AUTOMATYKA • 2008 • Tom 12 • Zeszyt 1

Krzysztof Dorosz*

Automatyczne sprawdzanie poprawności pisowni w języku polskim oparte na odległości Levenshteina

1. Wprowadzenie

1.1. Potrzeba sprawdzania poprawności pisowni tekstu

Problem sprawdzania pisowni wyrazów dla danego języka (a w szczególności dla języka polskiego) jest problemem nietrywialnym, biorąc pod uwagę jego usadowienie w całym procesie automatycznej oceny i analizy tekstu.

Internet stał się zdecydowanie najobszerniejszym kanałem dostępu do olbrzymiej liczby tekstów. Przy tak ogromnej ilości informacji do przetworzenia stale pojawia się pokusa do zautomatyzowania procesu pozyskiwania i analizowania tych informacji, których jednym z nośników jest właśnie tekst. Niestety łatwy, powszechny i tani dostęp do Internetu znacząco wpłynął na samą jakość tekstów. Ludzie przestali przykładać dużą wagę zarówno do poziomu merytorycznego, jak i poprawności językowej, w przeciwieństwie do czasów, gdy opublikowanie tekstu wymagało ogromnych nakładów finansowych[1)]. Szczególnie wyraźnie jest to zauważalne tam, gdzie tekst nie stanowi dłuższej wypowiedzi, a jest tylko krótką notką do przekazania informacji w sposób maksymalnie szybki (np. komunikatory internetowe, komentarze do artykułów w portalach WWW, blogi itp.).

Algorytmy lingwistyczne przetwarzające teksty są obecnie zupełnie nieodporne na występującą niechlujność językową. Tam gdzie człowiek podczas czytania nawet nie zwróci uwagi na fakt, iż wyraz napisano bez użycia polskich znaków diaktrycznych, tam automat rozpozna zupełnie inne (być może nieznane) słowo. Sytuacja ta ma miejsce także w przypadku wszelkich innych błędów, jak błędy ortograficzne, błędy czeskie itp.

Z tego powodu przed rozpoczęciem analizy danego tekstu algorytmem lingwistycznym należy sprawdzić, czy każde jego słowo znajduje się w podręcznym słowniku fleksyjnym języka. Jeśli dane słowo nie znajduje się w słowniku, to niestety jednoznacznie nie wynika z tego, że jest to niepoprawny wyraz. Trzeba rozważyć możliwość, że jest to poprawne słowo, a jedynie słownik fleksyjny jest niepełny, że jest to neologizm, wyrażenie

* Katedra Informatyki, Akademia Górniczo-Hutnicza w Krakowie

1) Na przykład przepisywanie książek ręcznie przez zakonników, składanie tekstów ręcznie przez zecera z czcionek itp.

branżowe (żargon), nazwa własna, lub ostatecznie – że jest to niepoprawnie napisany wyraz. Część wcześniej wymienionych możliwości można wykryć i obsłużyć z użyciem stemera, natomiast jedynie ostatnią możliwość obsługuje się mechanizmami spellcheckingu. Wzajemne użycie stemera i spellcheckera jest zagadnieniem bardzo skomplikowanym, wymaga doboru odpowiedniej strategii i nie zostało opisane w niniejszym artykule.

Istnieje także cała klasa przypadków, gdzie metody sprawdzania pisowni używane są do wsparcia osoby piszącej w trakcie pisania tekstu (np. podkreślając błędnie wprowadzane napisy). Są to tak zwane spellcheckery typu „check as I write" i stanowią trochę odmienne podejście do zagadnienia oceny tekstu. Między innymi chodzi o to, że osoba zauważając zaznaczone poprawne słowo, które automat sklasyfikował jako błędny napis, ma możliwość dodania go do słownika. Jest to typowy mechanizm ręcznego uczenia mechanizmu.

1.2. Potrzeba stosowania technik specjalizowanych dla języka

Do niedawna większość stosowanych metod sprawdzania poprawności językowych zostało wymyślonych z przeznaczeniem do języka angielskiego. Metody te działają także w przypadku zastosowania do innych języków, jednakże nie są optymalne w doborze podpowiedzi w przypadku błędnych napisów. Wynika to z faktu, iż albo są to na tyle ogólne metody, że nie uwzględniają bardzo ważnych cech charakterystycznych języka, albo są one nastawione na konkretne cechy języka angielskiego, które w większości innych przypadków nie mają zastosowania bądź też wprowadzają pogorszenie jakości w ocenie napisów nieanglojęzycznych.

Język polski ma wiele charakterystycznych cech, które z powodzeniem można wykorzystać w poprawie jakości algorytmów spellcheckingu. Charakterystyki te i metody ich wykorzystania zostaną opisane dalszym ciągu.

2. Odległość edycyjna (*Levenshtein Distance*)

Najbardziej popularną metodą stosowaną w szeroko rozumianych algorytmach spellcheckingu jest algorytm wyznaczania odległości edycyjnej pomiędzy dwoma napisami. Jest to metryka będąca miarą odmienności napisów (a dokładniej skończonych ciągów znaków z dowolnego alfabetu), którą w 1965 roku zaproponował rosyjski badacz teorii informacji Vladimir Iosifovich Levenshtein. W metryce tej zdefiniowane są następujące działania proste:

- wstawienie nowego znaku do napisu,
- usunięcie znaku z napisu,
- zamianę znaku w napisie na inny znak.

Całkowitoliczbowa wartość w tej metryce oznacza liczbę korekt z użyciem działań prostych, które należy przeprowadzić, aby z jednego napisu uzyskać napis drugi.

Załóżmy, że w metryce zdefiniujemy odległość jednego działania prostego jako 1, wtedy dla wyrazów: Tomek, Romek, Atomek, można zauważyć, że:

- wyrazy **Tomek** i **Romek** są od siebie odległe (w sensie odległości edycyjnej) o 1, ponieważ jedno działanie proste zamiany pierwszego znaku jest potrzebne, aby dokonać przejścia z jednego do drugiego wyrazu;

– wyrazy **Tomek** i **Atomek** są od siebie odległe także[2)] o 1, ponieważ potrzebna jest tylko jedna operacja dołożenia znaku;
– wyrazy **Atomek** i **Romek** są odległe o 2, ponieważ potrzeba jednej operacji usunięcia znaku i jednej operacji zamiany znaku[3)].

2.2. Algorytm wyznaczania wartości odległości Levenshteina pomiędzy dwoma napisami

Wyliczanie wartości tej metryki można zapisać w postaci algorytmu:

```
int LevenshteinDistance(char s[1..m], char t[1..n])
  // d is a table with m+1 rows and n+1 columns
  declare int d[0..m, 0..n]

  for i from 0 to m
    d[i, 0] := i
  for j from 1 to n
    d[0, j] := j

  for i from 1 to m
    for j from 1 to n
        if s[i] = t[j] then cost := 0
                else cost := 1
        d[i, j] := minimum(
        d[i-1, j] + 1, // deletion
        d[i, j-1] + 1, // insertion
        d[i-1, j-1] + cost // substitution
        )
return d[m, n]
```

Algorytm ten można bardzo często spotkać także w reprezentacji tablicowej, gdzie w górny poziomy wiersz wpisano słowo *s,* a w lewą pionową kolumnę słowo *t*. Wartości na przecięciu się wierszy i kolumn wyprowadzonych z poszczególnych liter tych napisów są wartościami odległości edycyjnej pomiędzy danymi podciągami tych słów ograniczonymi poprzez wybrane litery. Dla poprzedniego przykładu tabela wyglądałaby następująco (tab. 1). Wypełniając tabelę od lewej górnej komórki do prawej dolnej zgodnie z algorytmem zaprezentowanym powyżej, w ostatniej komórce tabeli uzyskamy wynik będący wartością odległości edycyjnej pomiędzy tymi słowami (w przykładzie wartość 2).

[2)] Zakładamy, że wielkość liter nie jest rozróżniana
[3)] J.w.

Tabela 1
Tabela przedstawiająca zasadę działania algorytmu znajdującego odległość edycyjną

		R	***O***	***M***	***E***	***K***
	0	1	2	3	4	5
A	1	**1**	2	3	4	5
T	2	2	**2**	3	4	5
O	3	3	2	3	4	5
M	4	4	3	**2**	3	2
E	5	5	4	3	**2**	2
K	6	6	5	4	3	**2**

2.3. Wykorzystanie algorytmu odległości edycyjnej w spellcheckingu

Celem każdego algorytmu spellcheckingu jest umiejętność odpowiedzi na pytanie, czy podane słowo jest poprawnym słowem danego języka, a jeśli nie – zaproponowanie najlepszej opcji poprawy tego napisu na słowo poprawne (algorytm powinien „domyślić" się, jakie słowo jest najlepsze).

Zakładając, że dysponujemy słownikiem wszystkich poprawnych napisów nad zadanym alfabetem, możemy dowolny napis porównać do wszystkich słów tego słownika. Idealnym przykładem takiego słownika jest słownik fleksyjny języka polskiego, gdzie zebrane są wszystkie wyrazy wraz z formami fleksyjnymi, które stanowią poprawne napisy języka polskiego[4)].

Wyznaczając dla dowolnego napisu N odległość edycyjną od wszystkich słów ze słownika, stwierdzić możemy jedną z następujących dwóch możliwości:

1) istnieje słowo należące do słownika takie, że N jest od niego odległe o wartość 0 – co oznacza, że napis N jest poprawnym napisem danego języka,
2) dla każdego słowa należącego do słownika, N jest od nich odległe o wartość większą od 0 – to oznacza, że N nie jest poprawnym słowem danego języka.

Rozważając dalej przypadek 2), możemy znaleźć takie słowo (a precyzyjniej zbiór słów) należących do słownika, dla którego wartość odległości od N było najmniejsze w zbiorze wszystkich wartości. Słowo takie (lub zbiór) stanowić będzie najlepszą propozycję do poprawy napisu N (zakładając, że traktujemy N jako błędnie napisane słowo w danym języku).

[4)] Z dokładnością do nazw własnych.

3. Przypadki specjalne w języku polskim

Analizując algorytm wyznaczania odległości edycyjnej napisów, można zauważyć, że algorytm ten operuje na najprostszej informacji zawartej w napisie (a mianowicie, że każdy napis składa się z liter alfabetu, które można dodać, usunąć lub zamienić w słowie). Nie ma w nim natomiast zawartej żadnej wiedzy o charakterystycznych cechach konstrukcji słów w danym języku, a w szczególności w języku polskim.

W poprzednim rozdziale celowo zwrócono uwagę na fakt, iż wynikiem odpowiedzi algorytmu spellcheckingu może być zbiór słów, a nie tylko pojedyncze słowo – w praktyce okazuje się, że najczęściej właśnie otrzymuje się listę słów, co stanowi potem problem w niektórych zastosowaniach.

Jednym z problemów jest np. zagadnienie automatycznej poprawy tekstów już napisanych bez ingerencji (opieki) człowieka. Algorytm taki otrzymując dla danego nieznanego słowa listę możliwych opcji korekty, nie będzie w stanie stwierdzić, która z opcji jest poprawna. Innym przykładem problemu jest optymalizacja wyników prezentowanych człowiekowi jako propozycje do poprawy: Które ze słów zawartych na liście korekt zaprezentować jako pierwsze?

Uzbrajając algorytm w dodatkowe informacje o naturze słów i typowych błędów danego języka, jesteśmy w stanie poprawić jakość tych sugerowanych podpowiedzi, zawężając i sortując listę słów do najbardziej prawdopodobnych korekt. Warto zwrócić uwagę na fakt, że działania takie cały czas ograniczają się do sfery informacji o języku, zostawiając jeszcze możliwości wykorzystania optymalizacji z użyciem parsera gramatycznego i tym podobnych skomplikowanych rozwiązań.

3.1. Znaki diaktryczne

Znaki diaktryczne (diakrytyczne) są to znaki nietypowe alfabetu narodowego. Dla języka polskiego są to znaki: *ą*, *ć*, *ę*, *ł*, *ń*, *ó*, *ś*, *ż*, *ź*.

Obecność Internetu i wymóg szybkości wpisywania tekstów w celu sprawnej komunikacji z innymi ludźmi sprawił, że często ludzie wpisują na klawiaturze teksty z pominięciem tych znaków (a precyzyjniej – zamieniając je na odpowiedniki niebędące znakami diaktrycznymi przez co łatwiejsze do wpisania z klawiatury). Jest to praktyka tak nagminna, że w większości zrezygnowano z wpisywania tych znaków specjalnych także w listach e-mail, które są przecież elektronicznym odpowiednikiem zwykłych listów pocztowych (jest to kolejny przykład, gdzie łatwość dostępu do technologii sprawiła pogłębienie się niestaranności językowej).

Z tego powodu należy przyjąć, że największy odsetek słów znalezionych w tekście jako niepoprawne słowa języka polskiego, to wyrazy napisane bez polskich znaków diaktrycznych i z tego powodu poszukiwanie odpowiednich propozycji powinno faworyzować słowa, które są tylko dopełnieniem w sensie polskich znaków specjalnych ponad innymi słowami, gdzie trzeba dodawać/usuwać litery.

Aby uwzględnić wyżej wymienioną informację w algorytmie, można zmodyfikować wagi w metryce tak, aby waga zamiany np. litery *a* na literę *ą* była wartością mniejszą od 1.

3.2. Dwuznaki

Język polski posiada bardzo specyficzny twór, jakimi są dwuznaki. Są one pozostałościami historycznymi po różnych transformacjach języka. Istotne z punktu widzenia spellcheckingu jest to, że obecnie pojedynczy dźwięk fonetyczny może być zapisany albo poprzez jedną literę (np. *a*, *o*) albo poprzez dwuznak (np. *dz*, *dź*, *sz*, *cz*, *ch*, *rz*). Z tego powodu pewne zbitki literowe pojawiają się w wyrazach języka statystycznie częściej niż inne (np. *sz* częściej niż *zs*). Tę informację można wykorzystać do optymalizacji algorytmu. Jednakże prawdziwym problemem są tutaj kwestie z pogranicza błędów ortograficznych i fonetyki. Mianowicie niektóre dźwięki potrafią posiadać podwójną reprezentacje (w przeszłości były różnie wymawiane, jednakże obecnie różnice w wymowie zanikły) np. *rz* i *ż*, albo *h* i *ch*. Omawiane przypadki są bardzo częstymi błędami ortograficznymi, więc także statystycznie będą pojawiać się częściej jako błędy w zapisie słowa niż przypadkowe zamiany litery np. *a* na *g*.

Niestety nie jest możliwe proste zaadoptowanie algorytmu odległości edycyjnej, aby obsługiwał dwuznaki. Konieczna jest ingerencja w strukturę algorytmu i zastosowanie odczytu litery z podglądnięciem kolejnej oraz mapy dwuznaków, wraz z odpowiednią zmianą wagi metryki dla takich przypadków.

3.3. Błędy ortograficzne

Jak każdy język naturalny, także i język polski posiada specyficzne dla siebie błędy ortograficzne. Są nimi dla przykładu błędy postaci *u* na *ó*[5)], albo *ż* na *rz*[6)], ale także błędy typu *ą* na *on*[7)] czy *ę* na *en*[8)] i tym podobne. Jak już zostało wspomniane powyżej, takie błędy z całą pewnością będą częściej popełnianymi pomyłkami od błędów losowych i trzeba traktować je w metryce algorytmu w sposób specjalny.

Inną możliwością na poradzenie sobie z tego typu problemem w implementacji jako algorytm odległości edycyjnej jest wstępne zmapowanie słów przed porównaniem na ich fonetyczne postacie poprzez zamienienie wszystkich wariantów błędów na specjalny dodatkowy znak alfabetu. Pozwoli to porównywać słowa klasycznie tak jakby błędy ortograficzne nigdy nie zostały popełnione.

Rozważmy przykład:

móha – słowo wejściowe,

oraz słownik:

mucha,
mocha[9)].

---

5) Kura, kóra
6) Żaba, rzaba
7) Błąd, błond
8) Męka, menka
9) Załóżmy, że takie słowo istnieje w języku polskim

Przed dokonaniem porównania zamieńmy we wszystkich słowach dźwięk *u, ó* na ***Ó*** oraz dźwięk *ch, h* na ***H***:

mÓHa – słowo wejściowe,

m***ÓH***a (mucha),
mo***H***a (mocha).

Po takim podstawieniu widać od razu, jakie słowo jest najbardziej prawdopodobną korektą dla słowa wejściowego. Metoda tu opisana nie wymaga zmian w strukturze algorytmu.

3.4. Pisownia łączna i rozdzielna

Kolejną cechą języka polskiego jest pisownia łączna i rozłączna bardzo różnych wyrażeń, w których najpowszechniejszym typem jest pisownia partykuły *nie* z innymi wyrazami. Warto wiedzieć, że język polski wyszczególnia o wiele więcej przypadków pisowni łącznej i rozdzielnej (pochodzi z [1]):

- pisownia zrostów typu *lwipyszczek*,
- pisownia zestawień typu *lwia paszcza*,
- pisownia wyrażeń typu *dziko rosnący, łatwo strawny, nowo otwarty*,
- pisownia zestawień typu *artysta malarz, lekarz chirurg* (...),
- pisownia przymiotników złożonych typu *jasnoniebieski* (...),
- pisownia połączeń z liczebnikami *pół*,
- pisownia połączeń z liczebnikami *ćwierć*,
- pisownia łączna i rozdzielna wyrażeń przyimkowych,
- pisownia przyimków złożonych,
- pisownia wyrażeń typu *ręka w rękę, sam na sam, od deski do deski*,
- pisownia formy zaimkowej *-ń* z przyimkami,
- pisownia wyrazów z przedrostkami,
- pisownia wyrażeń zaimkowych,
- pisownia zaimków złożonych typu *ten sam, taki sam*,
- pisownia wyrazu *jak*,
- pisownia wyrazu *indziej*,
- pisownia partykuł *bądź, bodaj, byle, chyba, ci, co, lada* (...),
- pisownia partykuł *-że, -ż, -li*,
- pisownia spójnika *że*,
- pisownia cząstek wyrazów *-ąd, -ędy, -dziesiąt, -dziesty* (...),
- pisownia końcówek *-(e)m, -(e)ś, -(e)śmy, -(e)ście*,
- pisownia łączna cząstek *-bym, -byś, -by, -byśmy, -byście*,
- pisownia rozdzielna cząstek *bym, byś, by, byśmy, byście*,
- pisownia łączna partykuły *nie*,
- pisownia rozdzielna partykuły *nie*.

Do tej klasy problemu (ze względu na podobną implementację rozwiązania) zaliczyć można także problematykę użycie łącznika, np. pisownia wyrazów w typu pseudo-Polak, eks-Amerykanin.

Problemy te charakteryzują się dwojaką naturą: z jednej strony błędami mogą być zlepione ze sobą dwa słowa, które powinny być pisane oddzielnie, z drugiej strony dwa oddzielnie pisane słowa, powinny być pisane razem. Z tego powodu implementacja tego typu przypadków wymaga szczególnego podejścia. Przede wszystkim do korekty należy podawać słowa wraz z kontekstem. Po drugie należy zastanowić się nad tym, czy utworzyć słownik listy wyjątków (co zawsze jest dość dobrym pomysłem w tego typu algorytmach), czy zaopatrzyć algorytm w generator wyrażeń, ponieważ w przypadku gdy napis będzie trzeba rozdzielić na dwa lub trzy słowa, sam słownik fleksyjny nie wystarczy.

4. Typowe ulepszenia niezależne od języka

Przy okazji omawiania ulepszeń algorytmu odległości edycyjnej o wiedzę charakterystyczną dla języka polskiego, warto wspomnieć jeszcze o dwóch ogólnych ulepszeniach, które wprawdzie nie są charakterystyczne dla naszego języka, ale dopełniają obraz próby skonstruowania algorytmu naprawdę optymalnego spellcheckera.

4.1. Pomyłki typograficzne

Pisząc na klawiaturze komputera, bardzo często mogą zdarzyć się tak zwane pomyłki typograficzne. Polegają one na wciśnięciu jednego z klawiszy znajdującego się na klawiaturze obok klawisza, który powinien być wciśnięty.

Wyróżniamy dwa rodzaje pomyłek typograficznych:

1) poziome,
2) pionowe.

Dla przykładu pomyłkami typograficznymi dla klawisza *s* mogą być[10]:

– klawisze *a, d* – pomyłki poziome,
– klawisze *q, w, e, z ,x, c* – pomyłki pionowe.

Z tego powodu należy nadać tego typu pomyłkom odpowiednie wartości w metryce, ponieważ są to pomyłki o większym prawdopodobieństwie od pomyłki losowej.

Koniecznie trzeba pamiętać także o fakcie, iż tego typu pomyłki wymagają dodatkowej wiedzy na temat tekstu – mianowicie o układzie klawiatury służącej do pisania. Niestety na świecie można spotkać się z bardzo wieloma różnymi układami klawiatur, choć najpopularniejszą z nich jest z całą pewnością klawiatura 101 klawiszy w układzie QWERTY. Z tego powodu powyższa metoda nie nadaje się do automatycznej poprawy tekstów, których pochodzenia nie znamy, a jedynie do implementowania w typowych spellcheckerach typu „check as I write".

4.2. Czeski błąd

Ostatnim omawianym przypadkiem jest tak zwany czeski błąd. Jest to błąd polegający na nieskoordynowaniu czasowym palców przy szybkim pisaniu na klawiaturze, wynikiem czego słowo zawiera parę zamienionych ze sobą miejscami liter.

---

[10] W standardowej klawiaturze 101 klawiszy o układzie QWERTY.

Sposobem na wykorzystanie tego błędu w poprawie algorytmu jest zaimplementowanie dodatkowej wartości dla metryki w przypadku wykrycia zamienionych miejscami par liter w napisie wejściowym a słowami w słowniku. Niestety wymaga to modyfikacji struktury algorytmu, ponieważ metoda ta wymaga odczytu litery słowa z kontekstem.

5. Optymalizacja wydajnościowa algorytmu

Znając już poszczególne przypadki charakterystyczne w języku polskim (jak i ogólne dobre praktyki), warto zastanowić się także nad kwestiami czysto implementacyjnymi algorytmu. To, w jakim stopniu algorytm będzie optymalny oraz jak dużych zasobów roboczych będzie wymagał, ma wpływ nie tylko na samą szybkość jego działania, ale także na możliwość zastosowania go w urządzeniach o małych możliwościach technicznych (np. w urządzeniach mobilnych). W urządzeniach takich mamy do dyspozycji o wiele słabsze jednostki CPU, a także o wiele mniejsze zasoby pamięci stałej i ulotnej.

5.1. Optymalizacja słownika fleksyjnego języka polskiego

Słownik fleksyjny języka polskiego w najprostszej płaskiej reprezentacji struktury danych (lista kolejnych słów posortowanych alfabetycznie) zajmuje przestrzeń kilkunastu megabajtów. Zaletą takiej reprezentacji jest ogromna prostota we wczytaniu do pamięci programu takiego słownika, a także późniejsze przeglądanie kolejnych słów. Przykładowy słownik fleksyjny języka polskiego może zaczynać się w następujący sposób (tab. 2).

Tabela 2
Przykładowy zbiór początkowych form słownika fleksyjnego języka polskiego

abakus	abandonu	abażurkami
abakusa	abandony	abażurki
abakusach	abazja	abażurkiem
abakusami	abazjach	abażurkom
abakusem	abazjami	abażurkowi
abakusie	abazją	abażurków
abakusom	abazje	abażurku
abakusowi	abazję	abażurom
abakusów	abazji	abażurowi
abakusy	abazjo	abażurów
abandon	abazjom	abażuru
abandonach	abażur	abażury
abandonami	abażurach	abażurze
abandonem	abażurami	abdominalna
abandonie	abażurek	abdominalną
abandonom	abażurem	abdominalne
abandonowi	abażurka	abdominalnego
abandonów	abażurkach	...

Zauważyć można od razu, iż charakterystyczną cechą takich słowników jest powtarzanie się pewnego początku (formantu) w grupie kolejnych wyrazów na liście. Jeśli prze-

analizować także większy fragment słownika, okaże się również, że znajdziemy także często powtarzające się końcówki słów. Taki zapis jest wysoce redundantny i powoduje bardzo duże zużycie pamięci na przechowanie tego typu struktury. Proste wyodrębnienie najdłuższego podciągu początkowego z grupy i zapisanie go wraz z listą końcówek okaże się bardzo efektywne. W tabeli 3 zaprezentowano zapis fragmentu słownika fleksyjnego w postaci listy początków słów wraz z końcówkami, z których można odtworzyć słownik prezentowany w tabeli 2.

Tabela 3
Fragment słownika fleksyjnego w postaci listy początków słów i zbioru końcówek

abakus: Ø, ach, ami, em, ie, om, owi, ów, y
abandon: Ø, ach, ami, em, ie, om, owi, ów, u, y
abazj: a, ach, ami, ą, e, ę, i, o, om
abażur: Ø, ach, ami, ek, em, ka, kach, kami, ki, kiem, kom, kowi, ków, ku, om, owi, ów, u, y, ze
abdominaln: a, ą, e, ego

Jak widzimy, już taki prosty zabieg pozwala znacząco zredukować miejsce zajmowane przez strukturę w pamięci (zarówno stałej, jak i ulotnej). Dużym wyzwaniem przy projektowaniu skrótu takiego słownika fleksyjnego jest tutaj dobór miejsca podziału słowa. Można przyjąć wiele strategii podziału – w szczególności takie, które uwzględniają ewentualną optymalizacje na zbiorze końcówek – czyli potrafią także zastępować licznie pojawiającą się końcówkę odpowiednio skróconym kodem. Praktyka pokazuje, że kilkunastomegabajtowy płaski słownik fleksyjny języka polskiego można zastąpić odpowiednią strukturą początków-końcówek o rozmiarach rzędu kilkuset kilobajtów, a więc o rząd wielkości mniejszą strukturą.

5.2. Obcinanie heurystyki wyznaczania odległości edycyjnej w słowniku fleksyjnym

Ze względu na to, że słownik fleksyjny języka polskiego zawiera bardzo dużo wyrazów (ponad milion form), problemem staje się czas wyznaczenia odległości edycyjnej pomiędzy zadanym napisem a wszystkimi słowami tego słownika. Ponieważ spellcheckery typu „check as I write" sprawdzają pisownię w trakcie wprowadzania tekstu przez człowieka, ich czas reakcji na sprawdzenie pojedynczego słowa wejściowego powinien być możliwie krótki (np. poniżej 1,5 sekundy). Nawet jeśli obecnie na coraz szybszych maszynach desktopowych nieoptymalne algorytmy wykonują się w zadowalającym przedziale czasu (co może być kontrargumentem do podejmowania prób optymalizacyjnych), to jednak należy zauważyć, że maszyny desktopowe nie są jedynym potencjalnym miejscem wykorzystywania takich algorytmów. Należy mieć tutaj na uwadze urządzenia mobilne, które nie dysponują dużą mocą obliczeniową. Czas wykonania takiego algorytmu może być liczony na takich urządzeniach przenośnych nawet w minutach, co uniemożliwiłoby użycie takich algorytmów w zastosowaniu rzeczywistym, a także w znaczący sposób przyczyniłoby się do szybkiego wyczerpania baterii urządzenia przenośnego. Warto więc dokonać wszelkich starań optymalizacyjnych jakie przyczynią się do zwiększenia wydajności spellcheckingu.

Z pomocą może przyjść dowolna klasa metod służących obcięciu heurystyki algorytmu wyznaczania odległości edycyjnej. Można zastosować podejście, które na podstawie pewnych przesłanek dokona predykcji momentu przerwania wyznaczania odległości edycyjnej całego słowa lub grupy słów. Jest to możliwe ponieważ algorytm wyznaczania odległości edycyjnej (patrz tab. 1) wykonywany jest jako ciąg sprawdzeń kolejnych podciągów początkowych słów. Na każdym etapie działania algorytmu możemy stwierdzić stopień zbieżności poszczególnych początków porównywanych słów i na tej podstawie dokonać predykcji.

Wykorzystując tę informację możemy:

– przerwać proces wyznaczania odległości edycyjnej w momencie, gdy dla dowolnego słowa ze słownika fleksyjnego odległość ta wyniosła 0 – oznacza to bowiem że sprawdzane słowo należy do słownika fleksyjnego, więc jednoznacznie zostaje potwierdzone jako prawidłowe i nie ma potrzeby dalszych sprawdzeń;
– przechowywać minimalną wartość odległości edycyjnej w trakcie przeszukiwania, jaką udało się uzyskać, i przerywać wyznaczanie tej wartości dla dowolnego słowa, którego początkowy podciąg słowa w wyznaczaniu odległości uzyskuje wartość większą od minimalnej.

5.3. Optymalizacja wag poszczególnych przypadków charakterystycznych

W rozdziale 3 wymieniono przypadki specyficzne dla języka polskiego, które można uwzględnić podczas tworzenia spellcheckera. Proponowano, aby każdy przypadek szczególny otrzymał specjalną wagę z dowolnego przedziału niekoniecznie równą stałej wartości 1, tak jak w klasycznym modelu odległości Levenshteina. Pozwala to na dowolne sterowanie procesem decyzyjnym spellcheckera. Ta dowolność sprawia jednak trudności w optymalnym określeniu wartości tych parametrów, a jest ich niemało:

– koszt dodania litery,
– koszt usunięcia litery,
– koszt zamiany znaku diaktrycznego na odpowiadającą mu literę,
– koszt zamiany dowolnej innej litery,
– koszt zamiany miejscem dwóch liter (błąd czeski),
– koszt zamiany błędu ortograficznego na poprawną pisownie,
– koszt dodania/usunięcia/zamiany znaku stanowiącego dwuznak,
– koszt pomyłki typograficznej,
– i inne...

Wyznaczenie tych wartości można dokonać poprzez analizę słów słownika fleksyjnego oraz dużego zbioru błędów (zbioru uczącego) i dobranie tych wartości w ten sposób, aby algorytm dawał najlepszą statystykę poprawy na zbiorze uczącym. Do tego celu można także wykorzystać metody automatyczne, jak np. algorytmy genetyczne, które po odpowiednim zakodowaniu tych wag oraz określeniu funkcji oceny w postaci wskaźnika statystyki poprawy spellcheckera pozwoli na automatyczny dobór tych parametrów. Jakość tych metod jest niestety zagrożona niewłaściwym doborem zbioru uczącego. Może się okazać, że

wartości doskonale sprawdzające się na zbiorze testowym będą dawały złe wyniki w rzeczywistych przebiegach. Aby tego uniknąć, można w algorytmach typu „check as I type" stosować podejście samo uczenia się algorytmu. Polega ono na przeliczaniu wartości wag każdorazowo po wyborze przez człowieka poprawnej korekty danego słowa, dzięki czemu wykorzystuje się wiedzę na temat preferencji użytkownika. Metoda taka jest o tyle lepsza od pozostałych, że pozwala nie tylko na uniknięcie problemów związanych ze słabym doborem zbioru uczącego, ale także pozwala na przyuczanie się algorytmu do konkretnych najczęściej popełnianych błędów przez danego użytkownika.

6. Podsumowanie

Przedstawione powyżej metody zostały zaimplementowane w postaci rozszerzenia algorytmu Levenshteina i przyniosły znaczące polepszenie przy sugerowaniu optymalnych korekt dla niepoprawnych napisów względem języka polskiego. Z całą pewnością są to metody potrzebne w dzisiejszym świecie, w którym zauważyć możemy tendencje do rosnącej niedbałości językowej.

Kolejnym krokiem w konstrukcji coraz to lepszych korektorów pisowni polskiej, będzie utworzenie zasad gramatyki języka polskiego w postaci jak najbardziej formalnej, którą będzie dało się wykorzystać do automatycznego sprawdzania gramatycznego tekstu. Być może powstaną także silne mechanizmy semantyczne, które pozwolą ostrzegać człowieka przed błędami o zupełnie innej klasie niż błędy językowe. Do tego jednak czasu wiele odkryć musi być poczynionych w dziedzinie przetwarzania języków naturalnych, aby urealnić takie życzenia.

Literatura

[1] Polański E., *Wielki słownik ortograficzny PWN z zasadami pisowni i interpunkcji*. PWN, Warszawa, 2003.

[2] Levenshtein V.I., *Binary codes capable of correcting deletions, insertions, and reversals*. Doklady Akademii Nauk SSSR, 163(4), 1965, 845–848.

[3] Lubaszewski W., Wróbel H., Gajęcki M., Moskal B., Orzechowska A., Pietras P., Pisarek P., Rokicka T., *Słownik Fleksyjny Języka Polskiego.* wyd. LexisNexis, 2001.

[4] Sankoff D., Kruskal J., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Rozdział pierwszy „An overview of sequence comparison". CSLI Publications, 1999.

[5] Kettunen K., *Low-level typographical spellchecking: A proposal.* Springer: Computers and the Humanities, 2004, ISSN 0010-4817.