POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Inżynieria systemów informatycznych (INS)

Projektowanie systemów z dostępem w języku naturalnym

Opracowanie algorytmu określającego możliwe zamienniki dla błędnych słów w tekstach w języku polskim

AUTOR:
Radosław Taborski - 209347
Piotr Konieczny - 209174

PROWADZĄCY PROJEKT:
dr inż. Dariusz Banasiak

OCENA PROJEKTU:

WROCŁAW 2018

Spis treści

Spis rysunków

Rozdział 1

Wstęp

1.1 Cele projektu

Głównym celem projektu jest stworzenie aplikacji, która będzie umożliwiała wpisanie przez użytkownika dowolnego tekstu w języku polskim oraz pozwoli na jego sprawdzenie pod względem użycia poprawnych słów. Następnie użytkownik będzie miał możliwość wstawienia za każde z wychwyconych błędnych wyrazów, innego, jednego z tych, które będą sugerowane przez aplikację, bądź też zignorowanie sugestii. Aplikacja wyposażona będzie w algorytm sprawdzający czy wyraz istnieje w słowniku języka polskiego, gdy takiego nie znajdzie, wyszukiwane będą słowa o podobnej budowie, lecz nie koniecznie pasujące znaczeniowo do kontekstu zdania.

1.2 Założenia projektowe

W projekcie wykorzystany został język programowania C#, z wykorzystaniem technologii WPF, umożliwiającej tworzenie graficznego interfejsu użytkownika w oparciu o składnię XAML.

W projekcie wykorzystany został również słownik języka polskiego dostępny na stronie słownika języka polskiego [1]. Słownik ten zawiera podstawowe formy słów wraz ze wszystkimi możliwymi odmianami. Słownik ten przewiduje jedynie słowa o maksymalnej długości wynoszącej 15 znaków i nie zawiera żadnych nazw własnych tzn. imion, nazw krajów, miast itp.

1.3 Zakres projektu

Zakres projektu dotyczy zaprojektowania i implementacji aplikacji umożliwiającej pisanie własnych tekstów, które na bieżąco są sprawdzane pod względem poprawności ortograficznej.

Umożliwione użytkownikowi zostało również wklejanie gotowych tekstów, w celu ich sprawdzenia.

Projekt został wyposażony w trzy algorytmy umożliwiające uzyskanie podpowiedzi dla użytkownika oraz opracowane zostały sposoby optymalizacji tak, aby aplikacja działała w czasie rzeczywistym mimo słownika, który zawiera prawie 3 miliony pozycji.

1.4 Opis rozdziałów

1. Rozdział I – opisuje założenia wstępne oraz zawiera informacje, na temat tego co projekt powinien zawierać. Rozdział ten ponadto stanowi wstęp, w którym zawarte są opisy

wszystkich rozdziałów niniejszego dokumentu.

2. Rozdział II – zawiera opis użytych w projekcie algorytmów wraz z diagramami.
3. Rozdział III – opisuje fazę projektową oraz szczegóły implementacyjne projektu oraz opis użytych metod przyśpieszających działanie programu.
4. Rozdział IV – podsumowanie projektu wraz z analizą osiągniętych celów.

Rozdział 2

Opis użytych algorytmów

Projekt został wyposażony w trzy algorytmy wyszukujące słów podobnych pod względem budowy dla bazowego ciągu znaków, który nie występuje w użytym słowniku.

2.1 Odległość Levenshteina

Pierwszym z użytych algorytmów jest algorytm na obliczanie długości Levenshteina (edycyjnej) [5]. Odległość ta jest miarą odmienności napisów. W algorytmie tym wyróżnia się takie pojęcie jak działanie proste na napisie. Do działań takich zaliczamy:

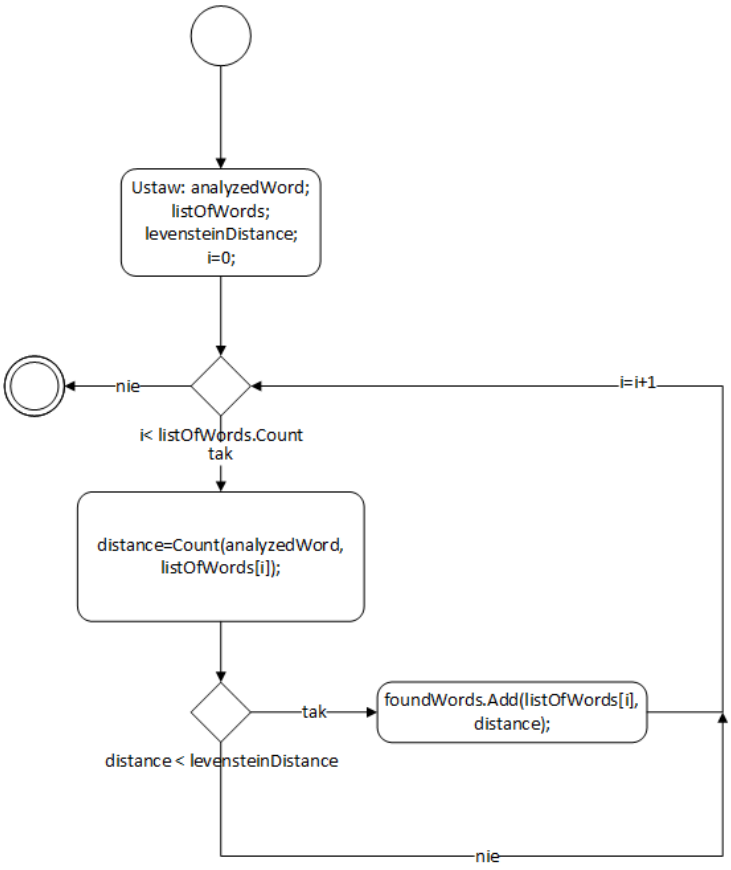
- wstawienie nowego znaku do napisu;
- usunięcie znaku z napisu;
- zamianę znaku w napisie na inny znak;

Idea algorytmu polega na stworzeniu dwuwymiarowej tablicy, o wymiarach n+1 na m+1, gdzie n i m to długości porównywanych słów. Pierwszy wiersz i kolumna uzupełniane są wartościami odpowiedni od 0 do n i m. W następnym etapie po kolei bierze się wartości wiersza i porównuje literę dotyczącą tego wiersza z literą dotyczącą kolumny. Dokonuje się porównania na zasadzie każdy z każdym. Jeżeli litery są identyczne to ustawia się koszt na 0, jeśli nie to na 1. Następnie daną komórkę wypełnia się wartością, którą jest minimum z poniższych trzech pozycji:

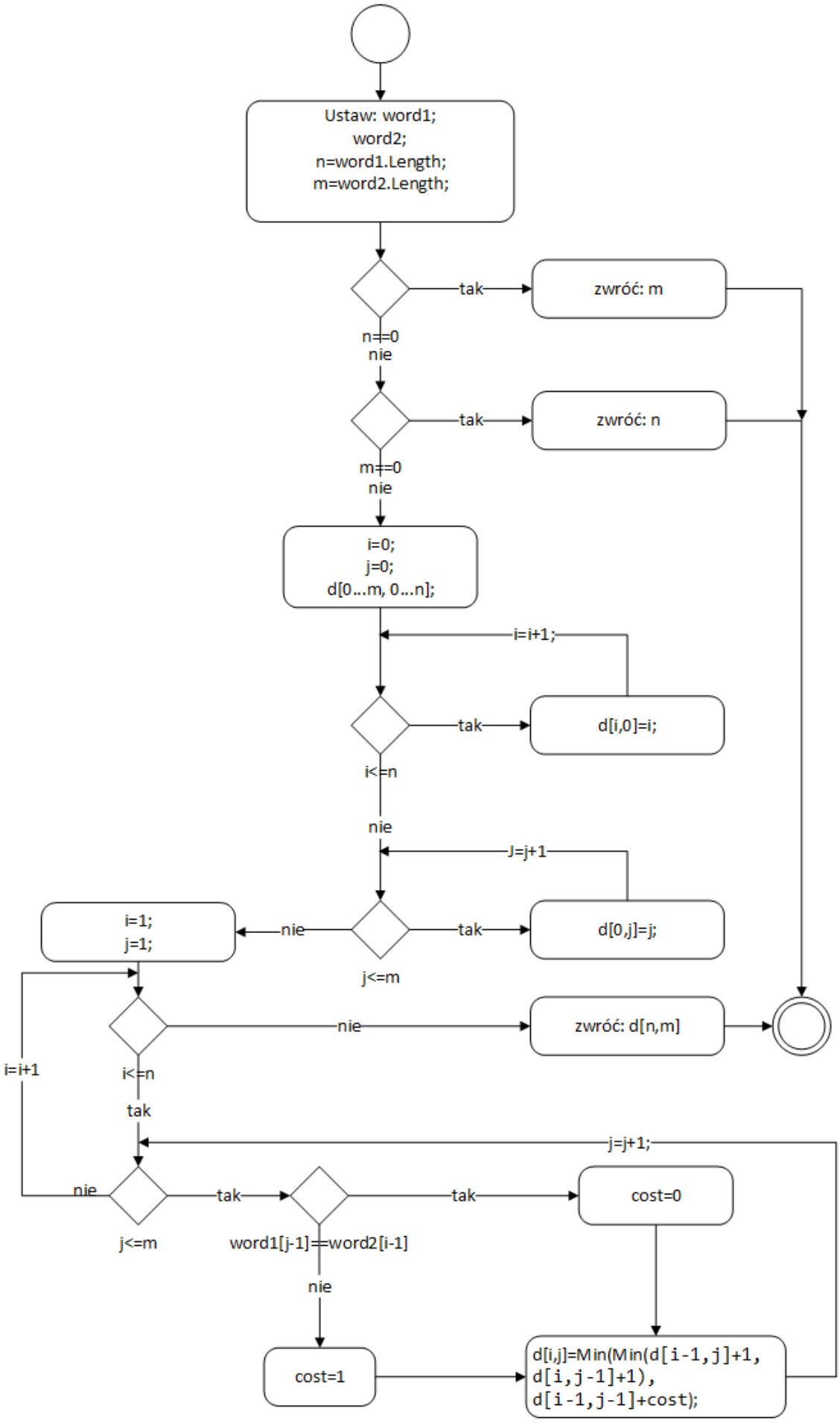
- wartości komórki leżącej bezpośrednio nad aktualnie badaną komórką zwiększoną o 1;
- wartości komórki leżącej bezpośrednio w lewo od naszej aktualnej komórki + 1;
- wartości komórki leżącej bezpośrednio w lewą-górną stronę od aktualnej komórki + koszt;

Po wykonaniu wszystkich porównań, odległością edycyjną będzie wartość w komórce [n+1, m+1]. Szczegółowy diagram działania został przedstawiony na rysunku 2.2.

Algorytm ten posłużył w projekcie do przeszukiwania wybranych fragmentów słownika i wybierania tych słów, których odległość Levenshteina nie przekracza odległości podanej przez użytkownika w jednym z parametrów dostępnych w interfejsie aplikacji. Diagram przedstawiający wykorzystanie odległości Levenshteina został zademonstrowany na rysunku 2.1.



Rysunek 2.1 Diagram dla działania algorytmu z wykorzystaniem odległości Levenshteina



Rysunek 2.2 Diagram obliczania odległości Levenshteina

2.2 Zamiana znaków

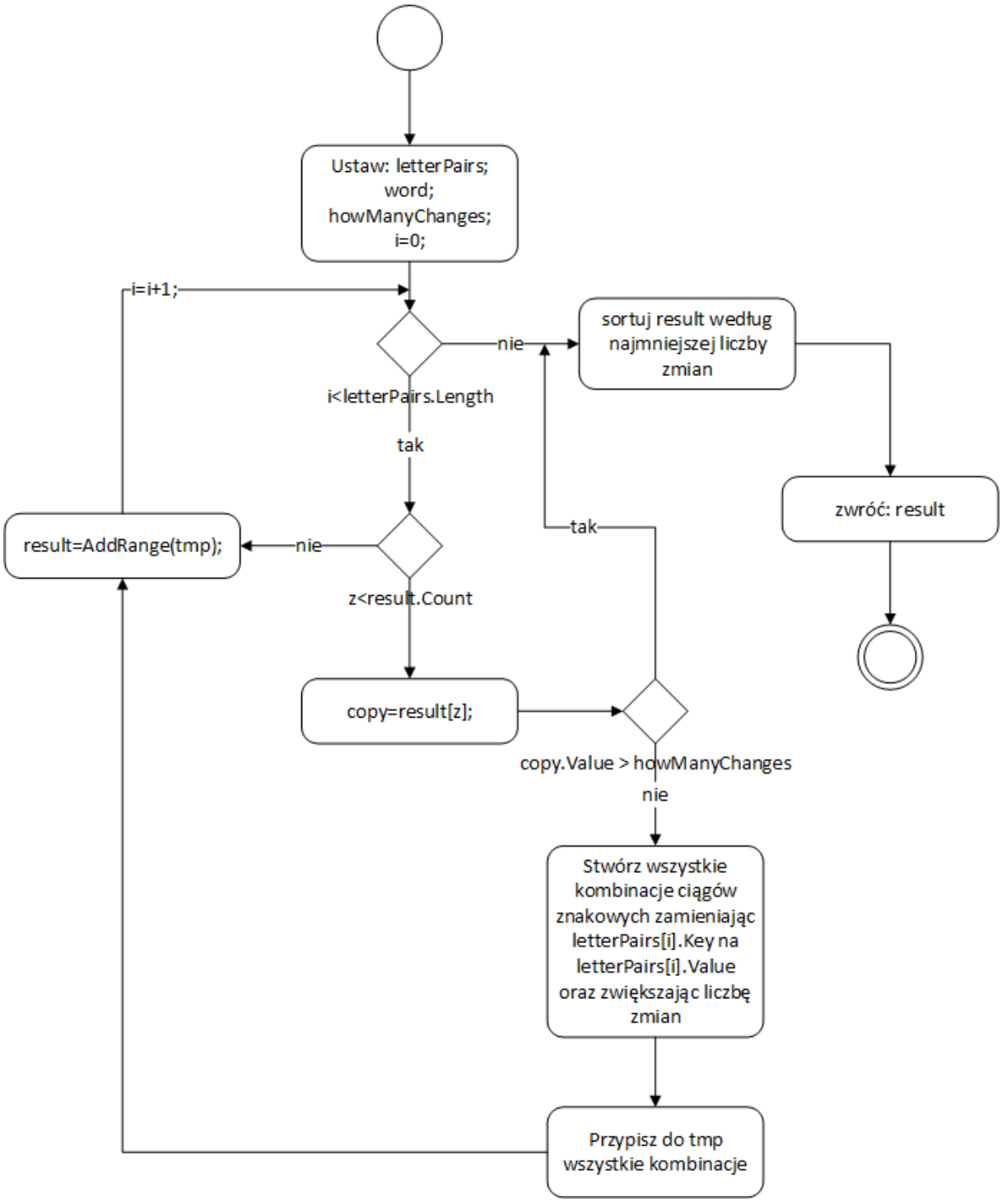
Ten algorytm do swojego działania potrzebuje listy najczęściej popełnianych błędów ortograficznych ([**??**]). Wybrane ciągi znaków i ich odpowiedniki ustawione zostały w liście _letterPairs, której elementy są typu KeyValuePair, składającego się z elementów jak Key i Value (oba są typu string), co przedstawione zostało na listingu 2.1. Są to według autorów projektu ciągi znaków, w których najczęściej popełniane zostają błędy ortograficzne.

```
1         private readonly static List<KeyValuePair<string, string>>
              _letterPairs = new List<KeyValuePair<string, string>>()
2         {
3             new KeyValuePair<string, string>("rz", "ż"),
4             new KeyValuePair<string, string>("ż", "rz"),
5
6             new KeyValuePair<string, string>("u", "ó"),
7             new KeyValuePair<string, string>("ó", "u"),
8
9             new KeyValuePair<string, string>("ch", "h"),
10            new KeyValuePair<string, string>("h", "ch"),
11
12            new KeyValuePair<string, string>("i", "j"),
13            new KeyValuePair<string, string>("j", "i"),
14
15            new KeyValuePair<string, string>("ę", "em"),
16
17            new KeyValuePair<string, string>("ji", "i"),
18            new KeyValuePair<string, string>("ji", "ii"),
19
20            new KeyValuePair<string, string>("rs", "rws"),
21
22            new KeyValuePair<string, string>("ą", "on"),
23            new KeyValuePair<string, string>("on", "ą"),
24            new KeyValuePair<string, string>("ą", "ę"),
25            new KeyValuePair<string, string>("ę", "ą"),
26            new KeyValuePair<string, string>("ą", "om"),
27            new KeyValuePair<string, string>("en", "ę"),
28            new KeyValuePair<string, string>("ę", "en"),
29            new KeyValuePair<string, string>("en", "ę"),
30            new KeyValuePair<string, string>("en", "ę"),
31
32            new KeyValuePair<string, string>("trz", "cz"),
33            new KeyValuePair<string, string>("cz", "trz"),
34
35            new KeyValuePair<string, string>("dź", "ć"),
36            new KeyValuePair<string, string>("ć", "dź"),
37
38            new KeyValuePair<string, string>("s", "ś"),
39            new KeyValuePair<string, string>("ś", "s"),
40            new KeyValuePair<string, string>("ł", "l"),
41            new KeyValuePair<string, string>("l", "ł"),
42            new KeyValuePair<string, string>("a", "ą"),
43            new KeyValuePair<string, string>("ą", "a"),
44            new KeyValuePair<string, string>("e", "ę"),
45            new KeyValuePair<string, string>("ę", "e"),
46            new KeyValuePair<string, string>("z", "ź"),
47            new KeyValuePair<string, string>("ź", "z"),
48            new KeyValuePair<string, string>("z", "ż"),
49            new KeyValuePair<string, string>("ż", "z"),
```

```
50              new KeyValuePair<string, string>("o", "ó"),
51              new KeyValuePair<string, string>("ó", "o"),
52              new KeyValuePair<string, string>("ć", "c"),
53              new KeyValuePair<string, string>("c", "ć"),
54              new KeyValuePair<string, string>("n", "ń"),
55              new KeyValuePair<string, string>("ń", "n"),
56
57              new KeyValuePair<string, string>("sz", "ż"),
58              new KeyValuePair<string, string>("ż", "sz"),
59          };
```

Listing 2.1 Lista par ciągów znakowych odpowiadająca najczęstszym błędom w języku polskim

Badany ciąg znaków, zostaje sprawdzony pod względem obecności ciągów znaków, z powyższej listy, z parametru Key. Następnie badany ciąg znaków zostaje przekształcany poprzez zamienianie wszystkich "podejrzanych o błąd"fragmentów na ich zamienniki, czyli na ciągi znaków z parametru Value. Na przykład ciąg znaków „grzegrzółka”, pod wpływem zamian wynikających z pierwszego elementu listy _letterPairs może przyjąć następujące trzy formy: „gżegrzółka”, „grzegżółka” oraz „gżegżółka”. Wszystkie te trzy formy zostają wychwycone przez algorytm i zwrócone do programu w dalszym etapie. Przy okazji tego ciągu znaków, badane są również wszystkie ciągi znaków z uwzględnieniem zamiany „a” na „ą” oraz „e” na „ę”, czyli m.in. powstają też takie kombinacje jak „gżęgrzółką”, „grzegrzółką”. Gdyby badany ciąg znaków tworzył „grzegrzulka” to dodatkowo badane byłyby wszystkie kombinacje zamieniające „u” na „ó” oraz „l” na „ł”. Algorytm ten powstał w celu łatwiejszego znajdywania tych kombinacji, których ilość podstawowych zamian na napisie, opisanych w poprzednim podrozdziale 2.1, może sugerować, że zostało wykonanych wiele operacji, a tak na prawdę został popełniony np. jeden błąd ortograficzny, dobrym przykładem jest tutaj zamiana „cz” na „trz”. Również biorąc pod uwagę, że z powodów optymalizacyjnych, przy mierzeniu odległości Levenshteina, nie jest badany cały słownik, a jedynie najbardziej prawdopodobne jego fragmenty (więcej w podrozdziale 3.5), ten algorytm doskonale wychwytuje pozostałe nieprawidłowości w badanym ciągu znaków. Schemat działania tego algorytmu został przedstawiony na diagramie z rysunku 2.3. Algorytm ten znajduje więc dość dużo różnych „dziwnych” kombinacji, jednak w dalszym etapie wszystkie te kombinacje podlegają sprawdzeniu, czy występują w słowniku i na tej podstawie określane jest czy zostaną podpowiedziane użytkownikowi czy też nie.

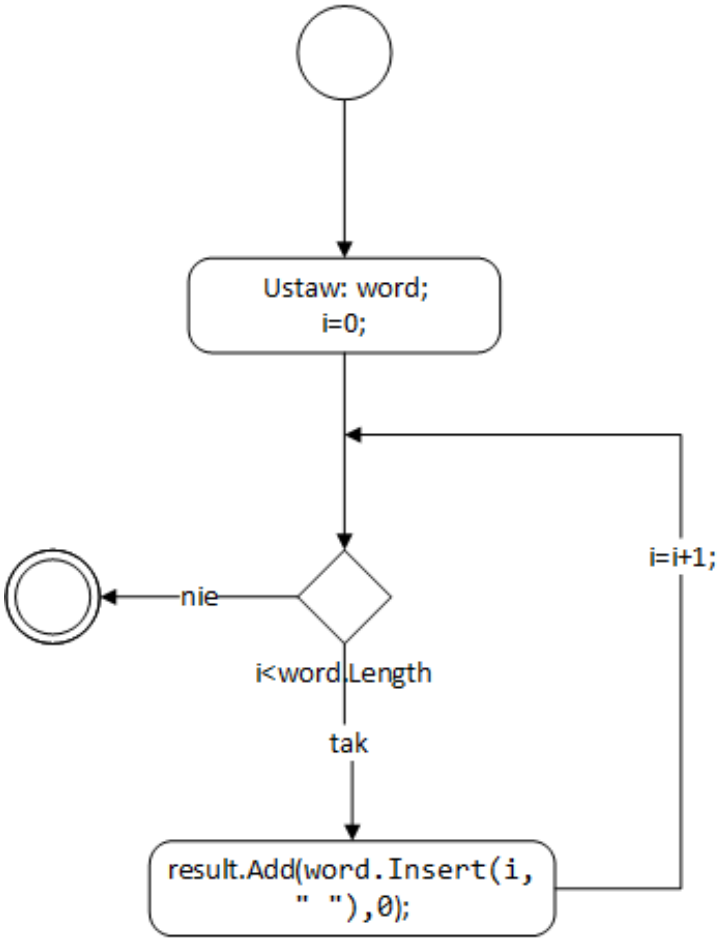


Rysunek 2.3 Diagram zamieniania ciągów znaków odpowiadających najczęstszym błędom ortograficznym

2.3 Podział na wyrazy

Algorytm ten powstał w celu wychwytywania, błędów, które na co dzień nie mają miejsca w piśmie ręcznym, jednak są powszechnym problemem w podczas opracowywania tekstów na komputerze. Mianowicie problem ten dotyczy braku wciśnięcia, bądź też reakcji komputera na wciśnięcie klawisza spacji. Algorytm ten bierze pod uwagę, że w jednym ciągu znakowym może

wystąpić tylko jedno takie zdarzenie. Na przykład między innymi z ciągu znaków "idziekotźostanie przekształcony do postaci "idzie kot", ale również do postaci "idzi ekot". Algorytm ten nie tworzy zbyt wielu kombinacji, jednak rozwiązuje dość często popełniany błąd w pisowni komputerowej. Dodatkowo w dalszym etapie działania programu, kombinacje uzyskane z tego algorytmu podlegają też zamianom w oparciu o algorytm LetterChanger opisanym w podrozdziale 2.2, dzięki czemu również znajdywane są sugestie dla słów, w których popełniony został błąd ortograficzny oraz zabrakło spacji. Diagram UML, dla tego algorytmu został przedstawiony na grafice 2.4.

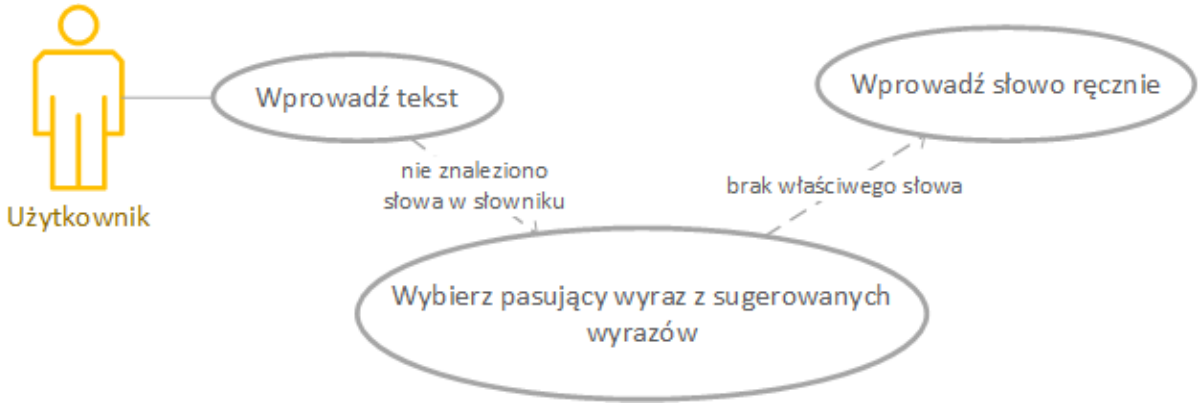


Rysunek 2.4 Diagram działania funkcji dodającej przerwy między wyrazami

Rozdział 3

Projekt i implementacja aplikacji

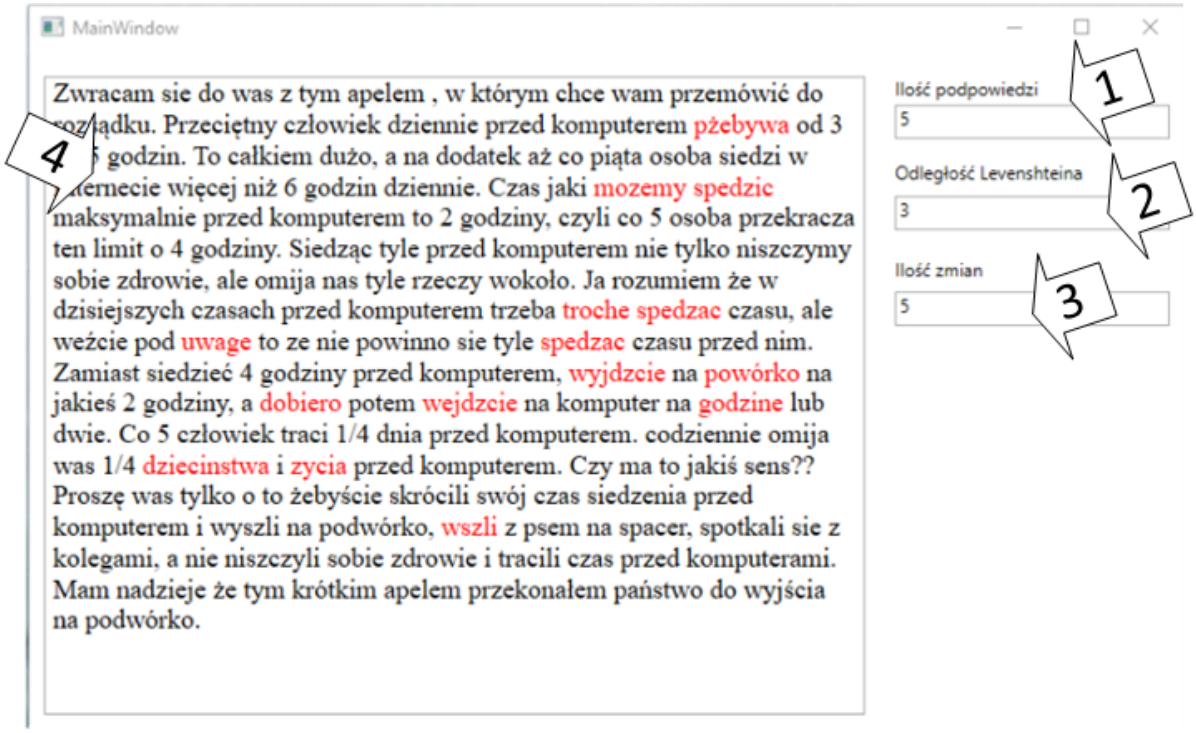
3.1 Funkcje aplikacji - diagram przypadków użycia



Rysunek 3.1 Diagram przypadków użycia

Głównym zadaniem aplikacji jest wyszukiwanie błędnych wyrazów w języku polskim oraz ich korekcja. Aby tego dokonać aplikacja porównuje wyrazy znajdujące się w tekście z wyrazami znajdującymi się w słowniku. Jeśli danego słowa nie znaleziono zostaje ono uznane za błędne. Aby dokonać korekcji błędnych wyrazów stosowane są algorytmy opisane w poprzednich rozdziałach. Dla optymalizacji działania aplikacji wyszukiwanie sugestii odbywa się w osobnych wątkach co znacząco przyspiesza działanie aplikacji.

3.2 Interfejs aplikacji



Rysunek 3.2 Interfejs aplikacji

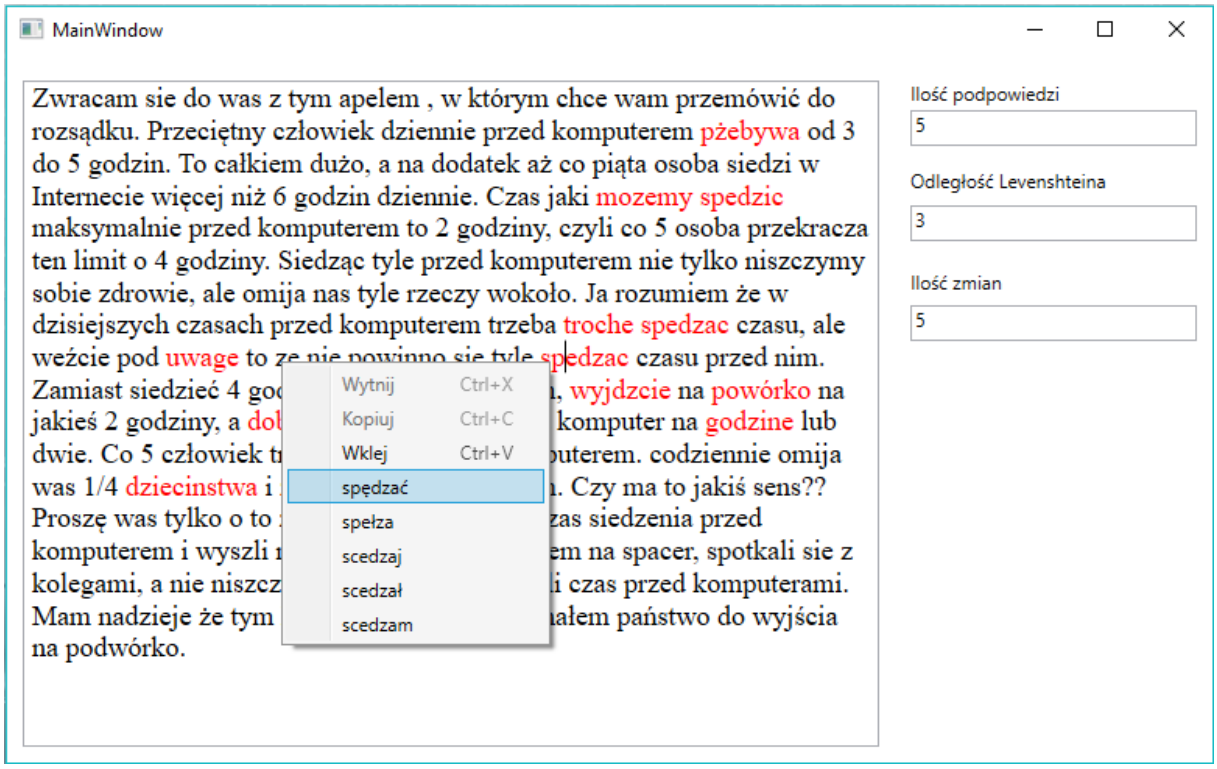
1. Ilość podpowiedzi - pozwala na ustawienie maksymalnej ilości podpowiedzi jakie mają się pojawić po kliknięciu prawym przyciskiem myszy na błędnie napisane słowo.

2. Odległość Levenshteina - pozwala na ustawienie odległości jaka ma być brana pod uwagę w przypadku algorytmu Levenshteina. Im większa liczba tym więcej podpowiedzi ale jednocześnie zmniejsza to szybkość działania aplikacji ze względu na dodatkowe obliczenia które muszą zostać wykonane.

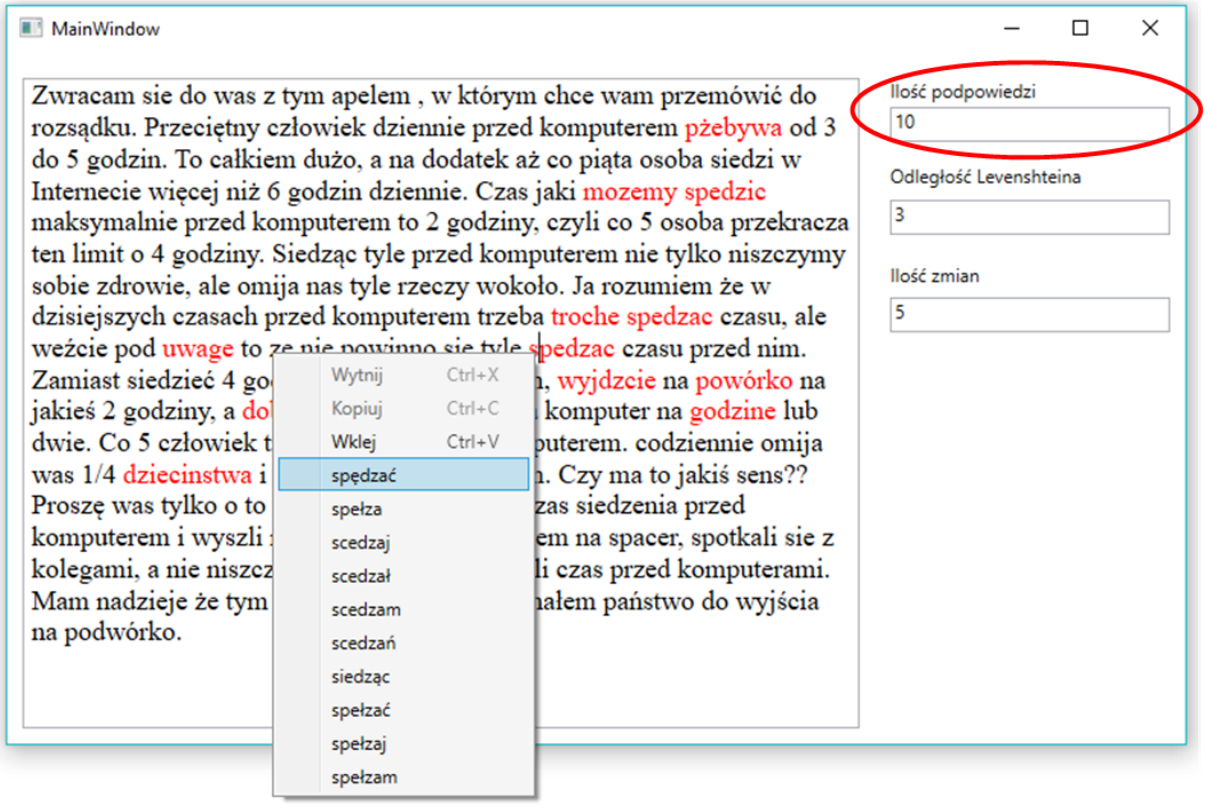
3. Ilość zmian - pozwala na ustawienie parametru ilości zmian dla algorytmu podmieniającego znaki diakrytyczne. Im większa liczba tym więcej podpowiedzi oraz dłuższy czas wykonywania algorytmu.

4. Edytor tekstu - pozwala na edycję tekstu. Wyszukiwanie błędów działa w czasie rzeczywistym (funkcja sprawdzająca poprawność uruchamia się z każdym kliknięciem spacji). Jeśli dane słowo zostało uznane za błędne zostaje zaznaczone kolorem czerwonym. Aby je poprawić należy kliknąć na nie prawym przyciskiem myszki. Zostanie wyświetlone menu kontekstowe zawierające możliwe zamienniki. Aby podmienić słowo wystarczy kliknąć na zamiennik. Możliwe jest również poprawienie tekstu ręcznie wpisując w miejscu błędnego słowa poprawne.

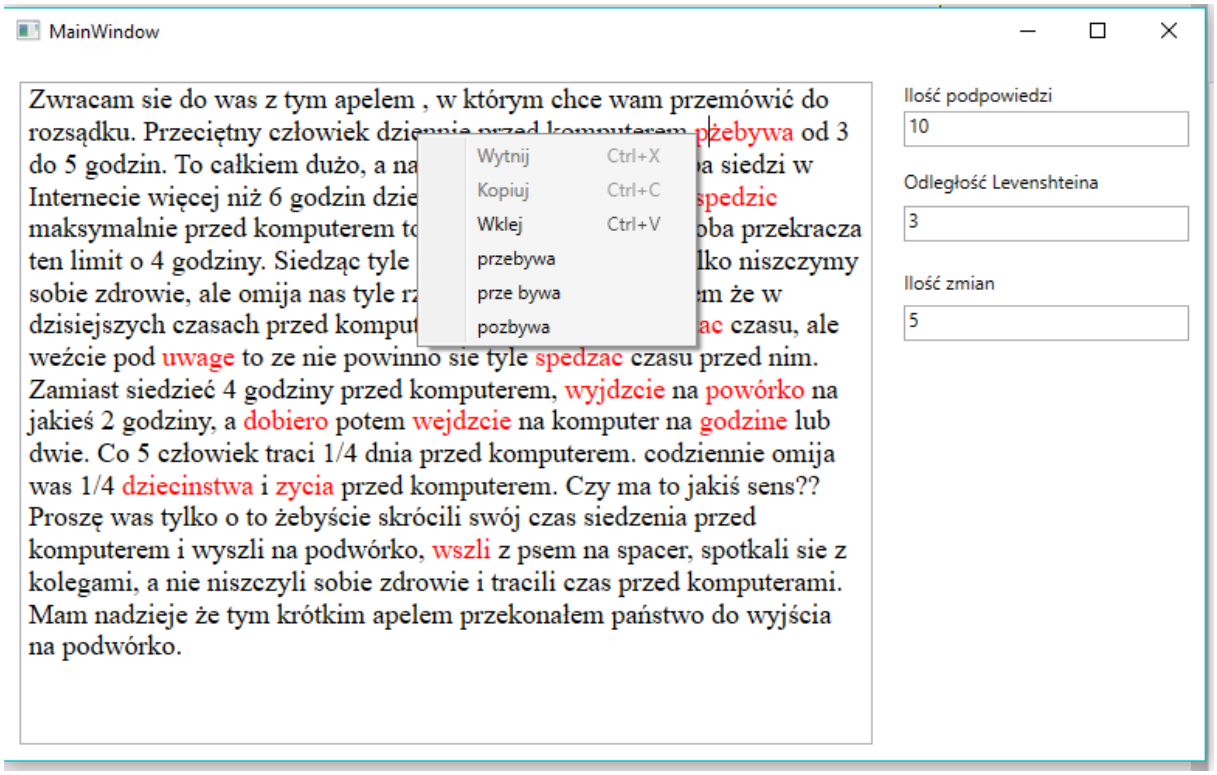
3.3 Wpływ parametrów na wyniki wyszukiwania



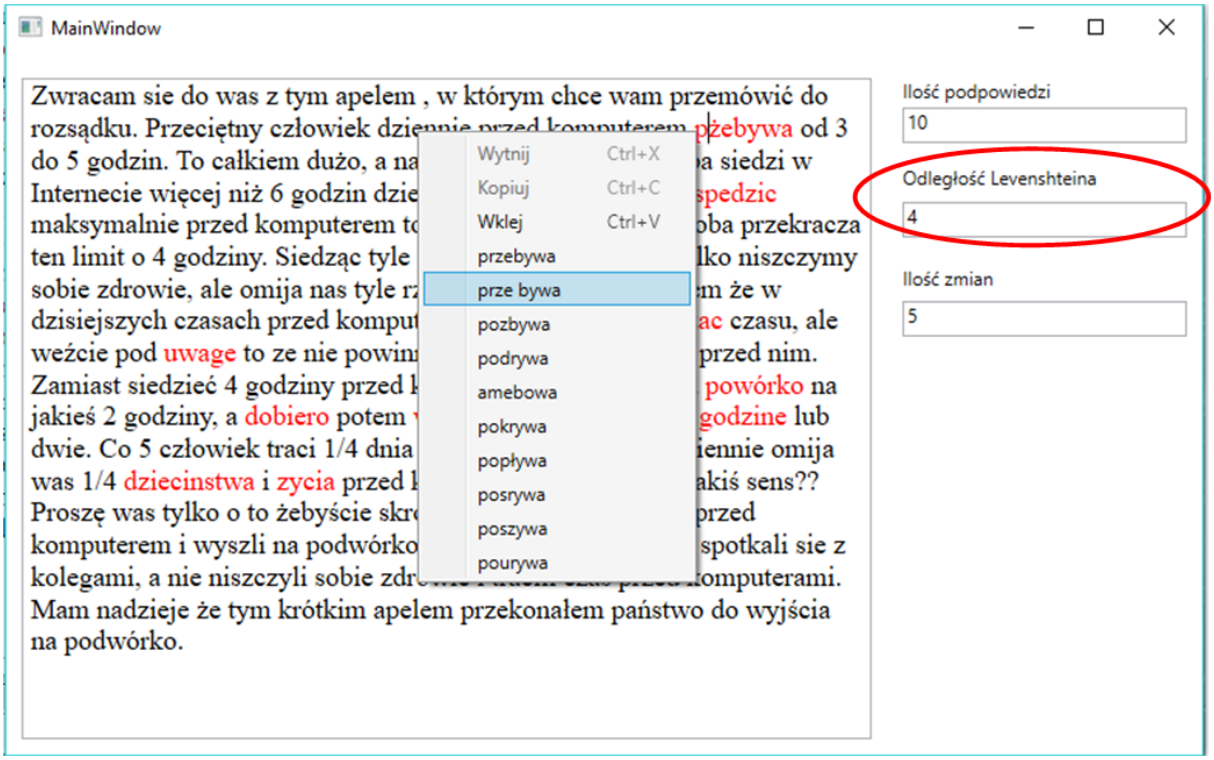
Rysunek 3.3 Przykładowy wynik wyszukiwania sugestii



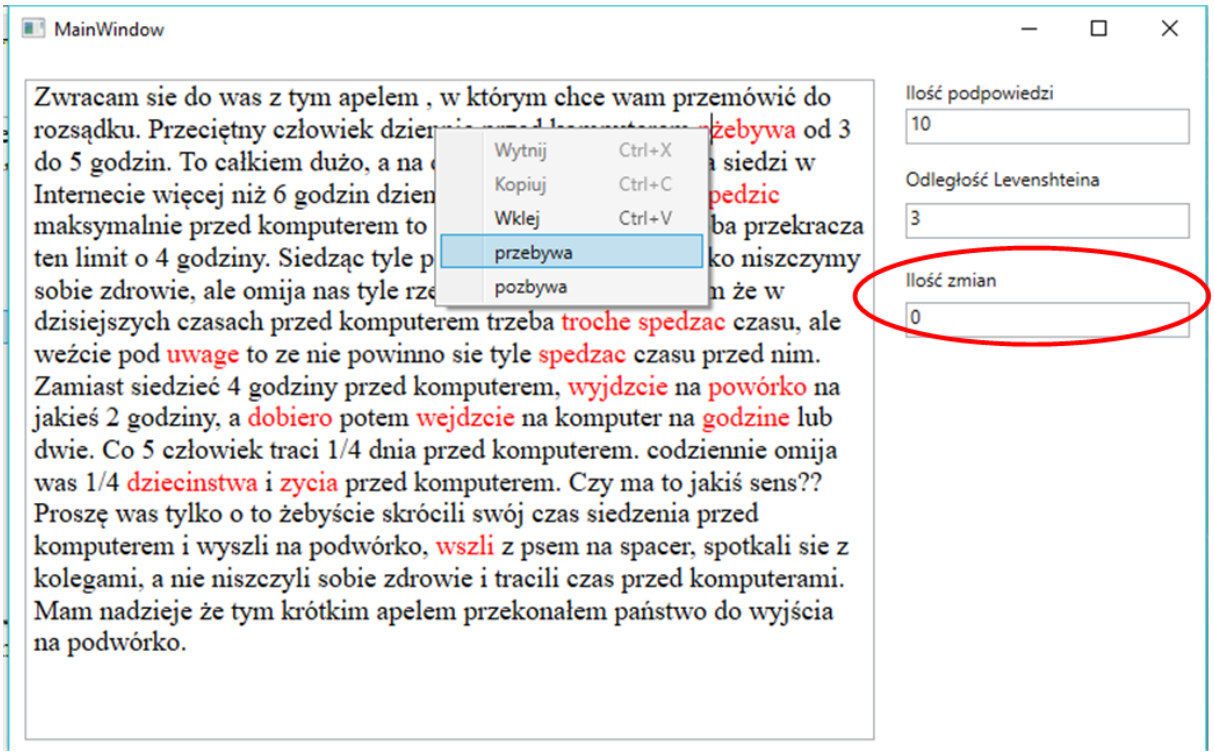
Rysunek 3.4 Zwiększenie ilości wyświetlanych sugestii



Rysunek 3.5 Przykładowy wynik wyszukiwania sugestii dla małej ilości podpowiedzi



Rysunek 3.6 Zwiększenie odległości Levenhstaina

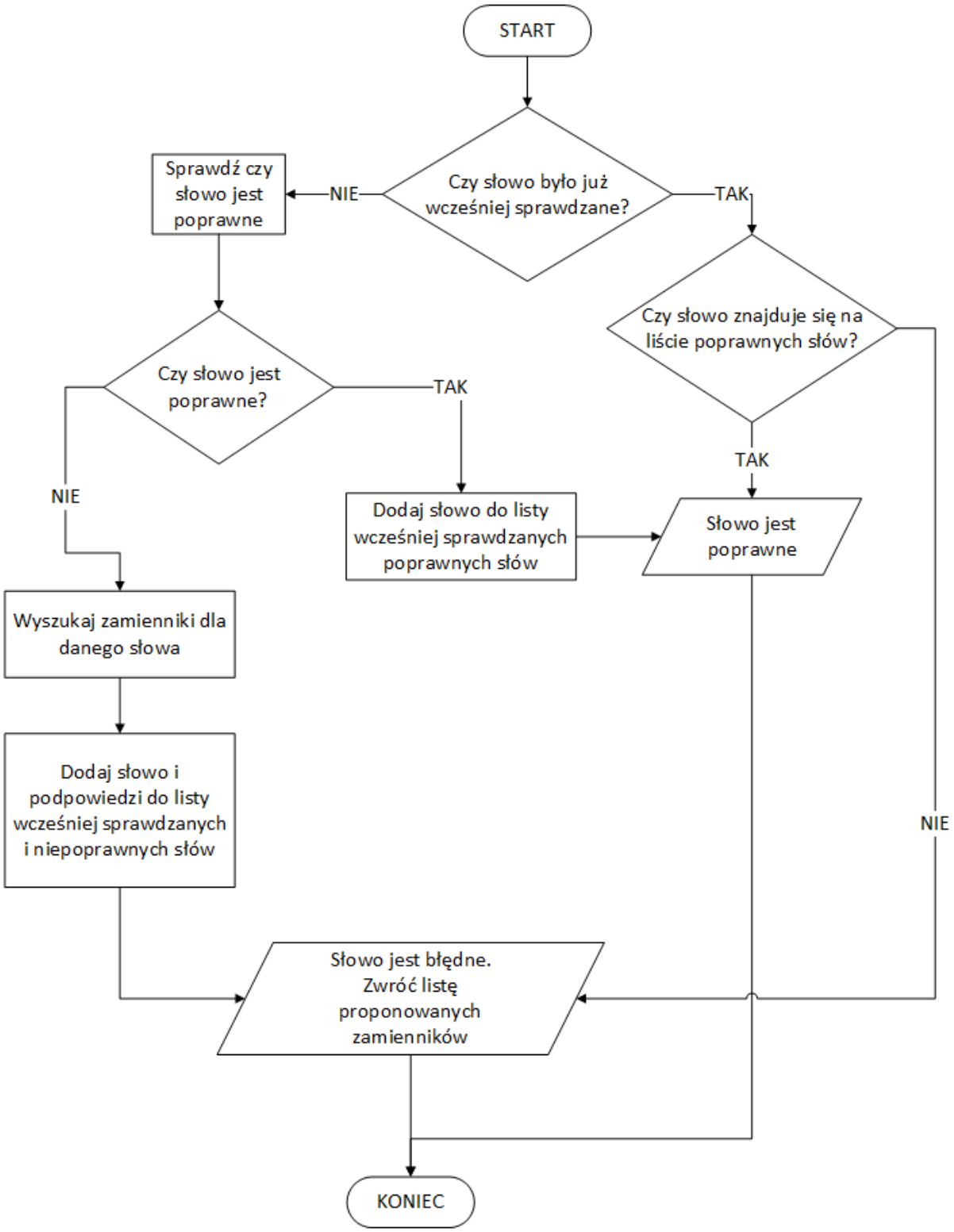


Rysunek 3.7 Zwiększenie ilości zmian

Odpowiednio zmieniając parametry znajdujące się w prawej części okna aplikacji można dostosować wyszukiwanie do potrzeb użytkownika. Rysunek 3.3 przedstawia przykładowe wyniki wyszukiwania podpowiedzi. Jeśli ilość wyświetlanych wyników nie jest odpowiednia można zmienić ich ilość tak aby uzyskać satysfakcjonujący wynik. Rysunek 3.4 przedstawia wpływ zwiększenia ilości wyświetlanych podpowiedzi. Jednocześnie ilość podpowiedzi nie ma wpływu na szybkość działania aplikacji.

Pozostałe parametry wpływają na podobieństwo sugestii do wprowadzonego błędnie słowa. Na rysunku 3.5 został przedstawiony przykładowy wynik wyszukiwania który przy danych parametrach nie wypełnia całej listy podpowiedzi. Na rysunku 3.6 zwiększono odległość Levenshteina tak aby uzyskać więcej sugestii. Natomiast na rysunku 3.7 całkowicie zminimalizowano wpływ algorytmu podmieniającego znaki tak aby pozbyć się niechcianych podpowiedzi zawierających spacje.

3.4 Realizacja wybranych funkcjonalności



Rysunek 3.8 Schemat korekcji tekstu z poziomu interfejsu

Główną funkcjonalnością aplikacji jest wyszukiwanie a następnie korekcja błędów. Rysunek 3.8 przedstawia schemat działania algorytmu odpowiedzialnego za powyższe zadanie.

Początkowo algorytm sprawdza czy dane słowo było już wcześniej wyszukiwane. Jeśli tak to sprawdza czy jest ono na liście poprawnych słów. Jeśli tak to słowo jest poprawne i algorytm kończy pracę. Jeśli nie to sprawdza on listę błędnie wprowadzonych słów i zwraca on podpowiedzi. Jeśli podpowiedzi zostały już raz wyszukane dla danego słowa to zostają one zapisane. Ma to na celu przyspieszenie działania algorytmu ponieważ najwięcej czasu pochłania wyszukiwanie sugestii.

Jeśli słowo jest wyszukiwane po raz pierwszy następuje sprawdzenie jego poprawności porównując je ze słowami znajdującymi się w słowniku. Jeśli jest poprawne to zostaje ono dodane do listy poprawnych sprawdzanych wcześniej słów i algorytm kończy pracę. Natomiast jeśli słowo jest błędne następuje wyszukiwanie sugestii. Po ich wyszukaniu zostają on zapisane wraz z niepoprawnym wyrazem na liście błędnych słów. Algorytm zwraca listę sugestii i kończy pracę.

Algorytm ten działa dla każdego słowa w osobnym wątku. Ma to za zadnie przyspieszenie działania aplikacji.

3.5 Optymalizacja

Słownik użyty w projekcie składa się z blisko trzech milionów haseł. Podczas działania programu słownik ten musi być wielokrotnie przeglądany, zarówno poprzez sprawdzanie każdego wyrazu wpisanego przez użytkownika, jak i badaniu każdej powstałej kombinacji z algorytmu zamiany znaków jak i podziału na wyrazy oraz przy badaniu odległości Levenshteina w najprostszej wersji, każdy badany wyraz powinien mieć mierzoną odległość z każdym wyrazem znajdującym się w słowniku. Przy zwykłym użytkowaniu programu sprowadzałoby się to do wielu milionów operacji, a w związku z faktem iż założono w projekcie, że program ma działać możliwie w czasie rzeczywistym, podjęto próbę rozwiązania problemu z tak dużą liczbą operacji przeglądania całego słownika oraz z ograniczeniem ilości wykonywania obliczania odległości Levenshteina. W poniższych podrozdziałach opisane zostały podjęte w tym zakresie czynności.

3.5.1 Serializacja

Serializacja jest mechanizmem zamiany obiektu zapisywania obiektów, który pozwala na binarny zapis całego drzewa obiektów. Tak zapisany obiekt można w później uruchomić w dowolnym momencie pomijając cały etap tworzenia obiektów, a zastępując go procesem deserializacji, który z reguły powinien trwać szybciej niż proces tworzenia nowych obiektów.

W projekcie serializacja użyta została do zapisania klasy odpowiedzialnej za przechowywanie słownika. Przy pierwszym uruchomieniu programu słownik jest wczytywany z pliku tekstowego, a jego hasła są w odpowiedni sposób organizowane (więcej o tym w kolejnym podrozdziale). Dlatego uruchamianie programu i organizowanie słownika za każdym razem od nowa byłoby dość mało optymalne. Klasa z wczytanym słownikiem została zserializowana i zapisana do pliku. Od tego momentu przy każdym ponownym uruchomieniu programu, sprawdzane jest czy ten plik istnieje we wskazanym miejscu i o ile nikt go nie usunął to zostaje wczytany. Operacja trwa o wiele krócej, niż odczyt pliku i organizacja słownika na nowo.

Przy wyborze metody serializacji posłużono się artykułem ze strony [2], z którego to wynika, że jedną z najszybszych metod serializacji i deserializacji odbywa się z wykorzystaniem biblioteki *ProtoBuf*, stworzonej przez firmę *Google*. Metoda ta opera się o zapisywanie stanu

klasy do pliku o strukturze XML, a każde pole i właściwość klasy musi zostać wyposażone w specjalny atrybut *[ProtoMember(1)]* , przy czym cyfra podana w nawiasie musi być dla każdego pola unikalna ([3]).

3.5.2 Organizacja słownika

Jak już zostało wspomniane w poprzednim podrozdziale, przy pierwszym uruchomieniu słownik jest wczytywany z pliku tekstowego i tworzona jest z niego klasa, która przechowuje słownik ten w odpowiednio zorganizowany sposób, który ułatwia jego przeszukiwanie.

Hasła są w pamięci przechowywane w dwojaki sposób:

- Jako zbiór list, z których każda przechowuje wyrazy o różnej długości oraz są posortowane w sposób alfabetyczny;
- Jako zbiór obiektów, z których każdy przechowuje wyrazy o różnej długości oraz rozpoczynające się na różną literę;

Pierwszy sposób przechowywania haseł został przystosowany specjalnie dla przeszukiwania słownika, poprzez przeglądanie haseł o wskazanej długości. Jest to szczególnie przydatne dla algorytmu wykorzystującego odległość Levenshteina. Badane słowo ma określoną długość, a zatem prawdopodobnie użytkownikowi również chodziło o słowo podobnej długości. Tak więc nie jest przeszukiwany cały słownik, a jedynie hasła o podobnej długości. W projekcie przyjęto że mogą to być hasła o długości z przedziału od -1 do +1 długości słowa badanego.

Drugi sposób przechowywania haseł został wykorzystany dla pozostałych dwóch algorytmów oraz dla sprawdzania czy dany wyraz znajduje się w słowniku. Przeszukiwanie słownika odbywa się w sposób dużo szybszy, gdyż znana jest długość słowa badanego oraz jego pierwsza litera, a oba te parametry odpowiadają indeksom dwuwymiarowego zbioru w którym przechowywane są hasła.

3.5.3 Praca na wielu wątkach

Rozdział 4

Podsumowanie

Aplikacja stworzona na potrzeby projektu spełnia wszystkie cele oraz założenia projektowe. Pozwala ona na sprawdzenie dowolnego tekstu w języku polskim wpisanego przez użytkownika. Użytkownik ma możliwość wstawienia w każde z błędnych słów innego zasugerowanego przez system. Dzięki zoptymalizowaniu algorytmów wyszukiwania jak i interfejsu użytkownika wyszukiwanie sugestii odbywa się w czasie rzeczywistym. Dzięki zastosowaniu słownika w postaci pliku tekstowego jego modyfikacja w formie dodawania kolejnych wyrazów jest bardzo prosta.

W celu wyszukiwania jak najtrafniejszych zamienników dla błędnie napisanych słów zaimplementowano trzy algorytmy:

- Odległość Levenshteina - oblicza on odległość edycyjną dwóch porównywanych napisów, odległość ta wyrażana jest przez liczbę operacji podstawowych. Jednak konieczność przeglądania całego słownika, w dość znacznym stopniu obciąża procesor, dlatego też w projekcie zastosowano wyłącznie porównywanie słów o podobnej długości (plus minus jeden znak).
- Zamiana znaków - algorytm ten zamienia ciągi znakowe odpowiadające częstym błędom ortograficznym w języku polskim i na tej podstawie utworzyć może wiele kombinacji i w pewnym stopniu uzupełnia uproszczenie zastosowane w przypadku użycia odległości Levenstheina, dla wybranych fragmentów słownika.
- Podział na wyrazy - algorytm ten przeciwdziała częstym błędom w piśmie komputerowym, wynikającym z braku wciśnięcia spacji.

Dzięki takiemu rozwiązaniu program dobrze radzi sobie niezależnie od rodzaju błędu jaki popełnił użytkownik.

Aplikacja została napisana w języku C# dlatego jest przeznaczona na komputery z systemem Windows. Korzystanie z niej nie wymaga instalacji. Dodatkowo jest to niezależna aplikacja, która do działania nie wymaga dodatkowych bibliotek.

Literatura

[1] strona internetowa: *http://sjp.pl/slownik/growy*, dostęp: 2017-11-15

[2] strona internetowa: *http://akubiak.pl/2015/02/27/porownanie-metod-serializacji-w-c/*, dostęp: 2017-11-08

[3] strona internetowa: *http://developers.google.com/protocol-buffers/docs/csharptutorial*, dostęp: 2017-11-10

[4] strona internetowa: *http://www.rjp.pan.pl/index.php?option=com_content&view=article&id=1101:ustalenia-dotyczce-bdow-ortograficznych&catid=54&Itemid=66*, dostęp: 2017-12-17

[5] strona internetowa: *http://pl.wikipedia.org/wiki/ Odleg%C5%82o%C5%9B%C4%87_Levenshteina*, dostęp: 2018-01-10