Довідка
про відсутність плагіату у випускній кваліфікаційній роботі

Міністерство освіти і науки України
Український державний університет науки та технологій

Кафедра «Комп'ютерні інформаційні технології»

ДОВІДКА

За результатами перевірки випускної кваліфікаційної роботи здобувача вищої освіти Кириченка Олександра Олександровича
(прізвище, ім'я, по батькові)
на тему: Дослідження структур даних електронного словника української мови для задач встановлення авторства текстів
в роботі не виявлено порушень академічної доброчесності.

Керівник ВКР _____ Віктор Шинкаренко

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Український державний університет науки і технологій

Кафедра Комп'ютерні інформаційні технології

«ДО ЗАХИСТУ»

Завідувач кафедри

_____ /Вадим ГОРЯЧКІН/

«__20__» _____12_____ 20__21__ р.

ДИПЛОМНА РОБОТА

на здобуття освітнього ступеня «магістр»

Галузь знань **12 Інформаційні технології**

Спеціальність **121 Інженерія програмного забезпечення**

Тема **Дослідження структур даних електронного словника української мови для задач встановлення авторства текстів**

Theme **Research of data structures of the electronic dictionary of the Ukrainian language for the tasks of determining the authorship of texts**

Керівник дипломної роботи проф. _____ Віктор ШИНКАРЕНКО

Нормоконтролер доц. _____ Олена КУРОП'ЯТНИК

Студент групи ПЗ2021 _____ Олександр КИРИЧЕНКО

Student Oleksandr KYRYCHENKO

Дніпро – 2021

Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна

Факультет Комп'ютерних технологій і систем Кафедра Комп'ютерні інформаційні технології

Спеціальність Інженерія програмного забезпечення

«ЗАТВЕРДЖУЮ»
Завідувач кафедри
_____ проф. Шинкаренко В.І.
(підпис)
«___»_____2021 р.

ЗАВДАННЯ

до дипломної роботи на здобуття ОС магістр
(освітній ступінь)

студента групи (ПЗ2021) 951-М Кириченка Олександра Олександровича
(номер групи) (ПІБ)

1 Тема дипломної роботи: Дослідження структур даних електронного словника української мови для задач встановлення авторства текстів затверджена наказом по університету від «18» листопада 2020 р. № 690ст.

2 Термін подання студентом закінченої роботи 3 грудня 2021р.

3 Вихідні дані до дипломної роботи _____
_____
_____
_____

4 Зміст пояснювальної записки (перелік питань до розробки) Призначення, постановка задачі та огляд аналогів, методика проведення дослідження, проектування і розробка інструментального програмного забезпечення, дослідження швидкодії структур даних, кластеризація текстів за морфологічними атрибутними послідовностями, охорона праці та безпека в надзвичайних ситуаціях, висновки.

5 Перелік демонстраційного матеріалу Постановка задачі, опис аналогів, методи вирішення, механізми реалізації, аналіз результатів, висновки.

6. Консультанти (з назвами розділів):

Розділ	Консультант	Підпис, дата	
		завдання видав	завдання прийняв
Техніко-економічні розрахунки	доц. Гненний М. В.	11.11.21	14.11.21
Охорона праці та безпека в надзвичайних ситуаціях	проф. Саблін О. І.		

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва розділів дипломного проекту	Термін виконання розділів проекту (роботи)	Примітка
1	Вступ	1.09.2020 – 10.09.2021	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	10.09.2020 – 15.09.2021	
3	Постановка задачі, технічне завдання	15.09.2020 – 30.09.2021	
4	Створення тестової програми	01.10.2020 – 15.10.2021	
5	Перші тестування	15.10.2020 – 22.10.2021	30%
6	Аналіз результатів	30.10.2020 – 11.11.2021	
7	Розрахунок економічних показників	11.11.2020 – 14.11.2021	
8	Охорона праці	14.11.2020 – 18.11.2021	60%
9	Оформлення пояснювальної записки	18.11.2020 – 01.12.2021	100%
10	Демонстраційні матеріали	5.11.2020 – 16.12.2021	

Дата видачі завдання «18» листопада 2020 р.

Керівник дипломного проекту _____ В.І. Шинкаренко
(підпис) (ПІБ)

Завдання прийняв до виконання _____ О.О. Кириченко
(підпис) (ПІБ)

РЕФЕРАТ

Об’єктом дослідження є процес визначення авторства природно-мовних текстів, вибір структури даних для ефективного зберігання та здійснення операцій вставки пошуку для текстових даних довільного розміру, а також структуризації словнику української мови для його компактного зберігання на жорсткому диску. Морфологічні атрибути, за якими проводитиметься аналіз авторства текстів.

Метою є дослідити можливості ефективного зберігання словника української мови з їх атрибутами, побудови швидкодіючої структури даних в оперативній пам’яті на його основі, використання послідовностей морфологічних атрибутів слів для визначення приналежності довільного тексту до певного автору.

Застосовані методи пошуку евклідової відстані між векторами, кластеризація за максимінними відстанями, статистичне зведення для аналізу обробки текстів за словником.

Вперше застосовано метод кластеризації за морфологічними атрибутами слів для розподілення текстів за авторами.

Пояснювальна записка складається зі вступу, 5 розділів, висновків, бібліографічного списку та 2 додатків:

– у вступі описується підстава для розробки, її актуальність. Складається із 4 сторінок;

– у першому розділі висвітлено аналіз сучасного стану дослідження швидкодіючих структур даних за науковими літературними джерелами, проаналізовано сучасний стану програмно-апаратного забезпечення для встановлення авторства текстів. Складається з 19 сторінок;

– у другому розділі надано обґрунтування експериментального методу дослідження, проводиться аналіз структур даних. Складається з 19 сторінок;

– у третьому розділі представлена розробка й тестування інструментального забезпечення для дослідження. Складається з 20 сторінок;

– у четвертому розділі описано виконані дослідження. Складається з 9 сторінок;

– у п‘ятому розділі розкриті питання охорони та безпеки праці. Складається з 10 сторінок;

– додатки містять технічне завдання, робочий проект, опубліковані тезиси та статтю.

Таблиць – 23, рисунків – 26, бібліографія – 63 джерела.

Ключові слова: лінгвістичний аналіз, швидкодіючі структури даних, кластерний аналіз, максимінний метод, мінімальна ідеальна хеш-функція, префіксальне дерево пошуку.

Зміст

Вступ

Дослідження у сфері розпізнавання плагіату ведуться досить давно, особливо актуальною дана проблема стала у вік інтернету і цифрових технологій. З часом кількість наукових публікацій становиться все більшою, однак частка повністю унікальних робіт невпинно зменшується. Дану проблему можна пов'язати з усебічно розвинутою сферою наукових досліджень. Усі винаходи сучасності можна сказати були вже розроблені в минулому в тому чи подібному вигляді. Крім того, на разі спостерігається підвищення комплексності робіт у зв'язку з необхідністю опрацювання великих об'ємів інформації.

Актуальність роботи. Визначення авторства тексту з метою виявлення плагіату – одна з ключових задач в сучасному світі. В епоху електронних носіїв інформації, коли все більша кількість інформації оцифровується, а вільний доступ до неї через мережу Інтернет має кожен. Однак копії оригінальних текстових робіт можуть бути недобросовісно використані третіми особами заради власних інтересів. Тому автори потребують захисту їх права на створені примірники та розповсюджені копії. Дане право на інтелектуальну власність забезпечує держава та міжнародні угоди.

У сфері текстових робіт (твори літератури, наукові дослідження, статті) немає точних засобів виявлення плагіату у зв'язку з тим, що всі люди користуються однаковими мовними одиницями – словами, для взаєморозуміння висловлених думок. Звісно, абсолютне відтворення тексту в його початковому вигляді можна відстежити, порівнявши 2 екземпляри. Однак не виключена й можливість часткової зміни оригіналу без втрати ідейного змісту шляхом заміни деяких слів синонімами або перестановкою речень і слів.

В українській мові, як і російській, можлива перестановка слів у реченні без втрати його основної думки та зі збереженням його засвоєння іншими людьми. Тому задача встановлення авторства потребує поглибленого аналізу тексту, наприклад, через атрибутивні показники, що дозволить абстрагуватися від слів та їх значень, та робити висновки виключно на їх морфологічних ознаках. Даний підхід дозволить аналізувати тексти незалежно від їх словників наявних унікальних слів, а також

відкриє можливість створення деякої узагальненої характеристики автора за його творами з подальшим включенням її до апарату аналізу та кластеризації текстів серед списку авторів.

Пропонована сфера використання програми – дослідження наукових або літературних робіт при їх публікації на наявність плагіату.

Об’єкт дослідження. Об’єктом дослідження є процес визначення авторства природно-мовних текстів, вибір структури даних для ефективного зберігання та здійснення операцій вставки та пошуку для текстових даних довільного розміру, а також структуризації словнику української мови для його компактного зберігання на жорсткому диску. Область лінгвістичного аналізу, засобами якої буде тексти розбиватимуться на морфологічні атрибути, за якими проводитиметься аналіз авторства текстів.

Предмет дослідження. Предметом дослідження є структури даних для зберігання словнику української мови. Словник української мови, що буде структурований для ефективнішого зберігання шляхом евристичного дослідження. Засобами якої тексти будуть розбиватися на морфологічні атрибути, за якими проводитиметься кластеризація текстів за відсотком співпадіння таких послідовностей атрибутів та формуватиметься висновок щодо авторства текстів.

Мета. Дослідити можливості ефективного зберігання словника української мови з їх атрибутами, побудови швидкодіючої структури даних в оперативній пам’яті на його основі, використання послідовностей морфологічних атрибутів слів для визначення приналежності довільного тексту до певного автору.

Завдання. В процесі розробки програми і документації необхідно було вирішити наступні задачі:

- дослідження та розробка методів для оптимізації структур даних для збереження українського словнику з атрибутами на жорсткому диску та в оперативній пам’яті;

- розробити програмний інструментарій для дослідження структур даних за швидкодією операцій вставки, пошуку, а також розміру в оперативній пам’яті;
- провести аналітичне порівняння розробленої структури з наявними аналогами за характеристиками розміру для зберігання її на жорсткому диску та в оперативній пам’яті, швидкодії операцій вставки, пошуку;
- виконати дослідження для розбиття вхідних текстів на морфологічні атрибути, здійснити їх кластеризацію за авторами та проаналізувати отримані результати для встановлення ефективності кластеризації розробленого програмного засобу.

Методи дослідження. Для встановлення кращої за швидкодією та зайнятим місцем в оперативній пам’яті і на диску структури проводилися емпіричні експерименти на спільних вибірках різнорідних даних. Проводилася серія тестів усіх сформованих структур на час виконання операції вставки, що досягалася заповненням їх даними словника, який будується із окремих текстових модулів під час запуску програми.

Після вибору кращої структури відбувався розбір вхідних текстів та проводилося формування n-розмірних кортежів атрибутів слів з них. Для цього було застосовано знайдення відстані між векторами за евклідовим методом, а також кластеризація методом максимінних відстаней для розподілення цих векторів по кластерам авторів. Крім того використовувалися методи математичної статистики для порівняння характеристик різних текстів з базою даних, а при розробці додатку застосовувався принцип об’єктно-орієнтованого програмування.

Наукова новизна. Розроблено структуру даних для зберігання словника української мови, що містить слова та їх атрибути та дозволяє найефективніше виконувати пошук будь-якого відомого українського слова в ньому серед усіх відомих структур даних, що доведено серією порівняльних тестів. На відміну від існуючих реалізацій призначений для української мови, а також застосовується структуризація словнику для зменшення його розміру на жорсткому диску без

втрати ключової інформації. Вперше застосовано метод кластеризації за морфологічними атрибутами слів для розподілення текстів за авторами, база даних яких наявна у програмі та може поповнюватися після кожного аналізу.

Практична значимість. Встановлення авторства текстів є добре розвинутою сферою, проте інструменту, здатного виконувати прогнозування приналежності тексту певному авторові, та що базується на основі повного словнику для україномовних текстів, досі немає. Аналіз морфологічних атрибутів слів на основі всеукраїнського словнику ВЕСУМ, що активно розвивається та поповнюється, може вирішити дану проблему. На основі отриманого програмного додатку можна проводити аналіз українських текстів на їх авторську приналежність та зробити висновки щодо унікальності текстів з точки зору структури формування речень автором. Розбір текстів на атрибутивні послідовності здійснюватиметься на основі розробленої швидкодіючої структури даних, що є кращою серед усіх відомих аналогів.

Апробація результатів дослідження. Виконано доповідь на міжнародній конференції Computer science and informational technologies 2021 [1], XIV Міжнародній науково-практичній конференції «Сучасні інформаційні та комунікаційні технології на транспорті, промисловості та освіті», Всеукраїнській науково-технічній конференції молодих учених, магістрантів та студентів «Науково-технічний прогрес на транспорті». Результати дослідницької роботи доповідались на семінарах кафедри КІТ 22.02.2021 р. та 09.12.2021 р.

Публікації за темою роботи. Підготовлено доповідь «Processing Words Effectiveness Analysis in Solving the Natural Language Texts Authorship Determination Task» у співавторстві з Демидович І. М., Куроп’ятник О. С., науковий керівник Шинкаренко В. І. для CSIT 2021 [1], що індексується у Scopus.

1 АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖЕННЯ ПРОБЛЕМИ ЗА НАУКОВИМИ ЛІТЕРАТУРНИМИ ДЖЕРЕЛАМИ

1.1 Призначення та область застосування

Встановлення авторства текстів є важливою складовою захисту авторського права та протидії використання плагіату. Інтелектуальна власність невіддільна від автора та не може бути передана іншій особі в повному обсязі. Ідеї завжди залишаться у їх творця, до того ж аутентичний стиль автора супроводжує його протягом всього життя, хоча й може зазнати певного впливу з набуттям досвіду. Таким чином можна виконувати аналіз належності текстів до того чи іншого автора за опосередкованим аналізом його наявних робіт.

Необхідність виявлення плагіату полягає у можливості ідентифікувати всі джерела, що хоч якось без відома автора використовують його інтелектуальну власність у вигляді текстової інформації на свою користь. Таке зловживання відкритістю інформації негативно впливає на загальний розвиток суспільства. Через злочинне привласнення чужої праці без затрат на це часу, людських ресурсів чи грошей плагіатори збагачуються, в той самий час автор залишається ні з чим. Це знецінює будь-які ідейні пориви, а творча діяльність залишається без стимулу для розвитку. В таких випадках прогрес у державі сповільнюється та спостерігається відплив інтелектуальних ресурсів.

Для попередження цього ще з давніх часів правителями усіляко заохочувалася винахідницька та творча праця, видавалися особисті грамоти, що давали право на розпорядження та впровадження винаходів у промисловість. Крім того обширно застосовувалися торгові знаки, авторські підписи, що виступали як рекламою для якісного продукту, так і гарантією його автентичності.

Підставою для розробки нового програмного засобу аналізу авторської належності викликана відсутністю на українському ринку повних аналогів для вільного користування, крім цього розроблений додаток можна буде інтегрувати в більшу систему з ширшим діапазоном дослідження текстів (наприклад, аналіз структури речень, словнику тексту).

Сфера застосування інструменту по виявленню плагіату обширна. Такого аналізу потребують усі наукові роботи, твори літератури, видання у засобах масової інформації, публіцистичні статті на різноманітних інтернет ресурсах та інших джерелах, що подають інформацію у відкритому доступі. Розроблений додаток легковагий та не потребує значних обчислювальних засобів і може бути інтегрований в наявні системи аналізу плагіату або стати сервісом з відкритим інтерфейсом взаємодії, що дозволить через клієнт-серверний ресурс проводити аналіз текстів дистанційно.

1.2 Постановка задачі

Розробка модулю побудови оптимальної структури даних, що буде зберігати словник слів української мови з їх морфологічними атрибутами, для найбільш оптимальної комбінації наступних характеристик:

- швидкодія операцій заповнення словнику та пошук у ньому довільних даних;
- групування однокореневих слів, до яких входять всі словоформи, утворені додаванням нової частки слова (префіксу, суфіксу, закінчення чи зміна форми слова через випадання чи чергування звуків при відмінюванні) з присвоєнням всім їм єдиного індексу основи;
- розмір, що буде займати кінцевий словник як на жорсткому диску, так і в оперативній пам’яті комп’ютера.

Розробка модулю дослідження авторського стилю за послідовностями наборів морфологічних атрибутів слів у тексті. Даний модуль використовуватиме попередньо побудований словник слів і атрибутів української мови. Аналіз проводитиметься на основі обробки відомих текстів авторів, після чого виконати кластерний аналіз вхідних текстів за наявним списком авторів.

Програма не передбачає відсіювання текстів за слабою приналежністю до наявних у базі даних текстів, будь-який текст буде віднесено до певного автора, навіть якщо справжнього автора у базі даних немає і орієнтується на чисельні показники близькості двох векторів атрибутів у n-вимірному просторі, число вимірів

якого є вибіркою 20% найчастіше вживаних послідовностей з 2-х атрибутів з кожного тексту. Дані вектори нормалізуються з метою можливості їх порівняння між собою. Для цього до кожного вектору додаються 0 у місця відсутніх атрибутів, що є в інших текстах. Крім цього, кожен показник кількості атрибутів у векторі ділиться на кількість слів у відповідному до цього вектору тексті, що дозволяє представити атрибути у відсотковій частці.

Результати кожної із структур порівнюватимуться в єдиній системі таким чином, щоб за трьома отриманими показниками (швидкодія операції вставки, пошуку та розмір структури в оперативній пам’яті) визначити кращу структуру, при чому показнику швидкодії пошуку надається пріоритет.

Послідовності атрибутів повинні будуватися наступним чином: починаючи з першого слова елемент на позиції i з i+1, i+1 з i+2…n-1 з n, де n-кількість слів у тексті, i є [1;n]. Всього послідовностей буде утворено n-1.

Для користувача програма буде надавати наступні можливості:

- вибір початкового словнику української мови, що буде основою для побудови структур даних;
- можливість аналізу будь-яких україномовних текстів, відкидаючи невідповідні побудованому словнику слова без втрати функціональної спроможності;
- вивід часових характеристик обробки програмою даних на кожному з кроків;
- збереження попередньо обробленого словнику у вигляді набору текстових фалів;
- збереження результатів кластеризації у текстовий файл.

1.3 Огляд літературних джерел

1.3.1 Огляд аналогів швидкодіючих структур даних

Однією з кращих структур для задач, що не потребують частої модифікації бази даних, та повинні бути найшвидшими в операції пошуку є хеш-таблиці. Хеш-таблиця – така структура даних, що реалізовує принцип асоціативного масиву, тобто

зберігає пари даних у вигляді (ключ; значення), а також дозволяє виконувати операції додавання нової пари, зміну значення за ключем та видалення пари.

Існують два базових варіанти хеш-таблиць:

- з відкритою індексацією (структура містить деякий масив даних H, елементами якого є пари (ключ; значення));
- з ланцюгами пар (масив даних H складається з множини списків пар (ключ; значення)).

Перший варіант, безсумнівно, має вищу швидкість при пошуку даних, що завжди буде рівною O(1), адже кожен елемент «значення» знаходиться в унікальній комірці пам’яті, доступ до якої ми можемо отримати за ключем, перейшовши за адресою комірки, що вираховується за ключем в один крок.

Однак, складності виникають при заповненні такої структури даних. Якщо взяти зв’язок між ключем та його репрезентацією у вигляді адреси пам’яті, то легко помітити, що застосування допоміжного механізму перетворення ключа в адресу неминуче. Є багато алгоритмів для здійснення подібних перетворень, однак жоден з них не може гарантувати, що на необмеженому просторі даних не знайдеться два різних ключа таких, що в кінцевому результаті будуть посилатися на одне й те ж значення. Така ситуація носить назву колізія. Для прикладу, ймовірність виникнення колізії при вставці в хеш-таблицю розміром 365 комірок лише 23 елементів перевищую 50% (за умови, що кожен елемент з рівною ймовірністю може потрапити в будь-яку комірку). Даний випадок розглянуто як парадокс днів народження [6]. Тому для вирішення колізій застосовують спеціальні механізми, що в свою чергу потребують додаткових ресурсів на виконання операції вставки.

Як один із варіантів вирішення колізій, що задовольняє умовам задачі даного проекту, є ідеальна хеш-функція, що буде розглянута далі.

Другий варіант вирішує дану проблему з колізіями шляхом влаштування комірок, на які посилаються декілька ключів, таким чином, щоб окрім значень в них зберігалася адреса наступної такої комірки. В результаті буде отримано ланцюжки комірок, а при виконанні операції пошуку за ключем дана хеш-таблиця буде

гарантувати вирішення колізій завдяки подвійному пошуку: за первинною адресою комірки, а також всередині ланцюга комірок зі значеннями.

Переваги даного варіанта заключаються в простому вирішенні проблем колізій, що дозволяє на необмеженому масиві пар ключів і значень організувати хеш-таблицю. Однак недоліком такого підходу є його сповільнення при операції пошуку, що неспинно буде прогресувати зі збільшенням об'єму даних у структурі. В гіршому випадку ми отримаємо ситуацію, коли хеш-таблиця перетвориться у довільну послідовність даних, коли всі ключі посилатимуться на єдину комірку списків значень. Тоді операція пошуку буде ідентична лінійному пошуку.

1.3.2 Аналіз мінімальної ідеальної хеш-функції

1.3.2.1 Принцип дії мінімальної ідеальної хеш-функції

Припустимо U - це всесвіт ключів. Нехай h: U→M буде хеш-функцією, яка відображає ключі з U до заданого інтервалу цілих чисел $M = [0, m-1] = \{0,1,\dots, m-1\}$.

Нехай $S \subseteq U$ це набір ключів n від U. За допомогою ключа x є S хеш-функція h обчислює ціле число в [0,m-1] для зберігання або отримання x в хеш-таблиці. Методи хешування для нестатичних наборів ключів можна використовувати для побудови структур даних, що зберігають S і підтримують запити на перевірку наявності "x є S?" в очікуваний час O(1). Однак вони включають певну кількість не витраченого простору через невикористані місця в таблиці та потребують додаткових витрат часу на вирішення колізій, коли два ключі хешуються в одну й ту ж комірку в таблиці.

Для статичних наборів ключів можна обчислити функцію, щоб знайти будь-який ключ у таблиці за один крок; такі хеш-функції називаються ідеальними (perfect hash function – PHF). Точніше, маючи набір ключів S, ми будемо говорити, що хеш-функція h: U→M є ідеальною хеш-функцією для S, якщо h є прямим відображенням на S, тобто немає колізій між ключами в S: якщо x і y знаходяться в S і x≠y, тоді h(x) ≠ h(y). На рис. 1.1 представлено ідеальну хеш-функцію. Оскільки колізій не відбувається, кожен ключ можна отримати з таблиці за один крок. Якщо m = n,

тобто таблиця має той самий розмір, що і S, то ми говоримо, що h це мінімальна ідеальна хеш-функція для S. Рис. 1.2 ілюструє мінімальну ідеальну хеш-функцію. Мінімальна ідеальна хеш-функція повністю вирішує проблеми втрати місця та часу. Ідеальна хеш-функція h є такою, що зберігає порядок, якщо ключі впорядковані в певному заданому порядку і зберігають цей порядок у хеш-таблиці.

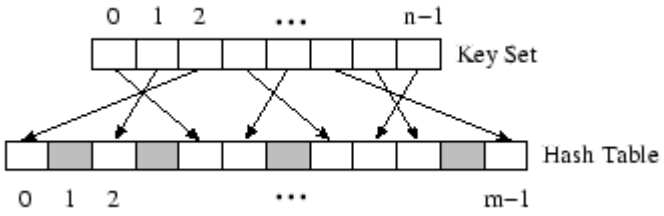


Рисунок 1.1 – Ідеальна хеш-функція

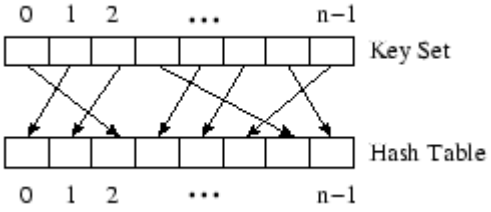


Рисунок 1.2 – Мінімальна ідеальна хеш-функція

Мінімальні ідеальні хеш-функції широко використовуються для ефективного зберігання пам'яті та швидкого отримання елементів із статичних наборів, таких як множина слів природних мов, зарезервовані слова в мовах програмування або інтерактивних системах, універсальні адреси ресурсів (URL) у веб-пошукових системах або набори елементів у інтелектуальному аналізі даних.

1.3.2.2 Переваги бібліотеки мінімальної ідеальної хеш-функції

Ідеальна хеш-функція відображає статичний набір з n ключів у набір з m цілих чисел без колізій, де m більше або дорівнює n. Якщо m дорівнює n, функція називається мінімальною.

Мінімальні ідеальні хеш-функції широко використовуються для ефективного зберігання пам'яті та швидкого отримання елементів із статичних наборів, таких як множина слів природних мов, зарезервовані слова в мовах програмування або інтерактивних системах, універсальні адреси ресурсів (URL) у системах веб-пошуку або набори елементів у інтелектуальному аналізі даних. Тому вони широко

застосовуються в інформаційно-пошукових системах, системах управління базами даних, мовного перекладу, електронної комерції, компіляторах, операційних системах тощо.

Використання мінімальних ідеальних хеш-функцій поки що обмежувалося сценаріями, коли набір хешованих ключів невеликий через обмеження існуючих алгоритмів. Але в реаліях сучасного світу, де об’єми збирання та обробки інформації невпинно ростуть, саме робота з масивним набором ключів є вирішальною. Отже, проект CMPH [7] надає спільноті безкоштовне програмне забезпечення у вигляді відкритого API, який працюватиме з наборами ключів, що близький до мільярда.

Напевно, найцікавішим додатком для мінімально досконалих хеш-функцій є їх використання як структури індексації для баз даних. Найпопулярнішою структурою даних, яка використовується як структура індексації в базах даних, є дерево B+. Насправді, дерево B+ популярне через свої широкі можливості для динамічних додатків з частими операціями вставки та видалення записів. Однак для додатків зі спорадичними модифікаціями та величезною кількістю запитів дерево B+ не є найкращим варіантом, оскільки практичне розгортання цієї структури надзвичайно складне і погано адаптоване для великих наборів ключів, що відіграє найважливішу роль в сучасних додатках.

Наприклад, у сфері пошуку інформації робота з величезними колекціями є щоденним завданням. Просте призначення ідентифікаторів веб-сторінкам колекції може бути складним завданням. Обмеження обробки трафіку сучасних баз даних спричинені переповненням оперативної пам’яті робочим набором URL-сторінок, що вирішується кращим влаштуванням даних у пам’яті за допомогою мінімальних ідеальних хеш-функцій, які можуть легко масштабуватися до сотень мільйонів записів, використовуючи стандартне обладнання.

Оскільки існує багато додатків, для яких питання швидкодії та зайнятого об’єму пам’яті є вирішальним, важливо реалізувати ефективні за пам’яттю та часом алгоритми для побудови мінімально досконалих хеш-функцій. Відсутність подібних бібліотек у світі вільного програмного забезпечення стало основною мотивацією для

створення бібліотеки C Minimal Perfect Hashing Library (gperf [8] дещо інший, оскільки він був задуманий для дуже швидкого створення ідеальних хеш-функцій для невеликих наборів ключів, а бібліотека CMPH була задумана для створення мінімальних ідеальних хеш-функцій для дуже великих наборів ключів). C Minimal Perfect Hashing Library — це портативна бібліотека LGPL [9] для створення та роботи з дуже ефективними мінімальними ідеальними хеш-функціями.

Бібліотека CMPH включає новітні та найефективніші алгоритми в простий у використанні, якісний і швидкий API. Бібліотека була розроблена для роботи з великими записами, які не поміщаються в оперативній пам'яті. Її успішно використовували для побудови мінімальних ідеальних хеш-функцій для наборів із понад 100 мільйонами ключів, і ми маємо намір збільшити це число до порядку мільярдів ключів. Хоча подібних бібліотек бракує, розробники відзначають наступні відмінні риси бібліотеки CMPH:

- швидкий пошук;
- ефективне використання місця з ретельно задокументованим використанням основної пам’яті;
- доступні найкращі серед сучасних алгоритмів генерації хеш-функцій;
- працює з наборами ключів на диску через використання шаблону адаптера;
- серіалізація хеш-функцій;
- портативний код C (наразі працює на GNU/Linux і WIN32 і, як повідомляється, працює в OpenBSD і Solaris);
- об'єктно-орієнтована реалізація;
- легко розширюваний;
- добре інкапсульований API, спрямований на бінарну сумісність серед різних версій;
- безкоштовне програмне забезпечення.

Серед шести реалізованих алгоритмів генерації хеш-функцій були протестовані всі, що задовольняли умови задачі побудови словника (максимально

можливий розмір ключа більший за 4 біти, тому FCH не тестувався). Найкращий результат, як і зазначалося розробниками бібліотеки, був досягнутий з використанням алгоритму Compress, hash and dispace – CHD [10].

1.3.2.3 Особливості алгоритму Compress, hash and dispace

Це найшвидший алгоритм для створення PHF і MPHF за лінійний час.

Головними параметрами продуктивності PHF є розмір представлення, час оцінки та час побудови. Розмір представлення відіграє важливу роль, коли вся функція поміщається в більш швидку пам’ять, а фактичні дані зберігаються в повільнішій пам’яті. Наприклад, компактні PHF можна повністю помістити в кеш ЦП, і це робить їх обчислення дуже швидкими, уникаючи промахів кешу – ситуації, коли завантажений блок даних в кеш для обчислень не має всіх необхідних для обчислень даних. У цьому контексті важливу роль відіграє алгоритм CHD. Його розробили Джамал Белацзугі, Фабіано К. Ботельо та Мартін Діцфельбінгер та представили на конференції [10].

Алгоритм CHD дозволяє отримувати PHF з розміром представлення, дуже близьким до оптимального, зберігаючи час побудови O(n) і час оцінки O(1). Наприклад, у випадку m=2n ми отримуємо PHF, яка використовує простір 0,67 біт на ключ, а для m=1,23n ми отримуємо простір 1,4 біта на ключ, що не було досягнуто раніше відомими методами. Алгоритм CHD створений на основі кількох відомих алгоритмів; основна нова особливість полягає в тому, що він поєднує модифікацію підходу, розробленого Пагом Р. [11] – "hash-and-displace" зі стисненням даних на послідовності індексів хеш-функції. Ця комбінація дозволяє значно зменшити використання простору, зберігаючи лінійний час побудови та постійний час запитів.

Алгоритм CHD також можна використовувати для k-ідеального хешування, де щонайбільше k ключів може бути зіставлено з тим самим значенням. Компактні PHF, створені алгоритмом CHD, можна використовувати в багатьох програмах, у яких ми хочемо призначити унікальний ідентифікатор кожному ключу без збереження будь-якої інформації про ключ. Одне з найбільш очевидних застосувань

цих функцій (або k-ідеальних хеш-функцій) — це коли у нас є невелика швидка пам’ять, в якій ми можемо зберігати ідеальну хеш-функцію, тоді як ключі та пов’язані з ними дані значень зберігаються в повільнішій, але більшій пам’яті. Розмір блоку або одиниці передачі можна вибрати таким чином, щоб k елементів даних можна було отримати за один доступ до читання. У цьому випадку ми можемо ґарантувати, що дані, пов’язані з ключем, можуть бути отримані в один крок у повільнішу пам’ять. Це використовувалося, наприклад, в апаратних маршрутизаторах [12].

Алгоритм CHD генерує найкомпактніші PHF та MPHF серед відомих наразі за O(n) час. Час, необхідний для оцінки згенерованих функцій, є постійним (на практиці менше 1,4 мікросекунд). Простір для зберігання отриманих PHF і MPHF віддалений від нижньої межі теоретичної інформації на фактор у 1,43 рази. Найближчим конкурентом є алгоритм Мартіна і Пага [13], але їх алгоритм не ґарантує виконання за лінійний час. Крім того, алгоритм CHD можна налаштувати так, щоб він працював швидше, ніж алгоритм BPZ [11] (найшвидший алгоритм, доступний у літературі на даний момент) і з результуючими функціями, що компактніші за розміром.

1.3.3 Аналіз префіксального дерева пошуку

1.3.3.1 Принцип дії префіксального дерева пошуку

В англійській мові trie походить від retrieval – операції отримання даних за певним критерієм, інакше кажучи пошуку. Їх також називають деревами цифрового або префіксального пошуку [15]. Trie пов’язані з Most-significant-first (найважливіші перші, далі MSF) сортуванням, як бінарні дерева пошуку з швидким сортуванням. У структуру trie літери розташовуються у вузлах графу від кореню таким чином, щоб при здійсненні обходу по дереву з верхнього рівня можна було прочитати всі слова, обходячи в певному порядку дочірні вузли. Так, кожен вузол матиме масив дочірніх вузлів розміром r (наприклад, r = 26, 128 або 256). Для зберігання рядку, ви просто здійснюєте спуск по дерево за асоційованими дочірніми вузлами для кожної літери в рядку від першої до останньої.

Наприклад, скажімо, що ми хотіли зберегти слова: and, bat, add, bag, cat, car. Для заповнення необхідної структури нам знадобиться визначити стоп-символ, який буде однозначно вказувати на успішне знайдення слова у дереві, а також відсіче можливі випадки хибного прийняття слова, що є частиною інших слів (наприклад, bat може бути частиною слова battery, battle і т. д.). Нехай даним символом стане $. Обходячи дерево по рівням літера за літерою ми переходимо на рівень нижче, якщо зустріли в даному рівні шукану літеру за порядком у слові, якщо в кінці потрапимо на комірку з символом $, то слово знайдено, якщо ж вказівник на наступний вузол буде пустим, то шуканого слова в дереві немає. Приклад побудованого дерева зображено на рис. 1.3.

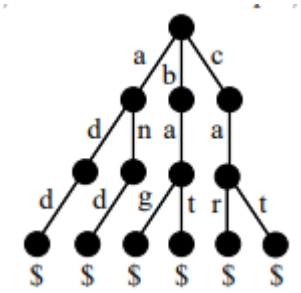


Рисунок 1.3 – Приклад префіксального дерева пошуку

1.3.3.2 Алгоритми покращення префіксального дерева пошуку

Час виконання пошуку становить лише O (довжина ключа) [14]. Те саме стосується виконання вставки, якщо ми розглядаємо r як константу. (Якщо r дійсно велике, тоді слід турбуватися про час, витрачений на виділення масивів дочірніх вказівників). Крім того, пошук за префіксами особливо простий (наприклад, якщо ви хочете створити текстовий редактор, який виконував би доповнення слів). Основним недоліком є високі накладні витрати на перехід за вказівником, кількість яких прирівнюється довжині ключа.

Наприклад, якщо ми додамо слово «automobile» до вищевказаного trie, кількість додаткових рівнів і відповідно переходів при зберіганні правила «одна літера на вузол» стане надмірною для даної задачі. Крім того, необхідно

враховувати фактор виділення r додаткового простору, оскільки потрібно мати масив розміру r у кожному вузлі. Проте концептуально дизайн структури настільки гарний, що були розроблені чисельні модифікації для його практичного покращення. Зокрема, можна зробити наступне:

- стиснути контури, які не розгалужуються в одне ребро;
- розділяти слово лише тоді, коли це необхідно.

Приклад модифікованого раніше описаного дерева з використанням даних вказівок зображений на рис. 1.4.

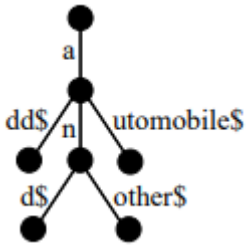


Рисунок 1.4 – Приклад префіксального дерева пошуку

1.3.4 Постановка задачі кластеризації

Задача ідентифікації авторства тексту – це задача встановлення авторства невідомого тексту за допомогою виділення особливостей авторського стилю та порівняння цих особливостей з іншими творами, авторство яких відомо. Такі особливості тексту носять назву авторський інваріант [2].

Авторський інваріант – це кількісна характеристика літературних текстів або певний параметр, який однозначно показує приналежність деякого тексту до автора або групи близьких за стилем написання авторів, та приймає суттєво відмінні значення для творів груп різних авторів.

Постановка задачі ідентифікації тексту [3] має наступний вигляд:

$$T = \{t_1,\ldots,t_k\} \text{ – множина текстів,}$$

$$A = \{\} \text{ – множина авторів.} \quad (1)$$

Для деякої підмножини текстів T` = { $t_1,\ldots,t_m$} $\subseteq$T автори відомі, тобто існує множина пар текстів «текст-автор» D = $\{(t_i, a_j)\}_{i=1}^{m}$. Необхідно встановити, хто із множини A є істинним автором для решти текстів (анонімних або спірних) T`` = = { $t_{m+1},\ldots,t_k$}$\subseteq$T.

Методика визначення авторства [3] включає послідовність наступних дій:

- вибір моделі представлення текстів як наборів ознак;
- вибір групи ознак для перевірки та формування з неї авторського інваріанту;
- вибір класифікаторів та їх параметрів;
- формування моделі авторського стилю, що дозволяє розділяти двох і більше авторів на основі отриманого авторського інваріанту та навченого класифікатора;
- безпосереднє визначення авторства невідомого тексту;
- прийняття підсумкового рішення автора тексту за рядом класифікаторів у разі, якщо вдалося знайти кілька інформативних груп ознак тексту.

Методи встановлення авторської приналежності, що ґрунтуються на підрахунку певних характеристик тексту, також різняться за способами підрахунку отриманих частот. Найбільш поширеними є наступні методи:

- міра «хі-квадрат»;
- інформаційна ентропія;
- міра Кульбака-Лейблера;
- інформація Фішера.

Крім того часто застосовують порівняння проекцій текстів у вигляді відстані між певними характеристиками текстів (послідовностями слів, атрибутів) за допомогою знайдення евклідової відстані між ними, косинусної подібності, тощо.

Ще одним підходом у пошуку подібності текстів є індекс Жаккара [4], також відомий як коефіцієнт пересічення за об’єднанням. Він є статистикою, що використовується для оцінки схожості та різності між вибірками наборів.

Визначається індекс як розмір пересічення, поділений на розмір об’єднання двох вибірок:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \quad (2)$$

Відстань Жаккара, що визначає різність двох вибірок, є доповненням до коефіцієнту Жаккара і визначається шляхом віднімання коефіцієнта від одиниці.

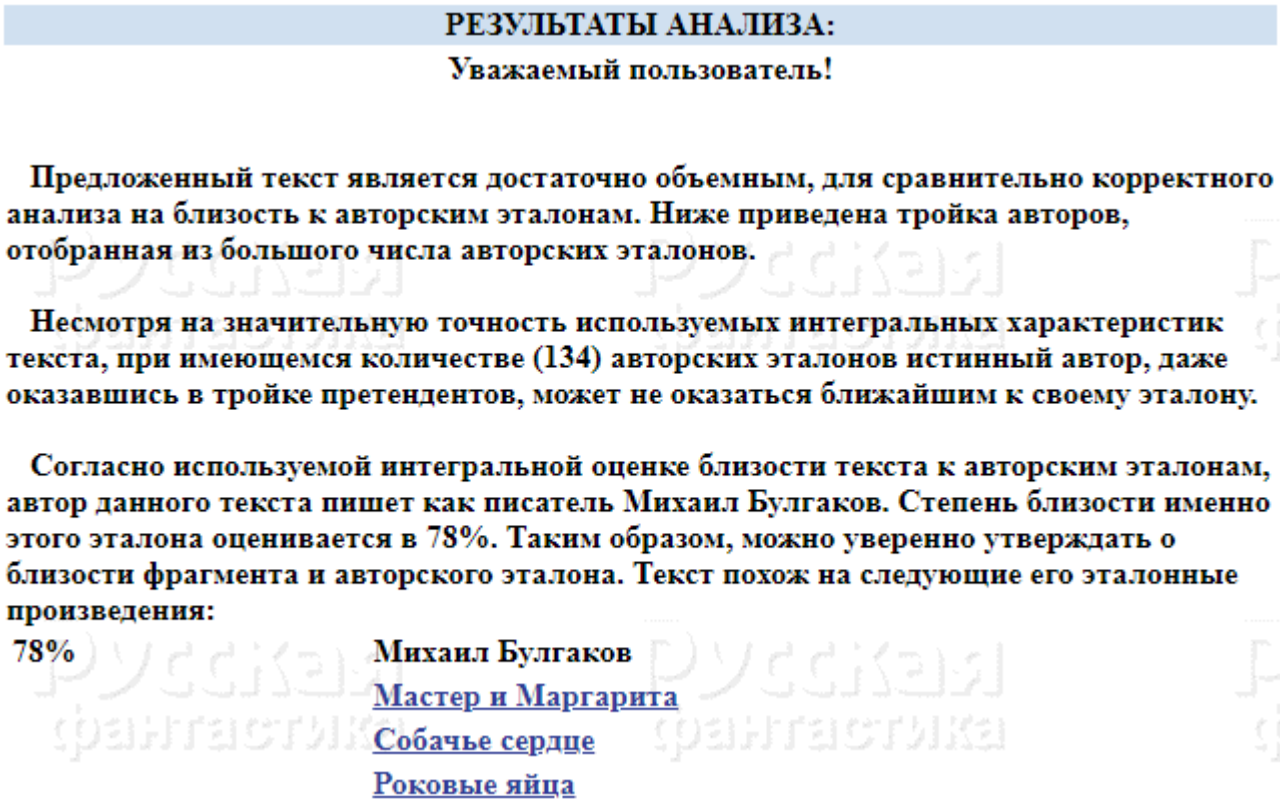
Аналіз наявних методів визначення авторства невідомих текстів показав, що досі не існує універсального рішення для стабільного і правдоподібного результату. Більшості алгоритмів необхідна обширна вибірка текстів, що дозволить знайти певні закономірності авторського стилю та застосувати отримані дані при співставленні з невідомим текстом. Однак і досі жоден з них не був застосований на таких об’ємах даних, що дозволяли б ґарантувати знайдення автору невідомого тексту.

1.4 Аналіз програмних аналогів «Лингвоанализатор» і «Штампомер»

В результаті пошуку програмних аналогів для аналізу текстів, написаних українською мовою, не було знайдено засобів у відкритому доступі. Однак в близькій до неї російській мові подібні додатки існують.

В іноземних системах встановлення авторства широко використовуються методи із теорії математичної статистики, розпізнавання образів, теорії ймовірності, алгоритми кластерного аналізу, нейронних мереж тощо.

Для детального аналізу було взято веб-ресурс «Лингвоанализатор» [17], що використовує принцип аналізу текстів через їх порівняння з авторськими еталонами – певною бібліотекою популярних російських творів, та формує деякі статистичні показники: відсоток схожості до низки авторів та їх відповідні твори, за якими було сформовано оцінку схожості. Методи, що використовуються додатком, наступні: марковські ланцюги, інформаційна ентропія. Співставлення текстів відбувається через порівняння близькості за частотною характеристикою графем (елементарних одиниць писемності, в даному випадку літер і знаків пунктуації) у текстах. Приклад роботи з завантаженим на ресурс фрагментом роману Михайла Булгакова «Майстер і Маргарита» на рис. 1.5.



РЕЗУЛЬТАТЫ АНАЛИЗА:

Уважаемый пользователь!

Предложенный текст является достаточно объемным, для сравнительно корректного анализа на близость к авторским эталонам. Ниже приведена тройка авторов, отобранная из большого числа авторских эталонов.

Несмотря на значительную точность используемых интегральных характеристик текста, при имеющемся количестве (134) авторских эталонов истинный автор, даже оказавшись в тройке претендентов, может не оказаться ближайшим к своему эталону.

Согласно используемой интегральной оценке близости текста к авторским эталонам, автор данного текста пишет как писатель Михаил Булгаков. Степень близости именно этого эталона оценивается в 78%. Таким образом, можно уверенно утверждать о близости фрагмента и авторского эталона. Текст похож на следующие его эталонные произведения:

78% **Михаил Булгаков**
Мастер и Маргарита
Собачье сердце
Роковые яйца

Рисунок 1.5 – Результат роботи сервісу «Лингвоанализатор»

Наступним програмним додатком для аналізу стане «Штампомер» [16]. Зустрічає програма користувача діалоговим вікном, що містить такі поля для заповнення:

- вхідний текст – введіть повне ім’я файлу або оберіть за допомогою інструменту перегляду всіх файлів у системі через Провідник;
- автор – вкажіть автора аналізованого тексту;
- твір – вкажіть назву твору;
- результати аналізу – вкажіть повне ім'я файлу у HTML форматі, до якого програма запише результат аналізу;
- викликати – вкажіть у рядку шлях до виконавчого файлу браузера, в якому здійснюватиметься перегляд;
- текст без розбиття на рядки/текст з розбиттям – якщо аналізований текст містить символи обриву рядка та (або) повернення каретки тільки для виділення нових абзаців, вкажіть перший пункт - "Текст без розбиття на

рядки". Якщо такі символи є в кінці кожного рядка, виберіть другий пункт - "Текст з розбиттям".

Вхідний файл не повинен містити високорівневого форматування (не підтримуються формати, подібні до ".doc" MS-Word. Якщо ж авторський текст у файлі формату ".doc", то його можна зберегти без форматування: меню "Файл" -> "Зберегти як..." та вказати тип файлу: "Текст DOS" (кодування CP866). У списку, що випадає, необхідно визначити кодування, в якому записаний аналізований текст. Це може бути: CP866 – кодування DOS для кирилиці, CP1251 – кодування Windows для кирилиці, KOI8-R – альтернативне кодування для кирилиці, будь-яке інше кодування, що підтримується встановленим Java стандартом.

У ході аналізу програма збирає різні статистичні дані та записує їх у файл результатів у вигляді таблиць відносин або процентних показників. У файлі наявні наступні характеристики: загальні дані, зміст розділових знаків, зміст завершальних розділових знаків, зміст речень в абзаці, зміст слів у реченні, зміст розділових знаків у реченні. Також записані таблиці аналізу штампів: повторення штампів n-го рівня, повторення штампів n-го рівня в одному абзаці, повторення штампів n-го рівня в одному реченні, де під штампом n-го рівня розуміється словосполучення із n слів. Тобто штамп 1-го рівня – це одне слово, а 5-го рівня – словосполучення з 5 слів.

На рис. 1.6 зображене діалогове вікно програми для аналізу тексту.

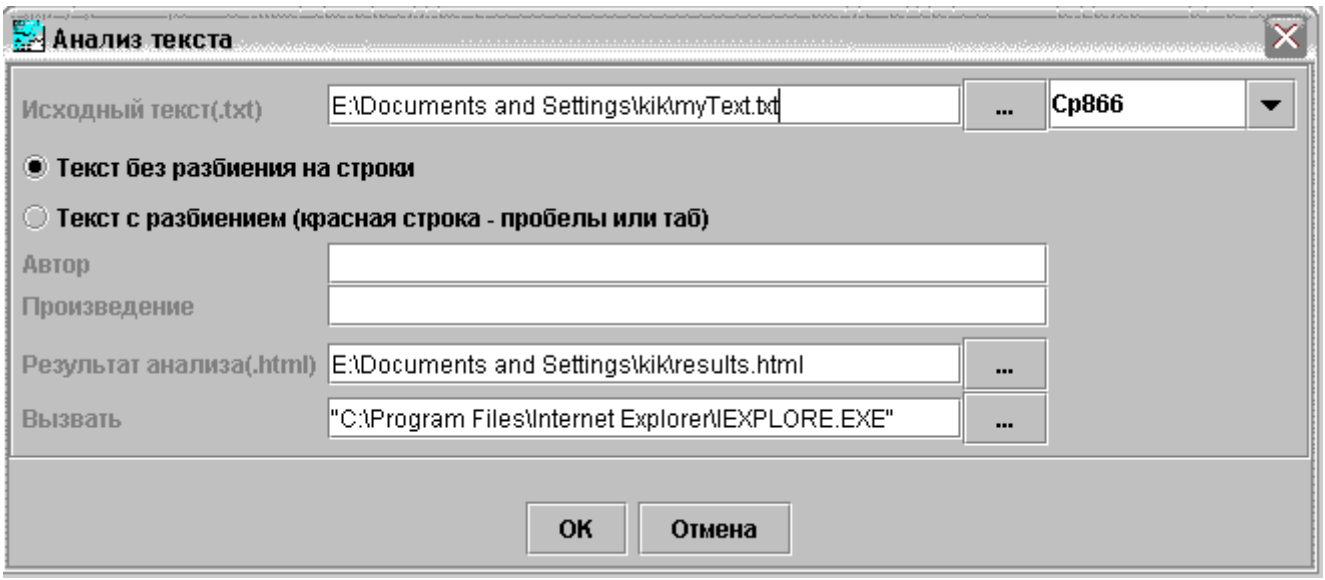


Рисунок 1.6 – Діалогове вікно аналізу для «Штампомер»

Друге діалогове вікно дозволяє заповнити дані для проведення порівняльного аналізу статистик текстів, щоб визначити подібність цих текстів. Для кожного тексту потрібно сформувати свій файл статистики. Після цього необхідно заповнити наступні поля:

- перший файл статистики, другий файл статистики – визначте повні імена файлів, що містять результати статистичного аналізу авторських текстів, що порівнюються;
- результати порівняння – вкажіть повне ім'я файлу у HTML форматі, в який програма запише результат порівняльного аналізу;
- викликати – вкажіть у рядку шлях до виконавчого файлу браузера.

У результаті порівняльного аналізу програма обчислює виражені у відсотках різниці відповідних таблиць даних, сформованих на кожен текст окремо на етапі статистичного аналізу текстів. Такі дані характеризують відмінність таблиць за числовими показниками, і що вище їх значення, то нижче ймовірність ідентичності авторства вихідних текстів. Діалогове вікно для порівняльного аналізу представлено на рис. 1.7.

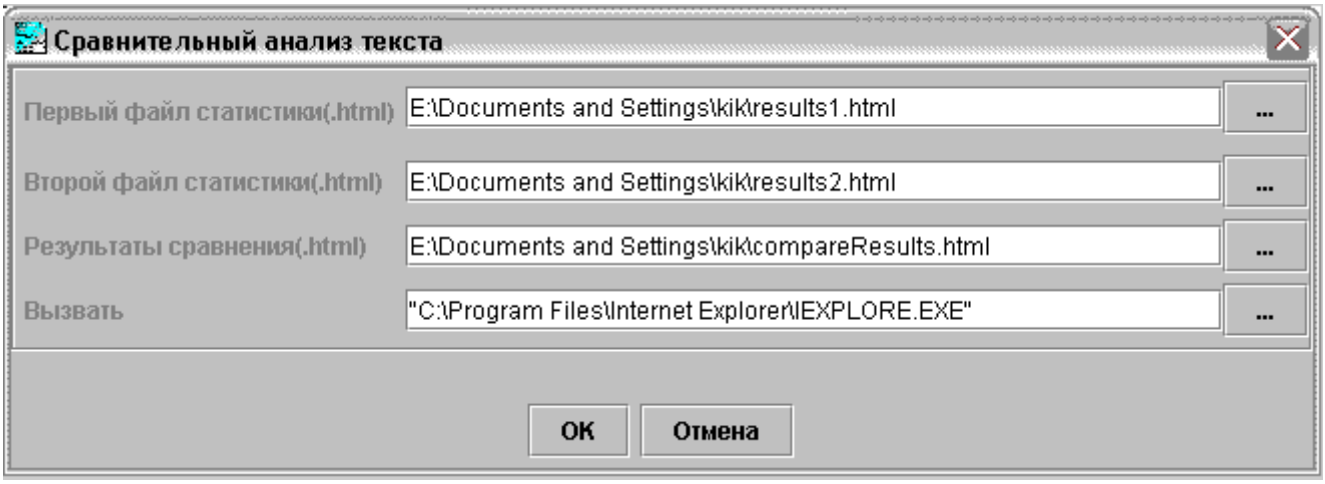


Рисунок 1.7 – Діалогове вікно порівняльного аналізу текстів через «Штампомер»

Зведені характеристики і можливості розглянутих вище програмних засобів у порівнянні з розроблюваним у даній праці програмним додатком представлені у табл. 1.1.

Таблиця 1.1 – Порівняння функціональних можливостей аналогів

Функціональна можливість	Наявність у програмі		
	Authorship Clusterizer	Лингвоанализатор	Штампомер
Розрахунок часової ефективності алгоритму	+	-	-
Можливість завантаження нових еталонів текстів	+	-	+
Можливість завантаження нового словнику	+	-	-
Мова аналізу текстів	українська	російська	будь-яка
Виведення статистичних показників схожості за декількома авторами	-	+	+
Ключові атрибути для формування оцінки схожості	послідовності з 2-х морфологічних атрибутів слів	число службових слів, словник морфем, складність граматичних конструкцій, словник автора	таблиці статистики граматичних атрибутів тексту
Використані структури даних	мінімальна ідеальна хеш-таблиця	словник (хеш-таблиця)	список
Об’єм тестових даних, на якому програми проходили перевірку достовірності оцінок	9 авторів, 5МБ	80 авторів, 128МБ	20 авторів, 10МБ
Вид програмного додатку	програма для Windows (.exe)	веб-ресурс	програма для Windows (.exe), Java-додаток (.jar)
Ведення обліку вгаданих творів	-	+	-
Діапазон розміру тексту для аналізу	[0; ∞]	[50;300] КБ	[0;25] МБ

Висновки до першого розділу

Аналіз програмного забезпечення для встановлення авторства на ринку свідчить про недостатньо розвиненість практичних аналогів, особливо для української мови. Прив’язка до мови важлива через різну інтерпретацію деяких частин мови в різних мовах, крім того словник кожної мови має свої унікальні властивості, що унеможливлює створення універсального рішення для всіх мов. Розглянуті аналоги використовують словник російської мови, а отже не відповідають вимогам задачі. До того ж жоден з них не проводить аналіз творів авторів за розглянутими у цій роботі морфологічними характеристиками, а обсяги тестування даних досить малі, щоб стверджувати про надійне та правдоподібне визначення авторства на вибірці всіх можливих існуючих писемних робіт.

Розглянуті дослідження у сфері швидкодіючих структур даних мають рішення, що задовольняють умовам задачі, але потребують інтеграції та модифікації для оптимальної роботи з розроблюваним словником. Тому варто виконувати експерименти з порівняння їх роботи вже після пристосування кожної структури до реальних даних.

2 РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ ДЛЯ ДОСЛІДЖЕННЯ СТРУКТУРИЗАЦІЇ СЛОВНИКУ УКРАЇНСЬКОЇ МОВИ В ПАМ'ЯТІ ТА КЛАСТЕРИЗАЦІЇ ТЕКСТІВ НА ОСНОВІ МОРФОЛОГІЧНИХ АТРИБУТІВ

2.1 Внутрішнє проектування

2.1.1 Опис функціональних характеристик

Програмний комплекс «AuthorshipClusterizer» виконує кластеризацію текстів української мови за авторами, враховуючи подібність використання порядку слів у тексті, аналізуючи морфологічні атрибути. Модулі програми написані на мові C#, для зберігання проміжних файлів з частинами слів та атрибутами застосовується формат файлу .txt.

Перший модуль має назву «Cutting» і включає функціонал для здійснення розбору та структуризації словнику ВЕСУМ з метою зменшення його розміру на жорсткому диску, пришвидшення операцій пошуку та співставлення слова з його атрибутом, а також індексації умовно прийнятих однокореневих слів як словоформи одного. Такими словами вважаються слова, що ідентичні за основою і списком закінчень, але мають різні префікси.

Другий модуль називається «Clusterizer» і виконує роботу з переведення текстів у морфологічні атрибутивні послідовності, нормалізацію векторів таких атрибутів серед всіх текстів, а також їх кластеризації.

Програмний додаток повинен володіти наступним функціоналом:

- розбити вхідний словник української мови на елементарні файлові одиниці для подальшої побудови структури даних у пам'яті за ними;
- виконувати реконструкцію слів за їх частинами в оперативну пам'ять у вигляді структури даних, що міститиме всі словоформи з їх індексами основ, а також морфологічні атрибути;
- виконувати розбиття множини вхідних текстів на токени (слова), видаляючи всі пробіли, знаки пунктуації та відступи;
- формувати вектори послідовностей морфологічних атрибутів за текстом;

– аналізувати вектори атрибутів з подальшою кластеризацією вхідних текстів за авторами.

2.1.2 Вхідні дані

Необхідні на вході програми дані наступні:

– словник української мови, що відповідає формату словника ВЕСУМ [18] (опціонально);
– множина текстових файлів у форматі .txt, що підлягають аналізу та подальшій кластеризації за авторами.

2.1.3 Вихідні дані

Список вихідних даних для програм наступний:

– для модулю першої програми – структурований словник української мови з мінімально необхідним набором даних для побудови відповідної структури;
– для другого модулю – текстовий файл з результатами кластеризації вхідних текстів за авторами, текстові файли з частотними характеристиками та кількістю розпізнаних слів за кожним вхідним текстом окремо.

Призначення компонентів системи:

– Dictionary cutting – виконує розбір словника української мови на наступні текстові модулі: основи слів (частини, що повторяються, а саме слово є унікальним), закінчення (згруповані за фактором їх повторення серед словоформ різних слів, після чого групи індексуються, а індекси прив’язуються до основ слів), літери, що чергуються при відмінюванні, індекси спорідненості слів (якщо слово з даною основою зустрічається вперше, то його порядковий номер, інакше номер основи, що відповідає першій зустрічі основи при підрахунку; якщо зустрічається основа, послідовність літер якої співпадає з якою-небудь у словнику, індекс закінчень співпадає, при цьому відмінність між ними лише у префіксі, тоді така

основа є префіксально утвореним словом і посилається на першу основу). При цьому модулі зберігається у файли за вказаною директорією.

– Text analyzer – розбиває масив вхідних текстів на список масивів слів. Проводить відсіювання всіх знаків пунктуації та пробілів. Шукає всі слова в словнику та привласнює їм відповідні індекси з нього;

– Clusterizator – виконує розподілення текстів за авторами на основі попередньо сформованих послідовностей атрибутів, що отримуються зі словника у пам’яті програми за індексом слова.

За допомогою моделювання складових частин системи через UML-діаграми можна відобразити її основні функціональні аспекти, шляхи потоків виконання та модулі, що будуть проводити обробку і відображення даних. Ці діаграми покладено в основу внутрішнього проектування програмного комплексу.

Метою створення діаграм є не лише документування розробки, а й окреслення у графічному вигляді статичної структури декларативних частин системи (класів, типів). Також вона буде містити в собі деякі елементи поведінки (операції), однак на відміну від інших діаграм рівень представлення діаграми класів показує безпосередні програмні одиниці, які надалі будуть представлені у вигляді програмного коду в такому ж вигляді. Однак на даному етапі програма все ще проектується, тому використовуються абстрактні поняття елементів. Важливе лише представлення і взаємодія класів, а не деталі реалізації, такі як продуктивність системи, інкапсуляція елементів, область їх видимості.

2.2 Зовнішнє проектування

2.2.1 Мова програмування і середовище розробки

Важливу роль при проектуванні займає і вибір мови, на якій здійснюватиметься програмування програми. Мова повинна володіти необхідним інструментарієм для її реалізації, крім того варто звернути увагу на наявні бібліотеки, що можуть полегшити процес програмування та зменшити час на розробку системи.

Для задачі моделювання кращої структури даних важливо виконати порівняльний аналіз із конкурентоспроможними зразками, що можуть в тих чи інших аспектах бути кращими для зберігання даних словнику. Крім того, предметна область лінгвістичного аналізу підходить для об’єктного керування, бо всі її частини можуть бути виражені через об’єкти для більш простого маніпулювання, а також немає необхідності в складних математичних розрахунках. Тому за мову програмування була взята C#.

C# - це об'єктно- і компонентно-орієнтована мова програмування. C# надає доступні та легко читабельні мовні конструкції, що реалізують ці концепції у роботі. Завдяки цьому C# підходить для створення та застосування програмних компонентів. З моменту створення мова C# збагатилася функціями для підтримки нових робочих процесів та постійно розширює стандарти щодо розробки ПЗ. C# - це об'єктно-орієнтована мова, що будується на створенні типів та заданні їх поведінки.

Наступні функції C# допомагають створювати надійні та довговічні програми:

– збір сміття автоматично відновлює пам’ять, зайняту недоступними іншим об’єктам областями або об’єктами, що підписалися на автоматичне знищення;

– типи, які допускають значення NULL, захищають від змінних, які не посилаються на виділені об’єкти$

– обробка виключних ситуацій забезпечує структурований і розширений підхід до виявлення та відновлення помилок;

– лямбда-вирази підтримують методи функціонального програмування;

– синтаксис мовного інтегрованого запиту (LINQ) створює загальний шаблон для роботи з даними з будь-якого джерела;

– уніфікована систему типів. Усі типи C#, включаючи примітивні типи, такі як int і double, успадковуються від одного кореневого типу object, тому мають

набір загальних операцій. Властивості будь-якого типу можна зберігати, транспортувати та використовувати узгоджено зі стандартами мови;

– підтримка як визначених користувачем типів посилань, так і типів значень;

– динамічне розподілення об'єктів та можливість зберігати вкладені структури, тобто виділення пам’яті середовищем буде виконано лише для верхньої структури з урахуванням усіх вкладених і лише один раз;

– підтримка загальних методів та типів, які забезпечують підвищену безпеку та продуктивність типів;

– наявність ітераторів, які дають змогу реалізаторам класів колекції визначати користувацьку поведінку для клієнтського коду.

Завдяки цим можливостям мови вона відмінно підходить для розробки дослідження користувацьких колекцій, які фактично є певними структурами даних, що реалізують інтерфейс для уніфікованого перебору даних, операції пошуку, вставки, видалення і т. д. Таким чином можна буде організувати серію тестів на різних розроблених структурах для зберігання словнику, а потім порівняти часові показники з відповідними вбудованими реалізаціями, що надає сама мова. Крім того, тестування програмних засобів значно прискориться за рахунок того, що всі реалізації будуть реалізовувати єдиний інтерфейс, тобто на виході буде отримано одні й ті ж дані, різниця полягатиме у швидкості виконання елементарних операцій CRUD (create, read, update, delete – створення, зчитування, оновлення та видалення даних).

Для задач даного дослідження важлива швидкість виконання лише двох операцій – зчитування та створення, бо оновлення існуючого словника шляхом зміни наявних слів чи видалення їх взагалі не передбачається як часта операція, адже стандарти мови хоч і змінюються з часом, однак не часто. До того ж важливо мати всі версії слів за весь часовий період становлення сучасної української мови (XVIII ст. – наші дні), тому видалення застарілих слів не потрібно.

2.2.2 Основні аспекти дослідження показників структур даних

Для визначення кращої структури даних перш за все необхідно сформувати список попередньо можливих варіантів базового типу структури для зберігання всіх слів, що буде кореневою. Наступним кроком буде формування вкладених елементів, таких як морфологічні атрибути слів. Кожен з описаних рівнів може бути представлений довільною структурою даних, перелік яких заздалегідь визначено на основі емпіричних суджень.

Так, на основі евристичних припущень, для зберігання словнику слів варто використати хеш-таблицю, що дозволить максимізувати швидкість пошуку слів, а множина слів мови змінюється не часто, тому перебудова структури з метою додавання нових чи видаленням старих слів не передбачається. Також конкурентоспроможним об’єктом дослідження можуть стати різноманітні дерева, бо багато слів мають словоформи, які можна було б компактно розмістити та призначити їм одну кореневу основу, що пришвидшило б пошук відповідного слова у словнику.

Після формування списку структур першого та другого рівнів необхідно визначити основні характеристики порівняння структур, а також їх ієрархію та вагу впливу на визначення кращої серед структур.

Оскільки основна задача словнику зводиться саме до пошуку слів серед статичної множини, то найбільший коефіцієнт матиме швидкодія структури за операцією пошуку.

Для тестування швидкодії виконання операцій кожна ключової функції обгорталася таймером, що вимірював час її виконання. Витрати ресурсів на таймер мізерні в порівнянні з об’ємом виконання робіт програмою на будь-якому із етапів, а точність досягає мілісекунд. Крім того запуск програми проводився в чистій системі та виконувався моніторинг навантаження процесору сторонніми програмами, до того ж ініціація процесу виконавчого файлу проходила вручну безпосередньо із операційної системи, а не з вбудованого у середовище тестування інструменту, що призначений для відлагодження та резервує додаткові ресурси на покрокове виконання програми при застосуванні точок зупину, здійснює моніторинг

ресурсів і т. д. Також серія контрольних виконань програми для тестування швидкодії проводилося багаторазово, а за підсумкові результати бралося середнє значення часу.

2.2.3 Розробка структури словника української мови на основі ВЕСУМ

Першою характеристикою для вимірювання, що буде спільною для всіх структур даних, стане об’єм всіх модулів словника на жорсткому диску для його відтворення у структуру під час виконання програми. Для компактного розміщення даних, що взяті зі словника ВЕСУМ зі словником слів у 6 213 574 одиниць, було проведено його розбір на унікальні морфологічні атрибути слів, а відповідним словам і словоформам призначені їх індекси.

Крім того, проведено поглиблений аналіз всіх слів. В результаті обробки всіх слів для їх групування за словоформами, а також однокореневими формами, проводиться конструювання структури українського словника слів (далі – основ) за критерієм їх співпадіння за коренями, співпадінням списку закінчень окремого слова. Розробка даного словника викликана необхідністю розбиття вхідного тексту на однозначні смислові одиниці (слова), пошук їх відповідностей у словнику та групування за семантичною подібністю, ігноруючи факт наявності омонімів у тексті. Словоформами вважаються всі відмінки, а також слова утворені додаванням префіксу. Відстеження омонімів неможливе без дослідження конструкції речення та аналізу контексту, тому їх наявність неможливо відстежити даним алгоритмом. При відстеженні відмінкових форм слова слід враховувати чергування голосних звуків всередині слів, щоб коректно співвіднести слово з основою на етапі аналізу текстів.

Для вирішення даної проблеми виконується ведення допоміжної інформації щодо розташування звуку, що чергується, та відповідної літери заміни. У списку основ зберігається звук, який міститься у базовій формі слова (інфінітиві, називному відмінку іменника). У списку закінчень всі елементи, що утворюють з основою, в якій чергується звук, слово, яке існує в українській мові, позначаються знаком «+». Це дозволяє відстежувати належність закінчення до альтернативної основи зі

звуком, що чергується, та повинно позитивно вплинути на швидкодію пошуку такого слова у словнику через зменшення кількості основ.

Відсічення префіксів напряму не відбувається, тому ведення обліку всіх префіксів також не здійснюється. Після формування всіх основ та привласнення їм індексів закінчень достатньо виконати перевірку на співпадіння послідовності літер кожної основи з усіма іншими основами. Сформуємо гіпотезу співпадіння коренів:

- з вибірки двох слів довжиною s і t коротше слово (t) повинно зустрічатися всередині довшого (s) таким чином, щоб літери з кінця і до символу на позиції s-t виключно. Частина довшого слова, що не співпадає з літерами коротшого в такому випадку називається префіксом;
- індекс списку закінчень обох основ порівнюваних слів збігається.

Складність такого алгоритму складатиме $O(n^2*m)$, де n – кількість слів, m – кількість порівнянь літер. Це значення не враховує середню різницю довжин слів, бо наперед невідомо, яку частку становлять слова з префіксами. Тому в реальності число m прирівнюється середній довжині слова із словнику, що приблизно показує фактичну кількість порівнянь. До того ж для кожного слова необхідно було б вираховувати його довжину, щоб правильно зіставити початки коренів, пропустивши префікс у довшому слові.

Для пришвидшення роботи можна виконувати посимвольне порівняння, починаючи з кінця слів. Тоді підрахунок довжин слів можна опустити, до того ж зменшиться кількість операцій порівняння символів у зв’язку з тим, що при першій розбіжності літер ми вже знатимемо, що дані слова мають різні корені, подальше порівняння не потрібно. Якщо зустрінемо на поточному кроці не співпадіння літер довшого і коротшого слів, то слова не співпадають, якщо ж усі літери збіглися – то довше слово вважаємо тим же словом, утвореним префіксальним способом, привласнюємо індексу його основи індекс коротшого слова. Завдяки цьому можна стверджувати, що число порівнянь тепер знаходиться в діапазоні [1;L], де L – середня довжина слова.

Прийнявши до уваги всі вищевказані проблеми було розроблено алгоритм, що повинен враховувати їх та видавати на виході готові модулі у вигляді текстових файлів, на основі яких можна буде заповнювати структуру даних словника.

Побудова словнику основ відбуватиметься у наступній послідовності:

- зчитування відсортованого базового словнику слів української мови зі всіма словоформами формату: слово у називному відмінку на початку рядку, перевірка, чи починається наступний рядок з пробілу, якщо так – це одна з його словоформ, інакше – це нове слово. Крім цього ведеться облік списку слів, що є словоформами одного слова, але починаються з іншої літери (вівця-овець, я-мені, він-йому і т. д.), щоб у кінці вони посилалися на базове слово у називному відмінку;
- переведення всіх слів у нижній регістр;
- пошук основи серед усіх словоформ окремого слова (кореня, що є незмінним);
- порівняння бази слів з їх словоформами для відсічення закінчень, якщо у даного слова є альтернатива (літера, що чергується), то закінчення починається зі знаку +, інакше /;
- якщо на етапі відсічення закінчення виявлено, що 1 літера всередині слова змінюється, а літери після неї у всіх словоформ повторюються – то у список альтернативних літер додається нове значення (номер по порядку змінної літери починаючи з 0 та альтернативна літера);
- пошук однокореневих слів, у яких співпадають списки закінчень (утворені префіксальним способом) та певна частина слова (передбачно – корінь);
- відділити всі морфологічні атрибути слів зі словнику, зберігаючи їх порядок. Сформувати список унікальних атрибутів;

– створити список індексів атрибутів, використовуючи отриманий список унікальних атрибутів та початковий порядок слів і атрибутів у словнику.

В результаті отримуємо наступні файли з даними для формування словнику:

– BasicIndexes – список індексів уніфікованих основ для кожної основи. На етапі розбору словника формуються певні групи основ, а об’єднує їх спільний корінь і список закінчень, таким чином відмінний від порядкового номеру індекс матимуть слова, що мають префікс у своєму складі. Загалом же основ сформовано 388936. Структура:

“**цифра**” **/n,**

де **/n** – перехід на новий рядок;

– UniqueEndings – список унікальних списків закінчень серед усіх слів української мови, розділених в рядку знаком / якщо чергування звуку у словоформі з участю даного закінчення немає і знаком +, якщо чергування наявне. Всього унікальних списків закінчень сформовано 2101. Структура:

{“/” | “+”} “**закінчення**” **/n.**

Примітка: закінченням може бути пусте місце, тобто 2 знаки + або / в ряд можуть зустрічатися один раз на список закінчень, що позначає факт того, що існує слово лише з кореня;

– EndingIndexes – індекси списків закінчень для відповідної основи, що вказують на рядок із файлу UniqueEndings. Структура:

“**цифра**” **/n;**

– Bases – список всіх основ слів без їх закінчень. Структура:

“основа” /n;

– UniqueProperties – список унікальних морфологічних атрибутів слів, таких як частина мови, відмінок і рід для іменника, час і стан для дієслова, ступінь порівняння для прикметника і т. д. Загальна кількість отриманих атрибутів 4365. Структура:

{“атрибут”:} /n;

– UniquePropertyIndexes – список індексів, що співставляють словоформу з її морфологічними атрибутами. Загальна кількість індексів рівна кількості слів зі словнику ВЕСУМ. Структура:

“число” /n;

– Switch – список звуків чергування та їх позицій у слові. Базуючись на ньому та знаку перед закінченням буде утворено правильне слово, яке буде відноситися до тої ж основи, що й інші слова цієї групи закінчень. Структура:

[“цифра” “літера”] /n.

Примітка: цифра позначає місце літери, що чергується в слові, індексація починається з одиниці, літера є літерою чергування, на яку буде проходити заміна у слові;

– Delimiters – незалежний від словника файл, що містить всі можливі роздільні знаки між словами для подальшого розбору тексту. Структура файлу:

“роздільний знак” \n.

Розроблена структура повинна компактно зберігати словник ВЕСУМ для його подальшого застосування в модулі розпізнавання тексту та кластеризації творів.

2.2.4 Визначення структур даних для порівняння

Після визначення типу даних, що будуть зберігатися, а також акцентуванні на операції пошуку варто зосередитись на тому, які структури будуть кращими для встановлених задач розбору текстів на морфологічні атрибути за словником зі швидкодіючою операцією пошуку.

Пошуковими структурами даних [5] є будь-які структури, що реалізують пошук конкретних елементів з певної множини даних, влаштованої в пам’яті певним чином. Найпростішим варіантом реалізації є невпорядкована послідовність всіх елементів. Якщо не застосовувати ніякого влаштування даних, то неминуче виникне ряд операцій, що потребують лінійного часу в гіршому та середньому випадку. Сучасні структури даних мають як ряд обмежень на здійснювані запити, так і відповідні привілеї, що при цьому дають приріст швидкодії на певних операціях. Окрім того, вартість побудови таких структур пропорційна кількості елементів n, тому її побудова доцільна навіть якщо кількість запитів багатократно менша за кількість елементів.

Структури даних мають наступні характеристики:

- тип (статичні виконують запити на незмінному наборі даних, динамічні передбачають реалізацію операцій зміни бази даних між виконанням запитів);
- час роботи (заповнення даними, пошук; для динамічних також оновлення даних та видалення). При цьому оцінюють мінімальний час, що необхідний для конкретної операції над даними, середній та найгірший час;
- використовувана пам’ять (зазвичай це значення складає O(n)).

Тепер необхідно провести попередній відбір серед можливих структур даних для формування тестової вибірки. Порівняльна характеристика обраних евристичним методом структур за середніми очікуваними показниками представлена у табл. 2.1. На основі поєднання показників їх розміру, швидкодії вставки та пошуку можна буде здійснювати вибір кращої структури.

Таблиця 2.1 – Порівняльна характеристика структур даних за складністю

Назва	Вставка	Пошук	Пам’ять	Опис
Відсортований індексований список (List C#)	O(n)	O(log n)	O(n)	Шаблонна структура даних у C#, що являє собою список, який реалізує операцію отримання даних за індексом, що дозволяє виконувати швидкий пошук на відсортованих даних
Словник (Dictionary)	O(1)	O(1)	O(n)	Різновид хеш-таблиці, що підтримує роботу з даними лише одного строго визначеного типу
Префіксальне дерево Trie	O(log n)	O(r), де r довжина ключу	O(2n)	Кореневий граф, кожною вершиною якого є унікальний для кожного рівня рядок

2.2.5 Дослідження показників структур даних

Розглянуті вище показники досить поверхнево розкривають сутність структур даних, тому необхідно розібрати, як і в яких ситуаціях діятимуть всі обрані структури на множині реальних даних, а також які адаптації вони можуть приймати для оптимальної роботи.

Першою характеристикою виберемо операцію вставки, бо саме вона є початковим етапом в роботі структури. Звісно, заповнення структури не так важливо в довгостроковій перспективі, як пошук у ній, однак варто проаналізувати всі можливі варіанти оптимізації в даній роботі, що при великих об’ємах даних позитивно відобразиться на кінцевому часі роботи. До того ж новому користувачу, мета якого буде кластеризувати образи творів за авторами при першому

завантаженні програми, доведеться в першу чергу взаємодіяти саме з програмою структуризації словнику, тому буде важливою швидкодія її формування.

Операція пошуку буде ключовою при виборі структури, тому треба зосередитися на точному вимірюванні часу на її виконання на найбільш повній вибірці тестових даних. Для цього буде обрано декілька джерел з різнотипними текстами, а саме:

- перелік статей із бази даних вільної енциклопедії – Вікіпедії (2.3ГБ) [20];
- повідомлення з месенджерів (3 МБ);
- твори класичної української літератури XVIII-XX ст. більш ніж від 100 авторів обсягом у 444 твори (96 МБ);
- фіктивний текст, що базується на творах української літератури різних авторів (194 МБ);
- тезиси від конференцій різної направленості, що вдалося відшукати у вільному доступі (16 МБ).

Завдяки різноплановості тематики текстів тестової вибірки повинна досягатися повнота перевірки як словникового запасу словника ВЕСУМ, так і рівномірність швидкодії пошуку слів, а також вплив механізму виділення цих слів із тексту, ігноруючи знаки пунктуації.

Наступним вимірюванням стане зайнятий структурою даних об’єм в оперативній пам’яті, що може варіюватися в залежності від способу розташування даних в пам’яті.

Для більшості структур дане значення буде близьке до об’єму, що потребується для розташування всіх слів у вигляді строкових даних в пам’яті на жорсткому диску. Оскільки вихідний словник після обробки першим модулем представляє собою по-новому структуровану версію для економії пам’яті, то дане значення необхідно буде вирахувати під час виконання програми. Таким чином, можна сформувати уявлення щодо передбачуваного розміру, якщо об’єднати всі кінцеві слова в єдиній змінній – масиві char, після чого підрахувати кількість

символів у ньому. Змінна типу char потребує 1 байту пам’яті для зберігання 1 символу, якщо користуватися звичайним кодуванням символів на зразок ASCII або Unicode.

Окрім самого слова у структурі також міститься індекс морфологічних властивостей даного слова, що значно економить пам’ять, адже значення індексу знаходиться в межах [0,4365], тобто для його представлення достатньо 4 знаки, що еквівалентно 4 байтам, а для зберігання повної характеристики знадобилося б в середньому 10 байт (для строкового представлення атрибуту).

Кінцевий теоретичний розмір всіх необхідних даних слова буде рівним розміру всіх слів плюс сукупний розмір індексів, а також довжина індексу базової основи, що для кожного слова буде змінною та варіюватиметься відповідно в діапазоні [0; 388936], що еквівалентно максимальній кількості основ. Таким чином, якщо виконати моніторинг у програмі, а саме підрахувати загальну довжину всіх даних у їх строковому вигляді, то отримаємо розмір у 80,25 МБ.

Дані цифри демонструють лише витрати на самі дані, однак для можливості здійснення пошуку необхідна навігація по структурі певним чином. Більша частина пам’яті витрачається саме на організацію влаштування даних у пам’яті. Так, для кожного вказівника на комірку пам’яті знадобиться ще 4 байти на змінну індексу типу int, а таких комірок для розробленого словнику знадобиться щонайменше по 1 на кожне унікальне слово. В початковому словнику наявні повтори слів, а саме словоформ, що виникає в результаті співпадіння певних відмінкових форм.

Заміри при виконанні програми показали, що всього таких унікальних слів 3 496 257, а отже щонайменше знадобиться додатково 14 МБ лише на вказівники.

Структура trie передбачає підвищені витрати пам’яті для зберігання одного символу в пам’яті в результаті того, що саме дерево містить повтори частин рядків для слів. Використовуючи структуру trie в модифікації стисненого префіксального дерева [14] досягається виграш у пам’яті на зберігання структури. Кінцевий розмір досить складно розрахувати чисельними методами для довільного набору даних, що в нашому випадку сформовані на основі словника української мови. Однак, завдяки вбудованим засобам профілювання відлагодження програми у середовищі розробки

Visual Studio можна відстежити приблизний об’єм оперативної пам’яті, що витрачається на зберігання структури. Результати представлені на рис. 2.8.

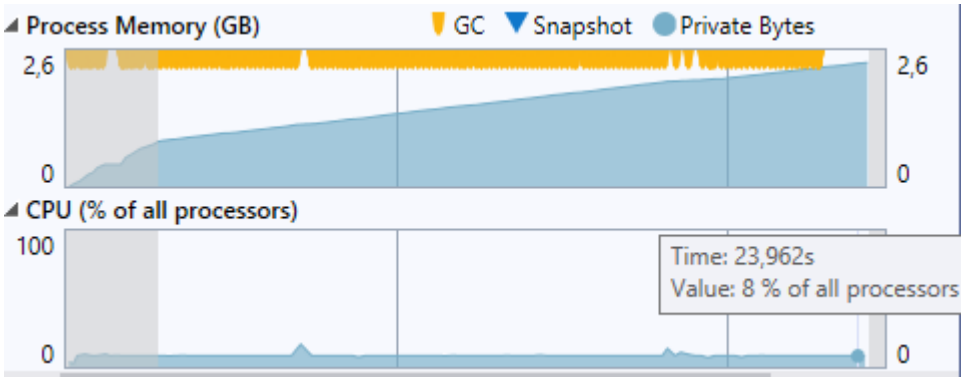


Рисунок 2.1 – Використання ресурсів програмою для структури trie

Теоретичні результати не в повному обсязі демонструють реальні витрати пам’яті, що наглядно видно на рис. 2.1. Тому при порівнянні структур потрібно орієнтуватися на реальні витрати, які можна отримати лише в умовах повноцінного експерименту на реальних тестових даних.

2.2.6 Модифікація структур даних

Перш ніж провести фінальне порівняння всіх структур необхідно провести ряд оптимізаційних дій над структурами, що дозволять дати виграш у швидкодії з урахуванням поставленої задачі.

Візьмемо структуру trie. Як розглядалося раніше, існують модифікації як для стиснення структури з метою економії пам’яті, так і для пришвидшення операції пошуку шляхом оптимального розподілення частин слова по вузлам дерева [14].

Наступним кроком є сортування кожного рівня дерева за відповідним порядком літер у частині слова на цьому рівні. Представимо, що у батьківського вузла міститься n вказівників на дочірні вузли. Асоціативно можна відобразити їх у масив, а пошук правильного шляху визначається співставленням відповідної частини слова зі значенням дочірніх вузлів почергово. Швидкість виконання такої операції без модифікації прирівнюється лінійному пошуку, тобто O(n).

Однак, якщо відсортувати всі елементи в асоційованому масиві, то можна застосувати бінарний пошук, складність якого складає O(log n), а саме також використати швидке сортування, складність якого O(n*log n) для сортування

кожного асоційованого масиву для кожного батьківського вузла, почергово спускаючись вниз по рівням. Завдяки цьому ми перекладемо частину витрат на обчислення на стадію заповнення структури даних, що в свою чергу відбувається лише один раз на сесію користування програмою, а операція пошуку необхідного слова у дереві отримає значний приріст продуктивності.

Щодо хеш-таблиці перш за все необхідно порівняти засоби, що надаються у пакеті зі стандартною бібліотекою .NET SDK, та розглянуту раніше бібліотеку CMPH, що постачається лише для операційних систем Linux. Інтеграція в програмний комплекс, що розроблений для операційної системи Windows, пакету із Linux неможливо без створення Інтернет-сервісів, що в свою чергу призведе до зниження продуктивності алгоритмів та їх практичну цінність. До того ж наявні у сучасному стандарті .NET алгоритми з реалізації структур даних безсумнівно повинні бути швидкими. Кінцеву оцінку швидкодії буде надано після проведення експериментів та порівняння програмних засобів на вибірці однакових даних.

Для репрезентативності порівняння структуру даних список також необхідно зробити максимально швидкою. Для цієї мети можна скористатися все тими ж можливостями стандартної бібліотеки мови та застосувати відсортований список в якості сховища даних. За замовчуванням він підтримує бінарний пошук за ключем зі складністю O(log n).

2.3 Організація процесу порівняння морфологічних атрибутів слів

Вирішальним етапом є переведення слів у їх морфологічні атрибути, на основі яких і проходитиме кластеризація. Для порівняння застосовуються послідовності з 2-х таких атрибутів. Таким чином із тексту, що містить n слів, буде отримано n-1 послідовність атрибутів, яка окремо індексується для кожного тексту.

Для цього необхідно на основі попередньо сформованої статистики з послідовності індексів морфологічних атрибутів слів побудувати вектори. Для нормалізації всіх векторів необхідно привести їх до єдиної розмірності, тобто сформувати з множини всіх текстів єдиний вектор, кожна комірка якого буде містити індекс сформованої послідовності з 2-х атрибутів. Доповнюючи кожен

вектор атрибутивних послідовностей текстів нульовими комірками до розмірності єдиного вектору ми отримаємо такі вектори, що можна буде порівнювати між собою.

Однак крім приведення векторів до єдиної розмірності необхідно також уніфікувати кількість атрибутів між ними. Більші тексти за замовчуванням матимуть більше таких послідовностей, а тому кожна отримана послідовність ділиться на кількість слів у тексті. таким чином ми переходимо від кількісних показників атрибутів, до відсоткових часток частоти їх зустрічі у тексті.

Завдяки аналізу послідовних, а не одиночних атрибутів, можна робити висновки щодо певного літературного стилю автора, що простежується у формі побудови словосполучень.

Висновки до другого розділу

В розділі сформовано специфікацію для програмного засобу, визначено його функціональні можливості, за допомогою яких можна буде провести експерименти та виконати дослідження. Спроектовано структуру даних словника для збереження на жорсткому диску та сформовано вибірку кращих структур даних, серед яких здійснюватиметься вибір на основі результатів вимірів часових характеристик операцій вставки та пошуку. Крім того визначено процес їх порівняння. Розроблено попередній алгоритм для переведення текстів у вектори атрибутів, за відстанями між якими виконуватиметься кластеризація та формуватиметься висновок щодо подібності текстів.

3 РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ СТРУКТУР І КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ

3.1 Застосування принципів програмування

Будь-який програмний продукт після завершення розробки повинен бути розгорнутий для подальшого використання. Навіть після завершення процесу розробки він може супроводжуватися програмістами для виправлення можливих критичних помилок, адаптації під нові вимоги або ж навіть повній реконструкції з інтеграцією певних його частин у новий продукт. Тому на стадії його програмування важливо дотримуватися певних принципів, що дозволять розробити якісний продукт з відповідним до поставлених задач функціоналом, продуктивною роботою завдяки використанню нових алгоритмів вирішення задач, а також спростить його подальше супроводжування.

Першим і найважливішим кроком є вибір методології програмування. Від будь-якої методології програмування ми очікуємо, що вона допоможе нам у вирішенні конкретних задач, поставлених предметною областю. Але однією з головних проблем у програмуванні є складність. Зі зростанням об’ємів програми підвищується її складність, а оскільки програму розроблюють зазвичай командою, то з часом все гостріше постає проблема управління як її розробкою, так і влаштування потоків управління в самому додатку. Для вирішення цієї проблеми можна розбити програму на невеликі, чітко окреслені частини, що доступні для розробки однією людиною.

Щоб подолати складність, необхідно абстрагуватися від дрібних деталей. З цією задачею відмінно може впоратися об’єктно-орієнтоване програмування (ООП). Ідеологічно ООП вирішує задачу програмування як моделювання інформаційних об’єктів, при цьому структурування цих об’єктів виконується з міркувань їх подальшого управління [21]. Розглянемо основні принципи даного підходу:

– наслідування – дозволяє влаштувати спорідненість між класами, що мають спільний функціонал. Таким чином досягається повторне використання коду, що зменшує обсяг його написання. При цьому кожен

похідний клас може бути використаний замість базового без втрати сенсу виконання, що допомагає організувати ряд шаблонних операцій, які можуть бути виконані над переліком споріднених типів;

– інкапсуляція – дозволяє організувати чітку ієрархічну керованість об’єкту через перелік чітко визначених відкритих методів, за допомогою яких можна керувати його станом;

– поліморфізм – можливість однією функцією опрацьовувати дані різних типів. Завдяки цьому досягається розгалуження потоку управління програми на етапі її виконання, тобто можна керувати ходом виконання функцій, їх порядком та функціональним призначенням в залежності від того, які типи даних прийняті на вході;

– абстракція – допомагає представляти об’єкти в системі не як програмні одиниці, а як абстракції з реального світу, в такому випадку можна легко перенести об’єкт із предметної області напряму як мінімально необхідний перелік методів та атрибутів для його представлення.

ООП дає можливість створювати розширювані системи (extensible systems) [22]. Це одна з найбільш значних переваг ООП і саме вона відрізняє цей підхід від традиційних методів програмування. Розширюваність (extensibility) означає, що в існуючу систему можна інтегрувати нові компоненти, причому без внесення до неї будь-яких змін. Компоненти можуть бути додані навіть на стадії виконання програми. Розширення типу (type extension) і поліморфізм змінних, що випливає з нього, виявляються корисними переважно в наступних ситуаціях:

– обробка різноманітних структур даних. Програми можуть працювати з об’єктами динамічних типів, значення яких конкретизується на стадії виконання. Нові типи можуть бути додані будь-якої миті;

- зміна поведінки під час виконання. На етапі виконання один об'єкт може бути замінений іншим. Такі підстановки можуть вирішувати хід потоку виконання програми;
- реалізація шаблонних компонентів. Алгоритми можна узагальнювати, щоб вони вже змогли працювати більше, ніж з одним типом об'єктів.
- поширення узагальнених компонент. Компоненти немає необхідності підлаштовувати під певний додаток. Їх можна зберігати у бібліотеці як ресурси для інших програм (semifinished products);
- розширення каркасу. Незалежні від додатку частини предметної області можуть бути реалізовані у вигляді каркаса і надалі розширені за рахунок додавання частин, специфічних для конкретної версії програми.

Багаторазового використання програмного забезпечення на практиці досягти не вдається через те, що існуючі компоненти вже не відповідають новим вимогам. ООП допомагає цього досягти без порушення роботи вже наявних клієнтів, що дозволяє витягти максимум із багаторазового використання коду.

Наступним застосованим принципом при побудові програмних засобів було використано SOLID, що з англійської є абревіатурою від ключових принципів:

- single-responsibility principle [23] (єдиної відповідальності) – визначає, що кожен об’єкт повинен мати єдину відповідальність в своїй роботі, що обмежена границями класу. Усі методи і поля об’єкту повинні бути направлені на вирішення конкретної задачі. Завдяки цьому досягається легка масштабованість коду через можливість легкого доповнення функціоналу новими модулями без необхідності редагувати старий код;
- open-closed principle [24] (відкритості-закритості) – програмні сутності повинні бути відкриті для розширення, але закриті для модифікації. Дотримання цього принципу ґарантує його легке супроводження в майбутньому, адже виключає ті самі витрати на редагування старого коду заради додання нового функціоналу;

– Liskov substitution principle [25] (принцип підстановки Лісков) – потребує, щоб функції, які використовують певний базовий тип мали можливість без додаткових засобів замінити цей тип на будь-який похідний від нього. В мові C# дана можливість реалізована завдяки шаблонним методам, завдяки яким можна організувати єдину функцію для виконання операцій вставки, пошуку, що буде приймати різні структури даних і таким чином змінювати свій хід роботи в залежності від цього. В результаті ми позбавимося повторів блоків коду та забезпечимо можливість змінювати структуру динамічно під час виконання програми;

– dependency inversion principle [26] (інверсія залежностей) – абстракції (сутності із реального вигляду у програмного коді, тобто об’єкти) не повинні залежати від деталей реалізації, тобто від низкорівневих рішень при програмуванні. Деталі повинні залежати від абстракцій. Таким чином ми застосовуємо конкретні програмні рішення заради досягнення кращих показників, а не корегуємо наші задачі під обмеження наявного інструментарію. Модулі вищих рівнів не повинні імпортувати дані із нижніх рівнів та бути тісно зав’язаними на них, всі операції повинні здійснюватися над абстракціями, що дозволить в будь-який момент підмінити окремий модуль на новий без змін у наявній системі;

– interface segregation principle [26] (розмежування інтерфейсів) – розроблені інтерфейси повинні надавати програмним сутностям якомога менший перелік методів і лише таких, що є невід’ємними у роботі. Невеликі інтерфейси дозволяють краще розділити обов’язки і чітко розмежувати обов’язки, що послабить зв’язність коду, тобто зменшить залежність між модулями програми. Програмні сутності не повинні залежати від методів, які вони використовують. Це підвищує гнучкість коду при його подальшому супроводженні з можливістю легко розширювати функціонал.

Наступним принципом стане DRY – don`t repeat yourself (не повторюй себе). Він частково пересікається із принципами SOLID і проголошує відмову від однотипних реалізацій для похідних класів. Замість цього організується процес зі зменшення повторів ділянок коду та винесення їх у окремі функції. Декомпозиція коду призводить до покращення його читабельності та полегшення подальшого супроводження. Доступ до функціоналу щодо вирішення однієї задачі повинен групуватися, тобто єдині за своїм принципом роботи ділянки коду повинні розташовуватися в одному місці, а не бути розкиданими по програмі. Даний принцип корисний для розроблюваної програми, адже повторне використання коду буде корисним при роботі з однотипною обробкою різних структур даних, а також аналізу різнотипних текстів за єдиним зразком.

Ще один не менш важливий у даній роботі принцип Yagni – you aren`t gonna need this (вам це не знадобиться). Через комплексність розроблюваного програмного продукту важливо чітко окреслити, який мінімально необхідний функціонал нам знадобиться в програмі, щоб досягти задач, поставлених у цій роботі. До того ж при дослідженні складно передбачити, чи справді знадобиться на перший погляд корисний функціонал в майбутньому.

Початок розробки функцій повинно проходити лише за чіткою потребою в них. До того ж надмірність функціоналу призведе до підвищення складності тестування, а крім того ж він може завадити надалі при реальній необхідності введення якогось методу через непередбачувані обмеження, що він може накладати при своїй роботі. Незадокументований функціонал так і може залишитися без відома користувачів, при цьому незримо впливаючи на хід роботи основного потоку виконання.

Крім дотримання перелічених принципів також варто використовувати новітні та ефективні засоби, які надаються сучасними програмними середовищами. Так наприклад, в .NET наявні 2 реалізації для типу рядкового значення. Строковий тип string є типом значення, що визначає необхідність здійснювання повне копіювання при передачі його у функцію. Це несло б величезні витрати на додаткову пам’ять. Однак, розробники середовища врахували дану проблему, тому

string реалізований як тип посилання, тобто в функцію передається лише 4-байтовий вказівник на область пам’яті зі значенням змінної.

Однак через це тип string при зміні свого змісту змінює місце в пам’яті на нове, заносячи туди результуючі дані. Стара область пам’яті помічається як незастосована та надалі збирається через Garbage collector (засіб для очищення та перерозподілу використовуваної пам’яті програмою).

Даний процес займає константний час для виділення вказівників та добре оптимізований, однак при оперуванні невеликими об’ємами даних та їх частому зміненні сукупні обчислювальні ресурси на дану операцію можуть навіть перевищувати ресурси на безпосередню зміну даних. Тому всюди, де передбачається повторне використання строкових даних було використано StringBuilder, що є вбудованим стандартним класом з підтримкою зміни рядку. В інших же випадках тип string дає виграш у використовуваній пам’яті та швидкості її виділення, тому де не потрібно змінювати рядок, був застосований саме він.

3.2 Проектування архітектури системи

В цьому розділі описані розроблені методи та їх взаємодія для обробки даних. Оскільки поставлена задача може бути умовно розділена на 2 незалежні модулі у зв’язку з тим, що структурування словника потрібне лише 1 раз, то виконано декомпозицію системи на 2 програми: структурування словнику, в якому використовується ВЕСУМ за базовий словник, а також розбір на кластеризацію текстів, що потребує в своїй роботі отриманий структурований словник.

Взаємодія між класами програми структурування словника влаштована наступним чином:

- запуск програми структуризації словнику, що починається з файлу Program.cs, що викликає метод класу DictionaryCutting для формування списків словоформ для кожної бази;
- передача управління наступній групі методів обробки слів (CutBasic виокремлює слово із словника, CompareWords порівнює кожну словоформу з основою, обираючи найкоротшу спільну частину, та

індексує звук чергування за потреби, GetEndingCompareToBasic утворює списки унікальних закінчень після формування основи, CutUniqueBases виконує пошук повторів основ на основі їх закінчень, CutPrefix на основі обернених слів через клас ReverseBase шукає слова з подібними основами, відмінні лише префіксом, та індексує їх на першу таку основу);

– після завершення опрацювання словнику всі сформовані частини нового словника записуються у відповідні текстові файли, а на екран виводяться показники швидкодії методу структуризації словнику.

Взаємодія між класами програми розбору та кластеризації текстів за атрибутивними послідовностями влаштована наступним чином:

– запуск програми кластеризації, що починається з файлу Program;
– завантаження всіх текстових файлів в оперативну пам’ять у вигляді списків у Program.cs;
– виділення всіх слів із текстів та формування списків на їх основі;
– виклик методу Clusterize, що в свою чергу ініціює заповнення всіх тестових структур даних через метод Insert інтерфейсу IBaseStruct, що реалізований кожною структурою окремо;
– пошук всіх слів з тексту через шаблонний метод Search, що також реалізується кожною структурою окремо;
– ClassesAllocation розпочинає процедуру формування векторів на основі сформованих атрибутивних послідовностей слів та викликає метод кластеризації за максимінним методом UseMaximin;
– сформовані результати кластеризації та відстаней між векторами текстів, а також всі виміри часу швидкодії методів виводяться на екран.

Програмна система розбита на два самостійні програмні додатки у зв’язку з тим, що модуль кластеризації може працювати на заздалегідь сформованому структурованому словнику, тобто операція розбору словника ВЕСУМ потрібна лише один раз і може проводитися лише за необхідності оновлення бази даних слів.

Розроблені діаграми класів представлені на рис. 3.1 і 3.2.

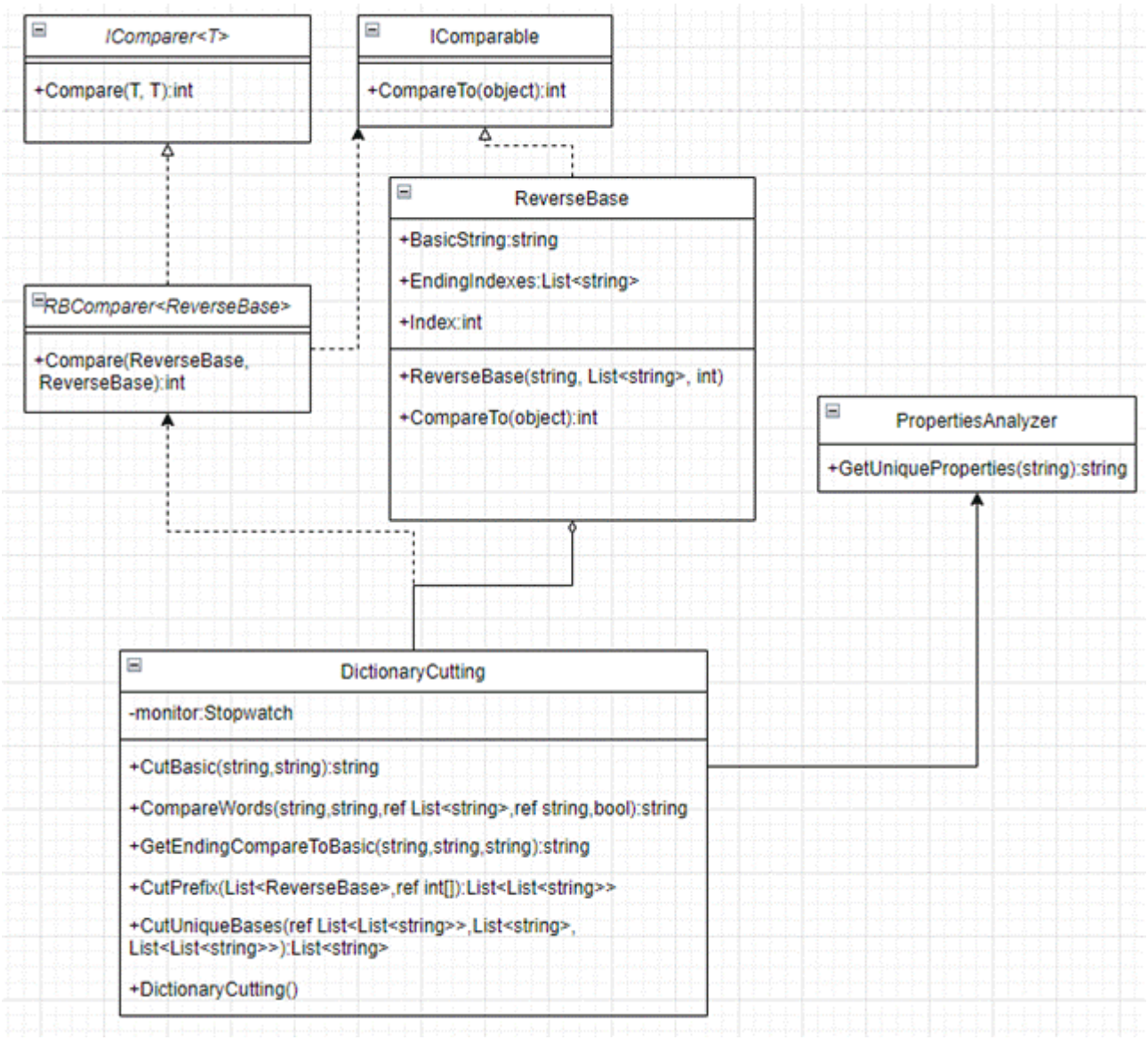


Рисунок 3.1 – Діаграма класів для структуризації словнику

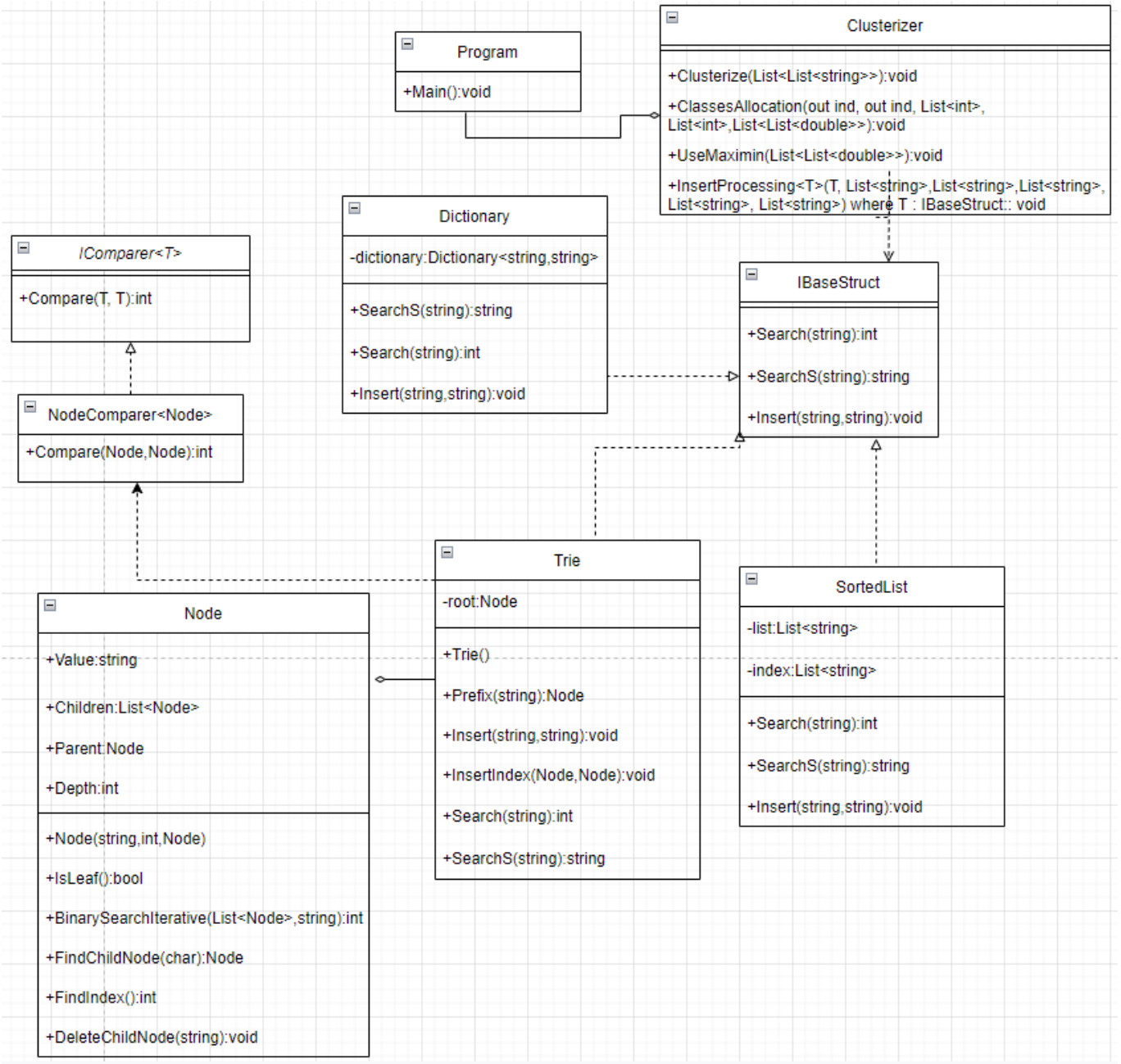


Рисунок 3.2 – Діаграма класів для розбору та кластеризації текстів

3.3 Тестування та налагодження програмного забезпечення

3.3.1 Аналіз методів тестування

Процес тестування є невід’ємною частиною розробки програмного забезпечення, що напряму відображається на його якості. Добре сформована система тестування поєднує ефективне витрачення робочого часу на виявлення критичних для роботи системи помилок. В результаті проходження тестування розроблений програмний засіб повинен задовольняти поставленим показникам

надійності та надавати функціональні можливості, що повністю відповідають технічному завданню.

Існує два підходи проведення тестування: методами чорної та білої скриньки. Перший передбачає перевірку програмного засобу з рівнів керування ним (інтерфейсної частини). На вхід програмі подаються сформовані тестові набори, що відповідають реальним умовам користування програмою, в результаті чого перевіряються її функціональні можливості. Перевагою такого підходу є простота формування вхідних наборів даних, що є інтуїтивно зрозумілими людині та можуть створюватися навіть не програмістами. Недоліком є неможливість організації повноти перевірки роботи програмного засобу, що зумовлено прихованістю деталей реалізації для простого користувача програмою, що може спричинити непередбачувані на перший погляд помилки.

Для поглибленого дослідження функціональних можливостей необхідно виконати тестування на рівні програмних модулів, що і передбачає тестування білою скринькою. такі тести можуть формуватися програмістом ще до написання безпосереднього коду, що підлягає перевірці. таким чином досягається і проектування програмного засобу і покриття його тестами в один крок. Однак даний підхід потребує всебічного розуміння предметної області, для якої розроблюється програма, а також чіткого усвідомлення мінімально необхідних функцій для реалізації запланованого функціоналу.

Набір сформованих тестів для білої скриньки може забезпечувати перевірку коду через покриття:

- операторів (кожен оператор повинен бути виконаний хоча б один раз);
- рішень (подібний до покриття операторів з тією відмінністю, що покриває розгалужені керуючими структурами гілки коду, як наприклад при використанні умовних операторів чи циклів);
- умов (всі умови в операторах повинні виконатись та не виконатись хоча б один раз);

– комбінаторне покриття умов (всі можливі комбінації ходу програми повинні бути виконані хоча б один раз).

Покриття операторів є найменш вдалим, адже може залишити без тестів блоки коду, що є в гілках умовних операторів чи циклів. Тому його не доцільно використовувати для повного тестування. З іншого боку комбінаторне покриття умов ґарантує всебічне тестування програми, однак потребує надлишкових ресурсів для тестування внаслідок циклічності перевірки деяких блоків коду.

3.3.2 Організація процесу тестування

Для тестування програмного засобу було розроблено серію тестів для ключових функціональних методів. Більшість тестів розроблена для модульного тестування, оскільки система розроблювалася однією людиною, а кожен наступний модуль потребував чітко налагодженої роботи всіх попередніх. Обрано метод тестування білої скриньки – покриття рішень, що є мінімально необхідним для повного покриття всіх гілок коду програми та забезпечить достатній рівень надійності для такого типу системи.

Тести розбиті за модулями системи та являють собою набори розроблених тестів для ключових методів кожного модуля.

Метод 1. Виділення основи з 2-х слів

Заголовок string CutBasic (string a, string b)

Метод приймає 2 слова у форматі string та знаходить спільну їх частину, якщо перші літери слів співпадають. Повертає спільну частину словоформ. Тест у табл. 3.1.

Таблиця 3.1 – Метод CutBasic

Номер тесту	Вхід	Вихід	a == b	a.Length > b.Length
1	a= “лося”, b= “лося”	“лося”	+	null
2	a= “лосеві”, b= “лосю”	“лос”	-	+
3	a= “лос”, b= “лося”	“лос”	-	-

Метод 2. Порівняння на співпадіння частин слів з формуванням закінчень

Заголовок string CompareWords (string a, string b, List<string> endingList, ref string drop, bool first)

Метод приймає 2 слова у форматі string, список закінчень List<string> і string літеру чергування, флаг першого входження у функцію для групи словоформ, та на основі спільної частини слів зберігає закінчення у список і звук чергування з індексом його позиції у слові, враховує можливість існування основи без закінчення. Повертає в явному вигляді спільну частину слів, в неявному список закінчень і звуків чергування. Тест у табл. 3.2.

Таблиця 3.2 – Метод CompareWords

Номер тесту	Вхід	Вихід	a == b	dropEnabled == true	a[i + 1] == b[i + 1]&& a[i]!=b[i]	!endingList.Contains (temp) && temp != "+"
1	a= “лося”, b= “лося” , endingList= “”, drop= “”, first=true	“лося”, endingList= “/”, drop= “”	+	-	-	-
2	a= “кіт”, b= “кота” , endingList= “”, drop= “”, first=true	“кіт” , endingList= “+a”, drop= “1a”	-	+	+	-
3	a= “лося”, b= “лосеві” , endingList= “/еві”, drop= “”, first=true	“лос” , endingList= “/еві”, drop= “”	-	-	-	+

Метод 3. Індексація основ з префіксами на базову основу та формування списків префіксів

Заголовок List<List<string>> CutPrefix (List<ReverseBase> revBs, int[] basicIndexes)

Метод приймає список із всіх обернених слів зі словнику та сформовані індекси основ слів для їх індексування з урахуванням однокореневих слів та формує список префіксів. Повертає в явному вигляді список списків префіксів (для ситуацій, коли на 1 основу існує декілька словоформ з різними префіксами), у неявному вигляді змінений масив всіх індексів основ. Тест у табл. 3.3.

Умовні позначення у табл 3.3:

revBs[i].bs.Length > 3 && !revBs[i].bs.Contains("-") – 1,

tempPrefix.Count != 0 – 2,

cnt == prefix.Count – 3.

Таблиця 3.3 – Метод CutPrefix

№ тесту	Вхід	Вихід	1	2	3
1	revBs = {{bs="номед", ed={/у/а/ові}}, {bs="номедіхра",ed={/у/а/ові}, {bs="вилжавіхра", ed={/ий/а/е/ому}} }, basicIndexes={2, 1, 0}	basicIndexes= {2, 1, 1}	+	+	+
2	revBs = {{bs="тук", ed={/у/а/ові/е}}, {bs="тукдів",ed={/ий/ого/ому/а/ій} }, basicIndexes={1,0}	basicIndexes= {1, 0}	-	null	null
3	revBs = {{bs=".ранжім", ed={/}}, {bs=".ртсва",ed={/} }, basicIndexes={3,0}	basicIndexes= {3, 0}	+	-	null
4	revBs = {{bs="умєовс", ed={/}}, {bs=" умєовс-оп ",ed={/} }, basicIndexes={1,0}	basicIndexes= {0, 0}	+	-	-

Метод 4. Формування списку унікальних закінчень

Заголовок List<string> CutUniqueEndings(List<List<string>> finEnding, int basesCount, List<List<string>> sumEnding)

Метод приймає пустий список списків закінчень, кількість основ слів та індекси для призначення основам списків їх закінчень. Повертає в явному вигляді список нових індексів закінчень, в неявному списки відсортованих унікальних списків закінчень. Тест у табл. 3.4.

Таблиця 3.4 – Метод CutUniqueEndings

Номер тесту	Вхід	Вихід
1	finEnding=null, basesCount=2, sumEnding={{“/”}, {“/a”, “/ий”, “/им”, “/ими”, “/их”, “/і”, “/ім”, “/ій”, “ого”, “/ою”, “/ому”, “/ої”, “у”, “ого” }}	finEnding ={{“/”}, {“/a”, “/e”, “/ий”, “/им”, “/ими”, “/их”, “/і”, “/ій”, “ім”, “/ого”, “/ої”, “/ому”, “ою”, “у” }}, endingIndexes={0,1}

Метод 5. Формування списку унікальних морфологічних властивостей слів

Заголовок void GetUniqueProperties (string properties)

Метод приймає рядок string зі всіма атрибутивними властивостями слів для кожної словоформи, розділених символом переходу рядка “/n”. Записує унікальні властивості, отримані після порівняння рядків кожного з кожним, у файл, крім того записує індекси цих атрибутів, додаючи 0 на початок, щоб сумарна кількість цифр була 4. Вивід результатів у файл послідовно для кожного слова. Тест у табл. 3.5.

Таблиця 3.5 – Метод CutUniqueBases

Номер тесту	Вхід	Вихід
1	properties= "noninfl:abbr/n noun:inanim:m:v_kly:nv:/n intj/n intj/n"	uniqueProperties = "noninfl:abbr/n noun:inanim:m:v_kly:nv:/n intj/n intj/n", uniquePropertyIndexes= {“0000”, “0001”, “0002”}

Метод 6. Ітеративний бінарний пошук

Заголовок int BinarySearchIterative(List<Node> inputArray, string key)

Метод приймає список елементів Node префіксального дерева, що є фактично повним рівнем дочірніх елементів для поточного вузла, а також string

ключ вузла. Виконує бінарний пошук у відсортованому списку і повертає індекс знайденого елемента, інакше -1. У табл. 3.6 опускаються всі атрибути класу Node у списку inputArray окрім string Value, що є ключом і потрібен для пошуку.

Таблиця 3.6 – Метод BinarySearchIterative

Номер тесту	Вхід	Вихід	min>max	key.CompareTo(inputArray[mid].Value) < 0
1	inputArray= {“3”, “2”, “1”}, key= “1”	index= 2	-	+
2	inputArray= {“1”, “2”, “3”}, key= “4”	index= -1	+	null
3	inputArray= {“3”, “2”, “1”}, key= “3”	index= 0	-	-

Метод 7. Привласнення точкам нових класів за перерахованими відстанями до центрів кластерів

Заголовок void ClassesAllocation(out int ind1, out int ind2, List<int> clusterCenters, List<int> points, List<List<double>> distances)

Метод приймає вихідні параметри типу int, в які буде записано результативні індекси максимінних відстаней (у ind1 мінімальна відстань між елементами кластерів, у ind2 максимальна в середині кластеру), список індексів кластерних центрів, список точок та список списків відстаней між усіма точками. В явному вигляді не повертає нічого. В неявному перераховує відповідно до нових центрів кластерів всі класи точок і привласнює їм нові значення в разі їх зміни, а також індекси максимінних відстаней ind1, ind2. Тест у табл. 3.7.

Таблиця 3.7 – Метод ClassesAllocation

Номер тесту	Вхід	Вихід
1	clusterCenters= {1, 2}, points= {0, 0, 0}, distances= {{0, 3, 4}, {3, 0, 5}, {4, 5, 0}}	ind1= 0, ind2= 1, points= {1, 1, 2}

Метод 8. Кластеризація за атрибутами слів векторів.

Заголовок void Clusterize (List<List<string>> propertiesByText)

Метод приймає список списків індексів атрибутів на кожен текст у послідовності, в якій розташовані відповідні слова в текстах. В неявному вигляді повертає евклідові відстані між сформованими векторами атрибутивних послідовностей. Тест табл. 3.8.

Таблиця 3.8 – Метод Clusterize

Номер тесту	Вхід	Вихід
1	propertiesByText={{“0000”, “0001”, “0002”, “0000”, “0001”}, {“0000”, “0001”, “0000”, “0001”}}	euclideanDistances={{0, 0.223}, { 0.223, 0}}

Метод 9. Метод максимінних відстаней.

Заголовок List<string> UseMaximin(List<List<double>> distances)

Метод приймає список списків всіх відстаней від кожного вектору в просторі до кожного. Повертає кластери у вигляді списку згрупованих індексів векторів. Тест у табл. 3.9.

Таблиця 3.9 – Метод UseMaximin

Номер тесту	Вхід	Вихід	distances[minMaxInd1] [minMaxInd2] > centresDistancesSum
1	distances= {{3, 0, 5}, {4, 5, 0}, {0, 3, 4}}	points= {0, 1, 0}	-
2	distances= {{0, 2, 2.83, 2}, {2, 0, 2, 2.82}, {2.83, 2, 0, 2}, {2, 2.83, 2, 0}}	points= {0, 1, 2, 3}	+

Метод 10. Заповнення структури словника

Заголовок void InsertProcessing<T>(ref T str, List<string> endingIndexes, List<string> basicIndexes, List<string> endings, List<string> drops, List<string> pos) where T : IBaseStruct

Метод приймає шаблонну структуру даних, що реалізує інтерфейс IBaseStruct, список індексів закінчень, список індексів основ, список унікальних закінчень, список звуків чергування з їх індексами, список атрибутів слів. Може додатково завантажувати список атрибутів слів, якщо вони досі не проіндексовані. Тест представлено у табл. 3.10. Повертає в неявному вигляді заповнену структуру даних, а також формує список індексованих атрибутів слів, якщо програма виконується вперше.

Умовні позначення у табл 3.10:

File.Exists – 1,

endingIndexes[count] != "" && drops[count] == "" – 2,

drops[count] != "" – 3,

endingIndexes[count] == "" – 4.

Таблиця 3.10 – Метод InsertProcessing

Номер тесту	Вхід	Вихід	1	2	3	4
1	endingIndexes={“0, 1”}, basicIndexes={“0, 1”}, endings= {“/+а+і+ові+ом+у”}, drops={“1о”}, pos={“0”}, ={“”}, bases={“кіт”}	str= {{“кіт”, “00000”}, {“кота”, “00001”}, {“коті”, “00002”}, {“котові”, “00003”}, {“котом”, “00004”}, {“коту”, “00005”}}, propertiesList={0, 1, 2, 3, 4, 5}	-	-	+	-
2	endingIndexes={“0, 1”}, basicIndexes={“0, 1”}, endings= {“/+а+і+ові+ом+у”}, drops={“1а”}, pos={“0”}, ={“”}, bases={“кіт”}	str= {{“кіт”, “00000”}, {“кота”, “00001”}, {“коті”, “00002”}, {“котові”, “00003”}, {“котом”, “00004”}, {“коту”, “00005”}}, propertiesList={}	+	-	+	-
3	endingIndexes={“0, 1”}, basicIndexes={“0, 1”}, endings={“/”}, drops={“”}, pos={“0”}, ={“”}, bases={“ тепло ”}	str= {“тепло”, “00000”}, propertiesList={}	+	-	-	+
4	endingIndexes={“0, 1”}, basicIndexes={“0, 1”}, endings={“//ам/ах/е/и/і/ів /ові/ом/у/”}, drops={“”}, pos={“0”}, ={“”}, bases={“жарт”}	str= {{“жарт”, “00000”}, {“ жартам ”, “00001”}, {“ жартах”, “00002”}, {“ жарте”, “00003”}, {“ жарти”, “00004”}, {“ жарті”, “00005”}, {“ жартів”, “00006”}, {“ жартові”, “00007”}, {“ жартом”, “00008”}, {“ жарту”, “00009”}}, propertiesList={}	+	+	-	-

Метод 11. Розбір текстів на слова та їх пошук у шаблонній структурі

Заголовок void SearchProcessing<T>(List<string> bases, string outDirName, T str , string dirName) where T : IBaseStruct

Метод приймає список основ (нова основа з нового рядку через “\n”), директорію для запису результатів та директорію з текстами для аналізу. Компонує розбір тексту на слова, пошук слів за словником та переведення їх у атрибути, кластеризацію. Записує дані статистики розібраних текстів у файл [index]map.txt (відсоток знайдених слів, складає словник тексту та частоту зустрічі кожного слова), а також індекси основ з індексом атрибуту у файл [index].txt, де [index] – порядковий номер тексту. Тест у табл. 3.11.

Таблиця 3.11 – Метод SearchProcessing

Номер тесту	Вхід	Вихід	Directory.Exists (dirName)	index!=“-1”
1	bases={“гром”}, outDirName= “ResultDictionary”, str= {{“кіт”,00000}, {“лис”,10001}}, dirName= “Samples”, words= “кіт\nлис\n”	posMap= “”, statRes= “”	-	null
2	bases={“гром”}, outDirName= “ResultDictionary”, str= {{“кіт”,00000}, {“лис”,10001}}, dirName= “Samples”, words= “кіт\nлис\n”	posMap= {“0000”, “0001”}, statRes= “unknown 0 total 2 1”	+	+
3	bases={“гром”}, outDirName= “ResultDictionary”, str= {{“кіт”,00000}, {“лис”,10001}}, dirName= “Samples”, words= “собака\nмиша\n”	posMap= “”, statRes= “unknown 2 total 2 0”	+	-

4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК ОБРОБКИ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТЕКСТІВ

Перш за все необхідно пересвідчитися в правдоподібності обраного методу моніторингу часу тестових методів та впливи на їх швидкодію з боку операційної системи, середовища розробки. Для цього необхідно провести серію тестових прогонів програми за різних умов.

Припустимо, що існує вплив середовища розробки при виконанні програми. Перевіримо це, порівнявши середній час виконання одної функції в середовищі Visual Studio та напряму з операційної системи. Для тестування взято метод для утворення структури даних з набору текстових файлів структурованого словника.

Використана структура даних словник. Робота методу полягає в утворенні з модулів словнику структури даних в оперативній пам’яті. Всі модулі попередньо завантажуються з жорсткого диску в оперативну пам’ять. Метод відтворює всі слова за проіндексованими модулями словнику та працює виключно в оперативній пам’яті, тому вплив файлової системи відсутній. Окрім тестового додатку в операційній системі під час тестів інші додатки, що потенційно можуть завантажувати центральний процесор (ЦП), вимкнені. Середнє навантаження на ЦП фонових програм становило 5-7%.

Даний вибір здійснено з метою економії часу на тестування, адже очікуваний час заповнення структури складає 3,5 секунди, а також відсутність впливу файлової системи через роботу всіх компонентів в оперативній пам’яті.

Серія з 10 тестових замірів часу виконання в операційній системі показала середній час 3,356 с, а з середовища розробки 3,698 с. Таким чином можна стверджувати, що внаслідок інтегрованих засобів відлагодження програми середовище розробки потребує додаткових ресурсів ЦП, тому всі наступні тести потрібно здійснювати безпосередньо з операційної системи.

4.1 Структуризація словника без втрат інформації

За базовий словник для розпізнавання слів української мови взято ВЕСУМ, що містить надмірну інформацію та може бути зменшений без втрати ключових даних для дослідження.

Першим кроком структурування є виділення частини слова серед словоформ, що є незмінною. При цьому необхідно врахувати факт чергування звуків, що можуть зустрічатися навіть на початку слова, як на рис. 4.1. У цьому випадку складно довести, що слово «овець» є словоформою від «вівця». Тому треба ввести нову індексацію утворених основ, що базується на наступних принципах:

– якщо у ВЕСУМ слово позначене як словоформа (зміщене на 2 пробіли), то індекс посилається на базове слово в називному відмінку;
– якщо певне слово має спільну частину з іншим словом у словнику з відмінністю лише префіксальної частини, а список закінчень співпадає, то індекс посилається на перше зустрічне слово за алфавітом в називному відмінку, зо відповідає критерію співпадіння.

```
вівця noun:anim:f:v_naz
  вівцю noun:anim:f:v_zna
  вівцею noun:anim:f:v_oru
  вівці noun:anim:f:v_mis
  вівце noun:anim:f:v_kly
  вівці noun:anim:p:v_naz
  вівці noun:anim:f:v_rod
  вівцям noun:anim:p:v_dav
  вівці noun:anim:p:v_zna
  вівці noun:anim:f:v_dav
  вівцями noun:anim:p:v_oru
  вівцях noun:anim:p:v_mis
  вівці noun:anim:p:v_kly
  овець noun:anim:p:v_rod
  овець noun:anim:p:v_zna
```

Рисунок 4.1 – Фрагмент представлення словоформ із ВЕСУМ

В результаті побудови нової структури, додавши до всіх основ їх списки закінчень, ми отримаємо множину слів із словнику ВЕСУМ, однак завдяки проведеній індексації загальний об’єм пам’яті, що займають усі модулі, становитиме 48,7 МБ, а отже вдалося провести стиснення початкового словнику у 5,66 рази без

втрати ключової інформації для подальшого переведення слів у їх атрибутивні одиниці на етапі розбору текстів. Загальний час для опрацювання словника ВЕСУМ модулем першої програми становить 19 хвилин і 31 секунду.

4.2 Дослідження операції виділення слів із тексту

Аналіз текстів здійснюватиметься на основі морфологічних атрибутів слів. Тому необхідно виділити всі слова з вхідних текстів, пропустивши всі роздільні знаки, а також перевести всі знайдені слова в нижній регістр, в якому перебувають всі слова в словнику.

Для тестування операції виділення слів було сформовано 5 тестових наборів різної тематики та змісту, всі файли вибірки використовують кодування UTF-8:

- перший – стягнені з бази даних Вікіпедії [18] статті українською мовою;
- другий – повідомлення із соціальних мереж;
- третій – 444 твори української літератури;
- четвертий – 18 збірників тезисів з різних конференцій;
- п’ятий – згенерований на основі речень попередніх файлів, не містить знаків пунктуації.

Всі файли окремих наборів злито в єдину структуру даних в оперативній пам’яті, щоб виключити вплив з боку файлової системи на швидкість виділення слів. У табл. 4.1 представлено час виділення слів із текстів кожної із сформованої вибірок даних. Таким чином можна простежити зміну часу виділення слів від тематики тексту та частоти вживання знаків пунктуації.

Таблиця 4.1 – Залежність швидкості розбору текстів від виду набору

Тестовий набір	Займана пам’ять, МБ	Час виділення, с	Швидкість, МБ/с
1	2228,1	101,4	21,98
2	3	0,13	23,08
3	94,1	4,32	21,78
4	16,3	0,72	22,63
5	189	9,24	20,45

З метою економії оперативної пам’яті під час виконання програми розбір тексту на слова проходить по рядкам. Сформований список слів із рядку подається на вхід функції пошуку, після чого заміщується наступним рядком. Функція запису пошукових результатів напроти зберігає статистичні дані проаналізованих в межах повного розбору одного текстового файлу. Після обробки статистичні дані записуються в файл і звільняють оперативну пам’ять, виділену на них.

Тому для отримання часових показників функцій розбору тексту та пошуку було проведено 2 тести. В першому (рис. 4.1) замірювався сумарний час на виконання обох операцій. У другому (рис. 4.2) вимкнено функцію пошуку.

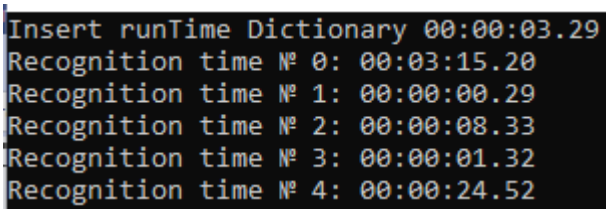


Рисунок 4.2 – Тест виділення слів + пошук

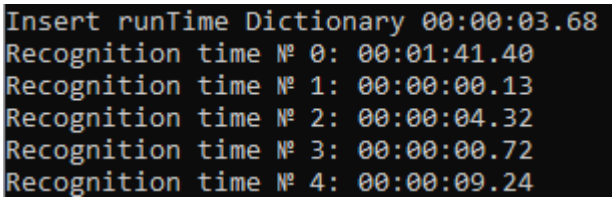


Рисунок 4.3 – Тест без пошуку

У табл. 4.2 наведено дані з обсягу тестових вибірок та частку розпізнаних у них слів.

Таблиця 4.2 – Частка розпізнаних слів серед різних тестових даних

Тестовий набір	Всього слів	Нерозпізнаних слів	К-ть унікальних слів	Частка нерозпізнаного, %
1	194 071 570	44 080 135	195 429	22,71
2	257 649	12 082	22 120	0,06
3	8 791 400	1 431 604	86 954	0,7
4	1 322 616	289 882	31 229	0,14
5	18 620 839	1 163 959	109 055	0,52

За результатами з табл. 4.2 можна стверджувати про значний процент розпізнавання текстів. Виключення становить база даних із Вікіпедії, однак дане явище викликано тим, що частка іномовних слів у ній явно більша, ніж у звичайних літературних творах. Тому можна стверджувати за іншими тестовими вибірками, що визначення виключно українських слів програмою здійснюється в достатньому обсязі, з чого робимо висновок про повноту словнику ВЕСУМ та його вдалому виборі для задач, поставлених у цій роботі.

4.3 Дослідження швидкодії структур даних

4.3.1 Моніторинг виконання операцій структур даних

Для вимірювання часу виконання операцій вставки та пошуку був застосований клас Stopwatch, що виконує роль таймеру та засікає момент часу до та після виконання тестових методів. Кінцевий час виконання методу буде рівний різниці часу початку та кінця його роботи.

Для знайдення розміру структури даних було застосовано засіб Diagnostic tools у Visual Studio. Під час виконання програми за допомогою розділу Memory usage можна відстежити споживання оперативної пам’яті всіма об’єктами в конкретний момент часу. Розмір програми в ОП у момент входу до функції заповнення структури даних складає 470 МБ і залишається статичним до моменту заповнення структур даних.

У табл. 4.3 представлено результати здійснених вимірів.

Таблиця 4.3 – Порівняльна характеристика структур даних

Характеристика	SortedList	Trie	Dictionary
Операція вставки, с	14,07	19,04	3,356
Операція пошуку, с	172,15	119,05	111,79
Займана оперативна пам’ять, МБ	258,71	778,5	197,4

Опираючись на отримані результати швидкодії структур можна вибрати кращу серед них. За всіма параметрами кращою виявилася структура даних Dictionary, що фактично є строго типізованою хеш-таблицею.

За часом пошуку близькою до неї стало префіксальне дерево trie, що, однак, має значну перевагу при необхідності видачі проміжних результатів пошуку, однак за споживаною ОП є надмірною. Завдяки цьому вона ефективно використовується в інтернет-системах пошуку, таких як Google, Yandex, Bing і т.д. Однак у цьому дослідженні так і не знайшли свого місця методи нечіткого пошуку, що теоретично могли бути застосовані для аналізу тексту з граматичними помилками. Однак під час аналізу наукового чи літературного тексту відсоток зустрічі помилок мізерний через його попередню перевірку видавництвом, тому було вирішено ігнорувати факт зустрічі помилок у тексті.

Відсортований список хоч і має досить ефективний алгоритм, однак далекий від хеш-таблиці. Його основна перевага в можливості послідовного обходу від одного елемента до іншого за час O(1), чого не може хеш-таблиця. Дана властивість корисна при аналізі серед групи слів, однак у цій роботі не потребується.

4.3.2 Порівняльний аналіз бібліотеки CMPH з кращою серед обраних структур даних

Розглянута раніше швидкодіюча структура даних мінімальна ідеальна хеш-таблиця, що будується за алгоритмом CHD реалізована лише у складі бібліотеки CMPH, що в свою чергу призначена для операційних систем Linux. Через це неможливо інтегрувати її модуль в розроблювану програму, однак можна провести тести в Linux цієї бібліотеки на тих же наборах даних і порівняти результати з кращою обраною структурою, доступною в .NET.

Для того, щоб порівняти швидкодію обох структур створено тестовий файл розміром 208 МБ зі словами, що заздалегідь розібрані за текстами з вибірок раніше. Таким чином порівнюватимуться операції вставки та пошуку між структурою Dictionary C# та структурою бібліотеки CMPH, що виконуватиметься в операційні системі Linux на віртуальній машині.

Результати проведеного тестування представлено на рис. 4.4 та 4.5.

```
Insert runTime Dictionary 00:00:03.30
Words search time 00:00:13.01
```

Рисунок 4.4 – Час виконання вставки та пошуку Dictionary

```
alex@alex-VirtualBox:~$ time cmph -v -m out66.txt.mph words.txt > /dev/null 2>/dev/null

real    0m37,066s
user    0m30,390s
sys     0m6,674s
alex@alex-VirtualBox:~$ time cmph -v -g out66.txt
Entering mapping step for mph creation of 3485604 keys with graph sized 7284913
Starting assignment step
Successfully generated minimal perfect hash function

real    0m3,831s
user    0m3,494s
sys     0m0,262s
```

Рисунок 4.5 – Тестування бібліотеки CMPH

З отриманих даних можна виділити наступні ключові показники: операція вставки 3.3 с у Dictionary проти 3.8 с CMPH, пошуку 13 с проти 37 с відповідно. Однак обидві операції CMPH обмежені можливостями файлової системи у своїй роботі через особливості роботи, чого позбавлена структура Dictionary. До того ж CMPH за швидкодією обмежена середовищем виконання віртуальної машини. Внаслідок цього можна стверджувати, що операція вставки CMPH щонайменше не гірша за швидкодією до Dictionary, а пошук у CMPH повинен бути не гіршим за теоретичними обґрунтуваннями мінімальної ідеальної хеш-функції. Тому Dictionary можна вважати наближеною за швидкодією до неї, що виводить її на рівень кращих хеш-таблиць.

4.4 Дослідження кластеризації текстів за авторами на основі морфологічних атрибутів

Для наступного експерименту сформовано вибірку з 9 творів 3 різних авторів, по 3 на кожного, в дужках розмір і порядковий номер у тесті відповідно:

– Михайло Стельмах – «Велика рідня» (3480 КБ, 1 і 10), «Казка про правду та кривду» (73 КБ, 2), «Дума про тебе» (1267 КБ, 3);

– Михайло Коцюбинський – «В путах шайтана» (43 КБ, 4), «Intermezzo» (41 КБ, 5), «Fata Morgana» (311 КБ, 6);
– Іван Франко – «Schon Schreiben» (20 КБ, 7), «Boa constriktor» (279 КБ, 8), «Без праці» (229 КБ, 9).

Для контрольної перевірки твір «Велика рідня» було розділено на 2 частини близькі за розміром. Таким чином можна перевірити точність знайдення відстаней в рамках одного авторського твору, що повинен відповідати єдиному стилю написання і побудови структури словосполучень і речень. Навіть твори одного автора можуть сильно різнитися внаслідок розвитку його стилю протягом життя, однак перевірка частин одного твору дасть чітку оцінку роботі програми щодо чіткості порівняння текстів за морфологічними атрибутивними послідовностями.

Далі необхідно знайти відстані між векторами. Для цього було застосовано евклідову відстань між точками у просторі [27]. Результати пошуку відстаней між векторами представлено на рис. 4.6.

	1)	2)	3)	4)	5)	6)	7)	8)	9)	10)
1)	0	0,01716	0,00732	0,01711	0,0193	0,01074	0,0187	0,01231	0,01301	0,00461
2)	0,01716	0	0,01782	0,02102	0,02223	0,01765	0,02198	0,01876	0,01926	0,01768
3)	0,00732	0,01782	0	0,01844	0,01987	0,01146	0,0202	0,01295	0,01283	0,00777
4)	0,01711	0,02102	0,01844	0	0,02142	0,01709	0,02072	0,01797	0,02019	0,017
5)	0,0193	0,02223	0,01987	0,02142	0	0,0193	0,02263	0,01945	0,0205	0,01944
6)	0,01074	0,01765	0,01146	0,01709	0,0193	0	0,01893	0,01278	0,01453	0,01087
7)	0,0187	0,02198	0,0202	0,02072	0,02263	0,01893	0	0,0175	0,01964	0,0185
8)	0,01231	0,01876	0,01295	0,01797	0,01945	0,01278	0,0175	0	0,01279	0,01221
9)	0,01301	0,01926	0,01283	0,02019	0,0205	0,01453	0,01964	0,01279	0	0,01325
10)	0,00461	0,01768	0,00777	0,017	0,01944	0,01087	0,0185	0,01221	0,01325	0

Рисунок 4.6 – Евклідові відстані між векторами атрибутивних послідовностей

Якщо проаналізувати відстані між векторами, то можна помітити, що найменша відстань між вектором 1 і 10, тобто між двома частинами твору «Велика рідня». Наступний за відстанню до 1-го є вектор 3, що також написаний М. Стельмахом і за жанром обидва твори є романами. За відносними відстанями ці вектори знаходяться в 2-3 рази ближче одне до одного, що свідчить про високу ступінь їх подібності.

Для більш наглядного аналізу решти текстів на основі отриманих відстаней треба здійснити кластеризацію за відстанями. Для цього було використано алгоритм максимінних відстаней. Результати кластеризації представлені на рис. 4.7.

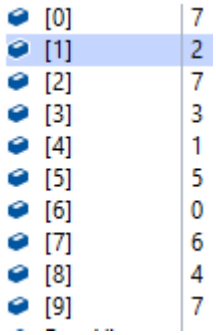


Рисунок 4.7 – Кластери творів

Порівнявши дані кластеризації з реальними авторами творів можна стверджувати, що правильно встановлений кластер творів лише автора Стельмаха, а саме 2-х частин «Велика рідня» та «Дума про тебе». Такий результат є результатом невдало вибраного методу кластеризації, що викликано подібною близькістю всіх векторів одне до одного. Лише 2 частини одного твору можуть бути точно інтерпретовані як єдиний кластер, інші ж кластери не формуються в достатньому радіусі через надто високу планку задання мінімальної внутрішньокластерної відстані між векторами внаслідок 2-х частин одного і того ж твору у вибірці.

Висновки до розділу 4

Після проведення експериментів було знайдено кращу структуру даних – Dictionary. Саме вона забезпечує найшвидші операції вставки та пошуку та потребує найменшого місця в ОП. Порівняльний аналіз із рядом структур та мінімальною ідеальною хеш-таблицею показав, що Dictionary є конкурентоспроможним аналогом для СМРН і задовольняє умовам швидкодійної та ефективної по займаній пам’яті для розробленого додатку, тому його використання доцільне для зберігання та пошуку в словнику українських слів.

5 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1 Вимоги безпеки праці під час виконання робіт за персональними електронно-обчислювальними машинами

Згідно до ст. 1 Закону України від 2002 р. «Про Охорону праці» [28] поняття охорона праці – це система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних і лікувально-профілактичних заходів та засобів, спрямованих на збереження життя, здоров'я і працездатності людини у процесі трудової діяльності.

Наступні нормативно-правові акти визначають норми щодо охорони праці за персональним комп’ютером:

- ДСанПіН 3.3.2.007-98 Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин затверджені постановою Головного державного санітарного лікаря України від 10.12.1998 № 7 [29];
- НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями» затверджені наказом Міністерства соціальної політики України від 14 лютого 2018 №207 [30];
- примірна інструкція з охорони праці під час експлуатації електронно-обчислювальних машин, впроваджена наказом Міністерства доходів і зборів України від 05.09.2013 № 443 [31].

Дані документи дають перелік вимог та спосіб роботи з електронно-обчислювальними пристроями на підприємствах. У документі [30] зведені основні вимоги безпеки при проведенні робіт з екранними пристроями, за яким інженер-програміст витрачає більше всього часу при розробці програмного забезпечення у робочий час. Основні вимоги під час роботи екранними пристроями наступні:

- щодня перед початком роботи необхідно візуально оглянути прилад та його комплектуючі на відсутність механічних пошкоджень, витирати прилад від пилу та інших забруднень;
- після завершення роботи екранні пристрої потрібно відключати від електричної мережі;
- у випадку виникнення аварійної ситуації необхідно негайно відключити екранний пристрій від електричної мережі.

Забороняється виконувати зміни в конструкції та складі екранних пристроїв, самостійно виконувати ремонт і налагодження екранних пристроїв на робочому місці під час безпосередньої роботи, працювати з екранними пристроями, які мають явно виражені несправності в роботі. Під час операторської роботи за екранними пристроями, що пов’язана із нервово-емоційним напруженням, мають дотримуватися оптимальні умови мікроклімату відповідно до ДСН 3.3.6.042-99 державних санітарних норм мікроклімату виробничих приміщень затверджених Головним державними санітарним лікарем України від 1 грудня 1999 року № 42 [32].

У ДСанПіН 3.3.2.007-98 [29] викладені вимоги до виробничих приміщень в яких розміщується електронно обчислювальна техніка, а також необхідні гігієнічні умови до організації роботи. Гігієнічні правила містять гігієнічні й ергономічні вимоги до організації робочих приміщень, та робочих місць параметрів робочого середовища, дотримання яких дасть змогу запобігти порушенням у стані здоров'я користувачів ЕОМ та ПЕОМ.

Основними вимогами до приміщень є: площа на одне робоче місце повинна бути не менше ніж 6 м^2, а об’єм не менше 20 м^3, при розміщенні робочих столів необхідно, щоб відстань між двома екранними приладами становила не менше 1,2 м, у приміщеннях обов’язкова наявність аптечка першої медичної допомоги, робочі місця з екранними приладами слід розташовувати так, щоб природне світло проходило через світлові прорізи, орієнтовані переважно на північ чи північний схід і забезпечували коефіцієнт природною освітленості (КПО) не нижче ніж

1,5%. Віконні прорізи приміщень для роботи з ВДТ мають бути обладнані регульованими пристроями для можливості зміни рівню освітленості. Заборонено розташування робочого місця у підвальних приміщеннях. При приміщеннях з ВДТ повинні бути обладнані побутові приміщення для відпочинку під час роботи.

Висота робочої поверхні столу з екранним приладом має регулюватися в межах 680-800 мм, робочий стілець має бути зручним, підйомно-поворотним, регульованим за висотою, екран повинен розташовуватися на оптимальній відстані від очей користувача, що становить від 600 до 700 мм та під кутом 30° до нормальної лінії погляду працюючого. Клавіатура розташовується на столі на відстані від 100 до 300 мм від краю, що направлений до робітника, поверхня клавіатури має бути матовою з коефіцієнтом відбиття 0,4, її розташування має забезпечувати повну видимість екрана для зручності ручного керування.

Шкідливі виробничі фактори – фактори, регулярний вплив яких на працюючого у визначених умовах протягом певного часу спричинить захворювання, зниження працездатності та негативного впливу на здоров'я нащадків. У залежності від рівня і часу впливу, шкідливі фактори можуть класифікуватися як небезпечні.

При роботі з програмою «AuthorshipClusterizer» програміст багато часу проводить за персональним комп’ютером з екранним пристроєм.

Робота на комп’ютері пов’язана з такими шкідливими факторами:

- зорове навантаження (тривале фокусування зору на близькому об’єкті, відблиски світла);
- порушення функцій опорного апарату;
- електромагнітні випромінювання;
- обмеження рухової активності протягом тривалого часу;
- синдром зап’ястного каналу.

Для зниження впливу шкідливих факторів робоче місце програміста має відповідати нормам, наведеним у документах [29-30].

Проблеми із зором одна з найбільших проблем під час роботи за ЕОМ через те що зорова система людини погано пристосована до роботи з комп'ютерним зображенням. Екранне зображення відрізняється від природного тим, що воно:

- не цілісне, а складається з дискретних точок – пікселів;
- має значно менший контраст, який знижується за рахунок зовнішнього освітлення;
- зображення на екрані мерехтливе, що збільшує навантаження на зоровий апарат;
- зображення на екрані має значні перепади яскравості порівняно з оточуючим середовищем.

Для зменшення зорового напруження під час роботи за комп'ютером рекомендується робити регулярні перерви. Поширеним є умовне правило 20 хвилин, що полягає у введенні відпочинку для очей кожні 20 хвилин роботи за ПК, а саме щонайменше на 10 секунд відводити погляд на віддалений об'єкт. Наступним кроком є вибір якісного монітору з великим показником точок на дюйм – dpi. Якщо монітор з певних причин не можна замінити на більш безпечний рідкокристалічний, то можна застосовувати спеціальні комп'ютерні окуляри, які знижують навантаження на зоровий апарат.

Ще одним негативним для зору фактором від монітора є відблиски на екрані, що є результатом неправильно влаштованого освітлення. Для запобігання дії даного фактору варто дотримуватись вимог державних будівельних норм ДБН В.2.5-28:2018 «Природне і штучне освітлення», затверджених наказом Міністерства регіонального розвитку, будівництва та житлово-комунального господарства України від 3 жовтня 2018 року №264 [36], або застосовувати рідкокристалічні монітори, конструкція яких майже повністю попереджує виникнення відблисків через менший коефіцієнт відбиття світла.

Порушення у роботі опорного апарату (сколіоз, кіфоз, лордоз) є поширеним захворюванням серед інженерів-програмістів. Основною профілактикою є комплекс

гімнастичних вправ та правильне облаштування робочого місця, в першу робочого стільця, столу та монітору з клавіатурою згідно до вимог документа [50].

Електромагнітне випромінювання має бути в межах гранично допустимих норм, які встановлені Державними санітарними нормами і правилами захисту населення від впливу електромагнітних випромінювань ДСН 239-96, затверджених Міністерством охорони здоров'я 1 серпня 1996 року №239 в редакції від 22 грудня 2017 року [35].

Головною профілактикою гіподинамії, що виникає внаслідок обмеженої рухової діяльності, є виконання комплекс простих фізичних вправ (кругові оберти голови, суглобів рук та ніг, присідання, випади вперед) в перервах від робіт або піша прогулянка під час обідньої перерви.

Синдром зап'ястного каналу – патологічний стан, що характеризується болем, відчуттям оніміння і поколювання в пальцях руки і самої, що виникає внаслідок здавлювання серединного нерву в зап'ястковому каналі протягом тривалого часу. У профілактиці захворювання допоможуть короткі перерви від роботи для виконання спеціального комплексу вправ з розминки кисті, суглобів рук та передпліччя або застосування спеціального бандажу під час роботи.

Мікроклімат у виробничих приміщеннях повинен дотримуватися згідно до норм мікроклімату у ДСН 3.3.6.042-99 [32]. Мікроклімат (метеорологічні умови) на робочому місці суттєво впливає на працездатність працівника та його стан організму. Характеризують клімат внутрішнього середовища виробничого приміщення показники температури, відносної вологості, швидкості руху повітря, теплового випромінювання нагрітих поверхонь, з якими взаємодіє працівник. Далі наведено ключові показники мікроклімату робочого приміщення та межі для норм оптимальної роботи:

- температура повітря в холодну пору року в робочому приміщенні 20 ºС, а в теплу пору року 24 ºС (при оптимальних нормах в холодну пору року – 16 – 24 ºС; в теплу пору року – 18 – 25 ºС);
- відносна вологість в діапазоні 45 – 55% (оптимальна 40 – 60%);

– швидкість руху повітря – в холодну пору року 0,2 м/с (при нормі 0,1 – 0,3 м/с) , а в теплу пору 0,4 (при нормі 0,2 – 0,4 м/с).

Рівень навколишнього шуму у робочому приміщені регламентується наказом Міністерства охорони здоров'я «Про затвердження Державних санітарних норм допустимих рівнів шуму в приміщеннях житлових та громадських будинків і на території житлової забудови» від 22 лютого 2019 року №463 [35]. Гучність звуку в приміщені становить 40 дБа (при допустимій нормі в 50 дБа), критерії шуму становлять 40, що задовольняє допустимій нормі.

Норми, які регламентують рівень виробничої загальної та локальної вібрації наведено в Державних санітарних нормах виробничої загальної та локальної вібрації ДСН 3.3.6.039-99, затверджених постановою Головного санітарного лікаря України від 1 грудня 1999 року №39 [36]. Допустимий показник для локальної вібрації в приміщенні, в якому проводиться розробка, становить 115 для віброшвидкості та 73 для віброприскорення впродовж восьмигодинного робочого дня, що відповідає нормам [36]. Рівень фонової виробничої вібрації в офісі також в межах норми.

Приміщення в яких встановлені персональні комп'ютери, повинні мати природне та штучне освітлення відповідно до ДБН В.2.5-28:2018 [34]. Мінімальна освітленість встановлюється в залежності від розряду виконуваних зорових робіт. Для робітників сфери розробки програмного забезпечення вона прирівняється до IV розряду і складає 300...500 лк.

Визначимо освітленість робочого місця програміста за ПК на відповідність характеру зорової роботи. При роботі з дисплеями датчик вимірювання рівню природної освітленості поверхні, де розташований ПК, показав 350 лк. Освітленість тієї ж поверхні відкритим небосхилом становить 20000 лк, тобто КПО = 1,75%, що відповідає нормативному КПО.

Для штучного освітлення як засіб допоміжного освітлення при хмарній погоді чи вечірній час у приміщенні наявні люмінесцентні лампи.

Розрахунок штучного освітлення здійснюватиметься для приміщення площею 12 м^2, ширина якого складає 3м, довжина – 4м, висота – 3,5м.

Застосуємо метод використання світлового потоку. Щоб визначити необхідну кількість освітлювальних пристроїв для забезпечення нормованого рівня освітленості, розрахуємо світловий потік, що падає на робочу поверхню за формулою 5.1:

$$F = \frac{E*K*S*Z}{\eta}, \quad (5.1)$$

де F – світловий потік, що розраховується, Лм;

E – нормована мінімальна освітленість для обраного виду роботи, Лк; E = 300 Лк;

S – площа приміщення (у нашому випадку S = 12м^2);

Z – відношення показнику середньої освітленості до мінімальної (зазвичай приймається рівним 1,1... 1,2, візьмемо значення Z = 1,1);

K – коефіцієнт резерву, який дозволяє врахувати зменшення світлового потоку освітлювачів у результаті їх експлуатації (значення варіюється в залежності від характеру робіт, типу приміщення та періоду робочого часу, в нашому випадку К = 1,5);

η – коефіцієнт використання світлового потоку. Виражається відношенням світлового потоку, що потрапляє на робочу поверхню, до загального потоку всіх освітлювачів, і обчислюється в долях одиниці. Залежить від характеристик освітлювача, об'єму приміщення, кольору і фактурних характеристик поверхонь стін і стелі, що характеризуються коефіцієнтами відбиття від стін ($\rho_{ст.}$) і стелі ($\rho_{стелі}$)), в нашому випадку значення коефіцієнтів дорівнюють $\rho_{ст}$ = 30% і $\rho_{стелі}$= 50%.

Розрахуємо індекс за формулою 5.2:

$$I = \frac{S}{h(A+B)}, \quad (5.2)$$

де S – площа приміщення, S = 12м^2;

h – розрахункова висота стелі, h = 3,5 м;

A – ширина приміщення, A = 3 м;

B – довжина приміщення, B = 4 м.

$$I = \frac{12}{3,5*(3+4)} = 0,49.$$

Отримавши індекс приміщення I, за таблицею із документу [54] знаходимо $\eta = 0{,}2$.

Підставимо всі значення у формулу для визначення світлового потоку F:

$$F = \frac{300*1{,}5*12*1{,}1}{0{,}2} = 29250 \text{ Лм.}$$

Для освітлення використані освітлювачі з люмінесцентними лампами типу ЛБ 40-1, світловий потік яких складає F = 4320 Лм. Розрахуємо мінімальну необхідну кількість світильників за формулою 5.3:

$$N = \frac{F}{Fл}, \tag{5.3}$$

де N – кількість ламп, що визначається;

F - світловий потік, прийняте значення F = 29250 Лм;

$F_л$- світловий потік лампи, $F_л$ = 4320 Лм.

$$N = \frac{29250}{4320} = 7.$$

У приміщенні розташовані світильники типу відкритий дволамповий (ВД). Кожен світильник комплектується двома лампами. Загалом необхідно обладнати приміщення щонайменше 4 світильниками із 7 працюючими лампами в них (з яких два знаходяться безпосередньо над робочим місцем).

5.2 Дії працівників у надзвичайних ситуаціях

Дії працівників при аварійних ситуаціях, зокрема при враженні електричним, регламентуються Порядком надання домедичної допомоги постраждалим при ураженні електричним струмом та блискавкою, затвердженим Міністерством охорони здоров'я України від 16 червня 2014 №398 [37].

Першим кроком є виклик медичної допомоги за номером 103. Перелік заходів першої допомоги [38] варіюється в залежності від стану потерпілого

припинення дії на нього електричного струму. Для визначення стану необхідно вжити наступні заходи:

- покласти потерпілого спиною на тверду поверхню;
- перевірити наявність у потерпілого дихання;
- перевірити наявність у потерпілого пульсу;
- з'ясувати стан зіниць, їх розширення вказує на погіршення кровопостачання.

У всіх випадках ураження електричним струмом [39] виклик лікаря є обов'язковим кроком незалежно від стану потерпілого.

Якщо потерпілий знаходиться при свідомості, його треба покласти у зручне положення і до прибуття лікаря забезпечити спокій, обов'язково спостерігаючи за диханням і пульсом. Обмежити рухову активність потерпілого, заборонити продовжувати роботу [40-43].

Якщо потерпілий у непритомному стані, його необхідно покласти на спину, розстебнути верхній одяг, забезпечити приплив свіжого повітря, дати понюхати нашатирний спирт, бризнути на нього водою і забезпечити спокій. Якщо дихання потерпілого уривчасте, рідке і судомне, йому необхідно робити штучне дихання і непрямий масаж серця [46-48].

Задля запобігання пожежі у робочому приміщенні необхідно дотримуватися загальних правил пожежної безпеки [49].

У виникненні ознак пожежі необхідно дотримуватися наступних правил:

- повідомити службу пожежної безпеки про факт пожежі за вашою адресою;
- за можливості вжити заходів протидії пожежі за допомогою вогнегасника;
- розпочати процес евакуації з приміщення;
- повідомити керівництво про факт пожежі;
- перед тим як схопитися за ручку дверей, доторкніться до неї тильною стороною долоні задля перевірки температури;

– обов‘язково зачиняйте за собою усі двері;

– при великій задимленості приміщення пересувайтесь поповзом.

Висновки до п‘ятого розділу

У даному розділі були розглянути основні питання з безпеки та охорони праці під час процесу розробки програмного додатку за персональним комп’ютером.

Розглянуті нормативні документи та законодавчі акти, в яких встановлені норми та вимоги до робочого процесу, середовища, де здійснюється розробка. Визначені основні параметри мікроклімату, вібрації, шуму та освітлення для інженера-програміста. Після їх визначення результати були співставленні з нормативними значеннями, описаними у відповідних документах.

Також були розглянуті порядки дій при наданні домедичної допомоги та дії при пожежній небезпеці та враженні струмом. Проаналізовані відповідні нормативні документи з протоколами дій для запобігання та під час виникнення надзвичайних ситуацій.

Висновки

В роботі досліджено низку структур даних для ефективного зберігання та доступу до словника української мови з атрибутами слів, проведено реструктуризацію словника ВЕСУМ для компактного зберігання даних, а також розроблено відповідні програмні засоби для вимірів швидкодії операцій над словником. Крім того, розроблено метод аналізу текстів на подібність через послідовності атрибутів слів, що також передбачає кластеризацію для наглядності.

В результаті отримано структурований словник без втрат необхідної інформації, що у 5,66 раз менший за розміром, порівняно з ВЕСУМ. Обрано кращу структуру даних для зберігання повного словнику з атрибутами в оперативній пам’яті – Dictionary зі стандартної бібліотеки .NET. Вона за всіма параметрами (розмір, швидкодія пошуку і вставки) переважає над іншими варіантами.

Розроблені програмні засоби для розбору текстів на слова, переведення їх в морфологічні атрибутивні послідовності, що являють собою вектори, між якими вимірюються евклідові відстані та проводиться кластеризація, що відображає подібність текстів, а отже може слугувати основою для прийняття рішення щодо належності тексту до автора, якщо роботи такого наявні в базі даних і тексти були віднесені до певного кластеру. Аналіз ефективності розпізнавання автору показав, що частини одного твору знаходяться в просторі набагато ближче, ніж всі інші твори, тобто можна стверджувати, що атрибутивні послідовності є авторським інваріантом і можуть застосовуватися при дослідженні авторства тексту.

Бібліографічний список

1. Демидович IEEE 16th International Conference on Computer Science and Information Technologies. 22-25 September 2021, Lviv, Ukraine. (виконано доповідь, знаходиться в процесі публікації)

2. Дослідження виявлення авторського інваріанту [Ел. ресурс]. Доступ: https://ru.wikipedia.org/wiki/Авторский_инвариант [Дата звернення: 11.11.2021]

3. Романов А. Шелупанов А. Бондарчук С. Обобщенная методика идентификации автора неизвестного текста // Доклады Томского государственного университета систем управления и радиоэлектроники : журнал. — 2010. — № 1(21). — С. 108-112. — ISSN 1818-0442.

4. Jaccard, P. (1901) Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin de la Société Vaudoise des Sciences Naturelles 37, 547—579.

5. Аналіз пошукових структур даних. [Ел. ресурс]. Доступ: https://neerc.ifmo.ru/wiki/index.php?title=Поисковые_структуры_данных [Дата звернення: 01.11.2021]

6. Парадокс розподілення днів народження. [Ел. ресурс]. Доступ: https://ru.m.wikipedia.org/wiki/Парадокс_дней_рождения[Дата звернення: 11.11.2021]

7. Бібліотека генерації мінімальної ідеальної хеш-функції. [Ел. ресурс]. Доступ: http://cmph.sourceforge.net [Дата звернення: 25.10.2021]

8. Бібліотека генерації ідеальної хеш-функції. [Ел. ресурс]. Доступ: https://www.gnu.org/software/gperf/ [Дата звернення: 24.10.2021]

9. Ліцензія вільного користування програмним забезпеченням. [Ел. ресурс]. Доступ: https://ru.wikipedia.org/wiki/GNU_Lesser_General_Public_License [Дата звернення: 06.11.2021]

10. F. C. Botelho, D. Belazzougui and M. Dietzfelbinger. Compress, hash and displace. In Proceedings of the 17th European Symposium on Algorithms (ESAвЂ™09). Springer LNCS, 2009.

11. F. C. Botelho, R. Pagh, N. Ziviani. Simple and space-efficient minimal perfect hash functions. In Proceedings of the 10th International Workshop on Algorithms and Data Structures (WADs'07), Springer-Verlag Lecture Notes in Computer Science, vol. 4619, Halifax, Canada, August 2007, 139-150.

12. B. Prabhakar and F. Bonomi. Perfect hashing for network applications. In Proceedings of the IEEE International Symposium on Information Theory. IEEE Press, 2006.

13. M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership. In Proceedings of the 35th international colloquium on Automata, Languages and Programming (ICALPвЂ™08), pages 385-396, Berlin, Heidelberg, 2008. Springer-Verlag.

14. Nebel, M. E. (1996). Digital search trees with keys of variable length. Informatique Theorique et Applications.

15. Robert S. Boyer, J Strother Moore. A Fast String Searching Algorithm. Communications of the ACM, USA, October 1977.

16. Аналіз засобу аналізу текстів для їх порівняння. [Ел. ресурс]. Доступ: http://www.shtampomer.narod.ru [Дата звернення: 10.10.2021]

17. Аналіз засобу аналізу текстів для встановлення авторства. . [Ел. ресурс]. Доступ: http://www.rusf.ru/books/analysis/ [Дата звернення: 09.10.2021]

18. Рисін А., Старко В. Великий електронний словник української мови (ВЕСУМ). 2005-2021. [Ел. ресурс]. Доступ: https://github.com/brown-uk/dict_uk

19. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю. М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. – 38 с.

20. База даних текстів із Вікіпедії. [Ел. ресурс]. Доступ: https://uk.wikipedia.org/wiki/Головна_сторінка [Дата звернення: 05.11.2021]

21. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ = Object-Oriented Analysis and Design with Applications / Пер. И.Романовский, Ф.Андреев. — 2-е изд. — М., СПб.: «Бином», «Невский диалект», 1998. — 560 с. — 6000 экз. — ISBN 5-7989-0067-3.

22. Chambers C. (1992) «The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages» // Stanford University, Ph.D. thesis.

23. Мартин С. Роберт. Быстрая разработка программ. Принципы, примеры, практика. — Вильямс, 2004. — ISBN 5845905583.

24. Meyer, Bertrand. Object-Oriented Software Construction. — Prentice Hall, 1988. — ISBN 0136290493.

25. Liskov, Barbara. Data abstraction and hierarchy (4 October 1987)

26. Мартин С. Роберт. Design Principles and Design Patterns. (2000)

27. Пошук евклідової відстані між точками. [Ел. ресурс]. Доступ: https://ru.wikipedia.org/wiki/Евклидова_метрика [Дата звернення: 08.11.2021]

28. Закон України «Про охорону праці» від 14 жовтня 1992 р. Редакція № 1667-IX від 15.07.2021

29. ДСанПіН 3.3.2.007-98 Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин затверджені постановою Головного державного санітарного лікаря України від 10.12.1998 № 7

30. НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями» затверджені наказом Міністерства соціальної політики України від 14 лютого 2018 №207

31. Примірна інструкція з охорони праці під час експлуатації електронно-обчислювальних машин, затверджена наказом Міністерства доходів і зборів України від 05.09.2013 № 443

32. ДСН 3.3.6.042-99 Державні санітарні норми мікроклімату виробничих приміщень затверджені постановою Головного державного санітарного лікаря України від 1 грудня 1999 року № 42

33. ДСН 239-96 Державні санітарні норми і правила захисту населення від впливу електромагнітних випромінювань затверджені Міністерством охорони здоров'я від 1 серпня 1996 року №239

34. ДБН В.2.5-28:2018 Державні будівельні норми України. Природне і штучне освітлення затверджені наказом Міністерства регіонального розвитку, будівництва та житлово-комунального господарства України від 3 жовтня 2018 року №264

35. Наказ Міністерства охорони здоров'я «Про затвердження Державних санітарних норм допустимих рівнів шуму в приміщеннях житлових та громадських будинків і на території житлової забудови» від 22 лютого 2019 року №463

36. ДСН 3.3.6.039-99 Державні санітарні норми виробничої загальної та локальної вібрації затверджені постановою Головного санітарного лікаря України від 1 грудня 1999 року №39

37. Порядок надання домедичної допомоги постраждалим при ураженні електричним струмом та блискавкою №398, затверджений Міністерством охорони здоров'я України від 16 червня 2014

38. Кодекс цивільного захисту України. Редакція від 17.06.2021

39. Кодекс законів про працю України від 10 грудня 1971 р. № 322-VIII. Редакція від 15.07.2021

40. Закон України від 23 вересня 1999 р. № 1105-XIV "Про загальнообов`язкове державне соціальне страхування". Редакція від 15.07.2021

41. Постанова Кабінету Міністрів України від 01 серпня 1992 р. № 442 "Про затвердження Порядку проведення атестації робочих місць за умовами праці". Редакція від 05.10.2016

42. Постанова Кабінету Міністрів України “Про затвердження Порядку розслідування та обліку нещасних випадків, професійних захворювань та аварій на виробництві” №337. Редакція від 05.01.2021

43. НПАОП 0.00-4.12-05 Типове положення про порядок проведення навчання і перевірки знань з питань охорони праці, затверджене наказом Держнаглядохоронпраці від 26.01.2005 №15. Редакція від 30.01.2017

44. НПАОП 0.00-7.14-17 Вимоги безпеки та захисту здоров’я під час використання виробничого обладнання працівниками, затверджене наказом Міністерства соціальної політики України від 28.12.2017 № 2072. Зареєстрованого в Мін’юсті України 23.01.2018 за №97/31549

45. ДСТУ 7299:2013 Дизайн і ергономіка. Робоче місце оператора. Взаємне розташування елементів робочого місця. Загальні вимоги ергономіки, затверджено та введено в дію наказом міністерства економічного розвитку і торгівлі України 14.10.2013 № 1231. Зареєстрованого в Мін’юсті України 14.02.2012 за №226/20539

46. «Перша медична долікарська допомога» [Ел. ресурс]. Доступ: https://www.bsmu.edu.ua/blog/6893-persha-medichna-dolikarska-dopomoga/. [Дата звернення: 11.11.2021].

47. «Інструкція з першої медичної допомоги» [Ел. ресурс]. Доступ: https://www.sop.com.ua/article/263-nstruktsya-z-nadannya-domedichno-dopomogi. [Дата звернення: 11.11.2021].

48. «Перша медична допомога при ураженні електричиним струмом» [Ел. ресурс]. Доступ: https://bozhedarivska-selrada.gov.ua/news/1576497483/. [Дата звернення: 11.11.2021]

49. Правила пожежної безпеки для навчальних закладів та установ системи освіти України від 15 серпня 2016 року.

50. V Міжнародна наукова мультидисциплінарна конференція студентів та молодих учених. ДНУЗТ, м. Дніпро, 25.11.2021.

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Перший проректор Дніпровського
національного університету
залізничного транспорту
імені академіка В. Лазаряна
Борис Боднар

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Технічне завдання
ЛИСТ ЗАТВЕРДЖЕННЯ
1116130.01202-01-ЛЗ

Завідувач кафедри КІТ

доц. Вадим Горячкін

Керівник розробки

проф. Віктор Шинкаренко

Виконавець

студент групи ПЗ2021
Олександр Кириченко

Нормоконтролер

доц. Олена Куроп'ятник

2021

ЗАТВЕРДЖЕНО
1116130.01202-01

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Технічне завдання

1116130.01202-01

Аркушів 22

2021

Анотація

Документ 1116130.01202-01 «Система дослідження ефективності структур даних та кластеризації творів за авторами на основі кращої структури. Технічне завдання» входить до складу програмної документації дипломного проекту.

У документі представлене призначення та область застосування програмного продукту, основні вимоги, стадії та терміни виконання проекту, техніко-економічні показники, що пред’являються до програмного продукту.

Зміст

Вступ

Система дослідження ефективності структур даних та кластеризації творів за авторами на основі кращої структури «AuthorshipClusterizer» являє собою програмний комплекс для формування компактного українського словнику з морфологічними атрибутами слів, на основі якого здійснюватиметься розбір текстів на морфологічні послідовності та формуватимуться кластери творів за ними. Система доступна як додаток і може функціонувати в операційній системі Windows.

Оскільки основна частина обчислювальних ресурсів зосереджена на етапі розбору тексту, то необхідно розробити швидкодіючу структуру для зберігання словника в оперативній пам’яті, а також організувати ефективне розбиття тексту на слова. Словник для зберігання всіх слів та їх атрибутів української мови повинен бути реструктуризований у зв’язку з його надмірністю та значним об’ємом. Формування кластерів текстів за їх схожістю відповідно до авторського стилю написання, що визначатиметься за послідовностями морфологічних атрибутів, повинно допомогти знайти плагіат на етапі попереднього аналізу текстових робіт.

Причиною замовлення продукту стала відсутність прямих аналогів на українському ринку, а також неможливість адаптації існуючих аналогів через відсутність відкритого доступу до коду.

Сферою застосування розробленого програмного продукту можуть бути всі видавничі та наукові галузі, де потрібна попередня перевірка на плагіат, дослідницькі відділи з аналізу текстових робіт, авторство яких не встановлено.

1 ПІДСТАВА ДО РОЗРОБКИ

Підставою для розробки є наказ ректора Дніпровського національного університету залізничного транспорту імені акад. В. Лазаряна Пшінька О.М. №690ст від 18.11.2020 р. «Про призначення наукових керівників та затвердження тем дипломних проектів» факультету «Комп'ютерні технології та системи» за спеціальністю 121 «Інженерія програмного забезпечення» по кафедрі «Комп'ютерні інформаційні технології».

Тема проекту «Дослідження структур даних електронного словника української мови для задач встановлення авторства текстів», керівник дипломного проекту проф. Шинкаренко В. І.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення

Програмний продукт призначений обробляти словник української мови для компактного зберігання, формувати на його основі швидкодіючу структуру даних в оперативній пам’яті та розбивати вхідні тексти на кластери за прогнозованими авторами, використовуючи евклідові відстані між векторами морфологічних атрибутивних послідовностей двох слів в ряд, що порівнюються в текстах.

2.2 Експлуатаційне призначення

Експлуатаційне призначення програмного засобу:

- інтеграція в системи обробки природно мовного тексту для підвищення їх ефективності;
- застосовування в наукових та видавничих сферах з метою початкового виявлення недобросовісного запозичення текстової інтелектуальної власності – плагіату.

3 ВИМОГИ ДО ПРОГРАМИ

3.1 Вимоги до функціональних характеристик

Слід виділити два незалежні модулі програмного комплексу: структуризації словнику та кластеризації текстів.

Програмний додаток повинен володіти наступним функціоналом:

- структурувати словник у форматі словника ВЕСУМ для його компактного зберігання на жорсткому диску, компенсуючи час формування з нього структури даних в оперативній пам’яті шляхом індексації компонентів;
- формувати швидкодіючу пошукову структуру на основі структурованого словнику, що реалізує операції вставки та пошуку;
- кластеризувати тексти на основі знайдених відстаней між їх векторами атрибутивних послідовностей.

Вхідні дані:

- словник української мови, що відповідає формату словника ВЕСУМ [18] (опціонально);
- множина текстових файлів у форматі .txt, що підлягають аналізу та подальшій кластеризації за авторами.

Вихідні дані:

- для модулю першої програми – структурований словник української мови з мінімально необхідним набором даних для побудови відповідної структури;
- для другого модулю – текстовий файл з результатами кластеризації вхідних текстів за авторами.

3.2 Вимоги до надійності

Список вимог до надійності програмного засобу в цьому розділі включає лише пункти, що стосуються програмного забезпечення, збої засобів обчислювальної техніки до нього не входять.

Вимоги до надійності програмного продукту наступні:

- наявність архівної копії тексту програми та файлів словнику на зовнішньому носії інформації;
- не більше ніж одна помилка на тисячу операторів;
- контроль коректності вхідної інформації, яка вводиться користувачем під час роботи з програмним продуктом.

3.3 Умови експлуатації

Даний програмний продукт може використовуватись в умовах, які відповідають вимогам [13].

Для коректного функціонування програмного продукту необхідно дотримання наступних вимог:

- програмний комплекс повинен використовуватись в приміщеннях, призначених для роботи ЕОМ у наступних кліматичних умовах: температура – 21-25 °С, відносна вологість повітря 40-60%;
- кваліфікація персоналу повинна бути на рівні користувача ЕОМ, що має досвід роботи з операційною системою Windows та пройшов підготовку за керівництвом користувача.

3.4 Вимоги до складу та параметрів технічних засобів

Мінімальна конфігурація комп’ютеру для забезпечення роботи програмного продукту:

- комп’ютер з тактовою частотою процесора 2 ГГц і розрядністю 64-біти;
- ОЗП об’ємом не менше 6 ГБ;
- вільний дисковий простір 1 ГБ;
- наявність CD/DVD приводу, USB роз’єму для встановлення ПЗ;

– монітор з роздільною здатністю екрану 1024x768 та більше;
– стандартні клавіатура та маніпулятор «миша».

3.5 Вимоги до інформаційної та програмної сумісності

Для функціонування програмного продукту необхідна Windows 7 або більш пізня версія.

3.6 Вимоги до маркування та упаковки

Програма може зберігатися на постійних запам’ятовуючих носіях (CD/DVD-диски, флеш-пристрої). Упаковка продукту повинна забезпечувати захист від механічних пошкоджень та мати маркування, зображене на рис. 3.1:

“AuthorshipClusterizer”, 1.0
Кириченко Олександр Олександрович
кафедра КІТ, ДНУЗТ 2021

Рисунок 3.1 – Штамп продукту

3.7 Вимоги до транспортування та зберігання

Умови транспортування та зберігання повинні забезпечувати захист носія від фізичних пошкоджень і відповідати вимогам [2].

Термін зберігання програмного продукту необмежений та залежить лише від умов зберігання та встановленого замовником періоду експлуатації.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

Програмна документація повинна включати наступні документи: технічне завдання та робочий проект у складі:

- специфікація;
- текст програми;
- опис програми;
- керівництво користувача. Керівництво структуризації словнику та кластеризації текстів за морфологічними атрибутами слів;

Вся документація до програми повинна задовольняти вимогам державного стандарту до оформлення програмних документів (ГОСТ 19.101-77).

5 ВИЗНАЧЕННЯ ВИТРАТ НА ПРОЕКТУВАННЯ ПРОГРАМИ

В цій дипломній роботі був розроблений комплекс програмних засобів для встановлення авторства українського тексту за морфологічними атрибутами слів з використанням унікальної структури даних для зберігання та пошуку слів «AuthorshipClusterizer».

Метою даного розділу є розробка техніко-економічного обґрунтування для розроблення нового програмного забезпечення заради оцінки запланованих витрат, а також дати висновок щодо доцільності його розробки та розгортання.

Першим кроком є розрахунок обсягу затрачених трудових ресурсів розробників завдяки оцінці розміру програмного коду. Основні методики визначення запланованих трудовитрат орієнтуються на кількісну або якісну оцінку.

Constructive Cost Model (COCOMO) [1] орієнтується на вимірюваний розмір проекту S в рядках коду LOC (KLOC), а обсяг трудових ресурсів у людино-місяцях, розрахунок наведено у формулі 5.1.

$$E = a \cdot S^b \cdot EAF \tag{5.1}$$

де E – необхідні людські ресурси на розробку проекту (в людино-місяцях);

S^b – обсяг коду (в KLOC);

EAF – фактор уточнення витрат (effort adjustment factor).

Для простих систем a = 2,4; b = 1,05.

Для визначення загальної кількості рядків коду можна скористатися засобами Microsoft Visual Studio, що є безкоштовною платформою для розробки програмного забезпечення, в якому проходила розробка всього комплексу програм, та підрахувати довжину всіх програмних модулів. Таким чином визначено, що загальна довжина коду програмного забезпечення становить 1954 рядків. Трудовитрати у людино-місяцях розраховано у формулі 5.2.

$$E = 2{,}4 \cdot 1{,}954^{1{,}05} \cdot 1 = 4{,}85 \tag{5.2}$$

Далі наведені розрахунки для обчислення вартості розробки програмного комплексу «AuthorshipClusterizator». За передбачувані напрямки витрат прийняті:

– грошовий оклад та премії працівників;

– відрахування на соціальні потреби;

– накладні витрати;

– витрати на персональний комп’ютер та ліцензійні програмні засоби, що застосовуються при розробці.

Оцінка основної заробітної плати (ОЗП) інженера-програміста, що займається розробкою програмного продукту, базується на кількості розробників, заробітної плати в перерахунок на години та тривалості розробки (у годинах). Розрахунок заробітної плати відбувається за даними середньої заробітної плати програміста в м. Дніпро [2] і представлений у табл. 5.1.

Таблиця 5.1 – Фонд місячної заробітної плати

№п/п	Посада виконавця	Оклад, грн/міс	Кількість		Сума зарплати грн.
			чол.	місяців	
1	інженер-програміст	17500	1	4,85	84875

Розробка представленого у цій роботі програмного продукту буде проводитися одним програмістом в період з 01.02.2021 до 25.06.2021, що становить 21 робочий тиждень або 105 днів. Робочий час на тиждень приймається за 40 годин. Погодинна оплата кваліфікованого інженера-програміста становить 104,17 грн/год. Таким чином, витрачений робочий час за форм. 5.3 розраховано у форм. 5.4:

$$t_{\text{розробки}} = N_{\text{чол}} \cdot N_{\text{тиж}} \cdot N_{\text{год}}, \quad (5.3)$$

де $N_{\text{чол}}$ – кількість виконавців, *чол*;

$N_{\text{тиж}}$ – тривалість розробки;

$N_{\text{год}}$ – витрати робочого часу, год;

$$t_{\text{розробки}} = 1 \cdot 21 \cdot 40 = 840 \text{ чол/год.} \quad (5.4)$$

ОЗП визначається за формулою 5.5:

$$\text{ОЗП} = t_{\text{розробки}} \cdot N \cdot K_{KB}, \quad (5.5)$$

де $t_{\text{розробки}}$ – витрати праці у чол/год;

N – зарплата за годину;

K_{KB} – коефіцієнт кваліфікації програміста, прийнято 0,75.

ОЗП становитиме за формулою 5.6:

$$\text{ОЗП} = 840 \cdot 104{,}17 \cdot 0{,}75 = 65627 \text{ грн.} \quad (5.6)$$

Відрахування на оплату соціальних внесків вираховуються у відсотковому відношенні від суми заробітної плати [3]:

$$C_{\text{соц}} = \frac{\text{ОЗП} \cdot 22\%}{100\%}$$

$$C_{\text{соц}} = \frac{65627 \cdot 22\%}{100\%} = 14438 \text{ грн.} \quad (5.7)$$

Отримані результати за пунктами (5.6) та (5.7) підсумовуються. В результаті вони складають суму 80065 грн., яку визначено основними прямими витратами.

Накладні витрати [4] включають загальногосподарські витрати на організацію робочого процесу: витрати на комунальні послуги в офісі, амортизація будівель, зарплату штатного обслуговуючого персоналу та інше. Вони визначаються в процентах (30 – 40%) від суми прямих витрат і розраховані за форм. 5.8 у 5.9:

$$\text{С}_{\text{накл}} = \frac{(\text{ОЗП} + \text{С}_{\text{соц}}) \cdot 35\%}{100\%}; \quad (5.8)$$

$$\text{С}_{\text{накл}} = \frac{(65627 + 14438) \cdot 35\%}{100\%} = 28\,022{,}75 \text{ грн.} \quad (5.9)$$

Впродовж усього терміну використання технічного устаткування підприємство щорічно витрачає певні кошти, пов’язані з її експлуатацією.

Витрати на експлуатацію [4] персонального комп’ютер визначаються на етапі планування розробки програмного засобу в залежності від вартості конкретної робочої станції. До експлуатаційних витрат відносяться:

– витрати на ремонт;

– оренда приміщення;

– вартість витратних матеріалів;

– заробітна плата ремонтника;

– додаткові витрати – обслуговування приміщення (прибирання, охорона, оренда, комунальні послуги);

– амортизаційні витрати на персональний комп'ютер та базовий пакет програмного забезпечення;

– витрати на електроенергію ($С_{ел}$).

Визначаються ці витрати за формулою 5.10:

$$С_{ел} = P \cdot B \cdot T_{розр}, \quad (5.10)$$

де Р – потужність комп’ютера та допоміжних споживачів електричної енергії, розрахована потужність для комп’ютера та монітору прийнята 0,5 кВт/год;

B – вартість 1 кВт/година, в Україні за поточним тарифом складає 1,68 грн;

$T_{розр}$ – час роботи з ЕВМ, прирівнюється робочому часу.

Витрати на електроенергію розраховані у форм. 5.11:

$$C_{ел} = 0{,}5 \cdot 1{,}68 \cdot 840 = 705{,}6 \text{ грн.} \quad (5.11)$$

Для визначення заробітної плати ремонтника ($С_{рем}$) необхідно перерахувати його середньомісячну заробітну плату в години, що витрачаються за нормою на один персональний комп’ютер. Кількість комп’ютерів, що обслуговуються одним

ремонтником приймається за 50, а його зарплата за місяць – 10000 грн. [7]. Тоді на один комп'ютер буде витрачено коштів за форм. 5.12:

$$C_{\text{рем}} = \frac{C_{\text{рем}}'}{N_{КОМ}} \cdot T_{\text{міс}}, \tag{5.12}$$

де $C_{\text{рем}}'$ – середньомісячна заробітна плата;

$N_{КОМ}$ – кількість комп'ютерів на одного ремонтника.

$T_{міс}$ – час розробки програмного продукту, міс.

Заробітна плата ремонтника ($C_{\text{рем}}$) розрахована у форм. 5.13:

$$C_{\text{рем}} = \frac{10000}{50} \cdot 4{,}85 = 970 \text{ грн.} \tag{5.13}$$

Крім цього необхідно врахувати витрати на комплектуючі вироби ($\text{С}_{КОМ}$) [5], на які будуть замінені несправні при ремонті. Вони приймаються за 10% від вартості комп'ютера впродовж терміну його експлуатації. Тепер перерахуємо ці витрати з урахуванням часу створення програмного засобу у форм. 5.14:

$$\text{С}_{КОМ} = \text{В}_{\text{ком}} \cdot \frac{N_{\text{Д}}}{N_{\text{експ}} \cdot 365} \cdot \frac{10\%}{100\%}, \tag{5.14}$$

де $B_{КОМ}$ – вартість персонального комп'ютеру;

$N_{\text{Д}}$ – тривалість розробки програмного продукту у днях;

$N_{\text{експ}}$ – термін експлуатації персонального комп'ютеру.

Прийнята вартість ПК з клавіатурою, мишею і монітором 20 000 грн [11]. Сума витрат на комплектуючі частини протягом періоду розробки програми визначається у форм. 5.15:

$$C_{\text{КОМ}} = 20000 \cdot \frac{105}{5 \cdot 365} \cdot \frac{10}{100} = 115{,}07 \text{ грн.} \tag{5.15}$$

Сума на витратні матеріали ($С_{вм}$) [5] протягом всього терміну експлуатації також становить приблизно 10% від вартості комп'ютеру. Робоча станція оцінюється у 20 000 грн., термін її експлуатації – 5 років. Витрати такі ж, що й на комплектуючі вироби:

$$C_{\text{ВМ}} = C_{КОМ} = 115{,}07 \text{ грн.} \tag{5.16}$$

Амортизаційні відрахування на персональний комп'ютер (АПК) [6] визначені з положення, що амортизаційний період в даний час дорівнює терміну зміни поколінь обчислювальних машин і приймається за 3 роки. Отже, кожні 3 роки амортизаційні витрати дорівнюють вартості такого комп'ютера. Вирахуємо частку витрат на амортизаційні відрахування в період розробки програми:

$$\text{АПК} = \text{В}_{КОМ} \cdot \frac{N_{\text{Д}}}{N_{\text{експ}} \cdot 365} = 20000 \cdot \frac{105}{3 \cdot 365} = 1917{,}8. \tag{5.17}$$

Для визначення суми амортизаційних відрахувань на програмне забезпечення (АПЗ) [6] необхідно враховувати час розробки проекту в межах часу дії пакету прикладних програм та операційної системи, що були застосовані. Розробка проводилася в операційній системі Windows, оновлення версій якої прийнято здійснювати кожні 5 років, а за середовище розробки для поставлених задач обрано Visual Studio 2019, що поширюється за моделлю щомісячної підписки, тому амортизаційні відрахування на програмне забезпечення дорівнюють його вартості в перерахунок на час розробки.

Операційна система для розробки взята Windows 10, середовище розробки програм – Microsoft Visual Studio 2019 Professional, що оплачується щомісячно за підпискою у 1200 грн/міс.

(5.18)

$$АПЗ_W = 6204 \cdot \frac{105}{5 \cdot 365} = 357 \text{ грн.}$$

$$АПЗ_V = 1200 \cdot 5 = 6000 \text{ грн.} \quad (5.19)$$

Розрахунок амортизаційних відрахувань на програмне забезпечення зведений у табл. 5.2.

Таблиця 5.2 – Застосоване програмне забезпечення

Найменування програмного забезпечення	Вартість програмного забезпечення, *грн*	Джерело придбання	Амортизаційні відрахування, *грн*
Windows 10 Professional	6204	https://soft.rozetka.com.ua/microsoft_fqc_09131/p3936301/	460
Visual Studio 2019 Professional	6000	https://visualstudio.microsoft.com/ru/vs/pricing/?tab=business	6000
Всього:	12204	-	6460

Додаткові витрати ($С_{дод}$) [6]: охорона, комунальні послуги, прибирання приміщень, тощо, досить складно оцінити, тому приймаються рівними 20% від заробітної плати інженера-програміста, а саме 3500 гривень на місяць.

Оренду офісу ($С_{ор}$) з площею 13 м^2 візьмемо 3000 гривень на місяць [8].

Загалом експлуатаційні витрати на один персональний комп'ютер будуть складати за форм. 5.20:

$$C_{експ} = C_{ел} + C_{рем} + C_{КОМ} + C_{ВМ} + АПК + АПЗ + C_{дод} + C_{ор} =$$

$$= 705{,}6 + 970 + 2 * 115{,}07 + 1917{,}8 + 6460 + 17500 + 15000 = 42783{,}54 \text{ грн} \quad (5.20)$$

Результати розрахунків приведено у табл. 5.3.

Таблиця 5.3 – Експлуатаційні витрати на ПК і ПЗ.

Найменування витрат	Витрати, грн
Витрати на електроенергію	705,6
Вартість витратних матеріалів	115,07
Витрати на ремонт	800
Амортизація персонального комп’ютера	1917,8
Амортизація програмного забезпечення	6460
Оренда приміщення	6000
Додаткові витрати	8750
Всього	42613,54

Підсумкові витрати на створення програмного продукту [6] вираховуються за форм. 5.21 у 5.22:

$$C_{\text{розробки}} = \text{ОЗП} + C_{\text{соц}} + C_{\text{накл}} + C_{\text{експ}}; \quad (5.21)$$

$$C_{\text{розробки}} = 65627 + 14438 + 28022{,}75 + 42783{,}54 = 150871{,}29 \quad (5.22)$$

Розрахунок витрат зведено у табл. 5.4.

Таблиця 5.4 – Кошторис витрат на розробку програмного засобу

Найменування витрат	Витрати, грн
Основна заробітна плата	65 627
Відрахування на соціальні потреби	14 438
Накладні витрати	28 022,75
Експлуатаційні витрати	42 783,54
Всього	150871,29

За розрахованими значеннями техніко-економічних показників проекту складено кошторис витрат на розробку актуального програмного забезпечення для дослідження авторства текстів «AuthorshipClusterizer», встановлена вартість розробки програмного засобу складає 150 871,29 грн.

6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії та етапи розробки програмного продукту представлені у табл. 6.1.

Таблиця 6.1 – Стадії та етапи розробки

Стадії розробки	Етапи розробки	Терміни виконання
1. Технічне завдання (ТЗ)	Функціональне та експлуатаційне призначення	23.09.21 – 24.09.21
	Вхідні та вихідні дані	24.09.21 – 27.09.21
	Вимоги до програмної документації	27.09.21 – 29.09.21
	Техніко-економічні показники	29.09.21 – 01.10.21
	Узгодження та затвердження ТЗ	01.10.21 – 04.10.21
2. Робочий проект	Розробка та програмування функціоналу програми	05.10.21 – 19.11.21
	Розробка і реалізація інтерфейсу користувача	19.11.21 – 26.11.21
	Налагодження програми	26.11.21 – 01.12.21
	Розробка, узгодження та затвердження програмної документації відповідно до вимог ГОСТ 19.101-77	01.12.21 – 03.12.21
	Проведення випробувань і корегування програми та документації за результатами випробувань	03.12.21 – 07.12.21
3. Впровадження	Підготовка і передача програми та програмної документації замовнику	08.12.21 – 10.12.21

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

Контроль здійснюється за допомогою виконання набору тестів з метою знаходження помилок в програмному продукті та його специфікації та їх виправленню. Контроль виконання роботи ґарантується головним керівником розробки проф. Шинкаренко В.І.

Прийом програмного продукту здійснюється уповноваженою комісією.

Бібліографічний список

1. Модель COCOMO. [Електронний ресурс]. Режим доступу: https://ru.wikipedia.org/wiki/COCOMO. Дата звернення: 25.11.21.

2. Дані щодо середньої зарплати програміста в Україні. [Електронний ресурс]. Режим доступу: https://www.work.ua/ru/salary-dnipro-web-программист/ Дата звернення: 25.11.21.

3. Закон України «Про збір та облік єдиного внеску на загальнообов'язкове державне соціальне страхування» із змінами від 1 липня 2021 року N 1617-IX

4. Липаев В.В. Экономика производства сложных программных продуктов. – М Синтег, 2008. 432 с

5. Благодатских В.А. и др. Экономика, разработка и использование программного обеспечения ЭВМ. – М.: Финансы и статистика, 1995. - 286 с.

6. Фатрелл Р.Т., Шафер Д.Ф., Шафер Л.И.. Управление программными проектами. Достижение оптимального качества при минимуме затрат. М.: Издательский дом “Вильямс”, 2004. – 1125 с.

7. Дані щодо середньої зарплати програміста в Україні. [Електронний ресурс]. Режим доступу: https://www.work.ua/ru/salary-компьютерный+мастер/ Дата звернення: 25.11.21.

8. Дані щодо оренди офісу в м. Дніпро. [Електронний ресурс]. Режим доступу: https://www.olx.ua/d/obyavlenie/sdam-ofis-13i20m-mozhno-obedinit-arenda-kabineta-pomeschenie-salon-tsent-IDNpGGQ.html#5d8a44c1ac;promoted Дата звернення: 25.11.21.

9. Ціна операційної системи Windows 10. [Електронний ресурс]. Режим доступу: https://soft.rozetka.com.ua/microsoft_fqc_09131/p3936301/ Дата звернення: 25.11.21.

10. Ціна на Visual Studio Professional. [Електронний ресурс]. Режим доступу: https://visualstudio.microsoft.com/ru/vs/pricing/?tab=business Дата звернення: 25.11.21.

11. ПК моноблок. [Електронний ресурс]. Режим доступу: https://ek.ua/ASUS-F5401WUAK-BA008M.htm Дата звернення: 26.11.21.

12. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю. М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. – 38 с.

13. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин: ДСанПІН 3.3.2.007-98

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Перший проректор Дніпровського
національного університету
залізничного транспорту
імені академіка В. Лазаряна
Борис Боднар

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Робочий проект
ЛИСТ ЗАТВЕРДЖЕННЯ
1116130.01202-01-ЛЗ

Завідувач кафедри КІТ

доц. Вадим Горячкін

_____

Керівник розробки

проф. Віктор Шинкаренко

_____

Виконавець

студент групи ПЗ2021
Олександр Кириченко

_____

Нормоконтролер

доц. Олена Куроп'ятник

_____

2021

ЗАТВЕРДЖЕНО
1116130.01202-01

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Специфікація

1116130.01202-01

Аркушів 2

2021

Позначення	Найменування	Примітка
Документація		
1116130.01202-01-ЛЗ	Лист затвердження	
1116130.01202-01 13 01-ЛЗ	Опис програми	
1116130.01202-01 І3 01	Керівництво структуризації словнику та кластеризації текстів за морфологічними атрибутами слів	
1116130.01202-01 12 01	Текст програми	

ЗАТВЕРДЖЕНО
1116130.01202-01-ЛЗ

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Опис програми

1116130.01202-01 13 01

Аркушів 20

2021

Анотація

Документ 1116130.01202-01 13 01 «Система дослідження ефективності структур даних та кластеризації творів за авторами на основі кращої структури. Опис програми» входить до складу програмної документації дипломного проекту.

У документі представлені типові варіанти використання програми, порядок її завантаження, встановлення та необхідні технічні засоби для роботи з нею. Описаний користувацький інтерфейс та перелік вхідних і вихідних даних для роботи програми, а також повідомлення для користувачів.

Зміст

1 ЗАГАЛЬНІ ВІДОМОСТІ

Програмний комплекс «AuthorshipClusterizer» здійснює структуризацію українського словника ВЕСУМ для зменшення розміру на жорсткому диску та зберігає його у вигляді окремих текстових файлів для подальшого відтворення у швидкодіючу структуру даних. На основі утвореної структури множина вхідних текстів розбиватиметься на слова, ігноруючи знаки пунктуації та текстової розмітки, що потім відображатимуться у морфологічні атрибути. На основі таких послідовностей атрибутів будуть обраховані відстані між текстами та здійснюватиметься кластеризація за авторами. Програма надає користувачу інструмент для дослідження авторської приналежності до даного тексту на основі подібності побудови словосполучень автором.

Програма функціонує в середовищі операційної системи Windows починаючи з версії 7 та вище. Програма розроблена на мові C# у середовищі Visual Studio 2019.

2 ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Програмний продукт призначений визначати подібність текстів за їх часткою співпадіння величини використаних морфологічних атрибутивних послідовностей, що може застосовуватися при попередній перевірці текстів на плагіат. Його задача найбільш ефективно використовувати ресурси центрального процесору при здійсненні пошуку слів із текстів за словником. Програма розробляється з метою протидії протиправному використанню чужої інтелектуальної власності.

Функціональні обмеження накладаються на обсяги одночасної кластеризації текстів, що за своїм об’ємом не можуть перевищувати 3 ГБ. Можливо опрацьовувати лише тексти, які мають переважну більшість україномовних конструкцій (більше 50%).

Сферою застосування розробленого програмного продукту можуть бути всі видавничі та наукові галузі, де потрібна попередня перевірка на плагіат, дослідницькі відділи з аналізу текстових робіт, авторство яких не встановлено.

3 ОПИС ЛОГІЧНОЇ СТРУКТУРИ

3.1 Алгоритм програми

Для демонстрації можливих варіантів використання програми була розроблена діаграма прецедентів, що вважається першим етапом при проектуванні системи, бо наглядно показує можливі шляхи управління в програмі зі сторони користувача.

Діаграма відображає різноманітні сценарії, що можуть проходити між акторами (рядовими користувачами) та прецедентами (варіантами використання), описує функціональні можливості продукту. Перевагою даної діаграми є її наочність навіть для осіб, що безпосередньо задіяні у сфері розробки чи предметній області, на яку націлений продукт.

Для детальнішого опису системи у мові UML є наявні наступні види відносин:

- асоціація – специфікує семантичні особливості взаємодії акторів і варіантів використання в графічній моделі системи;
- узагальнення – служить для вказівки того факту, що деякий варіант використання А може бути узагальнений до варіанту використання В;
- включення – вказує, що деяка задана поведінка для одного варіанта використання включається як складовий компонента в послідовність поведінки іншого варіанту використання.
- розширення – відзначає той факт, що один з варіантів використання може приєднувати до своєї поведінки деяку додаткову поведінку, певну для іншого варіанту.

В процесі моделювання на стадії створення проектного плану було створено діаграму прецедентів (рис. 3.1). Вона містить одного актора-користувача і надає наступні варіанти використання:

- робота з модулем словника, що дозволяє завантажити довільний український словник, який задовольняє специфікації програми;

– завантаження словнику з довільної директорії операційної системи;

– збереження оброблених модулів словника, що необхідні для роботи 2-го модулю;

– обрати директорію, що стане адресою за замовченням для збереження модулів словника;

– робота з модулем аналізатору текстів за словником (2-й модуль, окрема програма, що працює автономно за наявності модулів словника);

– попередній аналіз авторського стилю на основі довільної вибірки його творів (обирається користувачем з урахуванням жанру текстів, що йому необхідно проаналізувати);

– кластеризація текстів за авторами, що наявні у базі даних програми;

– збереження результатів кластеризації за авторами.

Рисунок 3.1 – Діаграма прецедентів

Діаграма активності являється ще одним з інструментів UML для відображення процесів у системі через взаємодію її функціональних модулів. Для створюваної програми вона допоможе продемонструвати взаємодію окремих взятих об’єктів

діяльності та подій, що поєднуються одне з одним потоками між вузловими точками (входами і виходами з елементів). Таким чином можна просто описати можливі шляхи використання програми користувачем в єдиному місці.

Користувачу доступно для запуску 2 модулі: для формування словнику та для кластеризації текстів. Робота другого модуля потребує дані словника, що отримуються в результаті роботи першого модуля. Однак надалі без потреби у розширенні словникового запасу чи зміни слів у зв’язку зі зміною стандартів мови виконання першого модулю може бути пропущене, а всі дані результуючого словника слів після першого його формування можуть використовуватися повторно без обмежень. На рис. 3.2 представлені відповідні діаграми активності для модулів системи.

Отримані на виході даної програми текстові модулі необхідно подати на вхід другому модулю, при першому запуску достатньо вказати директорію.

Діаграма станів дозволяє оглянути систему з точки зору робочих станів, множина яких досягається в результаті роботи машини станів. Таким чином для розроблюваної програми можна окреслити ключові етапи роботи під час виконання двох модулів програм. На рис. 3.3 зображені діаграми станів інтерфейсу для структуризації словнику та кластеризації текстів.

Діаграма послідовності – вид діаграми UML, що демонструє взаємодії між об’єктами у часі. На ній об’єкти можуть обмінюватися повідомленнями для послідовної взаємодії між собою. Всі процеси впорядковуються у вигляді вертикальних ліній. Надіслані повідомлення зображуються горизонтальними лініями та з’єднують 2 об’єкти, що взаємодіють. На рис. 3.4 і рис. 3.5 відповідно представлені діаграми послідовностей для типового ходу виконання програм структуризації словнику та кластеризації текстів за авторами цільовим користувачем.

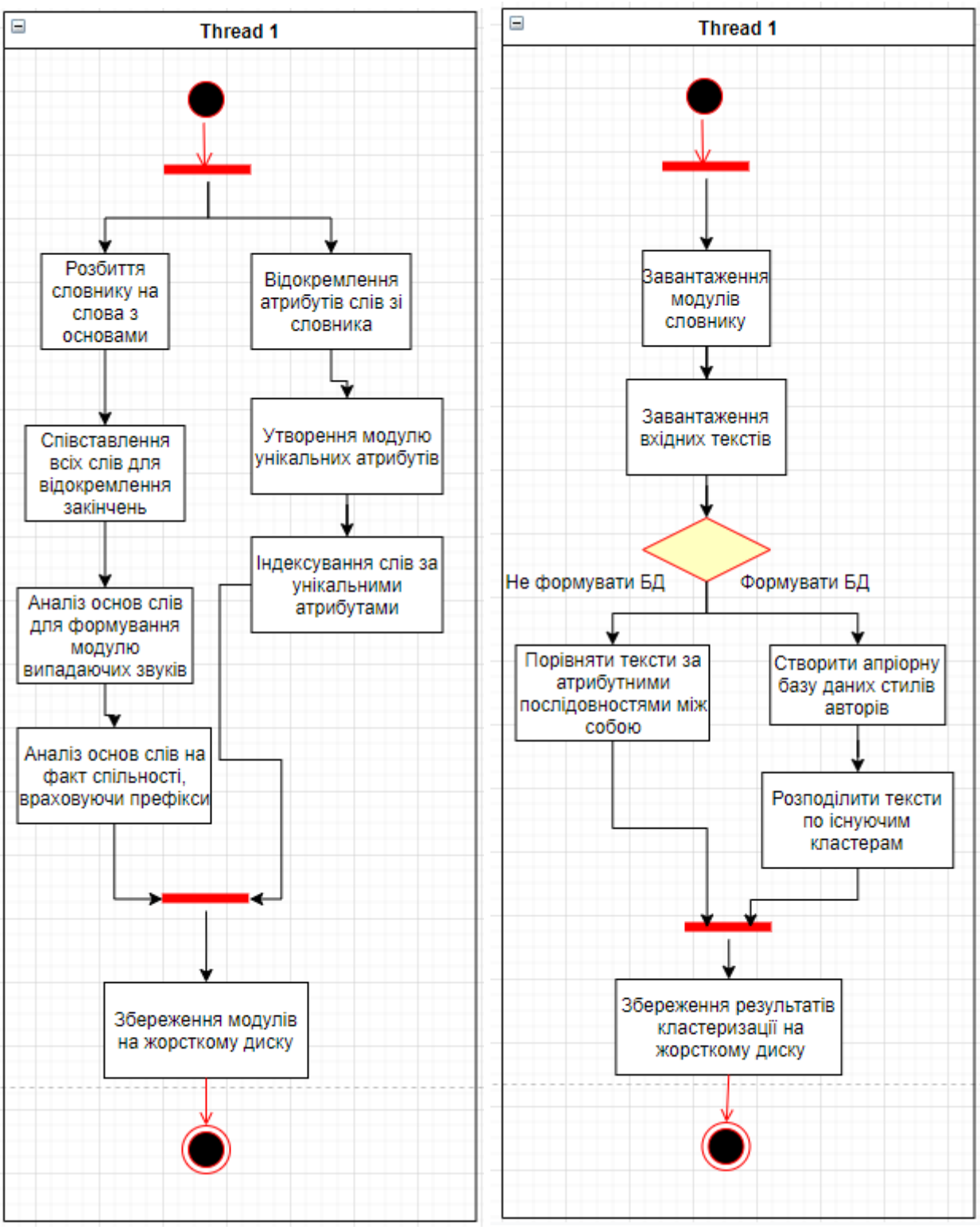


Рисунок 3.2 – Діаграма активності для модулів структуризації словника (ліва) та кластеризації текстів (права)

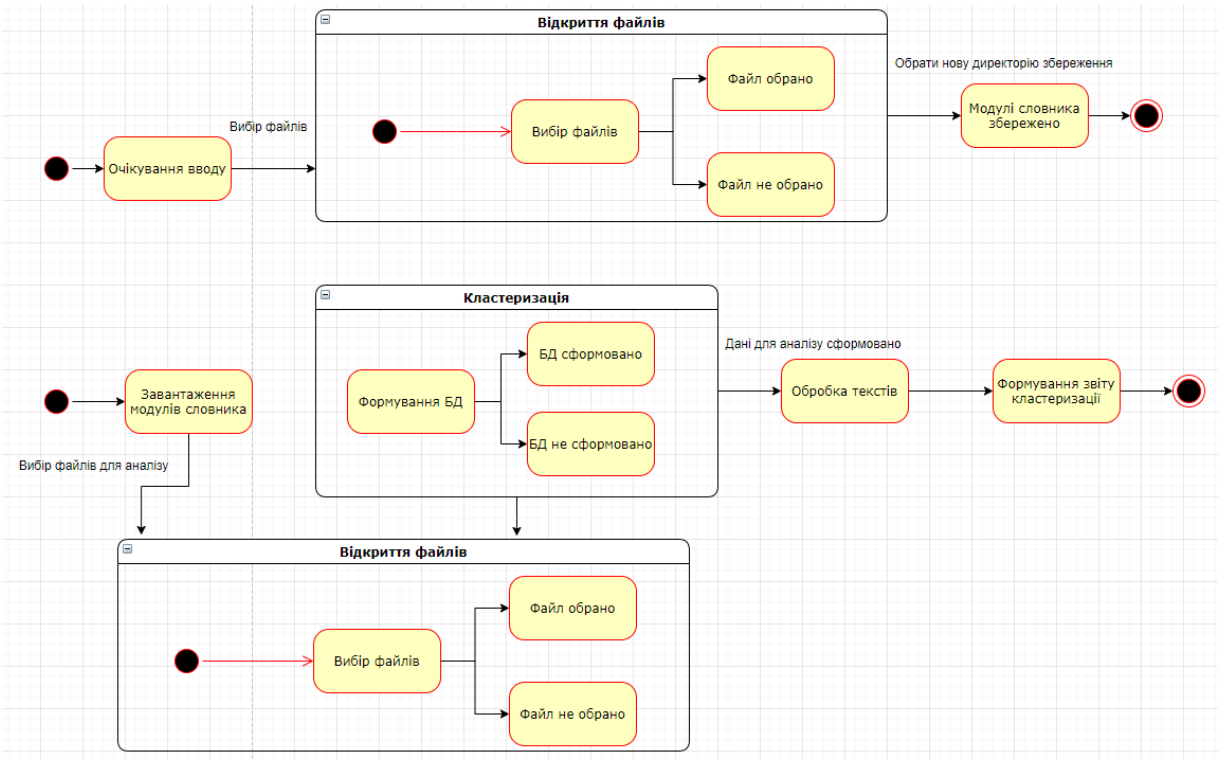


Рисунок 3.3 – Діаграма станів інтерфейсу користувача для 2-х модулів

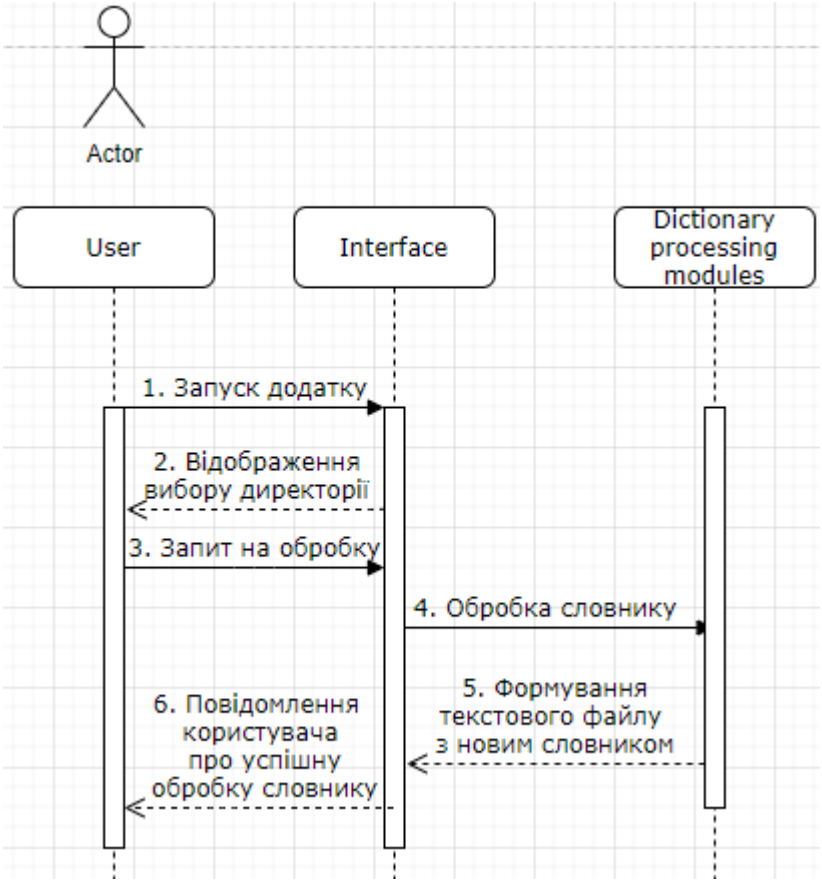


Рисунок 3.4 – Діаграма послідовності для структуризації словника

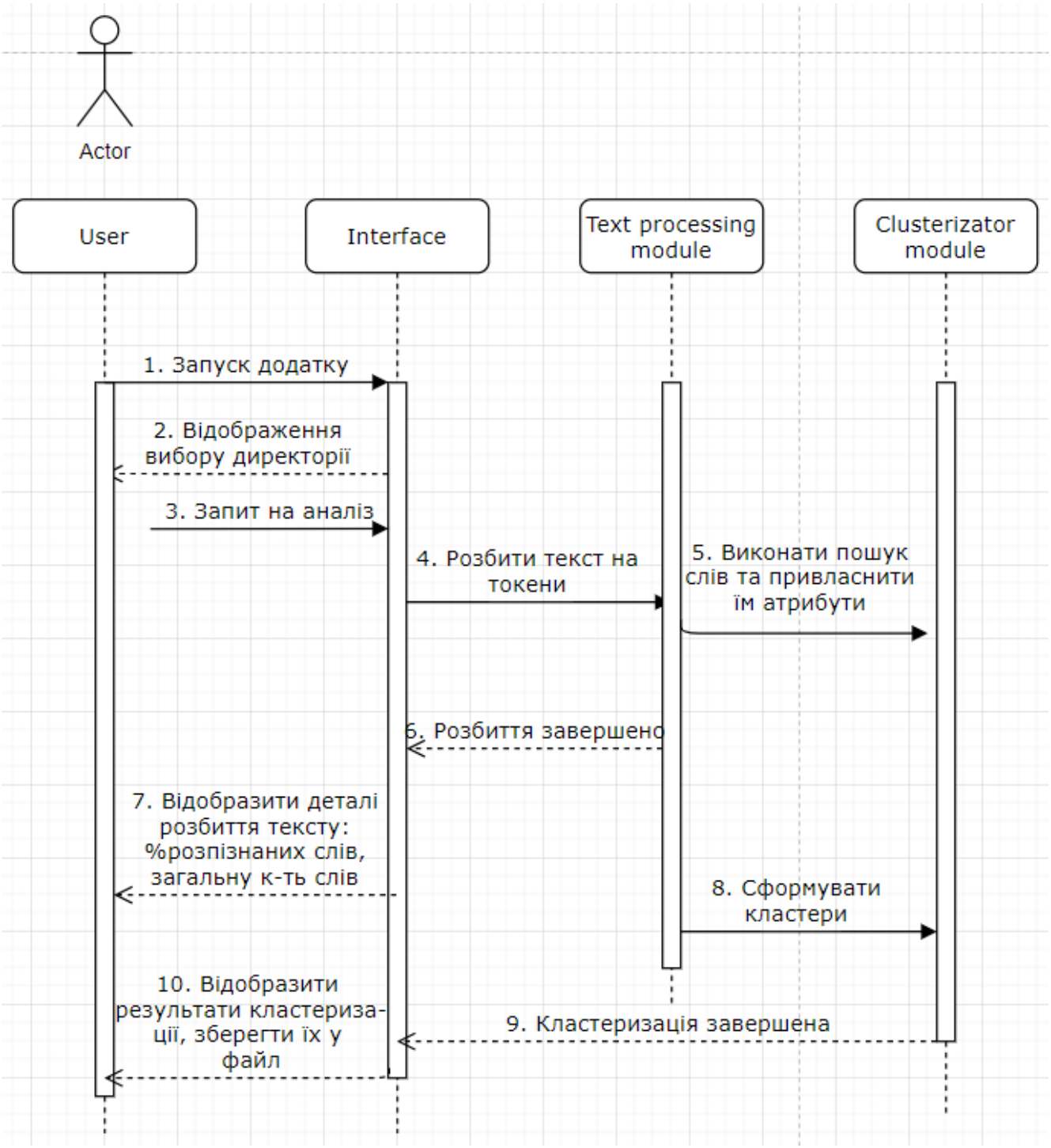


Рисунок 3.5 – Діаграма послідовності для кластеризації текстів

Діаграма компонентів дозволяє описати взаємодію між елементами програми на рівні їх фізичного представлення в операційній системі. Вона надає можливість описати архітектуру системи, що підлягає розробці, на основі елементарних компонентів системи, що в свою чергу можуть бути будь-яким видом виконуваного

коду: скрипти, виконавчі файли, сервіси та ін. Складовими частинами на діаграмі можуть виступами компоненти, інтерфейси та зв’язки між ними.

Для даної системи діаграма розроблюється з наступними цілями:

– графічне відображення структури початкового коду розроблюваної системи;

– специфікації ходу роботи програмної системи;

– можливість відображення багатократного використання окремих частин програмного коду;

– відображення взаємодії системи з іншими програмними засобами в ході своєї роботи.

Розроблена діаграма компонентів для «AuthorshipClusterizer» представлена на рис. 3.6. Стрілками показана послідовність виконання.

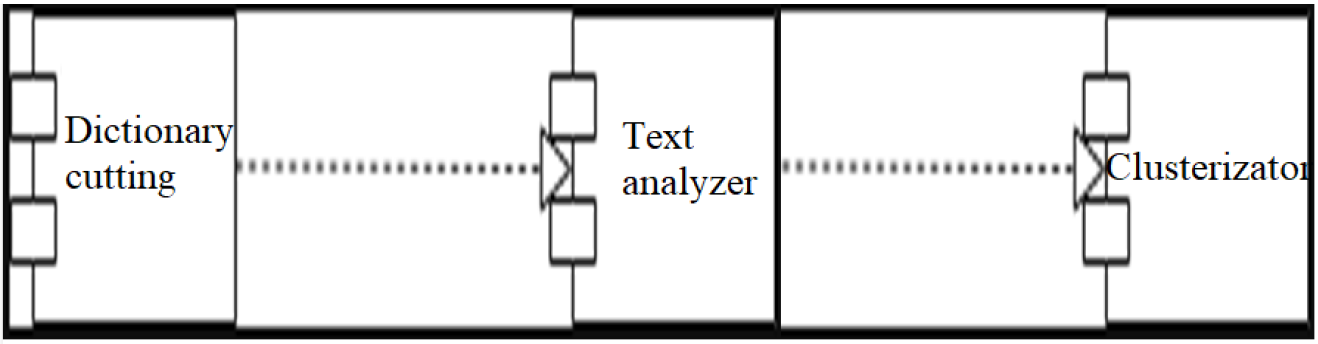


Рисунок 3.6 – Діаграма компонентів

3.2 Використані методи

Програмна система використовує методи часового оцінювання операцій вставки та пошуку Stopwatch(), що отримує часові характеристики з точністю до наносекунд, моніторингу зайнятого в оперативній пам’яті місця через діагностичні засоби Visual Studio.

Призначення модулів програми:

– Dictionary cutting – відповідає за стиснення та структуризацію вхідного

словнику української мови з формуванням текстових файлів для утворення швидкодіючої структури даних з них;

– Clusterizator – відповідає за розбір вхідних текстів на слова та їх кластеризацію за відповідними атрибутивними властивостями послідовностей слів.

Зв’язок між модулями програм здійснюється через відповідні текстові файли, що описані в розділі вхідні вихідні дані.

4 ВИКОРИСТАНІ ТЕХНІЧНІ ЗАСОБИ

Технічні засоби, які використовуються при роботі програми:

- ПК, під управлінням 64-розрядної ОС Windows;
- процесор з тактовою частотою не нижче 2 ГГц;
- оперативна пам’ять не менше 6 ГБ;
- простір на жорсткому диску не менше 1 ГБ;
- маніпулятор миша;
- клавіатура;
- наявність CD/DVD приводу, USB роз’єму або LAN-адаптеру для завантаження необхідного ПЗ в ОС;
- монітор з роздільною здатністю екрану 1024x768 та більше, з підтримкою не менш ніж 256 кольорів.

5 ВИКЛИК І ЗАВАНТАЖЕННЯ

Перед використанням програми необхідно виконати наступні дії:

- виконайте запуск виконавчого файлу програми Cutting.exe та дочекайтеся завершення формування структур словника;
- створіть папку в кореневій папці програми та помістіть в неї всі необхідні тексти для аналізу;
- виконайте запуск виконавчого файлу програми Clusterize.exe.

Після першого успішного формування структури словника на жорсткому диску виконання першого кроку можна опустити і приступати безпосередньо до кластеризаії текстів.

Після запуску програма повністю готова до обробки текстів, що розташовані у створеній папці.

6 ВХІДНІ ДАНІ

Вхідними даними програми, що розробляється, є:

- словник української мови, що відповідає формату словника ВЕСУМ [15] (опціонально);
- множина текстових файлів у форматі .txt, що підлягають аналізу та подальшій кластеризації за авторами.

7 ВИХІДНІ ДАНІ

Вихідними даними програми, що розробляється, є:

- для модулю першої програми – структурований словник української мови з мінімально необхідним набором даних для побудови відповідної структури;
- для другого модулю – текстовий файл з результатами кластеризації вхідних текстів за авторами, текстові файли з частотними характеристиками та кількістю розпізнаних слів за кожним вхідним текстом окремо.

8 ОПИС ПРИЗНАЧЕНОГО ДЛЯ КОРИСТУВАЧА ІНТЕРФЕЙСУ

Після до папки «AuthorshipClusterizer» можна виконати завантаження одного з двох наявних програмних засобів: структуризацію словнику на основі довільного українського словника, що відповідає структурі ВЕСУМ, або здійснення кластеризації текстів за атрибутивними характеристиками. Програма структурування словнику не потребує інтерфейсу через лінійність виконання. На рисунку 8.1 зображено інтерфейс програми кластеризації.

```
Insert runTime Dictionary 00:00:03.74
Input directory name with texts:
Literature
Recognition time № 0: 00:00:00.21
Recognition time № 1: 00:00:00.00
Recognition time № 2: 00:00:00.15
Recognition time № 3: 00:00:00.00
Recognition time № 4: 00:00:00.00
Recognition time № 5: 00:00:00.04
Recognition time № 6: 00:00:00.00
Recognition time № 7: 00:00:00.04
Recognition time № 8: 00:00:00.02
Recognition time № 9: 00:00:00.20
        1)      2)      3)      4)      5)      6)      7)      8)      9)      10)
1)      0       0,01716 0,00732 0,01711 0,0193  0,01074 0,0187  0,01231 0,01301 0,00461
2)      0,01716 0       0,01782 0,02102 0,02223 0,01765 0,02198 0,01876 0,01926 0,01768
3)      0,00732 0,01782 0       0,01844 0,01987 0,01146 0,0202  0,01295 0,01283 0,00777
4)      0,01711 0,02102 0,01844 0       0,02142 0,01709 0,02072 0,01797 0,02019 0,017
5)      0,0193  0,02223 0,01987 0,02142 0       0,0193  0,02263 0,01945 0,0205  0,01944
6)      0,01074 0,01765 0,01146 0,01709 0,0193  0       0,01893 0,01278 0,01453 0,01087
7)      0,0187  0,02198 0,0202  0,02072 0,02263 0,01893 0       0,0175  0,01964 0,0185
8)      0,01231 0,01876 0,01295 0,01797 0,01945 0,01278 0,0175  0       0,01279 0,01221
9)      0,01301 0,01926 0,01283 0,02019 0,0205  0,01453 0,01964 0,01279 0       0,01325
10)     0,00461 0,01768 0,00777 0,017   0,01944 0,01087 0,0185  0,01221 0,01325 0
Clusterize time 00:00:24.15
nSearch runTime Dictionary 00:00:25.98
```

Рисунок 8.1 – Структура головного вікна програми кластеризатору

Головне меню програми складається з наступних пунктів що є базовими для терміналу операційної системи:

– «Зменшити вікно» (відповідає за зменшення);

– «Збільшити вікно» (відповідає за збільшення вікна);

– «Закрити вікно» (відповідає за закриття вікна).

9 ПОРЯДОК РОБОТИ З ПРОГРАМОЮ

Перед початком роботи користувач повинен впевнитись в безпечності користуванням ПК або ноутбуком.

Підготуйте текстові дані для опрацювання, попередньо перевіривши, щоб всі файли використовували кодування UTF-8 та знаходилися в окремій директорії. Виділити ще одну папку, куди перемістити файли додатків Cutting.exe, Clusterizer.exe та словник ВЕСУМ dict_corp_vis.txt.

Запустіть програму структуризації словнику Cutting та дочекатися її завершення. Запустити Clusterizer та вказати директорію з файлами, що підлягають аналізу. Після завершення роботи програми на екран виведуться дані кластеризації текстів, евклідові відстані між ними, часові показники виконання операцій вставки, пошуку та кластеризації. Для поглибленого аналізу текстів у директорії з файлами для аналізу ознайомтеся з частотними показниками наявних слів та відсотком розпізнаних слів для кожного тексту окремо.

10 ПОВІДОМЛЕННЯ

В табл. 10.1 представлені повідомлення користувачу, що можуть з’явитися у процесі роботи програми.

Таблиця 10.1 – Повідомлення, що з’являються в процесі роботи програми

Текст повідомлення	Кому призначене	Опис ситуації	Рекомендовані дії
Директорію з текстами не знайдено	Користувач	Виконавчий файл не знайшов папку з файлами текстів.	Ввести правильне ім’я директорії при виконанні програми
Файли словнику не знайдені	Користувач	Виконавчий файл не знайшов файли структурованого словнику	Виконати програму Cutting.exe та перенести виконавчий файл Clusterizer.exe до директорії з модулями словнику

ЗАТВЕРДЖЕНО
1116130.01202-01-ЛЗ

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Керівництво користувача. Керівництво структуризації словнику та кластеризації текстів за морфологічними атрибутами слів

1116130.01202-01 І3 01

Аркушів 8

2021

Анотація

Документ 1116130.01202-01 І3 01 «Система дослідження ефективності структур даних та кластеризації творів за авторами на основі кращої структури. Керівництво користувача. Керівництво структуризації словнику та кластеризації текстів за морфологічними атрибутами слів» входить до складу програмної документації дипломного проекту.

У документі представлене призначення та умови застосування програми, підготовка до роботи з програмою та перелік наявних функцій у програми.

Зміст

Вступ

Програмна система «AuthorshipClusterizer» досліджує методами лінгвістичного аналізу тексти, за допомогою чого на основі нового авторського інваріанту виконує кластеризацію текстів за їх схожістю морфологічних атрибутивних послідовностей. Програма ефективно використовує ресурси комп’ютера як під час виконання, так і для збереження модулів словнику на диску, що досягається внаслідок дослідження вказаних аспектів та їх оптимізацію після експериментальних тестувань.

Для використання програми, користувачу необхідно вміти працювати в операційній системі Windows, мати навички користування мишею і клавіатурою, розуміти принцип роботи текстових редакторів для підготовки тестових текстів.

Перед роботою необхідно ознайомитись з наступними документами:

- опис програми;
- керівництво користувача.

1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

Програмний комплекс призначений для кластеризації текстів за їх атрибутивними послідовностями слів, що відображають слугують в якості авторського інваріанту та дозволяють виконувати дослідження схожості текстів на їх основі. Програма використовує постійно оновлюваний ВЕСУМ за базовий словник української мови, що забезпечує обширний словниковий запас, а також застосовує ефективну структуру даних під час аналізу для швидкого та якісного розбору текстів. Він може бути застосований у наукових та видавничих сферах для аналізу тексту на наявність плагіату.

Для виконання програми необхідна наявність ПК або ноутбуку, на якому встановлена Windows 7 або новіша версія. Конфігурація ПК для функціонування програмного продукту повинна бути наступна:

- ПК, під управлінням 64-розрядної ОС Windows;
- процесор з тактовою частотою не нижче 2 ГГц;
- оперативна пам’ять не менше 6 ГБ;
- простір на жорсткому диску не менше 1 ГБ;
- маніпулятор миша;
- клавіатура;
- наявність CD/DVD приводу, USB роз’єму або LAN-адаптеру для завантаження необхідного ПЗ в ОС;
- монітор з роздільною здатністю екрану 1024x768 та більше, з підтримкою не менш ніж 256 кольорів.

Для користування програмою користувач повинен вміти працювати в операційній системі сімейства Windows та аналізувати дані близькості текстів на основі статистичних показників.

2 ПІДГОТОВКА ДО РОБОТИ

2.1 Склад і зміст дистрибутивного носія даних

Дистрибутив програмного продукту містить у собі:

- програмний додаток Cutting.exe;
- файл словнику ВЕСУМ dict_corp_vis.txt;
- програмний додаток Clusterizer.exe.

2.2 Порядок завантаження даних і програм

Перед першим запуском програми Clusterizer попередньо виконайте структуризацію словнику через додаток Cutting. Після формування всіх необхідних модулів словнику у Clusterizer треба вибрати директорію в операційній системі, з якої будуть проаналізовані всі текстові файли та будуть сформовані кластери на їх основі.

2.3 Порядок перевірки працездатності

Для перевірки працездатності програми необхідно провести тестовий запуск програми з наступним переліком дій:

- сформувати папку, куди перемістити всі текстові файли, що підлягають кластерному аналізу на подібність;
- перемістити файли Cutting.exe, Clusterizer.exe та dict_corp_vis.txt в окрему папку;
- запустити Cutting.exe та зачекати закінчення виконання програми;
- запустити Clusterizer.exe та вказати шлях до директорії з файлами, що була сформована раніше;
- після закінчення виконання програми оцінити виведені в консоль дані кластеризації, а також перевірити сформовані файли статистичного аналізу словників текстів, що були сформовані на кожен файл окремо в директорії, де знаходяться всі тексти.

3 ОПИС ОПЕРАЦІЇ

Далі приведено список операцій та послідовність дій для їх виконання.

3.1 Функція структурування та індексації словника української мови

Функція виконує структурування словника для економії пам’яті на жорсткому диску, індексує відповідні частини слів, на які виконується розбиття початкового словнику, а також зберігає всі отримані структури у текстові файли. Також функція містить вбудований моніторинг швидкодії виконання операції структуризації.

Для виконання функції пересвідчіться у наявності словнику ВЕСУМ у директорії (dict_corp_vis.txt) та завантажте виконавчий файл Cutting.exe. Дочекайтеся завершення обробки словнику, про що свідчитиме вивід часових вимірів у консоль.

3.2 Функція розбиття тексту на слова та їх пошук на льоту

Функція з вхідного тексту виділяє всі слова та здійснює їх пошук у словнику з метою отримання морфологічного атрибута кожного слова та частотної статистики зустрічі слів у тексті.

Для виконання операції завантаження Clusterizer.exe, дочекайтеся завантаження структури даних у пам’ять, після чого оберіть директорію з текстами для аналізу. Очікуйте завершення розбору, про що свідчитиме вивід часових показників у консоль. Після цього у директорії з виконавчим файлом з’явиться папка зі статистикою для кожного файлу тексту, а також список індексів атрибутів, що відображає послідовність слів у текстах.

3.3 Функція формування векторів атрибутів та кластеризації текстів

Функція відповідає за знайдення евклідових відстаней між текстами за їх векторами атрибутивних послідовностей та кластеризує їх на основі цих відстаней.

Функція потребує попереднього виконання функції розбору тексту, після чого оберіть пункт продовжити роботу, що запустить кластеризацію текстів. Результати пошуку відстаней між векторами атрибутів текстів буде виведено у вікно консолі у вигляді таблиці.

4. АВАРІЙНІ СИТУАЦІЇ

При повторюваних відмовах технічних засобів, які унеможливлюють роботу з програмою, необхідно звернутись до системного адміністратора.

У випадку одиничного збою треба виконати перезавантаження програмного засобу або системи. При повторних збоях звертатися до системного адміністратора.

У разі виявлення помилок у даних необхідно пересвідчитися у правильності алгоритму користування програмою, перевірити вхідні дані, після чого перезапустити програму або систему, якщо проблема була підтверджена.

У разі виявлення некоректного виводу зображення на електронний вимкнути персональний комп’ютер та від’єднати його та периферійні пристрої від мережі, перевірити контакт кабелю дисплею від комп’ютеру до монітору. Якщо проблема не була усунута, викликати адміністратора.

У разі відмови магнітних або твердотільних носіїв треба встановити програму з диску на справний магнітний або твердотільних носій.

У випадках виявлення несанкціонованого втручання в дані одразу повідомити про це системного адміністратора.

В інших аварійних ситуаціях повідомити системного адміністратора про аварійну ситуацію.

ЗАТВЕРДЖЕНО

1116130.01202-01 12 01-ЛЗ

СИСТЕМА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРУКТУР ДАНИХ ТА КЛАСТЕРИЗАЦІЇ ТВОРІВ ЗА АВТОРАМИ НА ОСНОВІ КРАЩОЇ СТРУКТУРИ

Текст програми

1116130.01202-01 12 01

Аркушів 29

2021

Анотація

Документ 1116130.01202-01 «Система дослідження ефективності структур даних та кластеризації творів за авторами на основі кращої структури. Текст програми» входить до складу програмної документації дипломного проекту.

У документі представлений опис структури робочих модулів і їх текст.

Зміст

1 СТРУКТУРА ПРОГРАМИ

Комплекс програмних засобів «AuthorshipClusterizer» складається з двох модулів:

– Dictionary cutting – відповідає за стиснення та структуризацію вхідного словнику української мови;

– Clusterizator – відповідає за розбір вхідних текстів на слова та їх кластеризацію за відповідними атрибутивними властивостями послідовностей слів.

На рис. 1.1 зображена схема взаємодії модулів програми.

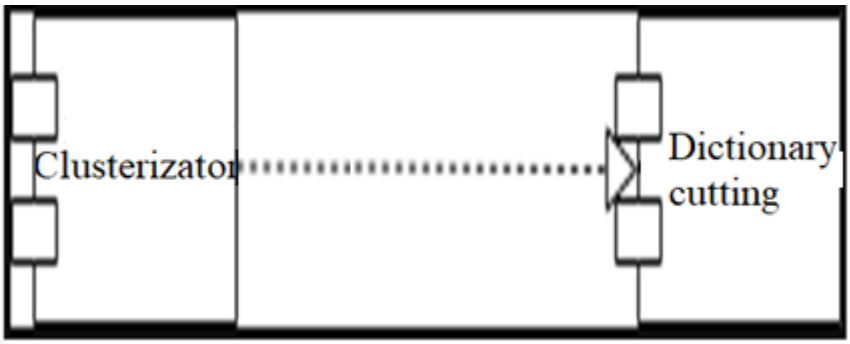


Рисунок 1.1 – Схема взаємодії модулів програми

Модуль Dictionary cutting складається з таких файлів написаних на мові C#, як: WordsCutting.cs, PropertiesAnalyzer.cs, RBComparer.cs, ReverseBase.cs.

WordsCutting.cs, PropertiesAnalyzer.cs, RBComparer.cs, ReverseBase.cs – група C# файлів, що виконує структуризацію словнику шляхом відділення закінчень з індексуванням їх унікальних списків для якомога більшого стиснення даних; індексація слів таким чином, щоб усі словоформи і однокореневі слова визначалися як одне і те ж слово; формування списку унікальних морфологічних атрибутів; збереження отриманих результатів у текстові файли;

Модуль Clusterizator складається з наступних файлів: IBaseStructure.cs, DictionaryAdapt.cs, ManagedList.cs, Trie.cs, Node.cs, NodeComparer.cs, Recognition.cs.

IBaseStructure.cs – інтерфейс для уніфікації всіх структур даних за базовими операціями (пошуку та вставки).

DictionaryAdapt.cs, ManagedList.cs, Trie.cs – адаптація 3-х структур даних (словник, відсортований список, префіксальне дерево), що будуть порівнюватися на швидкодію виконання операцій, на основі чого буде прийнято рішення, яка структура краща для даної задачі.

Node.cs, NodeComparer.cs – допоміжні класи для побудови префіксального дерева trie, а також порівняння 2-х вузлів дерева з метою сортування.

Recognition.cs – клас для виділення з текстів слів для їх подальшого переведення в морфологічні атрибути. Кластеризація текстів за послідовностями таких атрибутів.

2 ТЕКСТ ПРОГРАМИ

2.1 Module Cutting.cs

```
static string CutBasic(string a, string b)
    {
        if (a == b)
            return a;
        if (a.Length > b.Length)
        {
            return string.Concat(a.TakeWhile((c, i) => i <
b.Length && c == b[i]));
        }
        else
        {
            return string.Concat(b.TakeWhile((c, i) => i <
a.Length && c == a[i]));
        }
    }
    static string CompareWords(string a, string b,
List<string> endingList, ref string drop, bool first)
    {
        string ret = "";
        bool dropEnabled = false;
        string sign = "", temp;
        int length = Math.Min(a.Length, b.Length);
        if (a == b)
        {
            if (first)
                endingList.Add("/");
            return a;
        }
        for (int i = 0; i < length; i++)
        {
            if (a[i] == b[i])
                ret += a[i];
            else if (i + 1 < length)
            {
                if (a[i + 1] == b[i + 1])
                {
                    ret += a[i];
                    drop = i + b[i].ToString();
                    dropEnabled = true;
                }
            }
            else
            {
                if (dropEnabled)
                    sign = "+";
                else sign = "/";
                temp = sign + Biggest(ret, a, b);
                endingList.Add(temp);
                return ret;
            }
        }
        if (dropEnabled)
            sign = "+";
        else sign = "/";
        temp = sign + Biggest(ret, a, b);
        if (!endingList.Contains(temp) && temp != "+")
            endingList.Add(temp);
        return ret;
    }
    static string Biggest(string ret, string a, string b)
    {
        if (a.Length > b.Length)
        {
            if (a != ret)
                return b.Substring(ret.Length);
            return "";
        }
        else if (b != ret) return b.Substring(ret.Length);
```

```
        return "";
    }
    static List<List<string>> CutPrefix(List<ReverseBase>
revBs, int[] basicIndexes)
    {
        int cnt = 0, wcnt = 0;
        List<string> tempPrefix = new List<string>();
        List<List<string>> prefix = new
List<List<string>>();
        char[] temp;
        for (int i = 0; i < revBs.Count; i++)
        {
            if (revBs[i].bs.Length > 3 &&
!revBs[i].bs.Contains("-"))
            {
                for (int j = 0; j < revBs.Count; j++)
                {

                    if (revBs[i].bs[0] == revBs[j].bs[0])
                    {
                        if (revBs[j].bs.Length > 3 &&
!revBs[j].bs.Contains("-"))
                        {
                            while (revBs[i].bs.Length > wcnt &&
revBs[j].bs.Length > wcnt && revBs[i].bs[wcnt] ==
revBs[j].bs[wcnt])
                            {
                                wcnt++;
                            }
                            if (revBs[i].bs.Length == wcnt && wcnt
< revBs[j].bs.Length &&
revBs[i].endInd.SequenceEqual(revBs[j].endInd))
                            {
                                temp = revBs[j].bs.Substring(wcnt,
revBs[j].bs.Length - revBs[i].bs.Length).ToCharArray();
                                Array.Reverse(temp);
                                tempPrefix.Add(new String(temp) +
"-");
                                basicIndexes[revBs[j].index] =
revBs[i].index;
                            }
                            wcnt = 0;
                        }
                    }
                    else if
(revBs[i].bs[0].CompareTo(revBs[j].bs[0]) < 0)
                        break;
                }

                if (tempPrefix.Count != 0)
                {
                    while (cnt < prefix.Count)
                    {
                        if (prefix[cnt].SequenceEqual(tempPrefix))
                            break;
                        cnt++;
                    }
                    if (cnt == prefix.Count)
                        prefix.Add(new List<string>(tempPrefix));
                    tempPrefix.Clear();
                    cnt = 0;
                }
            }
        }
        return prefix;
    }
    static List<string>
CutUniqueEndings(List<List<string>> finEnding,
List<string> Nbases, List<List<string>> sumEnding)
    {
        int wc = 0;
        bool finEndingContains = false;
        List<string> endingIndex = new List<string>();
        List<string> ending = new List<string>();
        finEnding.Add(new List<string>(ending));
        for (int index = 0; index < Nbases.Count; index++)
```

```
        {
            ending = sumEnding[index];
            ending.Sort();
            if (ending.Count != 0)
            {
                while (wc < finEnding.Count())
                {
                    if (finEnding[wc].SequenceEqual(ending))
                    {
                        finEndingContains = true;
                        break;
                    }
                    wc++;
                }
                if (!finEndingContains)
                    finEnding.Add(new List<string>(ending));
            }
            endingIndex.Add((wc != 0 ? (wc - 1).ToString() :
""));
            finEndingContains = false;
            wc = 0;
        }
        finEnding.RemoveAt(0);
        return endingIndex;
    }
    static void Main(string[] args)
    {
        Stopwatch stopWatch = new Stopwatch();
        string elapsedTime;
        stopWatch.Reset();
        stopWatch.Start();
        string[] textLines = new string[6213575];
        string buff, basic, drop = "";
        int i = 0, k = 0;
        int size = 6213575;//6213561
        List<string> cut = new List<string>();
        List<List<string>> partOfSpeechIndexes = new
List<List<string>>();
        List<string> partOfSpeechFiltered = new
List<string>();
        List<List<string>> sumEnding = new
List<List<string>>();
        List<List<string>> finEnding = new
List<List<string>>();
        List<List<string>> prefix = new
List<List<string>>();
        List<string> bases = new List<string>();
        List<string> Nbases = new List<string>();
        List<string> ending = new List<string>();
        List<string> drops = new List<string>();
        List<string> endingIndex = new List<string>();
        int[] basicIndexes;
        Dictionary<string, List<string>> relatedBases = new
Dictionary<string, List<string>>();
        List<string> bs = new List<string>();
        bool rbf = false;
        List<string> uniquePropertyIndexes = new
List<string>();
        using (StreamReader sr = new
StreamReader("dict_corp_vis.txt",
System.Text.Encoding.Default))
        {
            while ((textLines[i] = sr.ReadLine()) != null)
            {
                i++;
            }
        }
        using (StreamReader sr = new
StreamReader("uniquePropertyIndexes.txt",
System.Text.Encoding.Default))
        {
            while ((buff = sr.ReadLine()) != null)
            {
                uniquePropertyIndexes.Add(buff);
            }
```

```
        }
        /*int u = 0;
        string buffer = "";
        int jn = 0, un = 0; ;
        bool testy = false;
        int ko = 0;
        List<string> tu = new List<string>();
        for (int j = 0; j < size; j++)
        {
           if (textLines[j][0] != ' ')
           {
              jn = j;
              while (j + 1 < size && textLines[j + 1][0] == ' ')
              {
                 u = j;
                 while (u + 1 < size && textLines[u + 1][0]
== ' ')
                 {
                    if (textLines[j +
1][2].CompareTo(textLines[u + 1][2]) < 0)
                    {
                       buffer = textLines[j + 1];
                       textLines[j + 1] = textLines[u + 1];
                       textLines[u + 1] = buffer;
                       testy = true;
                    }
                    u++;
                 }
                 if (testy)
                    tu.Add(textLines[j + 1]);
                 j++;
              }
              if (textLines[jn + 1][0] == ' ' &&
textLines[jn][0] != textLines[jn + 1][2])
              {
                 u = jn;
                 for (int ink = jn; ink < j; ink++)
                 {
                    for (int ik = jn; ik < j; ik++)
                    {
                       if (textLines[u + 1][2] !=
textLines[jn][0] && textLines[u + 2][2] == textLines[jn][0]
&& textLines[u + 2][0] == ' ' && textLines[u + 1][0] == ' ')
                       {
                          buffer = textLines[u + 1];
                          textLines[u + 1] = textLines[u + 2];
                          textLines[u + 2] = buffer;
                       }
                       u++;
                    }
                    u = jn;
                 }
              }
           }
           if (testy)
              ko++;
           tu.Clear();
           testy = false;
        }
        for (int j = 0; j < size; j++)
        {
        }
        using (StreamWriter sw = new
StreamWriter("dict_corp_vis.txt", false,
System.Text.Encoding.Default))
        {
           foreach (string line in textLines)
              sw.WriteLine(line);
        }*/
        for (int y = 0; y < size; y++)
           textLines[i] = textLines[y].ToLower();
        int ko = 0;
        string tests = "";
```

```
        char firstSign = '$';
        for (int j = 0; j < size; j++)
        {
            if (textLines[j][0] != ' ')
            {
                ko = j;
                while (j + 1 < size && textLines[j + 1][0] == ' ')
                {
                    if (textLines[ko][0] != textLines[j + 1][2] &&
firstSign != textLines[j + 1][2])
                    {
                        firstSign = textLines[j + 1][2];
                        textLines[j + 1] = textLines[j +
1].Remove(0, 2);
                        tests = textLines[j + 1];
                        rbf = true;
                        bs.Add(tests.Substring(0, tests.IndexOf("
")));
                    }
                    j++;
                }
            }
            if (rbf)
                relatedBases.Add(textLines[ko].Substring(0,
textLines[ko].IndexOf(" ")), new List<string>(bs));
            rbf = false;
            bs.Clear();
            firstSign = '$';
        }/*
        using (StreamWriter sw = new
StreamWriter("dict_corp_vis.txt", false,
System.Text.Encoding.Default))
        {
            foreach (string line in textLines)
                sw.WriteLine(line);
        }*/
        int checkCount = 0, baseNumber = 0, pos = 0;
        List<string> rbaseList = new List<string>();
        Dictionary<int, int> relatedBasesIndexes = new
Dictionary<int, int>();//child index, parent index
        List<string> prevEndings = new List<string>();
        string prevBasic = "", posTemp = "";
        int cj = 0;
        var tmpProperties = new List<string>();
        var propertiesIndexes = new List<int>();
        int colonRowCount = 0;
        for (int j = 0; j < size; j++)
        {
            if (textLines[j][0] != ' ')
            {
                pos = textLines[j].IndexOf(" ") + 1;
                posTemp = "";
                cj = j;
                do
                {
                    while (pos != textLines[cj].Length &&
textLines[cj][pos] != ' ' && colonRowCount !=
5)//textLines[j][pos] != ':' &&
                    {
                        if (textLines[cj][pos] == ':')
                            colonRowCount++;
                        posTemp += textLines[cj][pos];
                        pos++;
                    }
                    colonRowCount = 0;
                    pos = textLines[cj + 1].Remove(0,
2).IndexOf(" ") + 3;
                    tmpProperties.Add(posTemp);
                    posTemp = "";
                    cj++;
                } while (cj + 1 < size && textLines[cj][0] == '
');
                textLines[j] = textLines[j].Substring(0,
textLines[j].IndexOf(" "));
```

```
            basic = textLines[j];
            if (checkCount > 0)
            {
               relatedBasesIndexes.Add(Nbases.Count,
baseNumber);
               checkCount--;
            }
            else if (relatedBases.TryGetValue(basic, out
rbaseList))
            {
               checkCount = rbaseList.Count;
               baseNumber = Nbases.Count;
            }

            while (j + 1 < size && textLines[j + 1][0] == ' ')
            {
               buff = textLines[j + 1].Remove(0, 2);
               basic = CutBasic(basic, buff.Substring(0,
buff.IndexOf(" ")));
               j++;
               k--;
            }
            j = j + k;
            CompareWords(basic, textLines[j], ending, ref
drop, true);
            while (j + 1 < size && textLines[j + 1][0] == ' ')
            {
               buff = textLines[j + 1].Remove(0, 2);
               textLines[j + 1] = buff.Substring(0,
buff.IndexOf(" "));

               CompareWords(basic, textLines[j + 1],
ending, ref drop, false);
               textLines[j + 1] = "  " + textLines[j + 1];
               j++;
            }
            k = 0;
            ending.Sort();
            if (basic != prevBasic)
            {
               sumEnding.Add(new List<string>(ending));
               /*if (!bases.Contains(basic))
                  bases.Add(basic);*/
               Nbases.Add(basic);
               if
(!partOfSpeechFiltered.Contains(posTemp))
                  partOfSpeechFiltered.Add(posTemp);
               partOfSpeechIndexes.Add(new List<string>
{ (partOfSpeechFiltered.IndexOf(posTemp) < 10 ? '0' +
partOfSpeechFiltered.IndexOf(posTemp).ToString() :
partOfSpeechFiltered.IndexOf(posTemp).ToString()) });
               drops.Add(drop);
            }
            else if (!ending.SequenceEqual(prevEndings))
            {
               sumEnding.Add(new List<string>(ending));
               Nbases.Add(basic);
               if
(!partOfSpeechFiltered.Contains(posTemp))
                  partOfSpeechFiltered.Add(posTemp);
               partOfSpeechIndexes.Add(new List<string>
{ partOfSpeechFiltered.IndexOf(posTemp) < 10 ? '0' +
partOfSpeechFiltered.IndexOf(posTemp).ToString() :
partOfSpeechFiltered.IndexOf(posTemp).ToString() });
               drops.Add(drop);
            }
            else
            {
               if
(!partOfSpeechFiltered.Contains(posTemp))
                  partOfSpeechFiltered.Add(posTemp);
               if
(!partOfSpeechIndexes[partOfSpeechIndexes.Count -
1].Contains(partOfSpeechFiltered.IndexOf(posTemp) < 10 ?
'0' + partOfSpeechFiltered.IndexOf(posTemp).ToString() :
partOfSpeechFiltered.IndexOf(posTemp).ToString()))

partOfSpeechIndexes[partOfSpeechIndexes.Count -
```

```
1].Add(partOfSpeechFiltered.IndexOf(posTemp) < 10 ? '0' +
partOfSpeechFiltered.IndexOf(posTemp).ToString() :
partOfSpeechFiltered.IndexOf(posTemp).ToString());
                }
                prevEndings.Clear();
                prevBasic = basic;
                prevEndings.AddRange(ending);
                ending.Clear();
                drop = "";
            }
        }
        basicIndexes = new int[Nbases.Count];
        endingIndex = CutUniqueEndings(finEnding,
Nbases, sumEnding);
        List<ReverseBase> revBs = new
List<ReverseBase>();
        for (int h = 0; h < sumEnding.Count; h++)
        {
            revBs.Add(new ReverseBase(Nbases[h],
sumEnding[h], h));
        }
        revBs.Sort(new RBComparer());
        for (int h = 0; h < Nbases.Count; h++)
        {
            basicIndexes[h] = h;
        }
        prefix = CutPrefix(revBs, basicIndexes);
        int ei = 0;
        foreach (string st in Nbases)
        {
            cut.Add(st);
            ei++;
        }
        for (int g = 0; g < Nbases.Count; g++)
        {
            if (relatedBasesIndexes.TryGetValue(g, out ei))
                basicIndexes[g] = ei;
        }
        stopWatch.Stop();
        TimeSpan ts = stopWatch.Elapsed;
        elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
            ts.Hours, ts.Minutes, ts.Seconds,
            ts.Milliseconds / 10);
        Console.WriteLine("Cutting dictionary in " +
elapsedTime);
        Console.ReadKey();
        using (StreamWriter sw = new
StreamWriter("prefix.txt", false,
System.Text.Encoding.Default))
        {
            foreach (List<string> line in prefix)
                sw.WriteLine(string.Join("", line));
        }
        using (StreamWriter sw = new
StreamWriter("properties.txt", false,
System.Text.Encoding.Default))
        {
            foreach (string line in tmpProperties)
                sw.WriteLine(line);
        }
        using (StreamWriter sw = new
StreamWriter("finEnding.txt", false,
System.Text.Encoding.Default))
        {
            foreach (List<string> line in finEnding)
                sw.WriteLine(string.Join("", line));
        }
        using (StreamWriter sw = new
StreamWriter("output.txt", false,
System.Text.Encoding.Default))
        {
            foreach (string line in cut)
                sw.WriteLine(line);
        }
```

```
        using (StreamWriter sw = new
StreamWriter("partOfSpeech.txt", false,
System.Text.Encoding.Default))

        {

            foreach (string line in partOfSpeechFiltered)

                sw.WriteLine(line);

        }

        using (StreamWriter sw = new
StreamWriter("partOfSpeechIndexes.txt", false,
System.Text.Encoding.Default))

        {

            foreach (List<string> line in
partOfSpeechIndexes)

                sw.WriteLine(string.Join(" ", line));

        }

        using (StreamWriter sw = new
StreamWriter("switch.txt", false,
System.Text.Encoding.Default))

        {

            foreach (string line in drops)

                sw.WriteLine(line);

        }

        using (StreamWriter sw = new
StreamWriter("endingIndexes.txt", false,
System.Text.Encoding.Default))

        {

            foreach (string ind in endingIndex)

                sw.WriteLine(ind);

        }

        using (StreamWriter sw = new
StreamWriter("basicIndexes.txt", false,
System.Text.Encoding.Default))

        {

            foreach (int index in basicIndexes)

                sw.WriteLine(index);

        }

    }

  }

}

         static void GetUniqueAttributes()
         {
             int i = 0;
             var textLines = new
List<string>();
             var uniqueProperties = new
List<string>();
             var uniquePropertyIndexes = new
List<string>();
             var buff = string.Empty;
             using (StreamReader sr = new
StreamReader("properties.txt",
System.Text.Encoding.Default))
             {
                 while ((buff = sr.ReadLine())
!= null)
                 {
                     textLines.Add(buff);
                 }
             }

             for (i = 0; i < textLines.Count;
i++)
             {
                 if
(!uniqueProperties.Contains(textLines[i]))
                 {

uniqueProperties.Add(textLines[i]);

uniquePropertyIndexes.Add(uniqueProperties.Cou
nt > 999 ? uniqueProperties.Count.ToString() :
uniqueProperties.Count > 99 ? "0" +
uniqueProperties.Count.ToString() :
uniqueProperties.Count > 9 ? "00" +
uniqueProperties.Count.ToString() : "000" +
uniqueProperties.Count.ToString());
                 }
                 else
                 {

uniquePropertyIndexes.Add(uniqueProperties.Ind
exOf(textLines[i])> 999 ?
uniqueProperties.IndexOf(textLines[i]).ToStrin
g() : uniqueProperties.IndexOf(textLines[i]) >
99 ? "0" +
uniqueProperties.IndexOf(textLines[i]).ToStrin
g() : uniqueProperties.IndexOf(textLines[i]) >
9 ? "00" +
uniqueProperties.IndexOf(textLines[i]).ToStrin
g() : "000" +
uniqueProperties.IndexOf(textLines[i]).ToStrin
g());
                 }
             }

             using (StreamWriter sw = new
StreamWriter("uniqueProperties.txt", false,
System.Text.Encoding.Default))
             {
                 foreach (var property in
uniqueProperties)
```

```
                sw.WriteLine(property);
            }

            using (StreamWriter sw = new
StreamWriter("uniquePropertyIndexes.txt",
false, System.Text.Encoding.Default))
            {
                foreach (var index in
uniquePropertyIndexes)
                sw.WriteLine(index);
            }
        }
```

2.2 Module IBaseStruct.cs

```
    interface IBaseStruct
    {
        public int Search(string s);
        public string SearchS(string s);
        public void Insert(string s, string
index);
    }
```

2.3 Module SortedList.cs

```
    public class ManagedList : IBaseStruct
    {
        List<string> st;
        List<string> index;
        public ManagedList()
        {
            st = new List<string>();
            index = new List<string>();
        }
        public int Search(string s)
        {
            int ind = st.BinarySearch(s);
            if (ind > -1)
                return int.Parse(index[ind]);
            else return -1;
        }
        public void Insert(string s, string
ed)
        {
            int ind = st.BinarySearch(s);
            if (ind < 0)
            {
                st.Insert(~ind, s);
                index.Insert(~ind, ed);
            }
        }

        public string SearchS(string s)
        {
            int ind = st.BinarySearch(s);
            if (ind > -1)
                return index[ind];
            else return "-1";
        }
    }
```

2.4 Module DictionaryAdapt.cs

```
    public class DictionaryAdapt : IBaseStruct
    {
        Dictionary<string, string> dictionary
= new Dictionary<string, string>();
        public int Search(string s)
        {
            if (dictionary.ContainsKey(s))
                return
int.Parse(dictionary[s].ToString());
            else
                return -1;
        }

        public string SearchS(string s)
        {
            if (dictionary.ContainsKey(s))
                return dictionary[s];
            else
                return "-1";
        }

        public void Insert(string s, string
index)
        {
            if (!dictionary.ContainsKey(s))
                dictionary.Add(s, index);
        }
    }
```

2.5 Module Trie.cs

```
public class Trie : IBaseStruct
    {

        private readonly Node _root;

        public Trie()
        {
            _root = new Node('^'.ToString(),
0, null);
        }

        public Node Prefix(string s)
        {
            var currentNode = _root;
            var result = currentNode;

            foreach (var c in s)
            {
                currentNode =
currentNode.FindChildNode(c);
                if (currentNode == null)
                    break;
                result = currentNode;
            }

            return result;
        }

        public void Insert(string s, string
ed)
```

```
        {
            var commonPrefix = Prefix(s);
            var current = commonPrefix;

            for (var i = current.Depth; i <
s.Length; i++)
            {
                var newNode = new
Node(s[i].ToString(), current.Depth + 1,
current);
                IndexInsert(newNode, current);
                current = newNode;
            }
            IndexInsert(current, ed);
        }
        public void IndexInsert(Node nd,
string ed)
        {
            int ind =
nd.Children.BinarySearch(new Node(ed, nd.Depth
+ 1, nd), new NodeComparer());
            if (ind < 0)
            {
                nd.Children.Insert(~ind, new
Node(ed, nd.Depth + 1, nd));
            }
        }
        public void IndexInsert(Node nd, Node
cNd)
        {
            int ind =
cNd.Children.BinarySearch(nd, new
NodeComparer());
            if (ind < 0)
            {
                cNd.Children.Insert(~ind, nd);
            }
        }
        public void Delete(string s)
        {
            if (SearchS(s) == "1")
            {
                var node =
Prefix(s).FindChildNode('$');

                while (node.IsLeaf())
                {
                    var parent = node.Parent;

parent.DeleteChildNode(node.Value.ToString());
                    node = parent;
                }
            }
        }
        public int Search(string s)
        {
            var prefix = Prefix(s);
            if (prefix.Depth == s.Length)
                return
int.Parse(prefix.FindIndex().ToString());
            return -1;
        }

        public string SearchS(string s)
        {
            var prefix = Prefix(s);
            if (prefix.Depth == s.Length)
                return
prefix.FindIndex().ToString();
            return "-1";
        }
    }
```

2.6 Module Node.cs

```
public class Node
    {
        public string Value { get; set; }
        public List<Node> Children { get; set;
}
        public Node Parent { get; set; }
        public int Depth { get; set; }

        public Node(string value, int depth,
Node parent)
        {
            Value = value;
            Children = new List<Node>();
            Depth = depth;
            Parent = parent;
        }

        public bool IsLeaf()
        {
            return Children.Count == 0;
        }
        public static int
BinarySearchIterative(List<Node> inputArray,
string key)
        {
            int min = 0;
            int max = inputArray.Count - 1;
            while (min <= max)
            {
                int mid = (min + max) / 2;
                if (key ==
inputArray[mid].Value)
                {
                    return mid;
                }
                else if
(key.CompareTo(inputArray[mid].Value)<0)
                {
                    max = mid - 1;
                }
                else
                {
                    min = mid + 1;
                }
            }
            return -1;
        }
```

```
        public Node FindChildNode(char c)
        {
            int ind =
BinarySearchIterative(Children, c.ToString());
            if(ind>-1)
                    return Children[ind];

            return null;
        }
        public int FindIndex()
        {
            int ind;
            foreach (var child in Children)
                if (int.TryParse(child.Value,
out ind))
                    return ind;

            return -1;
        }
        public void DeleteChildNode(string c)
        {
            for (var i = 0; i <
Children.Count; i++)
                if (Children[i].Value == c)
                    Children.RemoveAt(i);
        }
    }
```

2.7 Module NodeComparer.cs

```
class NodeComparer:IComparer<Node>
    {
        public int Compare(Node p1, Node p2)
        {
            if (String.Compare(p1.Value,
p2.Value) > 0)
                return 1;
            else if (String.Compare(p1.Value,
p2.Value) < 0)
                return -1;
            else
                return 0;
        }
    }
```

2.8 Module Clusterizer.cs

```
    static class Program

    {

        static void SplitAlternative(out
List<string> ed, out List<string> edAlt,
string line)

        {

            var buff = string.Empty;

            ed = new List<string>();

            edAlt = new List<string>();

            int count = 0;

            while (count < line.Length)

            {

                if (line[count] == '/')

                {

                    count++;

                    while (count < line.Length
&& line[count] != '/' && line[count] != '+')

                    {

                        buff += line[count];

                        count++;

                    }

                    ed.Add(buff);

                    buff = "";

                    continue;

                }


                if (line[count] == '+')

                {

                    count++;

                    while (count < line.Length
&& line[count] != '/' && line[count] != '+')

                    {

                        buff += line[count];

                        count++;

                    }

                    edAlt.Add(buff);

                    buff = "";

                    continue;

                }

            }

        }


        static int GetBigger(List<int> pos)

        {

            int max = -1, maxInd = -1;
```

```
            for (int i = 0; i < pos.Count;
i++)
            {
                if (pos[i] > max)
                {
                    max = pos[i];
                    maxInd = i;
                }
            }

            return maxInd;
        }

        public static Dictionary<string,
List<int>>
CountEndingsPartOfSpeech(List<string>
endingIndexes, List<string> pos, List<string>
endings)
        {
            var posInd = new
Dictionary<string, List<int>>();
            var endingsSeparated = new
List<List<string>>();
            char[] delimiters = { '/', '+' };
            var uniqueED = new List<string>();
            List<int> temp;
            var resultPOS = new List<int>();
            foreach (string st in endings)
            {

endingsSeparated.Add(st.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries).ToList(
));
            }

            for (int i = 0; i <
endingIndexes.Count - 1; i++)
            {
                foreach (string st in
endingsSeparated[Convert.ToInt32(endingIndexes
[i])])
                {
                    if (posInd.TryGetValue(st,
out temp))
                    {

temp[Convert.ToInt32(pos[i])]++;
                        posInd[st] = temp;
                    }
                    else
                    {
                        uniqueED.Add(st);
                        posInd.Add(st, new
List<int>() { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0 });

posInd[st][Convert.ToInt32(pos[i])]++;
                    }
                }
            }
            foreach (string st in uniqueED)
            {
resultPOS.Add(GetBigger(posInd[st]));
            }
            return posInd;
        }
        public static void
InsertProcessing<T>(ref T str, List<string>
endingIndexes, List<string> basicIndexes,
List<string> endings, List<string> drops,
List<string> pos) where T : IBaseStruct
        {
            int count = 0, i = 0,
parsedPropertyIndex = 0, tempCntr = 0, posit =
0, propertyIndex, size = 6213575;
            var buff = string.Empty;
            var endingsList = new
List<string>();
            var endingsListAlt = new
List<string>();
            var propertiesFileExists = false;
            var dictIndexes = new
string[6213575];
```

```
            var propertyIndexes = new
string[6213575];

            var words = new DictionaryAdapt();

            var textLines = new string[size];

            var propertiesList = new
List<int>();


            if
(File.Exists("filteredPropertyIndexes.txt"))

            {

                propertiesFileExists = true;

                using (StreamReader sr = new
StreamReader("filteredPropertyIndexes.txt",
System.Text.Encoding.Default))

                {

                    while ((dictIndexes[i] =
sr.ReadLine()) != null)

                    {

                        i++;

                    }

                }

                i = 0;

                using (StreamReader sr = new
StreamReader("uniquePropertyIndexes.txt",
System.Text.Encoding.Default))

                {

                    while ((propertyIndexes[i]
= sr.ReadLine()) != null)

                    {

                        i++;

                    }

                }

                i = 0;

            }

            if (!propertiesFileExists)

            {

                using (StreamReader sr = new
StreamReader("dict_corp_vis.txt",
System.Text.Encoding.Default))

                {

                    while ((textLines[i] =
sr.ReadLine()) != null)

                    {

                        i++;

                    }

                }

            for (int j = 0; j + 1 < size; j++)

                {

                    if (textLines[j][0] != '
')

                    {

                        posit =
textLines[j].IndexOf(" ");


words.Insert(textLines[j].Remove(posit),
j.ToString());

                        while (j + 2 < size &&
textLines[j + 1][0] == ' ')

                        {

                            j++;

                            buff =
textLines[j].Remove(0, 2);


words.Insert(buff.Remove(buff.IndexOf(" ")),
j.ToString());

                        }

                    }

                }

            }

            i = 0;

            using (StreamReader sr = new
StreamReader("output.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    string bI =
basicIndexes[count];

                    if (endingIndexes[count]
!= "" && drops[count] == "")

                    {
```

```
                        endingsList =
endings[int.Parse(endingIndexes[count])].Split
('/').ToList();

                        foreach (string ed in
endingsList)

                        {
                    if (!propertiesFileExists)

                            {

propertyIndex = words.Search(buff + ed);
propertiesList.Add(propertyIndex);

                            }


parsedPropertyIndex =
int.Parse(dictIndexes[i]);

                            i++;
if(parsedPropertyIndex != -1)

                            {
str.Insert(buff + ed, bI +
propertyIndexes[parsedPropertyIndex]);

                            }

                            else tempCntr++;

                        }

                    }

                    else if (drops[count]!="")

                    {

                        SplitAlternative(out
endingsList, out endingsListAlt,
endings[int.Parse(endingIndexes[count])]);

                        foreach (string ed in
endingsList)

                        {
if(!propertiesFileExists)

                            {

                                propertyIndex
= words.Search(buff + ed);


propertiesList.Add(propertyIndex);

                            }




parsedPropertyIndex =
int.Parse(dictIndexes[i]);

                            i++;
if(parsedPropertyIndex != -1)

                            {

str.Insert(buff + ed, bI +
propertyIndexes[parsedPropertyIndex]);

                            }

                            else tempCntr++;

                        }

                        foreach (string edAlt
in endingsListAlt)

                        {

                            char[] a =
buff.ToCharArray();

                            if
(drops[count].Length == 2)


a[int.Parse(drops[count][0].ToString())] =
drops[count][1];

                            else

                            {


a[int.Parse((drops[count][0].ToString() +
drops[count][1].ToString()))] =
drops[count][2];

                            }

                            buff = new
string(a);



                            if
(!propertiesFileExists)

                            {

                                propertyIndex
= words.Search(buff + edAlt);


propertiesList.Add(propertyIndex);

                            }
parsedPropertyIndex =
int.Parse(dictIndexes[i]);

                            i++;
if(parsedPropertyIndex != -1)

                            {
str.Insert(buff + edAlt, bI +
propertyIndexes[parsedPropertyIndex]);

                            }

                            else tempCntr++;
```

```
                    }
                }
                else
                {
                    if
(!propertiesFileExists)
                    {
                        propertyIndex =
words.Search(buff);

propertiesList.Add(propertyIndex);
                    }

                    parsedPropertyIndex =
int.Parse(dictIndexes[i]);
                    i++;

                    if
(parsedPropertyIndex != -1)
                    {
                        str.Insert(buff,
bI + propertyIndexes[parsedPropertyIndex]);
                    }
                    else tempCntr++;
                }
                count++;
            }
        }

        if (!propertiesFileExists)
        {
            using (var sw = new
StreamWriter("filteredPropertyIndexes.txt",
false, System.Text.Encoding.Default))
            {
                foreach (var val in
propertiesList)
                    sw.WriteLine(val);
            }
        }

    public static IEnumerable<string>
SplitAndKeep(this string s, char[] delims)
    {
        int start = 0, index;
        //char[] delimiters = { ' ', '[',
']', '\"', ':', ';', '%', '@', '.', ',', '!',
'?', '#', '$', '(', ')', '{', '}', '>', '<',
'-', '+', '=', '/', '*', '^' };

        while ((index =
s.IndexOfAny(delims, start)) != -1)
        {
            if (index - start > 0)
                yield return
s.Substring(start, index - start);
            yield return
s.Substring(index, 1);
            start = index + 1;
        }

        if (start < s.Length)
        {
            yield return
s.Substring(start);
        }
    }
    public static List<List<string>>
DivideSentences(string buff,
List<List<string>> sentences, ref bool end)
    {
        var article = true;
        var added = false;
        foreach (char c in buff)
        {
            if (char.IsLetter(c) &&
!char.IsUpper(c))
            {
```

```
                    article = false;
                }
            }
            var regex = new Regex(@"([
…"",.%^&*();:\[\]!{}=+#$?@<>\-/_])",
RegexOptions.Compiled);
            var res =
regex.Split(buff.ToLower());
            var tokens = new List<string>();
            var tokenLine = res.Where(s => s
!= String.Empty);
            bool endNotAdded = false;
            string lastToken = string.Empty;
            foreach (string token in
tokenLine)
            {
                if (endNotAdded && token !=
"." && token != "!" && token != "?" && token
!= "…")
                {
                    if (!end)
                    {

sentences[sentences.Count -
1].AddRange(tokens);
                        added = true;
                    }
                    else
                    {
                        sentences.Add(tokens);
                        added = true;
                    }
                    tokens.Clear();
                    endNotAdded = false;
                    end = true;
                }
                if (token == "!" || token ==
"?" || token == "." || token == "…")
                    endNotAdded = true;
                tokens.Add(token);
                lastToken = token;
            }
            if (endNotAdded)
            {
                if (!end)
                    sentences[sentences.Count
- 1].AddRange(tokens);
                else sentences.Add(tokens);

                added = true;
                end = true;
            }
            else
            {
                if (article)
                    sentences.Add(tokens);
                if (!added)
                {
                    sentences.Add(tokens);
                }
                if (lastToken != "." &&
lastToken != "!" && lastToken != "?" &&
lastToken != "…")
                    end = false;
            }

            return sentences;
        }

        public static void
SearchProcessing<T>(List<string> bases, string
outDirName, T str, string dirName) where T :
IBaseStruct
        {
            string buff, elapsedTime;
            Stopwatch stopWatch = new
Stopwatch();
            TimeSpan ts;
            int tempCnt = 0, totalCnt;
```

```
            List<string> newBases = new
List<string>();
            List<int> newBasesInd = new
List<int>();
            Dictionary<string, int> result =
new Dictionary<string, int>();
            var posMap = new
List<List<string>>();
            List<List<string>> sentences = new
List<List<string>>();
            bool end = true;
            stopWatch.Reset();
            totalCnt = 0;
            List<string> st = new
List<string>();
            if (Directory.Exists(dirName))
            {
                string[] files =
Directory.GetFiles(dirName, "*.txt",
SearchOption.TopDirectoryOnly);
                foreach (var file in files)
                    posMap.Add(new
List<string>());
                for (int sa = 0; sa <
files.Length; sa++)
                {
                    end = true;
                    tempCnt = 0;
                    totalCnt = 0;
                    using (StreamReader sr =
new StreamReader(files[sa],
System.Text.Encoding.Default))
                    {
                        for (int i = -1; i <
bases.Count; i++)

result.Add(i.ToString(), 0);


                        stopWatch.Reset();
                        stopWatch.Start();
                        while ((buff =
sr.ReadLine()) != null)
                        {
                            st =
Regex.Matches(buff.ToLower(), @"\w+")
   .Cast<Match>()
   .Select(x => x.Value)
   .ToList();


                            foreach (var t in
st)
                            {
                                totalCnt++;
                                var index =
str.SearchS(t);


                                if (index !=
"-1")
                                {

result[index.Remove(index.Length - 4)]++;

posMap[sa].Add(index.Substring(index.Length -
4));
                                }
                                else
                                {
                                    tempCnt++;


posMap[sa].Add("-1");
                                }
                            }
                        }
                        stopWatch.Stop();
                        ts =
stopWatch.Elapsed;
                        elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                            ts.Hours,
ts.Minutes, ts.Seconds,
                            ts.Milliseconds /
10);
                    }
```

```
Console.WriteLine("Recognition time № " + sa +
": " + elapsedTime);


                    int outValue;

                    using (var sw = new
StreamWriter(outDirName + sa + ".txt", false,
System.Text.Encoding.Default))

                    {

                        sw.WriteLine("unknown
" + tempCnt + " " + "total " + totalCnt + " "
+ (double)tempCnt / totalCnt);

                        for (int i = 0; i <
result.Count; i++)

                        {


result.TryGetValue(i.ToString(), out
outValue);

                            if (outValue != 0)


sw.WriteLine(outValue + " " + bases[i] + " " +
i.ToString());

                        }

                        for (int i = 0; i <
newBasesInd.Count; i++)

                        {


sw.WriteLine(newBasesInd[i] + " " +
newBases[i] + " " + (i + 388826).ToString());

                        }

                    }

                    using (StreamWriter sw =
new StreamWriter(outDirName + sa + "map" +
".txt", false, System.Text.Encoding.Default))

                    {

                        for (int i = 0; i <
posMap[sa].Count; i++)


sw.WriteLine(posMap[sa][i]);

                    }

                    newBases.Clear();

                    newBasesInd.Clear();

                    result.Clear();

                    st.Clear();

                }

                stopWatch.Reset();

                stopWatch.Start();

                Clusterize(posMap);

                stopWatch.Stop();

                ts = stopWatch.Elapsed;

                elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",

                    ts.Hours, ts.Minutes,
ts.Seconds,

                    ts.Milliseconds / 10);

                Console.WriteLine("Clusterize
time " + elapsedTime);

            }

            Console.ReadKey();

        }


        public static void
Clusterize(List<List<string>>
propertiesByText)

        {

            var fourthList = new
List<Dictionary<string, double>>();

            var topFourthList = new
List<Dictionary<string, double>>();

            var sortList = new List<double>();

            var euclideanDistances = new
List<List<double>>();

            var counterProp = 0;


            foreach (var properties in
propertiesByText)

            {

                fourthList.Add(new
Dictionary<string, double>());

                topFourthList.Add(new
Dictionary<string, double>());

                euclideanDistances.Add(new
List<double> { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
});
```

```
            }

            foreach (var properties in
propertiesByText)
            {
                for (int j = 0; j <
properties.Count - 1; j++)
                {
                    if
(fourthList[counterProp].ContainsKey(propertie
s[j] + properties[j + 1]))
                    {

fourthList[counterProp][properties[j] +
properties[j + 1]]++;
                    }
                    else

fourthList[counterProp].Add(properties[j] +
properties[j + 1], 1);
                }
                counterProp++;
            }

            for (int i = 0; i <
fourthList.Count; i++)
            {
                foreach (var key in
fourthList[i])
                {
                    sortList.Add(key.Value);
                    sortList.Sort();
                }

                foreach (var key in
fourthList[i])
                {
                    if (key.Value >
sortList[sortList.Count - sortList.Count / 5]
&& key.Key.Length == 8)
                    {
topFourthList[i].Add(key.Key, key.Value);
                    }
                }
                sortList.Clear();
            }

            for (int i = 0; i <
topFourthList.Count; i++)
            {
                foreach (var key in
topFourthList[i])
                {
                    for (int j = 0; j <
topFourthList.Count; j++)
                    {
                        if (i != j &&
!topFourthList[j].ContainsKey(key.Key))
                        {

topFourthList[j].Add(key.Key, 0);
                        }
                    }
                }
            }

            for (int i = 0; i <
topFourthList.Count; i++)
            {
                for (int k = 0; k <
topFourthList.Count; k++)
                {
                    foreach (var set in
topFourthList[i])
                    {

euclideanDistances[i][k] +=
Math.Pow(topFourthList[i][set.Key] /
propertiesByText[i].Count -
topFourthList[k][set.Key] /
propertiesByText[k].Count, 2);
                    }
```

```
                    euclideanDistances[i][k] =
Math.Sqrt(euclideanDistances[i][k]);
                }
            }
            Console.Write("\t");
            for (int t = 1; t <
euclideanDistances.Count + 1; t++)
                Console.Write(t + ")\t");
            Console.WriteLine();

            for (int i = 0; i <
euclideanDistances.Count; i++)
            {
                Console.Write(i + 1 + ")\t");
                foreach (var str in
euclideanDistances[i])

Console.Write(Math.Round(str, 5) + "\t");
                Console.WriteLine();
            }
            Maximin(euclideanDistances);
        }

        private static void
ClassesAllocation(out int ind1, out int ind2,
List<int> clusterCenters, List<int> points,
List<List<double>> distances)
        {
            double min = 100000.0, maximin =
0.0;
            int minInd1 = -1, minInd2 = -1;
            ind1 = 0;
            ind2 = 0;
            for (int i = 0; i < points.Count;
i++)
            {
                for (int j = 0; j <
clusterCenters.Count; j++)
                {
                    if
(!clusterCenters.Contains(i))
                    {
                        if
(distances[i][clusterCenters[j]] < min)
                        {
                            points[i] =
points[clusterCenters[j]];
                            min =
distances[i][clusterCenters[j]];
                            minInd1 = i;
                            minInd2 =
clusterCenters[j];
                        }
                    }
                }
                if (min > maximin &&
!clusterCenters.Contains(i))
                {
                    maximin = min;
                    ind1 = minInd1;
                    ind2 = minInd2;
                }
                min = 100000;
            }
        }

        public static void
Maximin(List<List<double>> distances)
        {
            var max = 0.0;
            int maxInd1 = -1, maxInd2 = -1,
minMaxInd1, minMaxInd2;
            var clusterCenters = new
List<int>();
            var temp = string.Empty;
            var points = new List<int>();

            for (int i = 0; i <
distances.Count; i++)
            {
```

```
            points.Add(0);
        }

        for (int i = 0; i < points.Count;
i++)
        {
            for (int j = 0; j <
points.Count; j++)
            {
                if (distances[i][j] > max)
                {
                    max = distances[i][j];
                    maxInd1 = i;
                    maxInd2 = j;
                }
            }
        }
        max = 0.0;
        clusterCenters.Add(maxInd1);
        clusterCenters.Add(maxInd2);
        points[maxInd1] = 0;
        points[maxInd2] = 1;

        do
        {
            ClassesAllocation(out
minMaxInd1, out minMaxInd2, clusterCenters,
points, distances);

            var centresDistancesSum = 0.0;
            var iter = 0;
            for (int i = 0; i <
clusterCenters.Count; i++)
            {
                for (int j = i + 1; j <
clusterCenters.Count; j++)
                {
                    centresDistancesSum +=
distances[clusterCenters[i]][clusterCenters[j]
];
                    iter++;
                }
            }
            centresDistancesSum =
centresDistancesSum / 2 / iter;
            if
(distances[minMaxInd1][minMaxInd2] >
centresDistancesSum)
            {

clusterCenters.Add(minMaxInd1);
                points[minMaxInd1] =
clusterCenters.Count - 1;
            }
            else
            {
                break;
            }
        } while (true);
    }

    static void Main(string[] args)
    {
        List<string> drops = new
List<string>();
        List<string> endingIndexes = new
List<string>();
        List<string> basicIndexes = new
List<string>();
        List<string> endings = new
List<string>();
        List<string> bases = new
List<string>();
        List<string> delimiters = new
List<string>();
        List<string> partOfSpeechIndexes =
new List<string>();
        List<string> uniquePropertyIndexes
= new List<string>();
```

```
            Stopwatch stopWatch = new
Stopwatch();

            TimeSpan ts;

            ManagedList ML = new
ManagedList();

            DictionaryAdapt ht = new
DictionaryAdapt();

            Trie tr = new Trie();

            string buff = "";



            using (StreamReader sr = new
StreamReader("endingIndexes.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    endingIndexes.Add(buff);

                }

            }

            using (StreamReader sr = new
StreamReader("basicIndexes.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    basicIndexes.Add(buff);

                }

            }

            using (StreamReader sr = new
StreamReader("switch.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    drops.Add(buff);

                }

            }

            using (StreamReader sr = new
StreamReader("partOfSpeechIndexes.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {


partOfSpeechIndexes.Add(buff);

                }

            }

            using (StreamReader sr = new
StreamReader("finEnding.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    endings.Add(buff);

                }

            }

            using (StreamReader sr = new
StreamReader("output.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    bases.Add(buff);

                }

            }

            using (StreamReader sr = new
StreamReader("delimiters.txt",
System.Text.Encoding.Default))

            {

                while ((buff = sr.ReadLine())
!= null)

                {

                    delimiters.Add(buff);

                }

            }
```

```
            using (StreamReader sr = new
StreamReader("uniquePropertyIndexes.txt",
System.Text.Encoding.Default))
            {
                while ((buff = sr.ReadLine())
!= null)
                {

uniquePropertyIndexes.Add(buff);
                }
            }
            var endInd =
CountEndingsPartOfSpeech(endingIndexes,
partOfSpeechIndexes, endings);
            string elapsedTime;


            stopWatch.Start();//insert start
List
            InsertProcessing(ML,
endingIndexes, basicIndexes, endings, drops,
uniquePropertyIndexes);
            stopWatch.Stop();
            ts = stopWatch.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes,
ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Insert runTime
List " + elapsedTime);


            stopWatch.Reset();
            stopWatch.Start();//insert start
Trie
            InsertM(tr, endingIndexes,
basicIndexes, endings, drops,
uniquePropertyIndexes);
            stopWatch.Stop();
            ts = stopWatch.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes,
ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Insert runTime
Trie " + elapsedTime);


            stopWatch.Reset();
            stopWatch.Start();//insert in Dict
            InsertProcessing(ht,
endingIndexes, basicIndexes, endings, drops,
uniquePropertyIndexes);
            stopWatch.Stop();
            ts = stopWatch.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes,
ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Insert runTime
Dictionary " + elapsedTime);
string dirName Console.ReadLine();
            string outDirName =
@"C:\Users\illym\Source\Repos\Cutting\Recognit
ion\bin\Debug\netcoreapp3.1\ResultsTrie\result
";
            stopWatch.Reset();
            stopWatch.Start();//search start
List
            SearchProcessing(bases,
outDirName, ML, dirName);
            stopWatch.Stop();
            ts = stopWatch.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes,
ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Search runTime
List " + elapsedTime);


            outDirName =
@"C:\Users\illym\Source\Repos\Cutting\Recognit
ion\bin\Debug\netcoreapp3.1\ResultsList\result
";
            stopWatch.Reset();
```

```
            stopWatch.Start();//search start
Trie
            SearchProcessing (bases,
outDirName, tr, dirName);
            stopWatch.Stop();
            ts = stopWatch.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes,
ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Search runTime
Trie " + elapsedTime);


            outDirName =
@"C:\Users\illym\Source\Repos\Cutting\Recognit
ion\bin\Debug\netcoreapp3.1\ResultsHT\result";
            stopWatch.Reset();
            stopWatch.Start();//search start
Dictionary
            SearchProcessing(bases,
outDirName, ht, delimiters);
            stopWatch.Stop();
            ts = stopWatch.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes,
ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Search runTime
Dictionary " + elapsedTime);
            Console.ReadLine();
        }
    }
```

Міністерство освіти і науки України
Дніпровський національний університет залізничного транспорту
імені академіка В. Лазаряна

ТЕЗИ

Всеукраїнської науково-технічної конференції молодих учених, магістрантів та студентів «Науково-технічний прогрес на транспорті»

(29 березня 2021 року)

Дніпро
2021

СЕКЦІЯ 3
БУДІВЕЛЬНА ІНЖЕНЕРІЯ ТА ЕКОЛОГІЧНА БЕЗПЕКА
CIVIL ENGINEERING AND ENVIRONMENTAL SAFETY

Formation of an electronic dictionary for the Ukrainian language for the tasks of establishing the authorship of texts

O. O. Kyrychenko Research supervisor: V. I. Shynkarenko, Candidate of Technical Sciences, Associate Professor Language supervisor: A. O. Muntian, Senior Lecturer Dnipro National University of Railway Transport named after Academician V. Lazarian

The article provides constructing a dictionary of truncated words (hereinafter - bases) according to the criterion of their coincidence for word forms of a single word. The development of this dictionary is caused by the need to divide the text into unambiguous semantic units 35 (words), search for their correspondences in the dictionary and grouping by semantic similarity, excluding the fact of the presence of homonyms in the text. The aim of the work is to research the effectiveness of the use of combined data structures to preserve and use the created dictionary of words of the Ukrainian language to establish the authorship of texts.

The dictionary is a set of bases and a set of their possible endings, by which you can uniquely identify the input word. The research work is to find a better organization of the data structure for storing and searching for words of the Ukrainian language according to the criteria of speed of matching the word from the text of the word from the dictionary, as well as the size of such a structure. The basic dictionary of VESUM as a means of NLP (Natural language processing) of text analysis was taken as the most complete dictionary of words of the Ukrainian language. Based on it, a database was created that contains only the bases of words and their endings, thanks to which it was possible to compress the final dictionary file more than 30 times. This result was achieved by cutting off unnecessary morphological information, searching for unique lists of endings with their subsequent numbering, comparing the obtained ending indices with the bases to which they belong. In the process of vocabulary formation, the problem of alternation of vowel sounds within a word was considered, which negatively affected the clipping of endings and subsequent identification of the word in the dictionary.

The dictionary developed during the execution of the program is transformed into a certain data structure in RAM. The primary criterion in this work is the search for the index by comparing the word from the text of the word from the dictionary. Filling the data structure has little effect on the performance of the system due to the determinacy of the input data, as the dictionary is not updated with new objects during the analysis of input texts. Thus, the insertion operation has a predetermined execution time, moreover, relatively insignificant when parsing large texts. Preliminary testing yielded the following results: text files with a total size of 2.5 GB were processed in 612 seconds, while filling the dictionary took only 17 seconds.

The developed data structure is based on Trie, which is a common search tool, the best application of which is a search engine on the Internet. Their principle is applied in the constructed tree in which knowledge or even signs (for example, deficit of a point) which are dictionaries are created. To prevent intermediate errors of the second kind when comparing a word with a possible basis, the organization of search in the dictionary should be arranged so that the largest number of characters match when checking, and since the dictionary is sorted alphabetically, to achieve this is simple: search in reverse order. Thus, you can reduce the number of checks of sets of endings when comparing words with the basics and significantly increase performance. At the output of the program, you will get the frequency response of many words in the text that are semantically related.

At present, work has been done to compress the initial dictionary without losing key information for its operation. A data structure has been developed, as well as comparative performance tests on different volumes of data with template data structures. The next step will be to develop an algorithm for comparing the elements of the logical data structure of a set of physical structures.

Кластеризація текстів за приналежністю до автора на основі словнику атрибутів

Кириченко О. О., Шинкаренко В.І. Дніпровський національний університет
залізничного транспорту ім. акад. В. Лазаряна, Україна

В роботі досліджується розподілення вхідних текстів за авторами на основі попередньо опрацьованих творів завдяки співставленню векторів атрибутів цих текстів. Дана кластеризація передбачає оцінку схожості отриманих векторів для множини текстів та розподілення їх за авторами з аналізом отриманих даних. Актуальність такого підходу викликана необхідністю виявлення плагіату в електронних джерелах з метою захисту авторських прав. Мета роботи – розробка механізму встановлення авторства тексту шляхом вибору найближчої серед альтернатив структури письма автора, дослідження схожості текстових масивів даних за їх атрибутивними послідовностями у вигляді морфологічними даних слів.

Для формування векторів морфологічних атрибутів слів першочергово необхідно розділити вхідні тексти на множину слів та розпізнати всі їх входження. Для цього було розроблено комбіновану структуру даних на основі словнику, що в свою чергу є частковим випадком хеш-таблиці для наперед визначених типів даних – масивів символів. Така структура містить всі унікальні основи слів, що в свою чергу були взяті з відкритого словнику ВЕСУМ. Крім того застосовані й вибіркові атрибути слів з цього ж словнику, що обиралися за принципом найбільшого покриття значимих атрибутів української мови: частин мови, відмінків для іменників, роду, числа і т. д. Випробування на множині з 9 тестових творів української літератури дали в середньому покриття у 97.5% серед слів. Аналізуючи відсутні у словнику слова можна простежити, що більшу частину з них являють авторські слова, жаргонізми або застарілі слова, що більше не застосовується.

Наступним кроком є заміна слів їх атрибутивними представниками, що в свою чергу є індексами, сформованими попередньо з множини унікальних послідовностей атрибутів. Отримані масиви атрибутів розбиваються на пари почергово, перший елемент з наступним за ним, повтори залишаються. Таким чином з тексту розміром n слів формується n-1 пара атрибутів. Не знайдені у словнику слова також потрапляють у пари, однак в подальшому аналізі не враховуються. Для кожного тексту формується свій вектор атрибутів, у який потрапляють і записи сформованих пар інших текстів вибірки. Серед сформованих пар обирається 20% значимих, кількість яких найбільша. Далі кількість зустрічей атрибутів нормалізується з урахуванням кількості слів для кожного тексту окремо, щоб співставити відсоткові частки зустрічі атрибутів поміж всіх текстів та виконати кластеризацію за авторами. Останнім кроком є безпосереднє співставлення всіх векторів та вираження відстаней між ними через скалярне значення. За отриманими відстанями можна провести кластеризацію. Застосовано алгоритм максимінної кластеризації відстаней.

На виході програми буде сформовано кластери творів за авторами.

На разі проведена робота з формування словнику слів з їх атрибутами, розпізнавання слів з поданих текстів, утворення нормалізованих векторів з пар атрибутів та їх співставлення. Отриманий алгоритм опирається на морфологічні атрибути слів і аналізує звички побудови структури тексту автором. Для покращення кластеризації необхідно дослідити вплив різних атрибутів на процес кластеризації, а також дослідити природу розподілення утворених кластерів. Крім того, можна проаналізувати та утворити особистий словник унікальних слів автора, що не були віднайдені у сучасному словнику українських слів, та на його основі виконувати додаткову оцінку приналежності тексту його авторству. Результати роботи можуть бути використані для розподілення текстів за авторами.

Подана до публікації

Processing Words Effectiveness Analysis in Solving the Natural Language Texts Authorship Determination Task

Inna Demidovich, Viktor Shynkarenko, Olena Kuropiatnyk, Oleksandr Kirichenko
Department of Computer Information Technology
Dnipro National University of Railway Transport
named after academician V. Lazaryan
Dnipro, Ukraine
2019demidovichinn@gmail.com, shinkarenko_vi@ua.fm, olena.kuropiatnyk@gmail.com, illyminchik@gmail.com

Abstract— **The previously developed method establishes the natural language texts authorship based on frequency analysis, supplemented by indicators of text complexity and recurrent analysis. The authorship indication problem is reduced to the pattern recognition classical theory. To account for the different individual indicators information content, their weights are taken into account. They are determined according to the maximum number of the correctly established texts authorship from the training sample using a genetic algorithm. This method is used to study the effectiveness of the author's style representation that is based on different types of words processing: two types of words stems and 4-grams. To obtain stems, the adapted Porter stemmer is used and creating a dictionary of the foundations of the Ukrainian language original method is applied, respectively. Taking into account the calculated indicators weights, the reliability of establishing the text authorship in the control sample reached 88-91%.**

Keywords— natural language texts, authorship attribution, Porter stemmer, genetic algorithm, recurrent analysis, statistical analysis, text classification, dictionary, pattern recognition

I. Introduction

Authorship identification is a practical task of finding the most likely text author. However, the modern methods possibilities of the text authorship determining with various orientations and styles are still limited, and the results do not always correspond to reality. The determination of the text authorship is still an unsolved problem for many areas of human activity, such as education, jurisprudence, literary criticism, history, etc.

The most texts authorship is easy to recognize due to the fact that they are signed. However, there are situations in which the authorship of the text is either not known or is in doubt. In this work, we strive to create a convenient tool for the effective attribution of texts. This task is interdisciplinary and closely related to the natural structure of language.

There the comparison effectiveness of the author's style representation using various words processing methods is made to form the author's frequency profile with its subsequent use in the presented method for determining authorship of natural language texts.

II. Existing Research

There are a number of studies carried out in order to identify the most reliable method or tool for determining the text authorship of various styles in different fields; some are presented in the works [1-11].

Now most of the methods for determining the text authorship can be divided into two broad thrusts: the use of statistical methods [5, 2, 7,] and the use of machine learning [1, 3, 4, 6].

The attribution result is in the range from 74% to 92% of correctly identified cases [1-11]. These results varied depending on the used method, the language and the analyzed text style.

From the authorship determining accuracy point of view, today the most effective methods are based on analysis: frequency [7], N-gram [8, 9], words and morphemes [10, 11]. This analysis suggests that the use of these techniques makes it possible to most accurately reflect the author's syllable and style.

The Ukrainian language is characterized by great word forms variety and the order of their use in sentences. Due to this, certain patterns in the use of certain phrases and service parts of speech will make it possible to display the particular author narrative language characteristic, regardless of the subject and the text stylistic direction.

III. Research Means and Methods

To determine the natural language text authorship and comprehensive author's style study, indicators of several methods are used: frequency, the complexity of text perception, and modified recurrent analysis. On the basis of the obtained indicators, the text vector-image is constructed.

Various methods of words processing have been applied: splitting the text into a list of 4-grams used and word stems with the calculation of their use frequency for further analysis.

Consider all the tools and methods are used in the work.

A. Preparing the experiment

1) Presentation of text based on 4-gram

The text analysis method using N-gram is a relatively new method and in most cases is used to plagiarism detection in various text sources. This method also shows good results in determining the authorship of texts [12, 13, 14].

Text analyzing based on an N-gram, the length of characters' sequence is established, this frequency is

subsequently used to determine authorship. The method accuracy depends on the character sequence length. For the author, a frequency profile of the various N-grams usage is built, which makes it possible to display the features of his language. According to previous studies, the analysis of texts using 4-gram showed the greatest reliability [15, 16].

In this work, the splitting of words is performed with an overlap - each next 4-gram differs from the previous one by only one character. All punctuation marks, case letters and various formatting elements are not taken into account during forming 4-gram.

For example, for the phrase «Він йшов до мене» when it is divided into 4-grams, we get the following list: Він, ійшо, йшов, шовд, овдо, вдом, доме, омен, мене.

This approach allows you to analyze not only the words used in the author's speech, but also to reflect a certain extent reflects their sequence.

2) Used dictionaries

For the experiment in this work, two different dictionaries were created. The first dictionary (based on VESUM dictionary) was public the Large Electronic Dictionary of Ukrainian (VESUM) [17].

On its basis, a complex dictionary was built containing unique word stems, their endings and prefixes. To reduce its size, we pre-selected unique endings lists and assigned only an index from it to the word stem. Maintaining a vowel list in words is also supported.

To create lists of prefixes, the formed dictionary was analyzed for the presence of stems that differ only with the presence of a prefix by a simple search. As a result, the original vocabulary of stems was reduced - all key stems were assigned corresponding indexes from the list of prefixes, and unnecessary ones with prefixes were removed.

The advantage of the resulting dictionary is its support for taking into account all word forms for the stems, each of which will be assigned a unique index. Thus, in all cases, various forms of words, as well as words obtained by adding a prefix will unmistakably lead to a single stem.

For the second dictionary (based on text corpus dictionary), texts of various styles were used: fiction, journalism, scientific texts, official business documents and texts using colloquial vocabulary. Each dictionary consists of three parts.

The first includes a stems list of various words, the second one - endings and the third one - prefixes. Each stem corresponds to a list of all possible endings and prefixes for the word. The alternation of vowels in the word is also displayed in the process of changing its form.

The list of stems in the dictionary file is alphabetical, followed by three numeric elements. The first one is the line number from the file with endings, which lists all of them that can be added to the given stem. If using some ending requires an alternation of vowels in the stem, then there is a "+" symbol in front of it. The second place is taken by the number of the letter in the stem from its beginning, which takes part in the alternation, and the vowel for replacement. The third element is the line number from the file with prefixes specific to this stem. If any of these elements is missing, then it is replaced by "-".

An example of the stems:

- «важлив 0 - 20» (it has endings, hasn't alternation and also has prefixes);
- «абсолютніст 41 8о -» (it has endings, the 8th letter in it changes to о in some forms, it hasn't prefixes);
- «полудні - - 7» (it hasn't endings, alternations and prefixes).

Example endings for the above written stems (for number 1 and 2):

- «а, е, ий, им, ими, их, і, й, ім, ого, ої, ому, ою, у»;
- «ь, ю+е+и+і».

Example endings for the above written stems (for number 1 and 3):

- «архі, гіпер, мега, над, супер, ультра»;
- «о».

In further, the data of the dictionary was used to calculate the frequency of particular stem occurrence in the author's style and also to compile his own language profile.

Dictionary coverage ranges from 93% to 96% of words in every text.

3) Adapted Porter stemmer

Also, Porter's stemmer adapted to the Ukrainian language was applied [18, 19] to work directly with the various authors' texts and also to construct a profile of the using various stems frequency, characteristic for each author.

The stemming algorithm consistently applies some number of postfixes, endings, and suffixes cutoff rules, without using stems. The input is a list of word and morphemes classes for different parts of speech. The algorithm implemented in this work consists of the following steps:

1 word preprocessing (converting to lower case, replacing the apostrophe and "r");

2 checking if the word is infinitive, if yes - the algorithm terminates, otherwise - go to the next step.

3 separating a part of a word after the first vowel (RV);

4 removal of morphemes from RV (suffix, ending, postfix). If it was not possible to delete the certain part of the speech morpheme, then the algorithm proceeds to delete the morpheme of the next part in this order:

   a. verbs (with postfixes – ся, сь);
   b. adjectives and participles;
   c. verbs;
   d. nouns;

5 removal from RV letter "і";

6 removal word-forming morphemes from RV (sequence of 1-3 vowels at the beginning, ending with "-ість ");

7 removal from RV doubling of consonant "нн" and soft sign;

8 recreating a word from a part up to and including the first vowel and RV - stemma formation.

In this work, this algorithm is implemented in C # using the regular expression mechanism. Expressions describe the classes of morphemes for different parts of speech and their different forms:

- gerunds: ив, ивши, ившись;
- infinitives: ти, учи, ячи, вши, ши, ати, яти, ючи, ячи;
- returnable postfixes: ся, сь, си;
- adjectives: ими, ій, ий, а, е, ова, ове, ів, є, їй, єє, ее, я, ім, ем, им, ім, их, іх, ою, йми, іми, у, ю, ого, ому, ої;
- participles: ий, ого, ому, им, ім, а, ій, у, ою, ій, і, их, йми, их;
- verbs: сь, ся, ив, ать, ять, у, ю, ав, али, учи, ячи, вши, ши, е, ме, ати, яти, є;
- nouns: а, ев, ов, е, ями, ами, еи, и, ей, ой, ий, й, иям, ям, ием, ем, ам, ом, о, у, ах, иях, ях, ы, ь, ию, ью, ю, ия, ья, я, і, ові, ї, ею, єю, ою, є, еві, ем, єм, ів, їв, 'ю;
- word-forming parts.

4) Adapted recurrent analysis

This type of analysis was modified by us for its application in the processing of natural language texts. It is based on the quantitative analysis of recurrence plots used by Zbilut J. P. and Webber Jr. C. [20].

The text is converted to a time series. The value of each point in the series is the occurrence frequency of the N-gram, stem, or word stem, and the progress to the next corresponding element is taken as a unit of time. The use of recurrent analysis takes into account the internal structure of the text and the style of narration, which is characteristic exclusively for the author under study [23]. The process is described with more details in our works [15, 16].

Based on the resulting series, a phase space is built to display changes in state, as well as a recurrent plot that visually displays repetitions of certain words and even phrases in the text under study. The recurrent plot is used to calculate the indicators of recurrent analysis [15, 16].

5) Text perception complexity indicators

The work used text perception complexity indicators: the number of sentences, words, syllables and letters in the text, as well as the average number of words, syllables, letters in sentences and words. Initially, this type of analysis was used to study the English text and determine its perception complexity [21], but it also can be applied to other languages. The applied indicators are given in [15, 16].

6) Genetic algorithm

In order to increase the results reliability, in view of working with a large number of indicators in pattern recognition, weights were used for each parameter. Forming the value of the weights, the calculations were carried out using a genetic algorithm as in [16].

IV. Experiments to Determine the Text Authorship

A. Organization of the experiment

To determine the text authorship, the method of recognition based on the minimum distance to the standard is used [16, 22].

Each text is associated with its image, which includes all previously obtained data in the presented sequence: 4-gram frequencies or word stems; recurrent indicators; the text complexity indicators [16].

To conduct the experiment, 20 works of fiction by 11 Ukrainian authors were selected in the training sample.

The works of the following authors are presented: IB – I. Bahrianyi, AV – A. Vyshnia, MV – M. Vovchok, AD – A. Dovzhenko, MK – M. Kotsiubynskyi, HK – H. Kvitka-Osnovianenko, PM – P. Myrnbi, VN – V. Nestaiko, VP – V. Pidmohylnyi, IF – I. Franko, MK – M. Khvylovyi.

The choice of fiction is due to the reliable information about the authorship of the works availability and the characteristic syllable presence of each author. The control sample included 3 works by the same authors.

B. Experimental results

Table 1 shows the results of an experiment to determine the text authorship from the control sample.

TABLE I. Results of Using Different Word Processing Methods for Determining the Text Authorship

Author	4-gram	Based on VESUM dictionary	Based on text corpus dictionary	Stems
IB	IB	IB	IB	IB
IB	IB	IF	IB	IB
IB	IB	IB	IB	IB
AV	AV	AV	AV	AV
AV	AV	AV	AV	AV
AV	AV	AV	AV	AV
MV	MV	MV	AV	MV
MV	MV	MV	MV	MV
MV	MV	MV	MV	MV
AD	AD	AD	AD	AD
AD	AD	AD	AD	AD
AD	AD	AD	AD	AD
MK	MK	HK	MK	MK
MK	MK	MK	HK	MK
MK	MK	MK	MK	MK
HK	HK	HK	HK	AD
HK	HK	MK	HK	HK
HK	MK	HK	HK	MK
PM	PM	PM	PM	PM
PM	PM	PM	PM	PM
PM	PM	PM	PM	IB
VN	VN	VN	VN	VN
VN	VN	VN	VN	VN
VN	VN	VN	MV	VN
VP	VP	VP	VP	VP
VP	VP	IB	VP	VP
VP	VP	VP	VP	VP

Author	4-gram	Based on VESUM dictionary	Based on text corpus dictionary	Stems
IF	IF	IF	IF	IF
IF	IB	IF	IF	IF
IF	IB	IF	IB	IB
MK	MK	MK	IF	IF
MK	MK	MK	MK	MK
MK	MK	MK	MK	MK

At the intersection of the row with information about t author and the column with the name of the used method, the author is indicated, determined by the developed software. If the authorship of the studied text is determined correctly, then the cell is shaded.

The best result was obtained working with 4-grams - 91% of text authorship matches (30 out of 33). The results of working with word stem dictionaries and the stemming result were 88% correctly identified cases (29 out of 30).

CONCLUSIONS

According to the presented analysis data, all the studied methods gave a positive result and can be used to determine the authorship of texts. The best indicator - 91% of text authorship coincidences - was obtained working with 4 grams. Working with the words stems using dictionaries and stemming gave a result of 88%, which may not be so much a drawback of the method itself as a tolerance. In the future, it is planned to increase the size of the control sample for more accurate results.

Comparing the two approaches to working with word stems, stemming can be preferred. This method is faster and more convenient in calculations. Compilation and subsequent work with dictionaries take more time and resources, but the tolerance of the both methods is the same.

Since work with word stems does not provide data on the structure of phrases used by the author, in future works it is planned to build their models characteristic of different authors. This approach can more fully reflect the author's style, which will improve the results

REFERENCES

[1] R. Iyer, C. Rosé, A Machine Learning Framework for Authorship Identification FromTexts. ArXiv abs/1912.10204, 2019.

[2] R.A.Hardcastle, CUSUM: a credible method for the determination of authorship? Science & Justice : Journal of the Forensic Science Society, 37(2), 1997, pp. 129-138. doi: 10.1016/s1355-0306(97)72158-0

[3] J.Rygl, A. Horák, Authorship Attribution: Comparison of Single-Layer and Double-Layer Machine Learning, in: Sojka P., Horak A., Kopeček I., Pala K. (eds) Text, Speech and Dialogue. TSD 2012. Lecture Notes in Computer Science, vol 7499. Springer, Berlin, Heidelberg, 2012, pp 282-289. https://doi.org/10.1007/978-3-642-32790-2_34

[4] M. Lupei, A. Mitsa, V. Repariuk, V. Sharkan, Identification of authorship of Ukrainian-language texts of journalistic style using neural networks. Eastern-European Journal of Enterprise Technologies, 1(2), 2020, pp. 30–36. https://doi.org/10.15587/1729-4061.2020.195041

[5] V. Lytvyn et al. Development of the Quantitative Method for Automated Text Content Authorship Attribution Based on the Statistical Analysis of N-grams Distribution. Eastern-European Journal of Enterprise Technologies, 6 (2), 2019, pp. 28-51. doi:10.15587/1729-4061.2019.186834.

[6] V. Moshkina, I. Andreeva, N. Yarushkinaa, Solving the problem of determining the author of text data using a combined assessment. CEUR Workshop Proceedings 2782, 2020, pp. 112-118

[7] I. I. Drozdova, A. D. Obukhova, Opredelenie avtorstva teksta po chastotnyim harakteristikam (determining the authorship of the text by frequency characteristics), in: Tehnicheskie nauki v Rossii i za rubezhom: materialyi VII Mezhdunarodnoy nauchnoy konferentsii, Buki-Vedi, Moskva, 2017, pp. 18-21.

[8] Yunita Sari, Andreas Vlachos, Mark Stevenson, Continuous N-gram Representations for Authorship Attribution, in: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers, Valencia, Spain, 2017, pp.267-273. doi: 10.18653/v1/E17-2043.

[9] I. Markov, J. Baptista, O. Pichardo-Lagunas, Authorship Attribution in Portuguese Using Character N-grams. Acta Polytechnica Hungarica 14(3), 2017, pp. 59–78. doi: 10.12700/APH.14.3.2017.3.4.

[10] D. L. Hoover, Frequent word sequences and statistical stylistics. Literary and Linguistic Computing 17 (2), 2002, pp. 157-180. doi: 10.1093/llc/17.2.157.

[11] G. O. Sidorov, Automatic Authorship Attribution Using Syllables as Classification Features, Rhema journal 1, 2018, pp. 62-81.

[12] H. Gómez-Adorno, JP. Posadas-Durán, G. Sidorov, Document embeddings learned on various types of n-grams for cross-topic authorship attribution. Computing 100, 2018, pp. 741–756. doi: 10.1007/s00607-018-0587-8

[13] O. Marchenko, A. Anisimov, A. Nykonenko, T. Rossada, E. Melnikov, Authorship attribution system. Artificial Intelligence 2, 2016, pp. 77-85.

[14] G. Wimmer, G. Altmann, L. Hřebíček, S. Ondrejovič, S. Wimmerová, Uvod do analyzy textov, Univerzita Komenskeho v Bratislave, Bratislava, 2003.

[15] V.I. Shynkarenko, I.M. Demidovich Determination of the attributes of authorship of natural texts. Artificial Intelligence 3, 2018, pp. 27-35.

[16] V.I. Shynkarenko, I.M. Demidovich Authorship Determination of Natural Language Texts by Several Classes of Indicators with Customizable Weights, in: Proceedings of the 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2021). Volume I: Main Conference. Lviv, Ukraine, April 22-23, 2021, pp. 832-844.

[17] Great electronic dictionary of Ukrainian language (VESUM). URL: https://github.com/brown-uk/dict_uk.

[18] T. V. Golub, M. Yu. Tyagunova, Method of steaming Ukrainian-language texts for classification of documents based on Porter's algorithm. Scientific works of Donetsk National Technical University. Series: Informatics, cybernetics and computer engineering No 1(24), 2017, pp. 59–63.

[19] A. Hlybovets, V. Tochytsky, Algorithm of tokenization and steming for texts in Ukrainian, 2017.

[20] J. P. Zbilut, C. L. Webber Jr., Embeddings and delays as derived from quantification of recurrence plots. Physics Letters A 171 (3-4), 1992, pp. 199–203. doi: 10.1016/0375-9601(92)90426-M

[21] Yu. V. Rohushyna, Ispolzovanie kriteriev otsenki udobochitaemosti teksta dlia poiska informatii, sootvetstvuiushchei realnym potrebnostiam polzovatelia (the usage of criteria for evaluating the readability of the text to find information that meets the real needs of the user.). Problemy prohramiuvannia 3, 2007, pp. 76-88.

[22] P. S. Sisodia, V. Tiwari, A. Kumar, A Comparative Analysis of Remote Sensing Image Classification Techniques, in: International Conference on Advances in Computing, Communications and Informatics (ICACCI), IEEE, Delhi, 2014, pp. 1418-1421. doi: 10.1109/ICACCI.2014.6968245.

[23] V. Shynkarenko, O. Kuropiatnyk, Constructive Model of the Natural Language. Acta Cybernetica 23 (4), 2018, pp. 995–1015. doi: 10.14232/actacyb.23.4.2018.2.