1973.
- [Ken38] Maurice Kendall. A new measure of rank correlation. *Biometrika* 30, pp. 81-89, 1938.
- [KHB<sup>+</sup>07] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. *Annual*

- Meeting of the Association for Computational Linguistics (ACL), demonstration session, Prague, Czech Republic, 2007.*
- [KS10] Philipp Koehn and Jean Senellart. Fast approximate string matching with suffix arrays and a\* parsing. *Proceedings of AMTA 2010: The Ninth Conference of the Association for Machine Translation in the Americas*, 2010.
- [KWGS09] Mirosław Krzyśko, Waldemar Wołyński, Tomasz Górecki, and Michał Skorzybut. *Systemy uczące się*. Wydawnictwo Naukowo Techniczne, 2009.
- [Lev65] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845-848, 1965.
- [Lin11] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Publishers, 2011.
- [Lip07] Jarosław Lipski. Urównoleganie tekstów dwujęzycznych na poziomie zdania. *Praca magisterska, napisana pod kierunkiem dra Krzysztofa Jassema*, 2007.
- [LIS04] The Localisation Industry Standards Association LISA. Srx 1.0 specification. <http://www.ttt.org/oscarstandards/srx/srx10.html>, 2004.
- [luc] Apache lucene. <http://lucene.apache.org/>.
- [MKSW99] John Makhoul, Francis Kubala, Richard Schwartz, and Ralph Weischedel. Performance measures for information extraction. *Proceedings of DARPA Broadcast News Workshop*, pp. 249-252, 1999.
- [MNU00] Andrew McCallum, Kamal Nigam, and Lyle Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. *KDD'00 Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 169-178, 2000.
- [Moo02] Robert Moore. Fast and accurate sentence alignment of bilingual corpora. *Proceedings of the 5th Conference of the Association for Machine Translation in the Americas*, pp. 135-144, 2002.
- [Mue06] Uwe Muegge. An excellent application for crummy machine translation: Automatic translation of a large database. *Proceedings of the Annual Conference of the German Society of Technical Communicators in Stuttgart*, pp. 18-21, 2006.
- [mula] multiple. Kilgray translation technologies: memoq translator pro. <http://kilgray.com/products/memoq/>.
- [mulb] multiple. Wordfast community: Very large translation memory project. <http://www.wordfast.com/>.
- [Nag84] Makoto Nagao. A framework of a mechanical transla-

- tion between japanese and english by analogy principle. <http://www.mt-archive.info/Nagao-1984.pdf>, 1984.
- [Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR) Volume 33 Issue 1, March 2001*, 2001.
- [NByST00] Gonzalo Navarro, Ricardo Baeza-yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001, 2000.
- [NW70a] Saul Needleman and Christian Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48 (3), pp. 443–453, 1970.
- [NW70b] Saul Needleman and Christian Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Mol. Biol.* 48, pp. 444–453, 1970.
- [Pol] Wolters Kluwer Polska. System informacji prawnej lex. <http://www.wolterskluwer.pl/czytaj/-/artykul/marka-lex>.
- [RT99] Calyampudi Rao and Helge Toutenburg. *Linear Models: Least Squares and Alternatives*. Springer Series in Statistics, 1999.
- [Sel80] Peter Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms* 1 (4): pp. 359–73, 1980.
- [Som99] Harold Somers. Review article: Example-based machine translation. *Machine Translation* 14, pp. 113–158, 1999.
- [SP05] Geoffrey Sampson and Paul Martin Postal. *The 'language instinct' debate*. Continuum International Publishing Group, 2005.
- [Spe04] Charles Spearman. The proof and measurement of association between two things. *Americal Journal of Psychology* 15 (1904), pp. 72–101, 1904.
- [Sti89] Stephen M. Stigler. Francis galton's account of the invention of correlation. *Statistical Science*, 1989.
- [Thu27] Louis Thurstone. A law of comparative judgement. *Psychological Review* 34, pp. 273–286, 1927.
- [TI06] Dan Tufiş and Elena Irimia. Roco-news: A hand validated journalistic corpus of romanian, 2006.
- [Twi06] Greg Twiss. A comparative study of cat tools (maht workbenches) with translation memory components. *proz.com The translator workspace*, 2006.
- [Val05] Fotini Vallianatou. Cat tools and productivity: Tracking words and hours. *The Translation Journal Volume: 9 Issue: 4*, 2005.
- [VNH<sup>+</sup>05] Dániel Varga, László Németh, Péter Halácsy, András Kornai, Vik-



- tor Trón, and Viktor Nagy. Parallel corpora for medium density languages. *Proceedings of the RANLP 2005*, pp. 590-596, 2005.
- [WS99] Edward Whyman and Harold Somers. Evaluation metrics for a translation memory system. *Software - Practice and Experience*, 29(14), pp. 1265-1284, 1999.

# Spis tablic

1.1.	Przykładowe stemy słów . . . . .	8
3.1.	Ilości pracy w analizie produktywności . . . . .	32
3.2.	Produktywność tłumaczenia dla różnych narzędzi CAT . . . . .	32
4.1.	Wypełniona tablica w algorytmie Sellersa. . . . .	41
4.2.	Algorytm Sellersa: rozwiązanie pierwsze . . . . .	41
4.3.	Algorytm Sellersa: rozwiązanie drugie . . . . .	41
4.4.	Przykładowa tablica sufiksów . . . . .	43
4.5.	Posortowana tablica sufiksów . . . . .	44
5.1.	Wartości kar za różnice pomiędzy zdaniem $z_1$ i $z_2$ . . . . .	80
6.1.	Statystyki testowej pamięci tłumaczeń . . . . .	87
6.2.	Kompletność przeszukiwania pamięci tłumaczeń . . . . .	88
6.3.	Statystyki przeszukiwania . . . . .	89
6.4.	Porównawcza dokładność przeszukiwania pamięci tłumaczeń . . . . .	94
6.5.	Zgodność tłumaczy podczas oznaczania . . . . .	94
6.6.	Parametry techniczne maszyny testowej . . . . .	95
6.7.	Szybkość wyszukiwania algorytmu autorskiego i memoQ . . . . .	96
6.8.	Szybkość wyszukiwania - uruchomienia procedury testowej . . . . .	98
6.9.	Szybkość wyszukiwania - wyniki testów . . . . .	98
7.1.	Kompletność przeszukiwania w scenariuszach A i B . . . . .	101
7.2.	Przydatność przetworzonej pamięci tłumaczeń . . . . .	104
7.3.	Statystyki pamięci i wyniki eksperymentu . . . . .	104

## Spis rysunków

2.1.	Proste drzewo parsingu . . . . .	20
2.2.	Drzewo parsingu zdania w języku chińskim . . . . .	21
2.3.	Przykładowy algorytm EBMT . . . . .	24
4.1.	Naiwny algorytm wyszukiwania przybliżonego . . . . .	39
4.2.	Wypełnienie tablicy w algorytmie Sellersa . . . . .	42
4.3.	Algorytm generowania skrótu zdania . . . . .	46
4.4.	Algorytm dodawania zdania do indeksu . . . . .	47
4.5.	Funkcja <code>getLongestCommonPrefixes</code> . . . . .	48
4.6.	Obiekt <code>OverlayMatch</code> . . . . .	48
4.7.	Funkcja <code>search</code> . . . . .	49
4.8.	Algorytm Koehna i Senellarta - część pierwsza . . . . .	60
4.9.	Algorytm Koehna i Senellarta - część druga . . . . .	62
5.1.	Prosta reguła podziału SRX . . . . .	66
5.2.	Prosty wyjątek SRX . . . . .	66
5.3.	Reguła SRX, obsługująca wypunktowanie . . . . .	66
5.4.	Algorytm podziału tekstu na zdania . . . . .	67
5.5.	Zestaw reguły SRX i wyjątku od niej . . . . .	68
5.6.	Korelacja długości paragrafów tekstów w dwóch językach . . . . .	70
5.7.	Aglomeracyjny algorytm analizy skupień . . . . .	74
5.8.	Algorytm K-średnich . . . . .	75
5.9.	Algorytm Quality Threshold . . . . .	77
5.10.	Algorytm naliczania kar dla zdań . . . . .	81
5.11.	Autorski algorytm analizy skupień zdań . . . . .	81
6.1.	Faza przeszukiwania . . . . .	87
6.2.	Korelacja ocen dopasowania . . . . .	90
6.3.	Korelacja rang ocen dopasowania . . . . .	92
6.4.	Procedura testowa algorytmu Lucene . . . . .	97
6.5.	Procedura testowa algorytmu autorskiego . . . . .	97

## Dodatek A

# Wykaz publikacji z udziałem autora rozprawy

### A.1. Opublikowane

1. Rafał Jaworski: "Computing transfer score in Example-Based Machine Translation", *Lecture Notes in Computer Science 6008 (LNCS)*, Springer-Verlag, pp. 406-416, 2010
2. Rafał Jaworski, Krzysztof Jassem: "Building high quality translation memories acquired from monolingual corpora", *Proceedings of the IIS 2010 Conference*, pp. 157-168, 2010
3. Rafał Jaworski: "A sentence Clustering Algorithm for Specialized Translation Memories", *Speech and Language Technology (SLT) vol. 12/13*, pp. 97-103, 2011

### A.2. Przyjęte do druku

1. Rafał Jaworski: "Anubis - speeding up Computer-Aided Translation", *Computational Linguistics – Applications*, Springer-Verlag, 2012

## Dodatek B

# Reguły rozpoznawania jednostek nazwanych

W niniejszym dodatku przedstawione są reguły rozpoznawania jednostek nazwanych, oparte na wyrażeniach regularnych. Poniższy zestaw reguł został opracowany na potrzeby rozpoznawania jednostek nazwanych w tekstach prawniczych.

### B.1. Budowa reguły rozpoznawania

Reguła rozpoznawania jednostek nazwanych składa się z następujących elementów:

- **Language** - język tekstu, w którym ma być wyszukiwana jednostka;
- **Match** - wyrażenie regularne, pozwalające odszukać jednostkę;
- **Class** - typ jednostki nazwanej.

Przykładowa reguła rozpoznawania daty w tekście polskim jest następująca:

```
Language: pl
Match: ([0-9]{1,2})[\.\-/]([0-9]{1,2})[\.\-/]([0-9]{4})
Class: Date
```

### B.2. Lista reguł

Poniżej znajduje się pełna lista reguł, opracowanych przez autora rozprawy na potrzeby przetwarzania tekstów prawniczych.

```
# -- macro reprezentujące duże litery
UL=[A-ZĄŻŹŚĘĆŃÓŁ]

# -- macro reprezentujące małe litery
LL=[a-zążźśęćńół]
```

```
# -- macro tokenu pisanego z dużej litery
TUCF=({UL}[{UL}{LL}\-\.\}*)

NUM=([0-9]+)

TAG=(a|abbr|acronym|address|applet|area|b|base|basefont|
bdo|big|blockquote|body|br|button|caption|center|cite|
code|col|colgroup|dd|del|dir|div|dfn|dl|dt|em|fieldset|
font|form|frame|frameset|h[1-6]|head|hr|html|i|iframe|
img|input|ins|isindex|kbd|label|legend|li|link|map|menu|
meta|noframes|noscript|object|ol|optgroup|option|p|param|
pre|q|s|samp|script|select|small|span|strike|strong|
style|sub|sup|table|tbody|td|textarea|tfoot|th|thead|
title|tr|tt|u|ul|var|xmp)

#SINGLEORDPL=pierwszego|drugiego|trzeciego|czwartego|
piątego|szóstego|siódmego|ósmego|dziewiątego

#ORDPL=({SINGLEORDPL}|jedenastego|dwunastego|
trzynastego|czternastego|piętnastego|szesnastego|
siedemnastego|osiemnastego|dziewiętnastego|
dwudziestego|)

#SINGLECARDPL=(pierwszy|drugi|trzeci|czwarty|piąty|
szósty|siódmy|ósmo|dziewiąty)

MONTHCOMPL=(stycznia|lutego|marca|kwietnia|maja|czerwca|
lipca|sierpnia|września|października|listopada|grudnia)

# ----- FirstName -----

Language: *
Match: (@FIRST_NAME.*?@)
Class: HNERTFirstName

# ----- LastName -----
```

Language: \*

Match: (@LAST\_NAME.\*?@)

Class: HNERTLastName

# ----- StreetNum -----

Language: \*

Match: (@STREET\_NUM.\*?@)

Class: HNERTStreetNum

# ----- Street -----

Language: \*

Match: (@STREET.\*?@)

Class: HNERTStreet

# ----- City -----

Language: \*

Match: (@CITY.\*?@)

Class: HNERTCity

# ----- FormattingTag -----

Language: \*

Match: ((<|&lt;)/?{TAG}.\*?(>|&gt;))

Class: FormattingTag

# ----- JOLFull -----

Language: pl

Match: Dz\\.\\s\*U\\.\\s+z\\s+([0-9]{1,2})\\.([0-9]{1,2})

\\.([0-9]{4})\\s+nr\\s+{NUM},\\s\*poz\\.\\s\*{NUM}

Class: JOLFull

Language: pl

Match: Dz\\.\\s\*U\\.\\s+L\\s+{NUM}\\s+z\\s+([0-9]{1,2})

\\.([0-9]{1,2})\\.([0-9]{4})\\s\*,\\s+str\\.\\s\*{NUM}

Class: JOLFull

Language: en

Match: (Journal\s+of\s+Laws|OL|JOL)\s+of\s+([0-9]{4})  
/([0-9]{1,2})/([0-9]{1,2})\s+No\.\?\s\*{NUM},\s\*item\s\*{NUM}

Class: JOLFull

Language: en

Match: OJ\s+L\s+{NUM},\s+([0-9]{1,2})\.\.([0-9]{1,2})  
\.\.([0-9]{4}),\s+p\.\.\s+{NUM}

Class: JOLFull

Language: de

Match: Art\.\.\s+{NUM}\s+Abs\.\.\s+{NUM}\s+GG\s+vom  
\s+([0-9]{1,2})\.\.([0-9]{1,2})\.\.([0-9]{4})

Class: JOLFull

Language: de

Match: OJ\s+L\s+{NUM},\s+([0-9]{1,2})\.\.([0-9]{1,2})  
\.\.([0-9]{4}),\s+p\.\.\s+{NUM}

Class: JOLFull

Language: es

Match: DO\s+([0-9]{1,2})\.\.([0-9]{1,2})\.\.([0-9]{4})\s+L  
\s+{NUM}\s+p\.\.\s+{NUM}

Class: JOLFull

# ----- JOLYearOnly -----

Language: pl

Match: Dz\.\.\s\*U.\s+z\s+{NUM}\s\*(r\.\.|roku)\s+nr\s+{NUM},  
\s\*poz\.\.\s\*{NUM}

Class: JOLYearOnly

Language: en

Match: (Journal\s+of\s+Laws|OL|JOL)\s+of\s+([0-9]{4})  
\s+No\.\?\s\*{NUM},\s\*item\s\*{NUM}

Class: JOLYearOnly



```
Language: de
Match: Art\.\s+{NUM}\s+Abs\.\s+{NUM}\s+GG\s+vom
\s+([0-9]{4})
Class: JOLYearOnly

Language: es
Match: DO\s+([0-9]{4})\s+L\s+{NUM}\s+p\.\s+{NUM}
Class: JOLYearOnly

# ----- JOLWithLaterChanges -----

Language: pl
Match: Dz\.\s*U.\s*nr\s+{NUM},\s*poz\.\s*{NUM}\s*,
\s+z\s+późn\.\s*zm\
Class: JOLWithLaterChanges

Language: en
Match: (Journal\s+of\s+Laws|JOL|OL)\s+No\.\s*{NUM},
\s*item\s*{NUM},\s+with\s+later\s+changes
Class: JOLWithLaterChanges

Language: de
Match: Art\.\s*{NUM},\s*Abs\.\s*{NUM}\s+GG,\s+mit
\s+späteren\s+Korrekturen
Class: JOLWithLaterChanges

Language: es
Match: DO\s+L\s+{NUM}\s+p\.\s+{NUM},
\s*con\s+correcciones
Class: JOLWithLaterChanges

# ----- JOL -----

Language: pl
Match: Dz\.\s*U.\s*nr\s+{NUM},\s*poz\.\s*{NUM}
Class: JOL
```

Language: en  
 Match: (Journal\s+of\s+Laws|JOL|OL)\s+No\.\?  
 \s\*{NUM},\s\*item\s\*{NUM}  
 Class: JOL

Language: de  
 Match: Art\.\s+{NUM}\s+Abs\.\s+{NUM}\s+GG  
 Class: JOL

Language: es  
 Match: DO\s+L\s+{NUM}\s+p\.\s+{NUM}  
 Class: JOL

# ----- DateNoItem -----

Language: pl  
 Match: z\s+([0-9]{1,2})\.\s+([0-9]{1,2})\.\s+([0-9]{4})\s+  
 nr\s+{NUM},\s\*poz\.\s\*{NUM}  
 Class: DateNoItem

Language: en  
 Vector: \3\2\1\4\5  
 Class: DateNoItem

Language: de  
 Match: vom\s+([0-9]{1,2})\.\s+([0-9]{1,2})\.\s+([0-9]{4})\s+Art  
 \.\s\*{NUM},\s\*Abs\.\s\*{NUM}  
 Class: DateNoItem

Language: es  
 Match: de\s+([0-9]{1,2})\.\s+([0-9]{1,2})\.\s+([0-9]{4})  
 \s+Artículo\s\*{NUM},\s\*apartado\s\*{NUM}  
 Class: DateNoItem

# ----- YearNoItem -----

Language: pl  
 Match: z\s+([0-9]{4})\s\*(r\.|roku)\s+nr

```
\s+{NUM}, \s*poz\. \s*{NUM}
```

```
Class: YearNoItem
```

```
Language: en
```

```
Match: of\s+([0-9]{4})\s+No\.\?\s*{NUM}, \s*item\s*{NUM}
```

```
Class: YearNoItem
```

```
Language: de
```

```
Match: vom\s+([0-9]{4})\s+Art\.\?\s*{NUM}, \s*Abs\.\s*{NUM}
```

```
Class: YearNoItem
```

```
Language: es
```

```
Match: de\s+([0-9]{4})\s+Artículo\s*{NUM},  
\s*apartado\s*{NUM}
```

```
Class: YearNoItem
```

```
# ----- NoItem -----
```

```
Language: pl
```

```
Match: nr\s+{NUM}, \s*poz\. \s*{NUM}
```

```
Class: NoItem
```

```
Language: en
```

```
Match: No\.\?\s*{NUM}, \s*item\s*{NUM}
```

```
Class: NoItem
```

```
Language: de
```

```
Match: Art\.\?\s*{NUM}, \s*Abs\.\s*{NUM}
```

```
Class: NoItem
```

```
Language: es
```

```
Match: Artículo\s*{NUM}, \s*apartado\s*{NUM}
```

```
Class: NoItem
```

```
# ----- Date -----
```

```
Language: pl
```

```
Match: ([0-9]{1,2})[\.\-/]([0-9]{1,2})[\.\-/]([0-9]{4})
```

Class: Date

Language: pl

Match: ([0-9]{1,2})\s\*{MONTHCOMPL}  
\s\*([0-9]{4})\s\*roku

Class: Date

Language: en

Match: ([0-9]{4})[\.\-/]([0-9]{1,2})[\.\-/  
([0-9]{1,2})

Class: Date

Language: de

Match: ([0-9]{1,2})[\.\-/]([0-9]{1,2})  
[\.\-/]([0-9]{4})

Class: Date

Language: es

Match: ([0-9]{1,2})[\.\-/]([0-9]{1,2})  
[\.\-/]([0-9]{4})

Class: Date

# ----- CompanyLtd -----

Language: pl

Match: {TUCF}\s+Sp\.\s+z\s+o\.\s\*o\.

Class: CompanyLtd

Language: en

Match: {TUCF}\s+[Ll]td\.

Class: CompanyLtd

Language: de

Match: {TUCF}\s+GmbH

Class: CompanyLtd

Language: es

Match: {TUCF}\s+S\.R\.L\.

Class: CompanyLtd

# ----- CompanyPlc -----

Language: pl

Match: {TUCF}\s+S\.\s?A\.

Class: CompanyPlc

Language: en

Match: {TUCF}\s+[pP]\.\?[lL]\.\?[cC]\.

Class: CompanyPlc

Language: de

Match: {TUCF}\s+AG

Class: CompanyPlc

Language: es

Match: {TUCF}\s+S\.\s?A\.

Class: CompanyPlc

# ----- Email -----

Language: \*

Match: ([\w\.\_\d]+@\w+(\.\w+)\*)

Class: Email

# ----- Paragraph -----

Language: pl

Match: §\s\*{NUM}\s+ust\.\s\*{NUM}\s+pkt\s\*([a-z])(\)\.)

Class: Paragraph

Language: en

Match: §\s\*{NUM},\s\*point\s\*{NUM},\s\*sub\-point  
\s\*([a-z])(\)\.)

Class: Paragraph

Language: de

Match: §\s\*{NUM},\s\*Abs\.\s\*{NUM},\s\*S\.

\s\*([a-z])(\)\.)

Class: Paragraph

Language: es

Match: §\s\*{NUM},\s\*apartado\s\*{NUM},\s\*p\.

\s\*([a-z])(\)\.)

Class: Paragraph

# ----- ParagraphPoint -----

Language: pl

Match: ust\.\s\*{NUM}\s+pkt\s\*([a-z])(\)\.)

Class: ParagraphPoint

Language: en

Match: point\s\*{NUM},\s\*sub\ -point\s\*([a-z])(\)\.)

Class: ParagraphPoint

Language: de

Match: Abs\.\s\*{NUM},\s\*S\.\s\*([a-z])(\)\.)

Class: ParagraphPoint

Language: es

Match: apartado\s\*{NUM},\s\*p\.\s\*([a-z])(\)\.)

Class: ParagraphPoint

# ----- ParagraphSubPoint -----

Language: pl

Match: pkt\s\*([a-z])(\)\.)

Class: ParagraphSubPoint

Language: en

Match: sub\ -point\s\*([a-z])(\)\.)

Class: ParagraphSubPoint

Language: de

Match: S\.\s\*([a-z])(\)\.)

Class: ParagraphSubPoint

Language: es

Match: p\.\s\*([a-z])(\)\.)

Class: ParagraphSubPoint

# ----- ArtMultipleSec -----

Language: pl

Match: art\.\s\*([0-9]+[a-z]?)\s+ust\.\s\*{NUM}  
\s+i\s+{NUM}

Class: ArtMultipleSec

Language: en

Match: art\.\s\*([0-9]+[a-z]?)\s+sec\.\s\*{NUM}  
\s+and\s+{NUM}

Class: ArtMultipleSec

Language: de

Match: Art\.\s\*([0-9]+[a-z]?)\s+Abs\.\s\*{NUM}  
\s+und\s+{NUM}

Class: ArtMultipleSec

Language: es

Match: Artículo\s\*([0-9]+[a-z]?)\s+apartado  
\s\*{NUM}\s+y\s+{NUM}

Class: ArtMultipleSec

# ----- ArtSec -----

Language: pl

Match: art\.\s\*([0-9]+[a-z]?)\s+ust\.\s\*{NUM}

Class: ArtSec

Language: en

Match: art\.\s\*([0-9]+[a-z]?)\s+sec\.\s\*{NUM}

Class: ArtSec

Language: de  
Match: Art\.\s\*([0-9]+[a-z]?)\s+Abs\.\s\*{NUM}  
Class: ArtSec

Language: es  
Match: Artículo\s\*([0-9]+[a-z]?)\s+apartado\s\*{NUM}  
Class: ArtSec

# ----- Art -----

Language: pl  
Match: art\.\s\*([0-9]+[a-z]?)  
Class: Art

Language: en  
Match: art\.\s\*([0-9]+[a-z]?)  
Class: Art

Language: de  
Match: Art\.\s\*([0-9]+[a-z]?)  
Class: Art

Language: es  
Match: Artículo\s\*([0-9]+[a-z]?)  
Class: Art

# ----- SecPoint -----

Language: pl  
Match: ust\.\s\*{NUM}\s+pkt\.\s\*{NUM}  
Class: SecPoint

Language: en  
Match: sec\.\s\*{NUM}\s+item\s\*{NUM}  
Class: SecPoint

Language: de



Match: Abs\.\s\*{NUM}\s+S\.\s\*{NUM}

Class: SecPoint

Language: es

Match: apartado\s\*{NUM}\s+p\.\s\*{NUM}

Class: SecPoint

# ----- Sec -----

Language: pl

Match: ust\.\s\*{NUM}

Class: Sec

Language: en

Match: sec\.\s\*{NUM}

Class: Sec

Language: de

Match: Abs\.\s\*{NUM}

Class: Sec

Language: es

Match: apartado\s\*{NUM}

Class: Sec

# ----- HtmlEntity -----

Language: \*

Match: (&\d+)

Class: HtmlEntity

# ----- Punctor -----

Language: \*

Match: (\b(\d|\w)(\)|\.) )

Class: Punctor

# ----- MathNumber -----

Language: \*

Match: ({NUM}([\.\,]{NUM})?)

Class: MathNumber