ДВНЗ «Донецький національний технічний університет»
Факультет комп'ютерно-інформаційних технологій та автоматизації
(повне найменування інституту, назва факультету)
Кафедра прикладної математики та інформатики
(повна назва кафедри)

«До захисту допущено»
Зав. кафедри ПМІ

_____ Наталія МАСЛОВА
(підпис) (ім'я та прізвище)

"___" _____ 20___ р.

Випускна кваліфікаційна робота

бакалавра
(освітній ступінь)

на тему: Емулятор абстрактної обчислювальної машини нормальних алгоритмів Маркова

зі спецчастиною: Розробка online-емулятора для створення аналітичних моделей алгоритмів із використанням нормальних алгоритмів Маркова

Виконав: студент 4 курсу, групи ІПЗ-20

спеціальності 121 Інженерія програмного забезпечення

освітньої програми 121 Інженерія програмного забезпечення

Юрчак Микита Олегович
(прізвище та ініціали) (підпис)

Керівник проф. кафедри ПМІ, д.т.н., проф. Євген БАШКОВ
(посада, науковий ступінь, вчене звання, ім'я та прізвище) (підпис)

Рецензент проф. каф. ПЗКС НТУ "Дніпровська політехніка", д.т.н., доц. Іван Лактіонов
(посада, науковий ступінь, вчене звання, ім'я та прізвище) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Луцьк – 2024

ДВНЗ «Донецький національний технічний університет»

Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики та інформатики
Освітній ступінь бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітня програма 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Зав. кафедри ПМІ
_____/Н. Маслова/
«____»_____2024 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТУ

Юрчаку Микиті Олеговичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Емулятор абстрактної обчислювальної машини нормальних алгоритмів Маркова»

2. Спецчастина «Розробка online-емулятора для створення аналітичних моделей алгоритмів із використанням нормальних алгоритмів Маркова»

Керівник роботи Євген БАШКОВ, д.т.н., проф., проф. каф. ПМІ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом від «25» квітня 2024 року № 164

3. Строк подання студентом роботи «3» червня 2024 року

4. Вихідні дані до роботи Літературні джерела по нормальним алгоритмах Маркова та відповідним емуляторам. Online-емулятор для створення аналітичних моделей алгоритмів із використанням нормальних алгоритмів

5. Зміст розрахунково-пояснювальної записки:

1) аналіз предметної області та постановка задачі;
2) огляд та порівняння існуючих емуляторів нормальних алгоритмів;
3) розробка емулятору абстрактної обчислювальної машини нормальних алгоритмів Маркова;
4) порівняльний аналіз розробленого емулятора з існуючими аналогами;
5) тестування розробленого програмного модуля.
6) інструкції з використання розробленого емулятора;
7) розміщення програми в мережі Інтернет;
8) порівняльний аналіз розробленого емулятора з існуючими аналогами;
9) висновки.

6. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) робота містить 3 таблиці, 41 рисунок, 8 формул, 12 слайдів презентації та інший графічний матеріал, у кількості, достатній для демонстрації сутності роботи.

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтролер	доц. Назарова І.А.		

8. Дата видачі завдання 24 квітня 2024 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Прим.
1	Об'єктний аналіз предметної області і постановка завдання	05.05.2024	
2	Огляд літератури з предметної області	10.05.2024	
3	Обґрунтування структури додатку та інструментарію для розробки	15.05.2024	
4	Розробка структури програмного модулю, вибір даних для обробки та експериментів.	20.05.2024	
5	Тестування системи	26.05.2024	
6	Оформлення пояснювальної записки	02.06.2024	
7	Нормоконтроль пояснювальної записки, її перевірка антиплагіатною системою	12.06.2024	
8	Оформлення супутніх матеріалів (розробка графічного матеріалу, тощо) та рецензування роботи	14.06.2024	
9	Написання доповіді	14.06.2024	
10	Захист роботи	26.06.2024	

Студент _____ Микита ЮРЧАК
(підпис) (ім'я, прізвище)

Керівник роботи _____ Євген БАШКОВ
(ім'я, прізвище)

АНОТАЦІЯ

Юрчак Микита Олегович. Емулятор абстрактної обчислювальної машини нормальних алгоритмів Маркова / Випускна кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» за спеціальністю 121 «Інженерія програмного забезпечення». ДВНЗ ДонНТУ, Луцьк, 2024

Об’єктом дослідження є процеси виконання нормальних алгоритмів Маркова та їх емуляції.

Предметом дослідження є методи створення та функціонування емуляторів для нормальних алгоритмів Маркова.

Метою роботи є створення online-емулятора аналітичних моделей алгоритмів із використанням нормальних алгоритмів.

Виходячи з мети, завданням даної дипломної роботи є аналіз теоретичних основ нормальних алгоритмів Маркова та їхніх емуляторів з метою визначення їх ефективності, переваг та недоліків; розробка online-емулятора для НАМ; розроблення програмного модуля, використовуючи мову JavaScript.

Проведені дослідження в даній роботі базуються на сучасних методах аналізу та розробки програмного забезпечення, використанні веб-технологій для створення емулятора, а також на теоретичних знаннях з області формальних мов та їхніх застосувань.

Практичне значення полягає в авторській розробці методу створення онлайн-емулятора для нормальних алгоритмів, що відрізняється зручністю та ефективністю.

Дипломна робота складається з 6 розділів, включає 41 рисунок, 3 таблиці, 22 джерела інформації, загалом 90 сторінок.

Ключові слова: Нормальні алгоритми Маркова, емулятори, формальні мови , аналіз алгоритмів.

ABSTRACT

Mykyta Yurchak. Emulator of abstract computing machine for normal Markov algorithms / Bachelor's thesis for the degree of "Bachelor" in the specialty 121 "Software Engineering". – SHEE DonNTU, Lutsk, 2024

The object of the research is the processes of normal Markov algorithms and their emulators.

The subject of the research is the methods of creation and functioning of emulators for normal Markov algorithms.

The purpose of the work is to create an online emulator for creating analytical models of algorithms using normal Markov algorithms.

Based on the purpose, the tasks of this diploma work are the analysis of the theoretical foundations of normal Markov algorithms and their emulators to determine their effectiveness, advantages, and disadvantages; development of an online emulator for normal Markov algorithms; development of a software module using JavaScript.

The research conducted in this work is based on modern methods of analysis and software development, the use of web technologies for emulator creation, as well as theoretical knowledge in the field of formal languages and their applications.

The practical significance lies in the author's development of a method for creating an efficient and user-friendly online emulator for normal Markov algorithms.

The thesis consists of 6 chapters, includes 38 figures, 3 tables, 22 sources of information, totaling 90 pages.

Keywords: Normal Markov algorithms, emulators, formal languages, algorithm analysis.

ЗМІСТ

ВСТУП 9
1 ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО НОРМАЛЬНІ АЛГОРИТМИ МАРКОВА 10
1.1 Формалізація правил та визначень для нормальних алгоритмів 10
1.2 Приклади та тестування нормальних алгоритмів 14
2 АНАЛІЗ ІСНУЮЧИХ ЕМУЛЯТОРІВ НОРМАЛЬНИХ АЛГОРИТМІВ . 17
2.1 Мобільний застосунок 17
2.2 Веб-емулятор 20
2.3 Десктопний додаток 23
2.4 Переваги та недоліки розглянутих емуляторів 25
3 РОЗРОБКА ФУНКЦІОНАЛЬНОЇ ЧАСТИНИ ОНЛАЙН-ЕМУЛЯТОРА ДЛЯ НОРМАЛЬНИХ АЛГОРИТМІВ МАРКОВА 27
3.1 Загальний опис алгоритму 27
3.1.1 Ітеративна реалізація алгоритму 27
3.1.2 Рекурсивна реалізація алгоритму 29
3.1.3 Порівняльний аналіз рекурсивного та ітеративного підходів 31
3.2 Реалізація основного модуля використовуючи мову програмування JavaScript 33
3.3 Запобігання зациклюванню 36
3.4 Оцінка складності алгоритмів 39
4 РОЗРОБКА ІНТЕРФЕЙСУ КОРИСТУВАЧА 47
4.1 Діаграма варіантів використання 47
4.2 Інтерактивна таблиця правил підстановок 49
4.3 Робочий рядок та елементи контролю виконання алгоритму 50
4.4 Побудова графіка складності алгоритму 52
4.5 Функції експорту та імпорту файлів для НАМ 53
4.6 Логування 55

4.7 Теоретична довідка та інструкція користувача 56
5 РОЗМІЩЕННЯ ПРОГРАМИ В МЕРЕЖІ ІНТЕРНЕТ 58
6 ПОРІВНЯЛЬНИЙ АНАЛІЗ РОЗРОБЛЕНОГО ЕМУЛЯТОРА З ІСНУЮЧИМИ АНАЛОГАМИ 60
ВИСНОВКИ 63
ПЕРЕЛІК ПОСИЛАНЬ 65
Додаток А 67
Зауваження нормоконтролера 67
Додаток Б 68
Лістинг програмного застосунку 68

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

НАМ – Нормальний алгоритм Маркова

МП – Марківська підстановка

МТ – Машина Тюрінга

JS – JavaScript, мова програмування.

HTML – Hypertext Markup Language, мова розмітки гіпертексту.

CSS – Cascading Style Sheets, каскадні таблиці стилів.

JSON – JavaScript Object Notation, формат обміну даними.

GitHub – веб-сервіс для хостингу IT-проєктів та їх спільної розробки.

Алфавіт – сукупність символів, які використовуються для складання слів в алгоритмах.

Термінальне правило – правило, яке завершує виконання алгоритму.

Ітерація – один крок виконання циклу.

Рекурсія – метод програмування, при якому функція викликає сама себе.

Стек – структура даних, яка працює за принципом "останній прийшов – перший пішов".

Регресійний аналіз – статистичний метод для визначення залежностей між змінними.

Big O – нотація, що описує асимптотичну поведінку алгоритмів.

Емпіричні дані – дані, отримані шляхом спостережень або експериментів.

Алгоритмічна складність – характеристика, що описує обчислювальні витрати алгоритму (час, пам'ять).

Клієнт – в даному контексті, веб-браузер, що виконує JavaScript код.

Сервер – комп'ютер, що надає послуги іншим комп'ютерам в мережі.

Асимптотична поведінка – поведінка алгоритму при великих обсягах вхідних даних.

ВСТУП

Розвиток області обчислювальних наук завжди був пов'язаний з пошуком нових методів моделювання та вивчення обчислювальних процесів. Одним із важливих напрямків в цьому контексті є дослідження нормальних алгоритмів Маркова.

Аналіз нормальних алгоритмів Маркова розширює розуміння обчислювальної складності, та відкриває нові можливості для вирішення алгоритмічних проблем і формалізації обчислювальних процесів. Визначення алгоритмів у нормальному вигляді показує суттєві спільні риси з численням, і це є корисним у випадках, коли поняття числення в досліджуваному розділі математики або кібернетики застосовують, як, приміром, в математичній логіці або в математичній лінгвістиці [1].

В даному контексті існування емулятора для нормальних алгоритмів Маркова виявляється корисним у забезпеченні ефективності та зручності у вивченні та аналізі цих алгоритмів. Завдяки емулятору з’являється можливість автоматизувати процес експериментів із моделями алгоритмів, тестування їх на різних вхідних даних, а також аналізу їхньої поведінки.

Мета роботи полягає в дослідженні теоретичних основ нормальних алгоритмів Маркова та розробці online-емулятора для створення аналітичних моделей алгоритмів із використанням НАМ.

Завдання роботи полягає у:

–детальному вивченні понять, прикладів та властивостей НАМ з метою створення теоретичної бази для розробки емулятора;

–аналізі доступних онлайн та офлайн інструментів для емуляції абстрактних обчислювальних машин на базі НАМ;

–розробці архітектури емулятора та його реалізації;

–проведенні тестування емулятора та оцінці його ефективності;

–порівняльному аналізі розробленого емулятора з існуючими аналогами.

1 ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО НОРМАЛЬНІ АЛГОРИТМИ МАРКОВА

1.1 Формалізація правил та визначень для нормальних алгоритмів

Наприкінці 40-х років 20-го сторіччя радянським математиком О. О. Марковим було запропоновано нове уточнення поняття алгоритму, яке пізніше отримало його ім'я [2]. Нормальні алгоритми Маркова (НАМ) є однією із формалізацій інтуїтивного розуміння алгоритму.

НАМ представляються схемою, яка складається з сукупності підстановок, розташованих в певному порядку. Нехай A – скінчений алфавіт для запису вхідних і вихідних слів, та α, β, γ – деякі слова в алфавіті A. Алфавіт B називається розширенням алфавіту A, якщо A є підмножиною множини B. Це означає, що всі символи алфавіту A входять до складу алфавіту B, але B може містити додаткові символи. Нормальний алгоритм над алфавітом A можна розглядати як нормальний алгоритм у розширеному алфавіті B. Це означає, що правила та підстановки алгоритму для алфавіту A залишаються дійсними і можуть застосовуватися в алфавіті B, за умови, що вони перетворюють слова алфавіту A у слова того ж алфавіту. Таким чином, розширення алфавіту дозволяє зберегти функціональність алгоритму і правильно обробляти слова з алфавіту A, навіть при додаванні нових символів у B. Це розширює можливості застосування алгоритмів без зміни їх основних властивостей і логіки роботи.

Марківська підстановка (МП) – це операція над словами α, β, γ, що має вигляд $\alpha \to \beta$ і полягає у наступному:

– у вхідному слові γ шукається крайнє ліве входження слова α;

– якщо воно існує, тоді α замінюється на β у слові γ;

– отримане слово $\gamma`$ є результатом застосування марківської підстановки $\alpha \to \beta$ до слова γ;

– якщо слово α не входить у слово γ, то кажуть, що дана марківська

підстановка не може бути застосована до слова γ.

Слово α ще називають лівою частиною марківської підстановки, а β – правої частиною. У загальному випадку довжина слова α може не співпадати з довжиною слова β. Тобто один символ може бути змінений на декілька і навпаки. З цього слідує, що у загальному випадку довжини слів γ і $\gamma`$ також можуть не збігатися.

Існують випадки, коли α або β є порожніми. Нехай α є порожнім, тоді β приписуються перед словом γ, а результатом є нове слово $\gamma` = \beta\gamma$. Нехай β є порожнім, тоді слово α виключається зі слова γ, якщо воно там існує.

Якщо довжина слова α дорівнює нулю, то відповідне слово є порожнім. У МП порожні слова ніяк не позначаються і не займають ніякого місця [3]. Але у будь якому слові α є кілька входжень порожнього слова:

– перед першою буквою;
– після останньої;
– поміж довільною парою букв у середині слова.

При виконанні нормального алгоритму Маркова спочатку розглядається перше правило підстановки, шукається перше входження лівої частини правила у вхідне слово. Якщо воно є, виконується підстановка у вхідне слово правої частини правила замість лівої і робота триває з результатом підстановки [4]. Якщо входження лівої частини знайдено, то розглядається друге правило тощо. Після кожної виконаної підстановки перегляд правил починається спочатку (з першого правила). Якщо жодну підстановку не можна виконати або виконано підстановку за правилом виду $\alpha \to \beta\,.$ (термінальному правилу, додається символ крапки), то алгоритм вважається виконаним, а отримане в результаті підстановок слово – результатом або вихідним словом.

Дано $A=\{a, b\}$. Побудувати нормальний алгоритм Маркова, який замінює у вхідному слові всі символи a на b, а всі b на a.

Алгоритм зображено на рисунку 1.1.

1)	$*a \to b*$
2)	$*b \to a*$
3)	$* \to \ .$
4)	$\to *$

Рисунок 1.1 – НАМ, що замінює всі символи *a* на *b*, а всі *b* на *a*

Розглянемо дії алгоритму над словом *abba*: $abba \to^{(4)} *abba \to^{(1)} b*bba \to^{(2)} ba*ba \to^{(2)} baa*a \to^{(1)} baab* \to^{(3)} baab$. Оскільки виконано термінальне правило (3), виконання закінчено, і отримане слово є результатом. Можна оцінити складність як кількість виконаних підстановок, їх буде n+2, де n – довжина слова. Зауважимо, що НАМ із двох правил: $\{ a \to b \mid b \to a\}$ завдання не вирішує, оскільки зациклюється на будь-якому непустому слові над алфавітом *A*. Відбувається це тому, що ліва частина першого правила збігається з правою частиною другого, отже, після виконання другого правила завжди спрацює перше. Ліва частина другого правила збігається з правою частиною першого, отже, після застосування першого правила спрацює друге. Обидва випадки породжують нескінченний цикл. Також і будь-який нормальний алгоритм, який містить правила з порожньою лівою частиною та не містить до них термінальних правил, обов'язково зациклиться на будь-якому вхідному слові [5].

Нормальний алгоритм Маркова називається застосовним до вхідного слова α, якщо він зупиняється на цьому слові, і незастосовним – якщо зациклюється [6]. Безліч вхідних слів, до яких НАМ застосовується називається його областю застосовності.

У деяких випадках з вигляду правил НАМ можливо зробити висновок про його застосування до всіх вхідних слів. Справедливі дві достатні ознаки:

а) Якщо в НАМ у кожному правилі (за винятком термінальних) ліва частина не порожня і не містить літер, що входять до правих частин правил, то алгоритм застосуємо до будь-яких вхідних слів.

б) Якщо в НАМ у кожному правилі (за винятком термінальних) ліва частина довша за праву, то алгоритм застосуємо до будь-яких вхідних слів.

Обґрунтуємо першу ознаку. Із застосуванням кожного правила, що задовольняє умові, кількість входжень літер, що використовуються для запису лівих частин, зменшується, отже, зменшується кількість можливих спрацьовувань правил. Нових входжень лівих частин у слово не з'являється, оскільки праві частини не містять потрібних при цьому букв. НАМ зупиниться.

Приклад НАМ такого роду зображено на рисунку 1.2.

1) $a \to 01$
2) $b \to 10$
3) $c \to 11$

Рисунок 1.2 – Приклад НАМ для першої ознаки

Обґрунтуємо другу ознаку. Після кожного спрацьовування правила, що задовольняє умові, довжина слова зменшується, тому через не більше ніж n підстановок (де n – довжина слова) алгоритм зупиниться.

Теза Маркова, також відома як принцип нормалізації, стверджує, що будь-який алгоритм може бути реалізований у вигляді нормального алгоритму Маркова. Також справедливими є твердження про еквівалентність формалізмів, запропонованих Марковим та Тюрінгом [7], тобто:

– для будь-якої даної машини Тюрінга можна побудувати нормальний алгоритм Маркова, який здійснює те саме перетворення вхідних слів у вихідні;

– для будь-якого даного нормального алгоритму Маркова можна побудувати машину Тюрінга, що здійснює те саме перетворення вхідних слів у вихідні.

Отже, нормальні алгоритми Маркова та машини Тюрінга мають різні методи представлення та побудови, проте обидва вони можуть обчислювати ту ж саму множину функцій. Це робить їх еквівалентними в плані обчислювальної потужності, що є ключовим фактором у формалізації поняття обчислювальної моделі. Таким чином, машини Тюрінга та нормальні

алгоритми Маркова є двома різними способами опису та реалізації алгоритмів, але вони здатні до виконання тих самих завдань за рахунок їхньої еквівалентності в обчислювальній потужності.

1.2 Приклади та тестування нормальних алгоритмів

Для більшого розуміння та ілюстрації принципів роботи нормальних алгоритмів Маркова, в цьому розділі наведено декілька прикладів. Ці приклади демонструють застосування НАМ для вирішення конкретних завдань.

Приклад 1.1. Побудувати нормальний алгоритм, що перетворить вхідний рядок у рядок в алфавіті $A=\{a, b, c\}$, у якому букви розташовано за абеткою.

На рисунку 1.3 зображена реалізація алгоритму вирішення цієї задачі.

1) ba → ab
2) ca → ac
3) cb → bc

Рисунок 1.3 – НАМ, що розташовує букви за абеткою

На рисунку 1.4 зображено тестування алгоритму на слові «*bcab*».

$$b\underline{ca}b \overset{2}{\rightarrow} \underline{ba}cb \overset{1}{\rightarrow} ab\underline{cb} \overset{3}{\rightarrow} abbc$$

Рисунок 1.4 – Тестування НАМ із прикладу 1.1 на слові «*bcab*»

Приклад 1.2. Реалізувати алгоритм, що здійснює операцію віднімання в унарній системі числення. $A=\{1, -\}$ (рисунок 1.5).

1) 1−1 → −
2) −1 → −
3) − → .

Рисунок 1.5 – НАМ віднімання для унарних чисел

На рисунку 1.6 зображено тестування алгоритму на словах «*1111–11*» та «*–1*», та «*1–*»

$$1)\ 1111\underline{-11} \xrightarrow{1} 11\underline{1-1} \xrightarrow{1} 11\underline{-} \xrightarrow{3} 11\ ;$$
$$2)\ \underline{-1} \xrightarrow{2} \underline{-} \xrightarrow{3}\ ;$$
$$3)\ 1\underline{-} \xrightarrow{3} 1\ .$$

Рисунок 1.6 – Тестування НАМ для прикладу 1.2 на різних випадках

Приклад 1.3. Реалізувати алгоритм, що в алфавіті $A=\{a, b\}$ стирає все, окрім послідовності символів *aab* (рисунок 1.7).

1) *A → a*	8) #a → #
2) *B → b*	9) #b → #
3) ## →	10) #A → a*
4) *# → .	11) #B → b*
5) aab → #AAB#	12) a →
6) a# → #	13) b →
7) b# → #	14) → .

Рисунок 1.7 – НАМ, що стирає все, окрім послідовності символів *aab*

Спочатку знаходиться послідовність aab та виділяється за допомогою підстановки 5. Далі стираються всі символи, що знаходяться перед цією послідовністю, за допомогою підстановок 6 та 7. Потім стираються всі символи, що знаходяться після цієї послідовності, за допомогою підстановок 8 та 9. Розшифровка #AAB# здійснюється підстановками 10, а потім 1 та 2. Підстановка 4 видаляє допоміжні символи та завершує роботу алгоритму. Якщо в слові немає такої послідовності, воно стирається за допомогою підстановок 12 та 13.

На рисунку 1.8 зображено тестування алгоритму на слові «*abab*».

$$\underline{a}bab \overset{12}{\rightarrow} \underline{b}ab \overset{13}{\rightarrow} \underline{a}b \overset{12}{\rightarrow} \underline{b} \overset{13}{\rightarrow} \ ;$$

Рисунок 1.8 – Тестування НАМ для прикладу 1.3 на слові «*abab*»

На рисунку 1.9 зображено тестування алгоритму на слові «*aaabb*».

$$a\underline{aab}b \overset{5}{\rightarrow} \underline{a\#}AAB\#b \overset{6}{\rightarrow} \#AAB\underline{\#b} \overset{9}{\rightarrow}$$
$$\overset{9}{\rightarrow} \underline{\#A}AB\# \overset{10}{\rightarrow} a \underline{* A}B\# \overset{1}{\rightarrow} aa \underline{* B}\# \overset{2}{\rightarrow} aab \underline{* \#} \overset{4}{\rightarrow} aab$$

Рисунок 1.9 – Тестування НАМ із прикладу 1.3 на слові «*aaabb*»

Приклад 1.4. Збільшити число, записане у трійковій системі числення, на одиницю.

1) *0 -> 0* ;Гонимо маркер у кінець числа
2) *1 -> 1*
3) *2 -> 2*
4) 0* -> 1 ;Збільшуємо останній розряд
5) 1* -> 2
6) 2* -> #0 ;З перенесенням до наступного розряду
7) 0# -> 1 ;Збільшуємо розряд
8) 1# -> 2
9) 2# -> #0 ;З перенесенням до наступного розряду
10) ## -> 1. ; Забираємо зайві символи на початку
11) # -> .
12) -> *# ; Ставимо маркери початку та кінця

Рисунок 1.10 – НАМ збільшення числа у трійковій системи числення на 1

На рисунку 1.11 зображено тестування алгоритму на слові «*112*».

$$_112 \overset{12}{\rightarrow} \# \underline{* 1}12 \overset{2}{\rightarrow} \#1 \underline{* 1}2 \overset{2}{\rightarrow} \#11 \underline{* 2} \overset{3}{\rightarrow} \#112 \underline{* } \overset{6}{\rightarrow} \#1\underline{1\#}0 \overset{8}{\rightarrow} \underline{\#}120 \overset{11}{\rightarrow} 120$$

Рисунок 1.11 – Тестування НАМ із прикладу 1.4 на слові «*112*»

Розглянуті приклади нормальних алгоритмів Маркова наочно демонструють їх ефективність та універсальність у вирішенні різних завдань. НАМ представляють собою потужний інструмент для формалізації та реалізації алгоритмічних процесів. Їх застосування охоплює широкий спектр завдань, від простих замін символів до складних математичних операцій.

2 АНАЛІЗ ІСНУЮЧИХ ЕМУЛЯТОРІВ НОРМАЛЬНИХ АЛГОРИТМІВ

2.1 Мобільний застосунок

Серед мобільних застосунків у магазині AppStore [8] було знайдено емулятор марковських алгоритмів від автора, підписаного як «Konstantin Mishukov» [9].

Додаток має головне вікно із двома пунктами меню, а саме: «Algorithm» та «Settings». Нижче знаходиться текстове поле для робочого рядка, а також кнопка «Execute Algorithm» (рисунок 2.3, а).

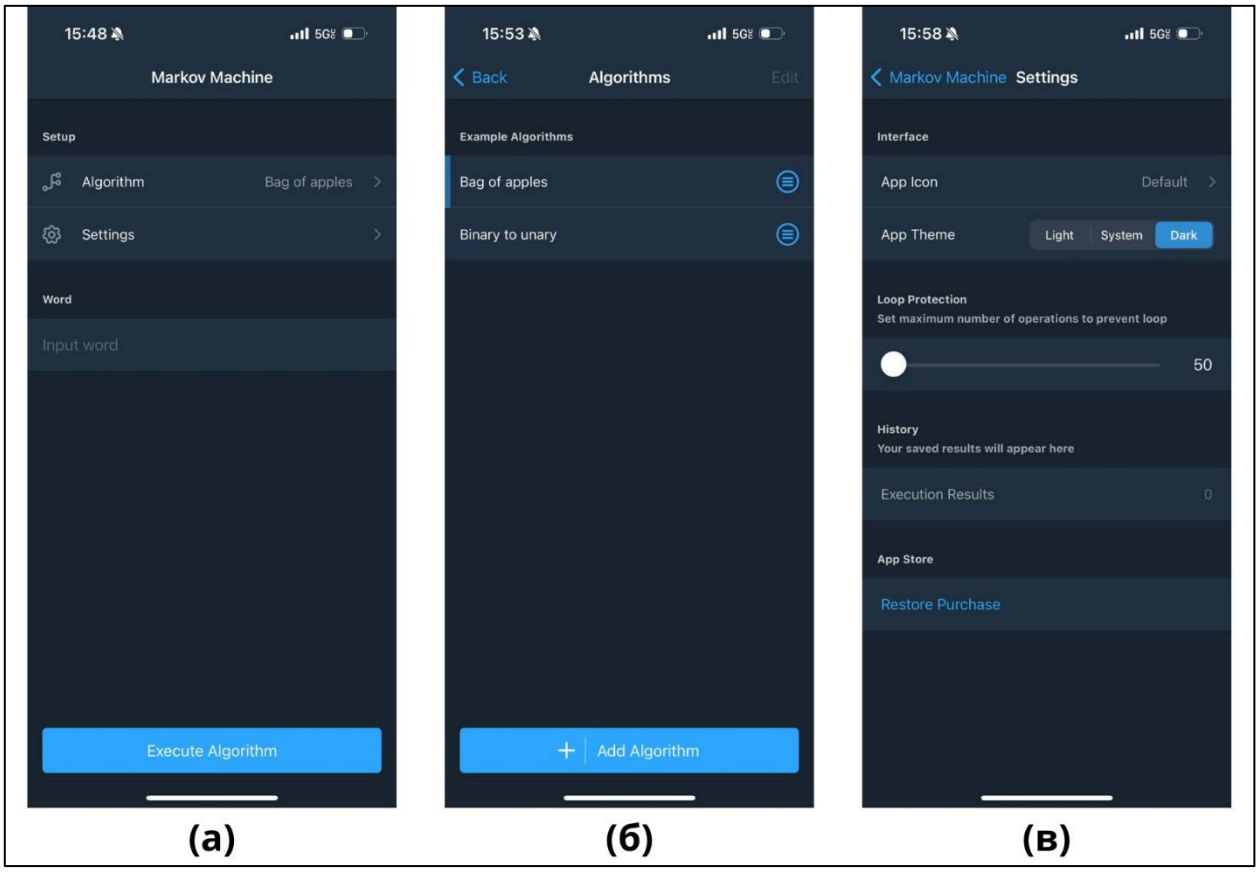

Рисунок 2.3 – Основні вікна додатку

При виборі пункту «Algorithm» відкривається вікно із двома доступними прикладами, а також можливістю додати свій алгоритм, натиснувши кнопку «Add Algorithm» (рисунок 2.3, б). При виборі пункту «Settings» відкриється вікно із налаштуваннями (рисунок 2.3, в), де можна змінювати зовнішній

вигляд інтерфейсу, встановлювати обмеження циклів (від 50 до 5000), переглядати історію виконання алгоритмів, а також відновити доступ до внутрішніх покупок.

При натисканні кнопки «Add Algorithm» (рисунок 2.3, б) з'являється можливість додати свій алгоритм. Програма запропонує дати йому ім'я, після чого відкриється вікно, зображене на рисунку 2.4.

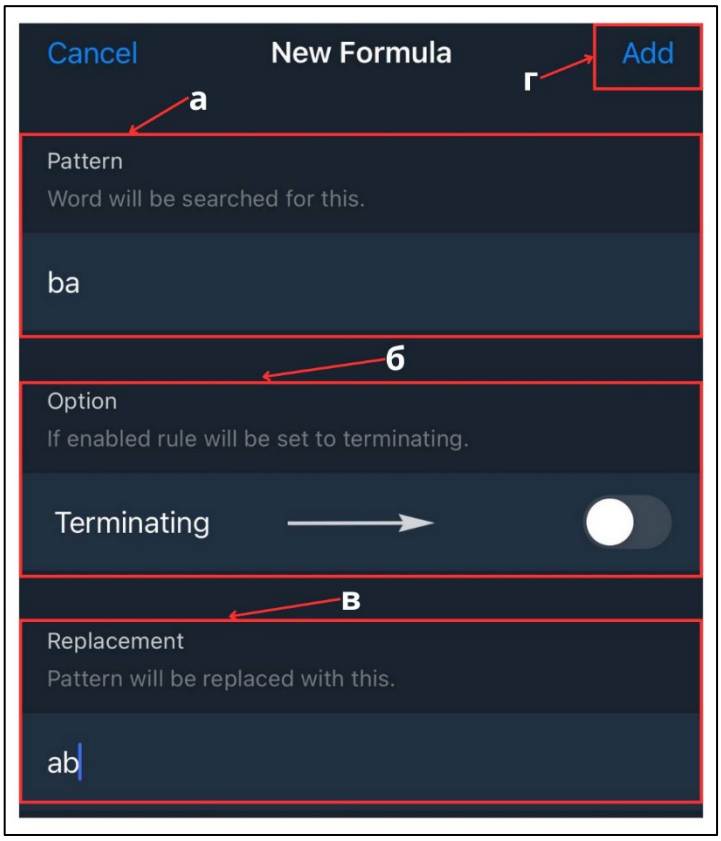

Рисунок 2.4 – Вікно додавання нової формули

Ліва сторона правила підстановки записується у поле «Pattern» (рисунок 2.4, а), права сторона правила підстановки записується у поле «Replacement» (рисунок 2.4, в). Якщо правило, що додається, є термінальним – необхідно перевести у стан «включено» кнопку, що знаходиться у підпункті «Option» (рисунок 2.4, б). Натиснувши кнопку «Add» (рисунок 2.4, г) це правило підстановки додається до загального списку правил.

Для тестування програми було додано новий алгоритм «Double word». Його мета – подвоїти слово, що складається з однакових символів ($A=\{x\}$).

На рисунку 2.5 відображено команди, записані у відповідному вікні мобільного додатку.

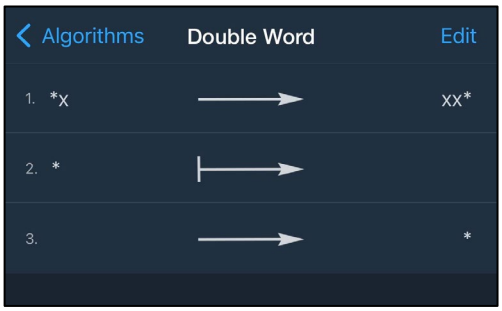

Рисунок 2.5 – Алгоритм НАМ у вікні мобільного додатку

Після запуску цього алгоритму зі словом *xx* відкривається нове вікно із результатами роботи. Це вікно відображає характер завершення роботи (термінальне правило або завершення через обмеження), назву алгоритму, кількість підстановок, вхідні та вихідні дані. В даному випадку програма спрацювала правильно, видавши подвоєне слово (рисунок 2.6).

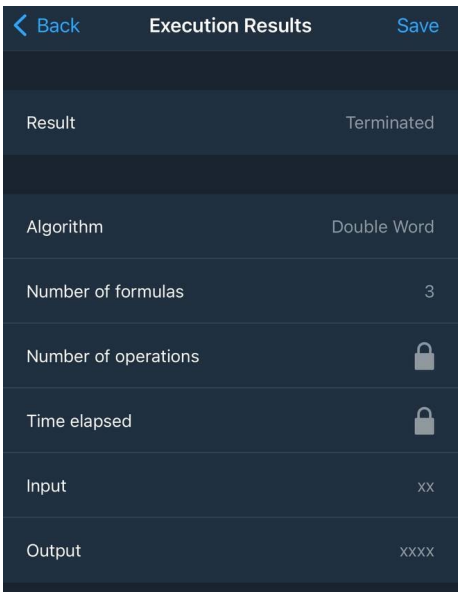

Рисунок 2.6 – Вікно з результатами роботи програми

Для збереження результатів, перегляду логування, кількості операцій та витраченого часу застосунок вимагає звершити внутрішню покупку за додаткову плату.

2.2 Веб-емулятор

У веб-середовищі необхідно визначити один найбільш функціональний веб-емулятор, який було виявлено. Такий веб-емулятор було знайдено на сайті з матеріалами з практикуму для студентів ВМК МДУ ім. М.В. Ломоносова [10].

У даній роботі, оскільки інтерфейс веб-сайту повністю представлений російською мовою, було використано розширення від Google [11] для автоматичного перекладу інтерфейсу на українську мову. Це розширення дозволяє користувачеві зручно працювати з веб-сайтом, не витрачаючи час на ручний переклад текстів. Далі таке рішення буде використовуватись за необхідністю, тому як таке рішення дозволяє підтримувати мовну політику.

На рисунку 2.1 зображено головний інтерфейс веб-сайту з деякими помітками.

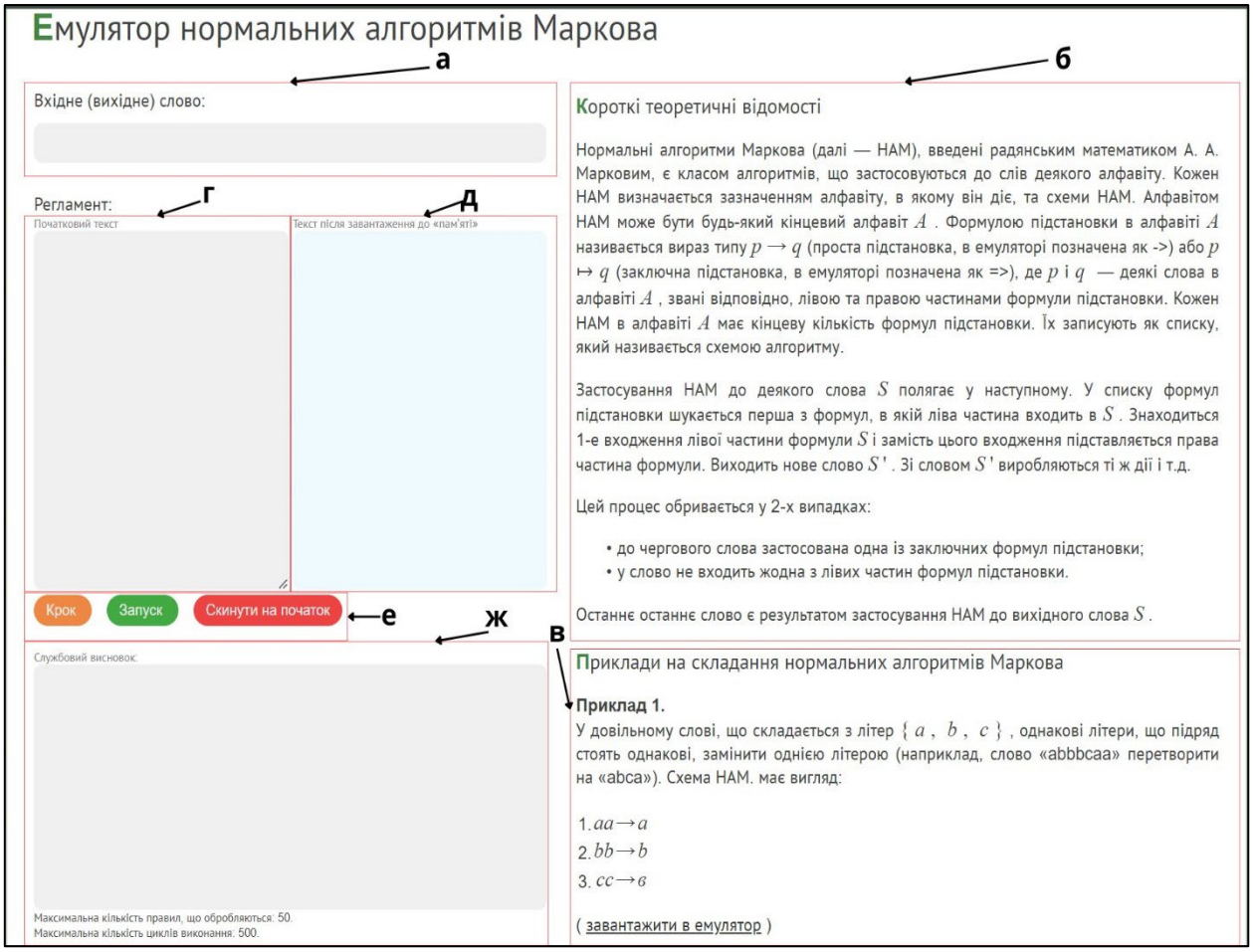

Рисунок 2.1 – Інтерфейс веб-емулятора

Умовно сторінку можна поділити на 7 частин (рисунок 2.1). Автор додав короткі теоретичні відомості з прикладами на головну сторінку емулятора (рисунок 2.1, б і в), а також можливість завантажувати приклади напряму в емулятор одним натисканням кнопки, що полегшує тестування цих прикладів, а також розуміння протоколу написання правил підстановки у відповідному полі. Текстове поле для вводу вхідного, а також виводу вихідного слова, тобто робочого рядка (рисунок 2.1, а). Нижче розташовані два текстові поля для введення правил підстановок (рисунок 2.1, г і д). Поле "початковий текст" доступне для введення, і користувач може вводити свої команди з дотриманням певних правил введення (рисунок 2.2). Поле, що розташоване справа від нього (рисунок 2.1, д), є закритим для введення і лише дублює ці команди у визначеному програмним засобом стилі та шрифті, роблячи рядки з правилами підстановок динамічними та підсвічуючи той рядок, який застосовується на поточному кроці. Ще нижче розташовано три кнопки (рисунок 2.1, е), які дозволяють зробити один крок алгоритму, запустити алгоритм від початку до кінця, або почати виповнення алгоритму з самого початку. Останнім елементом є текстове поле для службового виводу (рисунок 2.1, ж). Воно виводить застосовані до робочого рядка кроки, тобто відбувається логування.

Емулятор ставить певні обмеження перед користувачем, а саме:

– максимальна кількість правил, що обробляються: 50;

– максимальна кількість циклів виконання: 500.

З точки зору сайтобудування такі обмеження є виправданими. Вони можуть забезпечувати: стабільність роботи (запобігання виконанню непродуктивних або безкінечних алгоритмів), контроль навантаження на сервер та запобігання перевантаженню системи [12], а також зменшення ризику зловживання (зниження ризику виникнення проблем з безпекою системи). З іншого боку, ці обмеження можуть унеможливити аналіз або використання деяких типів нормальних алгоритмів, які вимагають більшої кількості правил або більшої кількості циклів для виконання.

Приклад виконання алгоритму програмою зображено на рисунку 2.2.

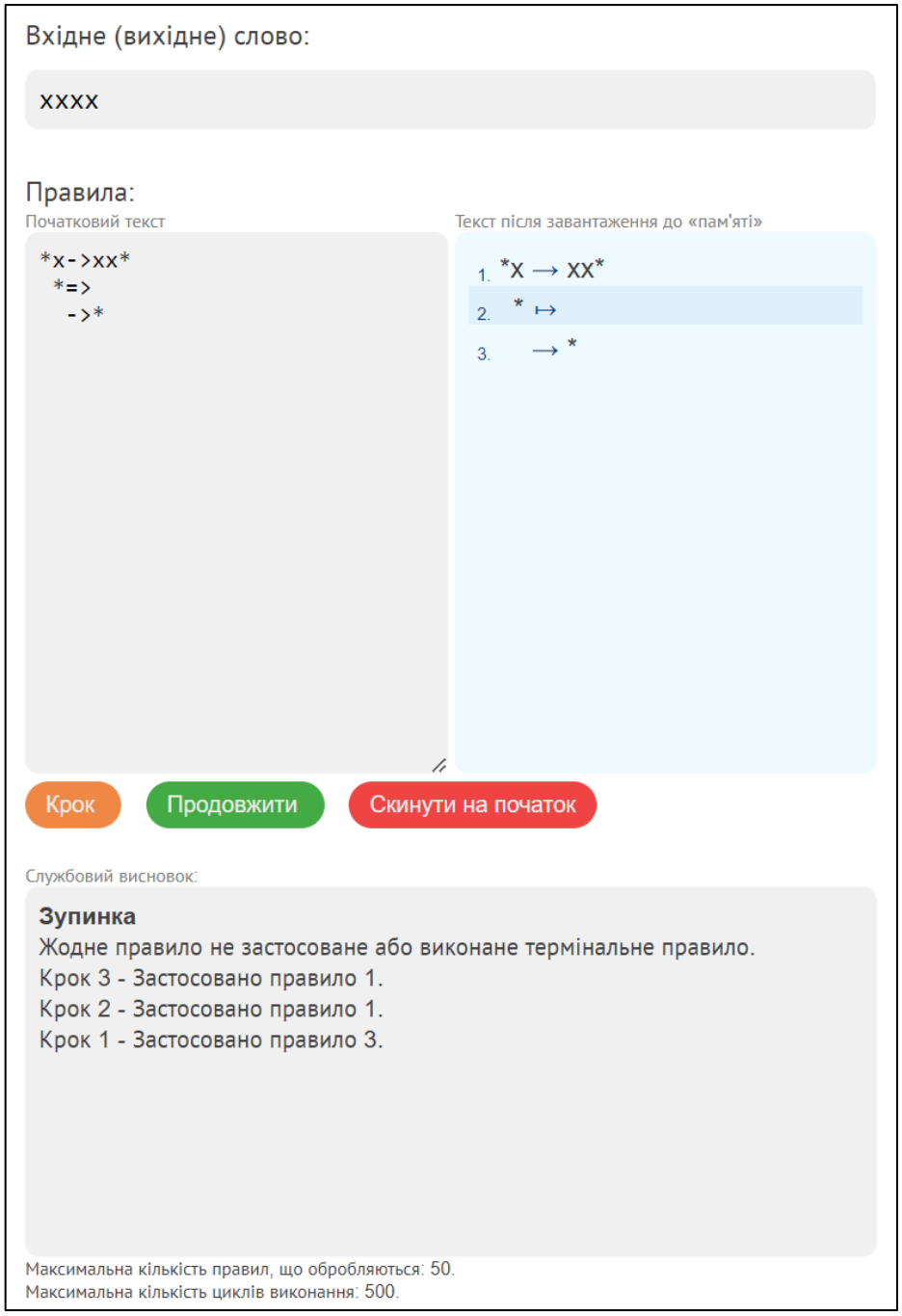

Рисунок 2.2 – Приклад виконання алгоритму програмою

Щодо працездатності самого програмного продукту, під час тестування серйозних помилок виявлено не було. Натомість кнопка «Скинути на початок» очищує лише поле логування, але не скидає робочий рядок до первісного стану, а також залишає підсвіченим рядок, який був застосований востаннє.

2.3 Десктопний додаток

Серед всіх розглянутих емуляторів своїм функціоналом найбільше виділяється десктопна версія авторства Констянтина Полякова [13]. Програмний продукт повністю безкоштовний до завантаження та використання, та не потребує інсталяції. Архів завантажується із виконавчим файлом програми та бібліотекою файлів-прикладів алгоритмів. На рисунку 2.7 зображено головне вікно програми.

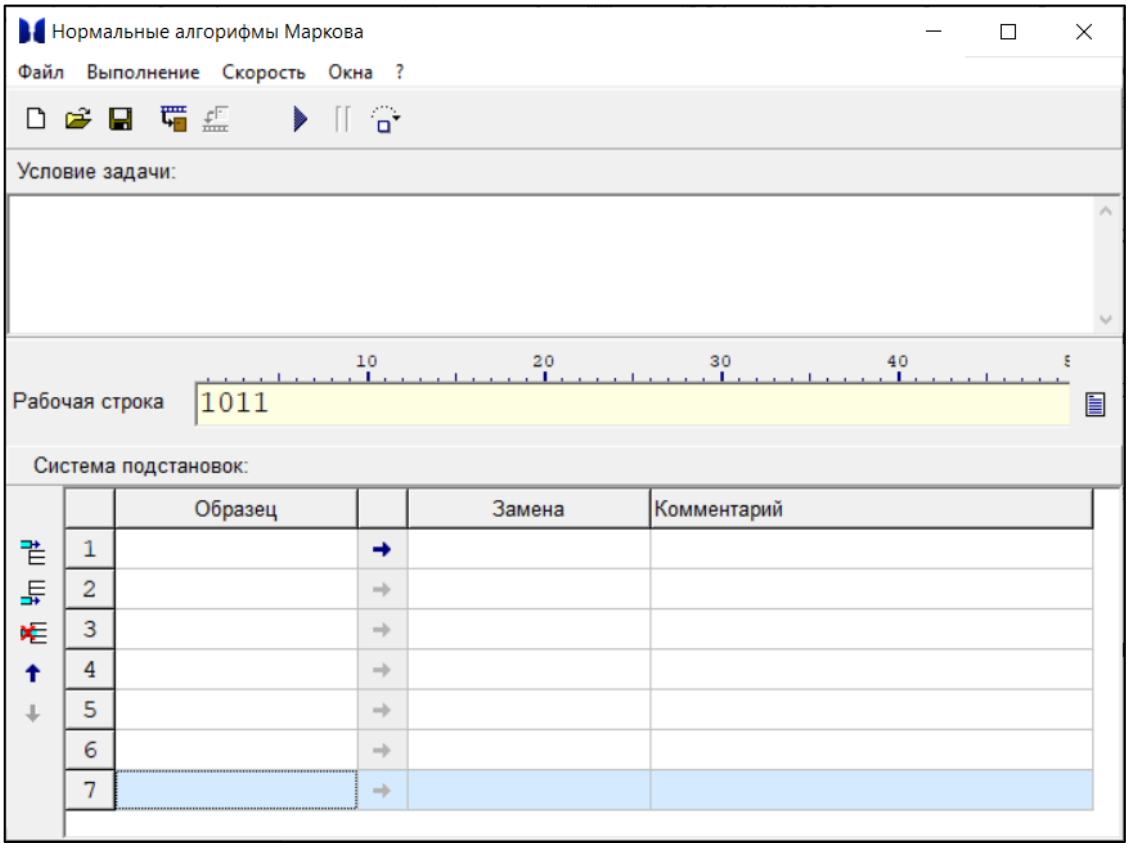

Рисунок 2.7 – Інтерфейс десктопного емулятора

На відміну від вищерозглянутих емуляторів, даний емулятор має динамічну таблицю для запису кожного правила підстановки та можливість додати коментар до кожного з них. Автором передбачено виділення кожного рядка таблиці та проведення з ним наступних операцій:

– додавання нового рядка вище або нижче над виділеним;

– видалення виділеного рядка;

– перенесення виділеного рядка нижче або вище за таблицею;

Серед відмінностей також можна виділити:

– робочий рядок, що має лінійку, яка позначає довжину слова;

– текстове поле для запису умови задачі;

– можливість зберегти алгоритм та стан машини шляхом імпорту та експорту файлу із розширенням «.nam»;

– задання швидкості виконання алгоритму та можливість поставити це виконання на паузу;

– можливість зберегти стан робочої строки окремою кнопкою та відновити цей стан іншою кнопкою.

Окрім цього програма має окреме вікно, де записується покрокове виконання програми, а також довідка з інформацією про автора, основні теоретичні відомості, та інструкція користувача.

На рисунку 2.8 зображено процес виконання програмою алгоритму для переведення числа з двійкової системи числення в унарну.

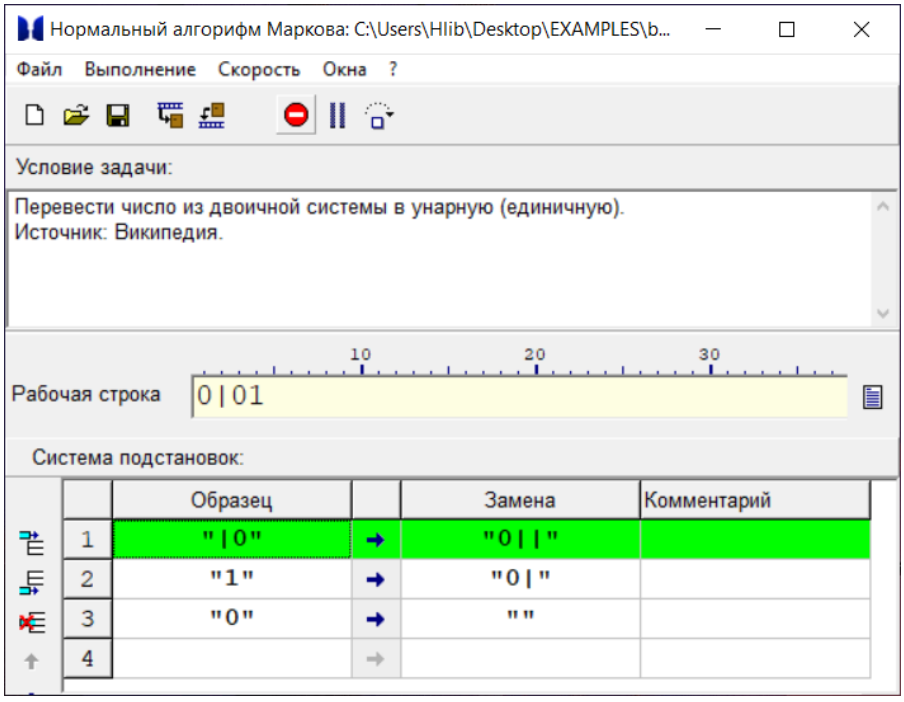

Рисунок 2.8 – Емулятор під час виконання алгоритму

Щодо обмежень у циклах чи кількості правил, автор вказівок не надав. Помилок під час тестування не було виявлено.

2.4 Переваги та недоліки розглянутих емуляторів

Переваги веб-емулятору полягають у:

– зручному доступі (запускається у браузері з будь-якого пристрою з встановленим браузером) [14];

– коротка теоретична інформація по нормальним алгоритмам та приклади на головній сторінці емулятора.

Серед недоліків веб-емулятору можна виділити обмеження в кількості правил та циклі, а також в неможливості збереження алгоритму для подальшого використання.

Переваги мобільного застосунку для IOS полягають у можливість офлайн-роботи на смартфоні [15] та збереження алгоритмів (у безкоштовній версії програми – не більше одного). Недоліком є неможливість запуску на інших ОС окрім IOS.

Серед переваг десктопної версії емулятора можна зазначити наступне:

– безкоштовність, не вимагає інсталяції;

– динамічна таблиця для правил підстановки з можливістю додавання коментарів;

– широкий функціонал, включаючи збереження алгоритмів в окремих файлах.

Недоліком десктопної версії є можливість використання лише на платформі Windows.

Отже, залежно від потреби та вимог користувача, можна визначити найбільш підходящий варіант емулятора. Якщо потрібен швидкий доступ та базова робота з алгоритмами без необхідності зберігання даних – веб-емулятор буде оптимальним рішенням. Для користувачів iOS, які потребують можливості роботи офлайн, мобільний додаток буде найзручнішим. Якщо ж потрібна розширена функціональність та можливість детальної роботи з алгоритмами без обмежень у кількості правил та циклів, десктопна програма-емулятор буде найбільш оптимальним рішенням.

У таблиці 2.1 наведено порівняння параметрів емуляторів.

Таблиця 2.1 – Порівняння параметрів розглянутих емуляторів

Доступні функції	**Веб-емулятор**	**Застосунок IOS**	**Десктопний додаток**
Офлайн-робота	Ні	Так	Так
Збереження алгоритму	Ні	Так	Так
Кількість правил	До 50	Необмежено	Необмежено
Кількість циклів	До 500	До 5000	Необмежено
Наявність прикладів	Так	Так	Так
Динамічна таблиця	Ні	Ні	Так
Додавання коментарів	Ні	Ні	Так
Регулювання швидкості виконання	Ні	Ні	Так
Платформа	Браузер	iOS	Windows
Детальний звіт	Ні	Так (платно)	Так
Безкоштовність	Так	Частково	Так

Виходячи з проведеного аналізу, виникає логічне рішення розглянути можливість розробки модифікованого онлайн-емулятора, який буде поєднувати в собі найкращі якості кожного з вищезазначених варіантів. Такий емулятор може забезпечити усі необхідні функції та інструменти для роботи з нормальними алгоритмами Маркова, а також надати основні теоретичні відомості з даної теми українською мовою.

Такий проєкт також буде актуальним у контексті росту зацікавленості українських користувачів у вивченні та використанні формальних мов програмування.

3 РОЗРОБКА ФУНКЦІОНАЛЬНОЇ ЧАСТИНИ ОНЛАЙН-ЕМУЛЯТОРА ДЛЯ НОРМАЛЬНИХ АЛГОРИТМІВ МАРКОВА

3.1 Загальний опис алгоритму

Вхідними даними для онлайн-емулятора є правила підстановок та робочий рядок (word). Правила підстановок можуть бути представлені у вигляді двох масивів: зразки (A) та заміни (B). При цьому необхідно забезпечити, щоб кількість елементів у кожному з цих масивів була однаковою (n), що дозволяє гарантувати коректність та повноту перетворень, які буде виконувати алгоритм.

Для реалізації емулятора необхідно розробити інтерфейс введення даних, який забезпечить користувачу можливість введення правил підстановок у зручному та інтуїтивно зрозумілому форматі. Це передбачає створення механізму перевірки та валідації введених даних, щоб кількість зразків та замін відповідала встановленим вимогам.

Після введення правил підстановок та робочого рядка, дані будуть передаватися у відповідну функцію для подальшої обробки (рисунок 3.1).

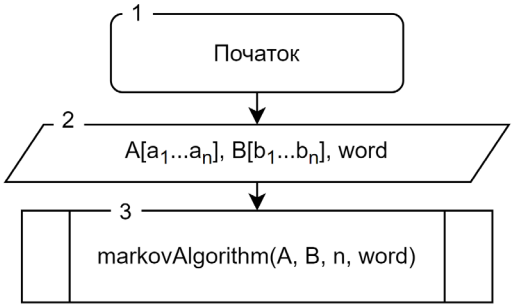

Рисунок 3.1 – Передача вхідних даних у відповідну функцію

3.1.1 Ітеративна реалізація алгоритму

Головна ідея алгоритму функції markovAlgorithm() полягає у використанні циклу do...while, який триває, поки у рядку word знаходяться зразки з масиву A. У межах цього циклу здійснюється ітерація через всі

правила підстановок за допомогою циклу for. Для кожного зразка A[i] функція шукає його входження у рядку word. Якщо зразок знайдено, здійснюється підстановка B[i], після чого перевіряється, чи правило B[i] є термінальним. Якщо так, алгоритм завершує роботу. Якщо ні, алгоритм продовжує обробку оновленого рядка. Процес триває, поки у рядку word можна знайти зразок для підстановки. У результаті, функція повертає остаточно оброблений рядок.

Блок-схему функції markovAlgorithm зображено на рисунку 3.2.

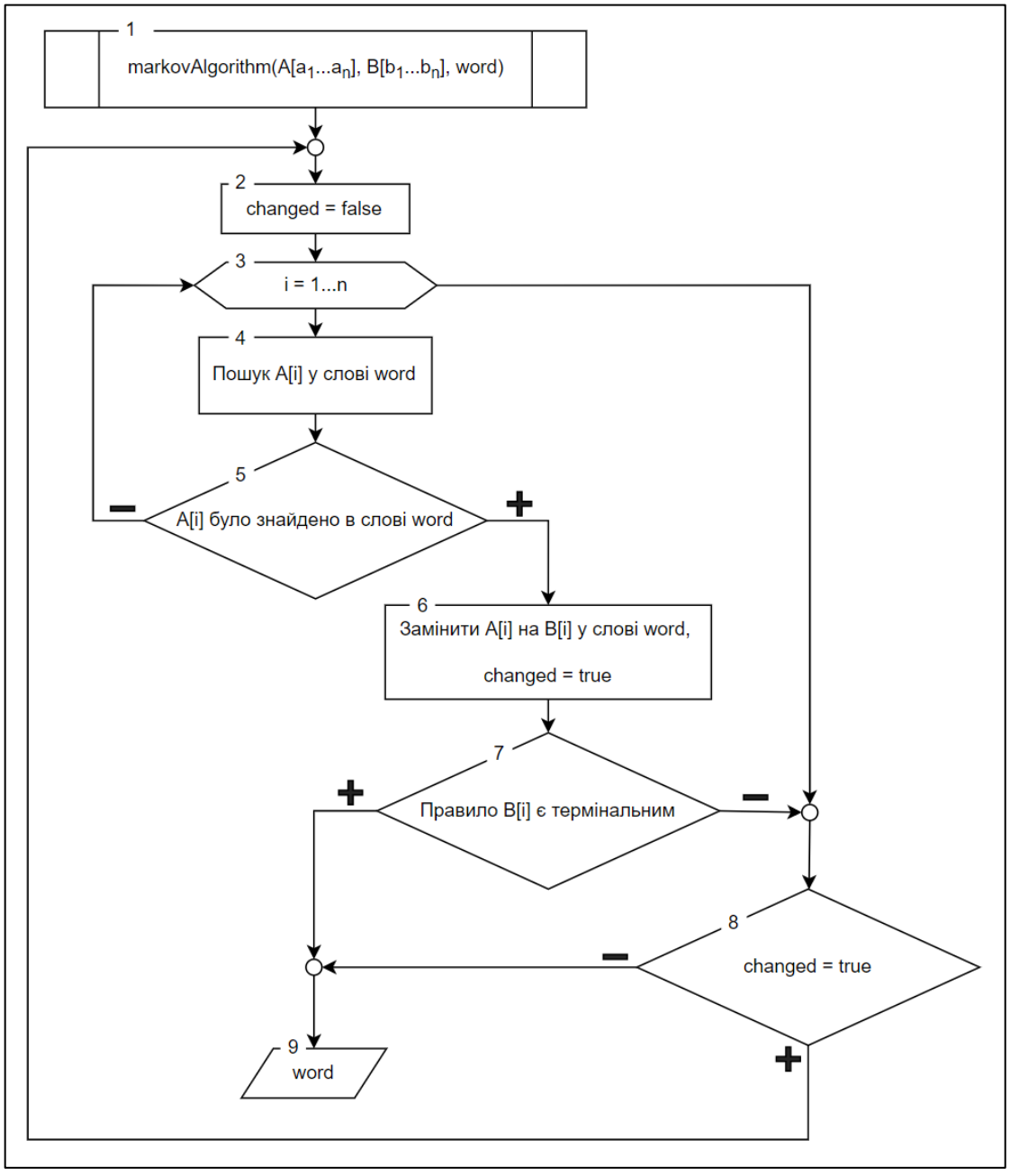

Рисунок 3.2 – Блок-схема функції markovAlgorithm()

Покроковий опис алгоритму з рисунку 3.2:

1) початок функції markovAlgorithm();

2) встановлення «прапорця» в стан false, тобто зміни у слові поки що не відбулись;

3) цикл for для проходження по правилах підстановок;

4) пошук зразка A[i] у слові word;

5) якщо A[i] у word не було знайдено, програма повертається до кроку 3;

6) якщо A[i] у word було знайдено, відбувається заміна зразка A[i] на відповідний елемент масива B, а стан «прапорця» встановлюється в значення true;

7) перевірка правила B[i] на термінальність, та, якщо воно є термінальним, алгоритм переходить до кроку 9;

8) якщо правило B[i] не є термінальним, перевіряється значення «прапорця», та, при значенні true, алгоритм повертається до кроку 2, інакше – до кроку 9;

9) вивід обробленого слова word та вихід із функції.

3.1.2 Рекурсивна реалізація алгоритму

Як альтернативний підхід до імплементації алгоритму, було розроблено рекурсивну функцію.

Рекурсія – це метод організації обчислювального процесу, за якого функція викликає сама себе. Такі виклики називаються рекурсивними, а функція, що містить їх, – рекурсивною [16]. Існують два типи рекурсії: пряма та непряма. Пряма рекурсія відбувається, коли функція викликає саму себе безпосередньо, тоді як непряма рекурсія передбачає взаємні виклики двох або більше функцій.

Метою розробки такої функції була демонстрація можливості реалізації алгоритму без використання ітеративної конструкції do…while, що, в теорії, може сприяти спрощенню логіки розуміння алгоритму та зменшення кількості

змінних.

Блок-схему рекурсивної функції markovAlgorithm() зображено на рисунку 3.3.

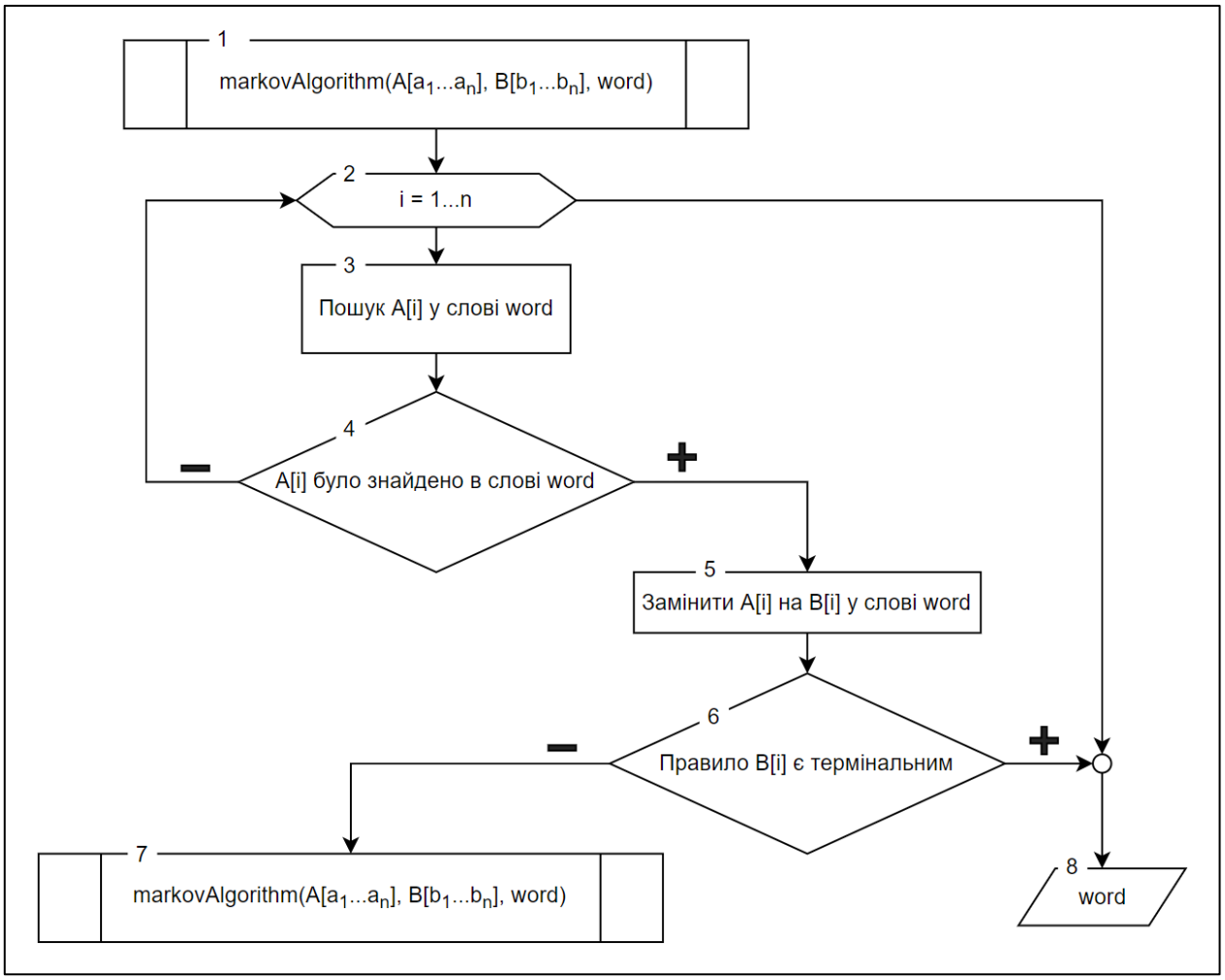

Рисунок 3.3 – Блок-схема рекурсивної реалізації алгоритму

Покроковий опис алгоритму з рисунку 3.3:

1) початок функції markovAlgorithm();

2) цикл for для проходження по правилах підстановок;

3) пошук зразка A[i] у слові word;

4) якщо A[i] у word не було знайдено, програма повертається до кроку 2;

5) якщо A[i] у word було знайдено, відбувається заміна зразка A[i] на відповідний елемент масива B;

6) перевірка правила B[i] на термінальність, та, якщо воно є термінальним, алгоритм переходить до кроку 8;

7) якщо правило B[i] не є термінальним, функція звертається сама до себе, та знов викликає markovAlgorithm(), повертаючись до кроку 1;

8) вивід обробленого слова word та вихід із функції.

Таким чином, було усунуто необхідність у додатковій змінній-прапорці changed та циклі do...while. Незважаючи на те, що блок-схема рекурсивної функції виглядає більш лаконічно, для оцінки її ефективності необхідно провести більш детальний аналіз. Необхідно провести тестування та порівняння з ітеративною імплементацією алгоритму, враховуючи такі аспекти продуктивності, як час виконання, використання пам'яті та обробка великих обсягів даних.

3.1.3 Порівняльний аналіз рекурсивного та ітеративного підходів

У процесі реалізації емулятора алгоритмів Маркова виникає вибір між використанням рекурсії та ітерації. Цей вибір впливає на ефективність та надійність програмного забезпечення, особливо при роботі з великими вхідними даними. В даному розділі буде розглянуто застосування рекурсивного та ітеративного підходів у реалізації алгоритму Маркова, а також будуть проаналізовані переваги та недоліки кожного з підходів у контексті даної роботи.

У реалізації алгоритмів Маркова рекурсивний підхід може здаватися природним вибором для організації послідовності підстановок. Однак, при роботі з великими вхідними даними рекурсивна реалізація програми може стикатися з низкою проблем:

- переповнення стеку;
- надмірне використання пам’яті;
- складність відлагодження.

Переповнення стека відбувається у разі глибокої рекурсії, коли функція викликає сама себе багато разів, може виникнути помилка. Це обмеження

пов'язане з тим, що кожен виклик функції займає місце в стеку, який має обмежений розмір. В даній роботі це призводить до неспроможності алгоритму обробити великі вхідні дані.

Надмірне використання пам'яті пов'язано з тим, що кожен рекурсивний виклик займає певний обсяг пам'яті для зберігання локальних змінних та повернення адреси. Це може призвести до надмірного використання пам'яті при великій кількості рекурсивних викликів.

Відлагодження рекурсивних алгоритмів може бути складнішим через велику кількість викликів функції, що утворюють складну структуру стеку викликів. Це ускладнює пошук і виправлення помилок у коді.

Ітеративний підхід запобігає вищевказані проблеми, оскільки ітеративні алгоритми не залежать від глибини рекурсії, не мають проблем з переповненням стеку викликів, та їх легше відлагоджувати завдяки простій структурі циклів.

У таблиці 3.1 наведено порівняння рекурсивного та ітеративного підходів у реалізації емулятора алгоритмів Маркова.

Таблиця 3.1 – рекурсивного та ітеративного підходів

Критерій	Рекурсія	Ітерація
Використання пам'яті	Високе через збереження стеку викликів	Низьке, постійний обсяг пам'яті
Можливість переповнення стеку	Висока, особливо при великих вхідних даних	Відсутня
Відлагодження	Складніше через глибоку рекурсію	Простіше завдяки лінійній структурі циклів
Читабельність коду	Висока для задач з природним поділом на підзадачі	Може бути менш читабельною для складних алгоритмів

Отже, рекурсивний підхід до реалізації емулятора нормальних алгоритмів Маркова має свої переваги у випадках, коли задача має природний поділ на

підзадачі, і коли кількість рекурсивних викликів є відносно невеликою. Однак, для задач з великими вхідними даними або для алгоритмів, що потребують великої кількості рекурсивних викликів, ітеративний підхід є більш ефективним і надійним. Ітеративні алгоритми використовують постійний обсяг пам'яті та запобігають проблемам з переповненням стеку викликів, що робить їх придатними для обробки великих обсягів даних у контексті НАМ.

Таким чином, для реалізації онлайн-емулятора НАМ з підозрою на можливість великих вхідних даних доцільно використовувати ітеративний підхід. Це забезпечить стабільність роботи програми, оптимальне використання пам'яті та полегшить процес відлагодження. У даній роботі ітеративна реалізація онлайн-емулятора НАМ була обрана як основний підхід через її переваги у продуктивності та надійності при роботі з великими вхідними даними.

3.2 Реалізація основного модуля використовуючи мову програмування JavaScript

Оскільки емулятор НАМ буде реалізовувано в онлайн-середовищі, для його розробки доцільно використовувати переважно мову програмування JavaScript. Це обумовлено декількома ключовими причинами. JavaScript є основною мовою програмування для веб-розробки і підтримується всіма сучасними браузерами, що забезпечує крос-платформену сумісність. Також JavaScript дозволяє створювати інтерактивні і динамічні веб-додатки, що є важливим для зручного та ефективного використання емулятора. Крім того, наявність численних бібліотек і фреймворків для JavaScript спрощує процес розробки та тестування, забезпечуючи високу продуктивність та масштабованість додатка.

Таким чином, використання JavaScript для розробки онлайн-емулятора нормальних алгоритмів Маркова є оптимальним вибором з огляду на його технічні можливості та широке розповсюдження в веб-розробці.

Реалізацію функції markovAlgorithm() зображено у фрагменті

програмного коду на рисунку 3.4.

```
function markovAlgorithm(rules, word, maxIterations = 1000) {
    let index;
    let count = { applied: 0 };

    do {
        // Перевіряємо, чи не досягнуто максимальну кількість ітерацій
        if (count.applied >= maxIterations) {
            console.log("Досягнуто максимальну кількість ітерацій (ітерація)");
            return { word, count };
        }

        // Скидаємо індекс перед кожною ітерацією
        index = -1;
        // Ітерація по всім правилам
        for (let i = 0; i < rules.sample.length; i++) {
            // Зберігаємо поточне правило заміни
            const ai = rules.sample[i];
            const bi = rules.substitution[i];

            // Знаходимо індекс першого входження поточного правила
            index = word.indexOf(ai);
            if (index !== -1) {
                // Збільшуємо лічильник застосованих правил
                count.applied += 1;

                // Перевіряємо, чи закінчується правило на крапку
                if (bi[bi.length - 1] === '.') {
                    // Якщо так, замінюємо та виходимо з функції
                    word = word.slice(0, index) + bi.slice(0, -1) +
word.slice(index + ai.length);
                    //console.log("Rule with dot applied. Exiting...");
                    return { word, count };
                } else {
                    // Інакше замінюємо та переходимо до наступної ітерації
                    word = word.slice(0, index) + bi + word.slice(index +
ai.length);
                    break;
                }
            }
        }
    // Продовжуємо ітерації, поки є можливість застосувати правило
    } while (index !== -1);
    // Повертаємо слово після застосування всіх можливих правил
    return { word, count };
}
```

Рисунок 3.4 – Програмний код реалізації ітеративної функції

В ітераційній реалізації алгоритму використовувалася змінна changed, яка позначала, чи було здійснено заміну в поточній ітерації. Ця змінна дозволяла визначити, коли завершити виконання циклу. В JavaScript-реалізації змінну changed було усунуто, і натомість використовується змінна index, який оновлюється в кожній ітерації циклу. Це зміна була продиктована особливостями мови JavaScript, яка дозволяє більш ефективно використовувати конструкцію do...while з перевіркою умови index != -1, що вказує на наявність підстановок, які можна виконати.

Незважаючи на усунення змінної changed, суть алгоритму залишається незмінною. Алгоритм все ще виконує ітеративну обробку рядка word, шукаючи та замінюючи зразки з масиву A на відповідні заміни з масиву B, аж доки не залишиться жодних можливих підстановок. Це забезпечує однаковий логічний підхід до вирішення задачі, але реалізований через специфіку синтаксису та конструкцій JavaScript.

Протестуємо алгоритм на наступному прикладі. Дано $A=\{a, b\}$. Записати слово в зворотному порядку. НАМ для цього завдання зображено на рисунку 3.5.

1) $*A \rightarrow aX$	7) $*b \rightarrow B$
2) $*B \rightarrow bX$	8) $Bb \rightarrow bB$
3) $XA \rightarrow aX$	9) $Ba \rightarrow aB$
4) $XB \rightarrow bX$	10) $Ab \rightarrow bA$
5) $X \rightarrow .$	11) $Aa \rightarrow aA$
6) $*a \rightarrow A$	12) $\rightarrow *$

Рисунок 3.5 – НАМ для реверсування слова

Для того щоб продемонструвати застосування функції на прикладі заданих правил підстановки, було написано фрагмент коду, що зображено на рисунку 3.6.

```
const rules = {
    sample: ["*A", "*B", "XA", "XB", "X", "*a", "*b", "Bb", "Ba", "Ab", "Aa",
""],
    substitution: ["aX", "bX", "aX", "bX", ".", "A", "B", "bB", "aB", "bA", "aA",
"*"]
};
const word = "aabbab";
console.log("Вхідне слово: " + word);
const result = markovAlgorithm(rules, word);
console.log("Вихідне слово: " + result.word);
console.log("Кількість застосованих правил: " + result.count.applied);
```

Рисунок 3.6 – Застосування основного модуля для вирішення НАМ

У наведеному фрагменті коду визначаються правила підстановки у вигляді двох масивів: sample і substitution, які містять відповідно вхідні шаблони та їх заміни. Початкове слово, на якому будуть виконуватися підстановки, задається як word = "aabbab". Результат виконання такої програми зображено на рисунку 3.7.

```
Вхідне слово: aabbab
Вихідне слово: babbaa
Кількість застосованих правил: 35
```

Рисунок 3.7 – Результат виконання НАМ для реверсування слова

Спостерігається, що результати для цього прикладу є вірними, що підтверджує правильність реалізації ітеративного підходу до виконання алгоритму Маркова.

3.3 Запобігання зациклюванню

У процесі реалізації емулятора нормальних алгоритмів Маркова існує ризик виникнення зациклювання, коли алгоритм може нескінченно повторювати одні й ті ж підстановки без досягнення кінцевого результату. Це може статися через повторювані ітерації, що призводять до циклічних підстановок. Зациклювання є критичною проблемою, оскільки воно

призводить до неефективного використання обчислювальних ресурсів та робить неможливим отримання результату.

Наприклад, якщо запустити вищезгаданий алгоритм (рисунок 3.5) на порожньому слові, може виникнути проблема зациклювання. Це відбувається через правило підстановки $\rightarrow$ *, яке додає символ зірочки до порожнього рядка. В результаті алгоритм буде постійно повторювати таку підстановку, оскільки ніякі нові символи не з’являтимуться у вхідному рядку. Це призводить до нескінченного додавання символу зірочки, що викликає зациклення.

Одним з ефективних методів запобігання зациклюванню є встановлення максимальної кількості ітерацій. Це означає, що алгоритм припиняє виконання після досягнення визначеного ліміту ітерацій, навіть якщо не було досягнуто кінцевого результату. Встановлення ліміту ітерацій гарантує, що алгоритм не зациклиться безкінечно, а також дозволяє обмежити використання обчислювальних ресурсів, зокрема часу виконання та пам'яті, запобігаючи надмірному використанню процесорного часу. Окрім того, додавання ліміту ітерацій є простим у реалізації і не потребує значних змін у структурі алгоритму.

У розділі 2.4 необмеженість кількості ітерацій та правил розглядалася як перевага десктопного емулятора нормальних алгоритмів Маркова. Це дозволяє здійснювати підстановки без жорстких обмежень, забезпечуючи гнучкість та універсальність алгоритмів. Однак, ця необмеженість має свої обмеження у практичному застосуванні, особливо в контексті десктопного та веб-емуляторів.

У десктопному емуляторі НАМ немає можливості запустити алгоритм миттєво та отримати результат негайно. Виконання алгоритму здійснюється покроково, що дозволяє користувачу слідкувати за кожною підстановкою і контролювати процес виконання. Хоча можна налаштувати швидкість виконання, це все ще буде покроковий процес, який не забезпечує миттєвого результату. Крім того, десктопний емулятор використовує обчислювальні

ресурси машини, на якій він запущений. Це означає, що продуктивність алгоритму залежить від потужності апаратного забезпечення користувача. У випадку великих вхідних даних та складних правил підстановок, необмежена кількість ітерацій може призвести до значного навантаження на систему, що може викликати її збої.

У випадку веб-емулятора планується додати можливість виконання алгоритму як покроково, так і миттєво. Це забезпечить більшу гнучкість та зручність для користувачів. Миттєве виконання алгоритму дозволить отримати результат значно швидше, що є важливим для аналізу великих обсягів даних або складних підстановок. Однак, у зв'язку з необмеженою кількістю ітерацій, виникає ризик зациклювання алгоритму, що може призвести до збоїв програми. Так як програма буде написана на JavaScript, всі операції виконуються на стороні клієнта, тобто в браузері користувача. Узагальнюючи, можна сказати, що веб-емулятор нормальних алгоритмів Маркова, виконуючи всі обчислення на стороні клієнта, пропонує оптимальний баланс між доступністю, продуктивністю та надійністю. Встановлення максимального ліміту ітерацій забезпечує захист від зациклювання, що робить його ефективним інструментом для широкого спектру застосувань.

На рисунку 3.8 зображена поведінка програми при встановленому значенні максимальної кількості ітерацій 100.

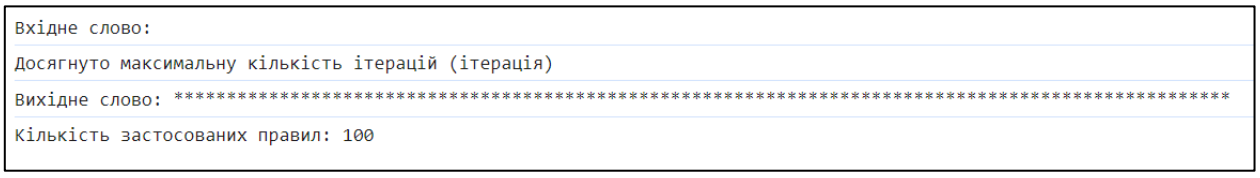

Рисунок 3.8 – Спрацьоване обмеження в 100 ітерацій

У повній версії емулятора планується додати повзунок із різними значеннями, що надасть користувачеві можливість самостійно встановлювати максимальну кількість ітерацій. Це забезпечить додаткову гнучкість у

використанні алгоритмів Маркова, дозволяючи адаптувати виконання алгоритму до конкретних потреб користувача та обмежень системи.

Для забезпечення ефективної роботи емулятора необхідно визначити максимальну кількість ітерацій, яка буде достатньою для обробки відносно великих вхідних даних, але не буде створювати значного навантаження на сервер. Це вимагає проведення серії тестів та аналізу результатів для різних обсягів вхідних даних і правил підстановки. Враховуючи результати тестів, можна буде встановити рекомендації щодо оптимальної кількості ітерацій для різних умов.

3.4 Оцінка складності алгоритмів

У процесі розробки емулятора нормальних алгоритмів Маркова важливим аспектом є оцінка складності алгоритму. Оцінка складності дозволяє визначити, як ефективно алгоритм обробляє вхідні дані різних розмірів та виявити можливі проблеми з продуктивністю. Для цього було розроблено програмне забезпечення, яке дозволяє здійснювати збір емпіричних даних, аналіз отриманих результатів та побудову теоретичних моделей.

Метою оцінки складності алгоритму є виявлення залежності кількості ітерацій алгоритму від довжини вхідного слова. Це дозволяє зрозуміти, наскільки ефективно алгоритм працює при збільшенні розміру вхідних даних, та які оптимізації можуть бути застосовані для покращення його продуктивності.

У традиційному аналізі складності алгоритмів широко використовують O-нотацію (Big O notation) для опису асимптотичної поведінки алгоритму при збільшенні розміру вхідних даних. Big O (у своїй найпростішій формі) зменшує порівняння між алгоритмами до однієї змінної. Ця змінна вибирається на основі спостережень або припущень. Наприклад, алгоритми сортування зазвичай порівнюються на основі операцій порівняння, тобто порівняння двох параметрів для визначення їх відносного впорядкування. При

цьому передбачається, що порівняння "коштує" дорого. О-нотація дозволяє зрозуміти, як змінюється продуктивність алгоритму з ростом обсягу даних, але вона не враховує специфічні деталі та можливі варіації в поведінці алгоритму, що можуть мати значний вплив на його ефективність у реальних умовах.

Однак, у випадку нормальних алгоритмів Маркова, застосування класичної О-нотації може бути складним і не завжди точним з кількох причин. НАМ можуть демонструвати нестабільну поведінку залежно від конкретних правил підстановок та структури вхідного слова, що призводить до значних варіацій у кількості ітерацій навіть для вхідних даних однакового розміру. Це означає, що навіть незначна зміна вхідних даних може призвести до суттєвих змін у часі виконання алгоритму. У деяких випадках НАМ можуть зациклюватися або виконувати велику кількість ітерацій без досягнення кінцевого результату, що робить оцінку складності складнішою і менш передбачуваною. Крім того, складність алгоритму може залежати не тільки від довжини вхідного слова, але й від інших факторів, таких як структура слова та специфічні правила підстановок. Це додає додаткову складність до оцінки алгоритму, оскільки ці фактори можуть значно впливати на його поведінку.

Зважаючи на ці обмеження, для оцінки складності НАМ було обрано регресійний метод. Регресійний аналіз дозволяє емпірично визначити залежність між розміром вхідного слова та кількістю ітерацій алгоритму. Цей підхід має декілька переваг. Регресійний аналіз базується на реальних даних, що дозволяє отримати точнішу модель залежності. Крім того, регресійний метод підтримує різні типи функцій із відповідними коефіцієнтами, включаючи лінійні, квадратичні, кубічні, експоненціальні та логарифмічні. Це дозволяє підібрати найбільш відповідну модель для конкретного алгоритму з високою точністю, забезпечуючи краще розуміння складності та ефективності алгоритму.

Збір емпіричних даних здійснюється шляхом виконання алгоритму Маркова для слів різної довжини. Для кожної довжини слова комбінаторно перебираються всі можливі варіанти складання слова з введеного алфавіту.

Після цього для кожної довжини визначається максимальна кількість ітерацій, необхідна для досягнення кінцевого результату. Таким чином, для кожної довжини слова вибирається «найгірший» варіант, де кількість ітерацій була найбільшою.

Для аналізу зібраних емпіричних даних використовується JS-бібліотека regression. Бібліотека забезпечує проведення регресійного аналізу, дозволяючи визначити найкращу теоретичну модель для опису залежності між довжиною вхідного слова та кількістю ітерацій. Бібліотека підтримує кілька типів функцій, включаючи лінійні, квадратичні, кубічні, експоненціальні та логарифмічні моделі.

Регресійний аналіз є статистичним методом, який дозволяє визначити залежність між змінними. У контексті даного дослідження змінною незалежною є довжина слова, а залежною – кількість ітерацій алгоритму. Використовуючи регресійний аналіз, можна визначити, яка з теоретичних моделей (лінійна, квадратична, кубічна, експоненціальна або логарифмічна) найкраще підходить для опису отриманих даних.

Лінійна регресія передбачає, що залежність між змінними можна описати рівнянням прямої лінії. У даному випадку рівняння представлено як (формула 3.1):

$$y = a \cdot x + b, \tag{3.1}$$

де y – кількість ітерацій, x – довжина слова, a та b – коефіцієнти регресії.

Квадратична регресія використовує параболічну залежність між змінними (формула 3.2):

$$y = a \cdot x^2 + b \cdot x + \text{c}. \tag{3.2}$$

Кубічна регресія описує залежність за допомогою кубічної функції (формула 3.3):

$$y = a \cdot x^3 + b \cdot x^2 + \text{c} \cdot x \ + d. \tag{3.3}$$

Експоненціальна регресія описує залежність за допомогою експоненціальної функції (формула 3.4):

$$y = a \cdot e^{b \cdot x}. \quad (3.4)$$

Логарифмічна регресія використовує логарифмічну залежність (формула 3.5):

$$y = a \cdot ln(x) + b. \quad (3.5)$$

Однією з ключових задач регресійного аналізу є визначення коефіцієнтів регресійної моделі. Ці коефіцієнти визначаються шляхом мінімізації різниці між спостережуваними значеннями (емпіричними даними) та значеннями, передбаченими моделлю. Бібліотека regression використовує метод найменших квадратів для визначення коефіцієнтів. Цей метод мінімізує суму квадратів різниць між спостережуваними та передбаченими значеннями, забезпечуючи найбільш точну відповідність моделі до даних.

Під час розробки та тестування алгоритму з'ясувалося, що використання бібліотеки regression значно підвищує точність визначення коефіцієнтів у порівнянні з ручними методами. Бібліотека забезпечує автоматичне визначення коефіцієнтів за допомогою методу найменших квадратів, що дозволяє уникнути помилок, пов'язаних з ручними обчисленнями. Це робить процес аналізу більш надійним та точним, забезпечуючи кращу відповідність моделі емпіричним даним.

Після проведення регресійного аналізу та визначення теоретичної моделі, результати відображаються у вигляді графіків. На графіку по осі X відкладається довжина слова, а по осі Y – кількість ітерацій. Емпіричні дані відображаються як точки, а теоретична модель – у вигляді лінії.

Проведемо тестування на наступному прикладі. Дано *A={1, 0}*.

Дублювати слово та розділити символом #. НАМ для цього завдання зображено на рисунку 3.9.

1) *1 → 1x*	7) y0 → 0y
2) *0 → 0y*	8) x# → #1
3) * → #	9) y# → #0
4) x1 → 1x	10) # → #.
5) x0 → 0x	11) → *
6) y1 → 1y	

Рисунок 3.9 – НАМ дублювання бінарного слова

Проведемо тестування НАМ вказаного на рисунку 3.9 на слові «101». Результати тестування наведено на рисунку 3.10.

$$_101 \rightarrow \underline{*\,1}01 \rightarrow 1x\,\underline{*\,0}1 \rightarrow 1x0y\,\underline{*\,1} \rightarrow 1x0y1x\,\underline{*} \rightarrow 1\underline{x0}y1x\# \rightarrow 10\underline{xy1}x\# \rightarrow$$
$$\rightarrow 10\underline{xy1}x\# \rightarrow 10\underline{x1}yx\# \rightarrow 101xy\underline{x\#} \rightarrow 101\underline{xy\#}1 \rightarrow 101\underline{x\#}01 \rightarrow$$
$$\rightarrow 101\underline{\#}101 \rightarrow 101\#101$$

Рисунок 3.10 – Тестування НАМ дублювання бінарного слова

До цього прикладу було застосовано метод регресійного аналізу, описаний вище.

Результати роботи програми, яка відображає зображено графіки із функціями тимчасової складності НАМ для дублювання слова на рисунку 3.11.

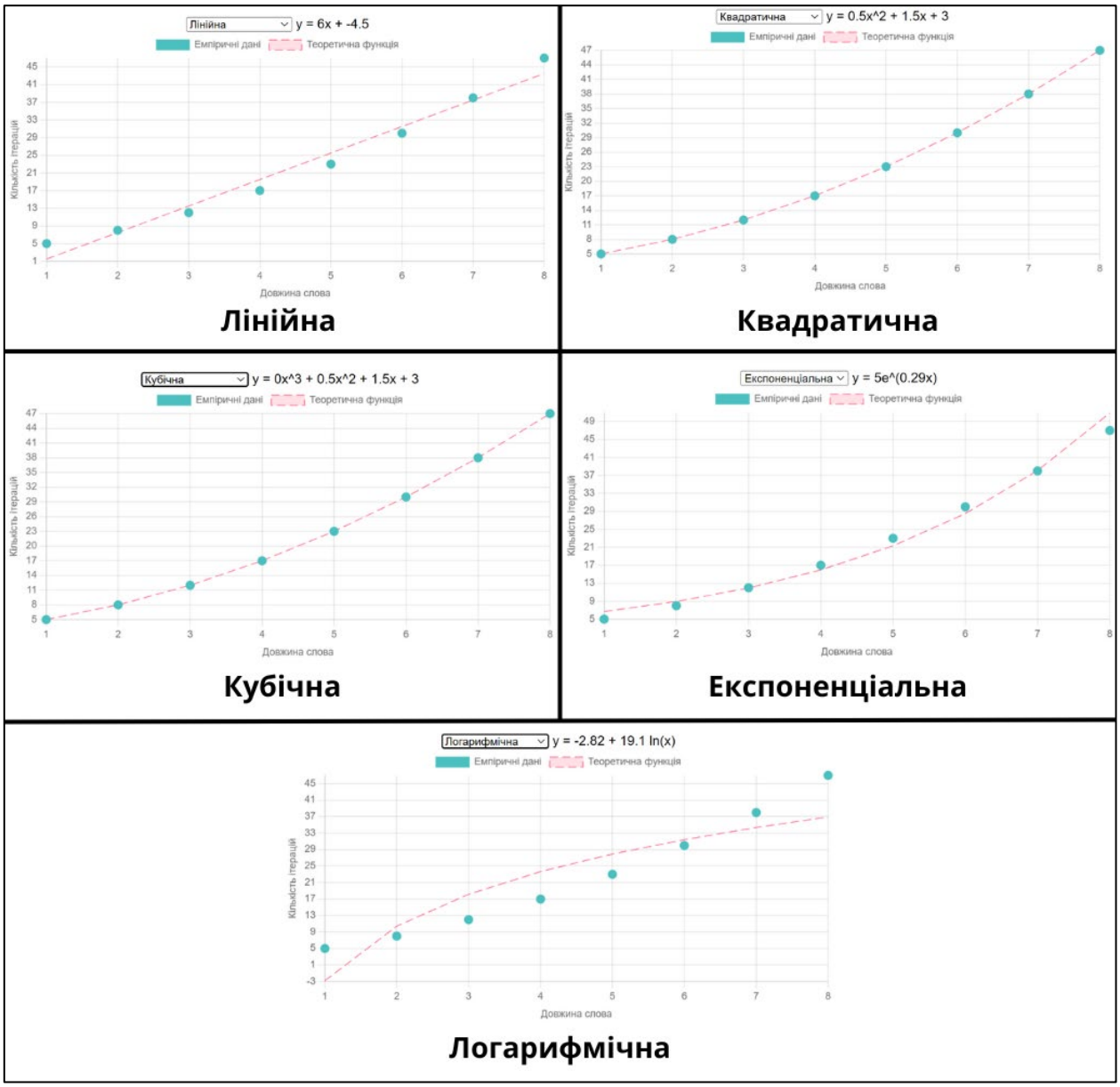

Рисунок 3.11 – Графіки складності HAM для дублювання слова

Як можна побачити з рисунку, для цього конкретного HAM, ідеальну точність показують квадратична та кубічна функції. Лінійна модель демонструє певний рівень точності, але має деякі погрішності. Також можна побачити, що складність алгоритму зростає повільніше, ніж це припускає експоненціальна залежність. Логарифмічна модель виявилася найменш точною для даного HAM.

Результати такого аналізу є важливими для подальшої розробки та вдосконалення HAM, забезпечуючи їх ефективну роботу та стабільність у різних умовах.

Функцію побудови графіку із емпірично отриманими даними та спроби автоматичного визначення складності (побудови теоретичної функції) алгоритму буде додано до інтерфейсу користувача. Це дозволить користувачам безпосередньо візуалізувати результати роботи алгоритму та оцінити його складність у зручний спосіб. Однак, ця задача є важкою в прорахуванні, оскільки для побудови такого графіку потрібно комбінаторно перебрати всі можливі варіанти слів різної довжини. Кількість можливих варіантів слів можна розрахувати за допомогою формули розміщення з повтореннями (формула 3.6):

$$\overline{A_n^k} = n^k, \tag{3.6}$$

де n – кількість символів в алфавіті, а k – довжина слова.

Щоб уникнути надмірного навантаження на систему, інтерфейсом буде реалізовано обмеження: довжина слова від 1 до 8, а кількість символів в алфавіті не більше 5. Це забезпечить зручне користування програмою, зберігаючи при цьому її продуктивність.

З урахуванням встановлених обмежень, максимальна кількість слів, які потрібно перебрати, становитиме (формула 3.7):

$$\sum_{k=1}^{8} 5^k \tag{3.7}$$

Виконаємо обчислення (формула 3.8):

$$5^1 + 5^2 + 5^3 + 5^4 + 5^5 + 5^6 + 5^7 + 5^8 = 5 + 25 + 125 + 625 + 3125 + \\ +15625 + 78125 + 390625 = 483280 \tag{3.8}$$

Отже, максимум переберется 483280 слів.

Важливо зазначити, що це обмеження в довжині алфавіту буде введено лише для побудови графіку складності алгоритму і не впливає на роботу основного модуля алгоритму.

Щодо рекомендованого обмеження максимально кількості ітерацій, пропонується встановлення обмеження в один мільйон циклів. Справедливо, що для виконання НАМ дублювання (копіювання) слова довжиною n необхідно здійснити $\frac{n^2}{2} + 1.5n + 3$ ітерацій (рисунок 3.11). Отже, для виконання ітерацій близько до одного мільйона необхідно ввести слово довжиною 1412. При цьому кількість ітерацій складе 998993.

Результати виконання такого алгоритму програмою зображено на рисунку 3.12.

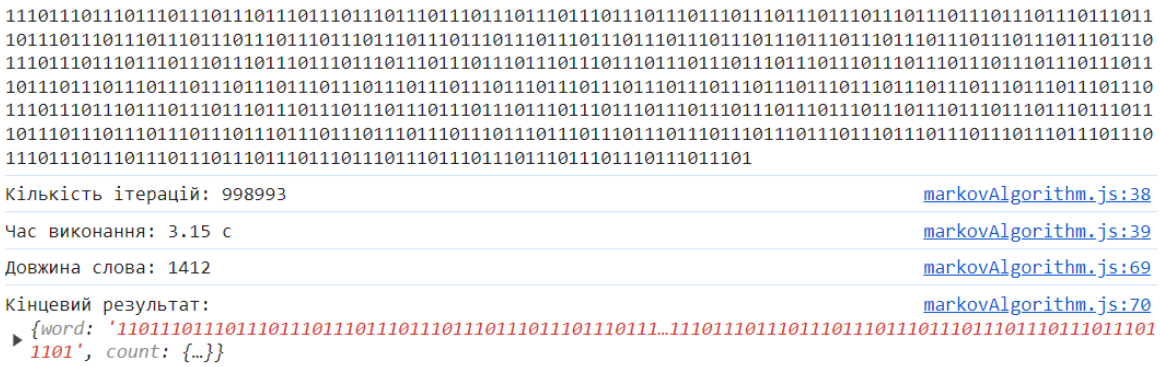

Рисунок 3.12 – Результат виконання НАМ із заміром часу

Отже, щоб пройти близько мільйону ітерацій, програмі потребувалося 3.15 секунд, що є прийнятним результатом. В інтерфейсі користувача буде додано повзунок, на якому можна буде встановити обмеження кількості ітерацій від п’яти тисяч до одного мільйону. Також буде додано режим необмеженої кількості ітерацій із повідомленням-застереженням для користувача, що це може призвести до збою програми в деяких випадках.

Оскільки програма написана повністю на JavaScript, вона буде виконуватися на стороні клієнта. Це означає, що всі обчислення будуть проводитися в браузері користувача, а не на сервері, що запобігає виникненню проблем з сервером та забезпечує його безпеку. Такий підхід також дозволяє зберегти серверні ресурси, забезпечуючи швидке та ефективне виконання алгоритму без зайвого навантаження на інфраструктуру сервера.

4 РОЗРОБКА ІНТЕРФЕЙСУ КОРИСТУВАЧА

4.1 Діаграма варіантів використання

Діаграма варіантів використання відображає функціональну поведінку системи, зображуючи взаємодії між користувачами та системою, яку розробляють [17]. Актори на діаграмі представляють користувачів системи, а варіанти використання описують поведінку системи стосовно акторів.

Діаграму варіантів використання для емулятора нормальних алгоритмів Маркова наведено на рисунку 4.1.

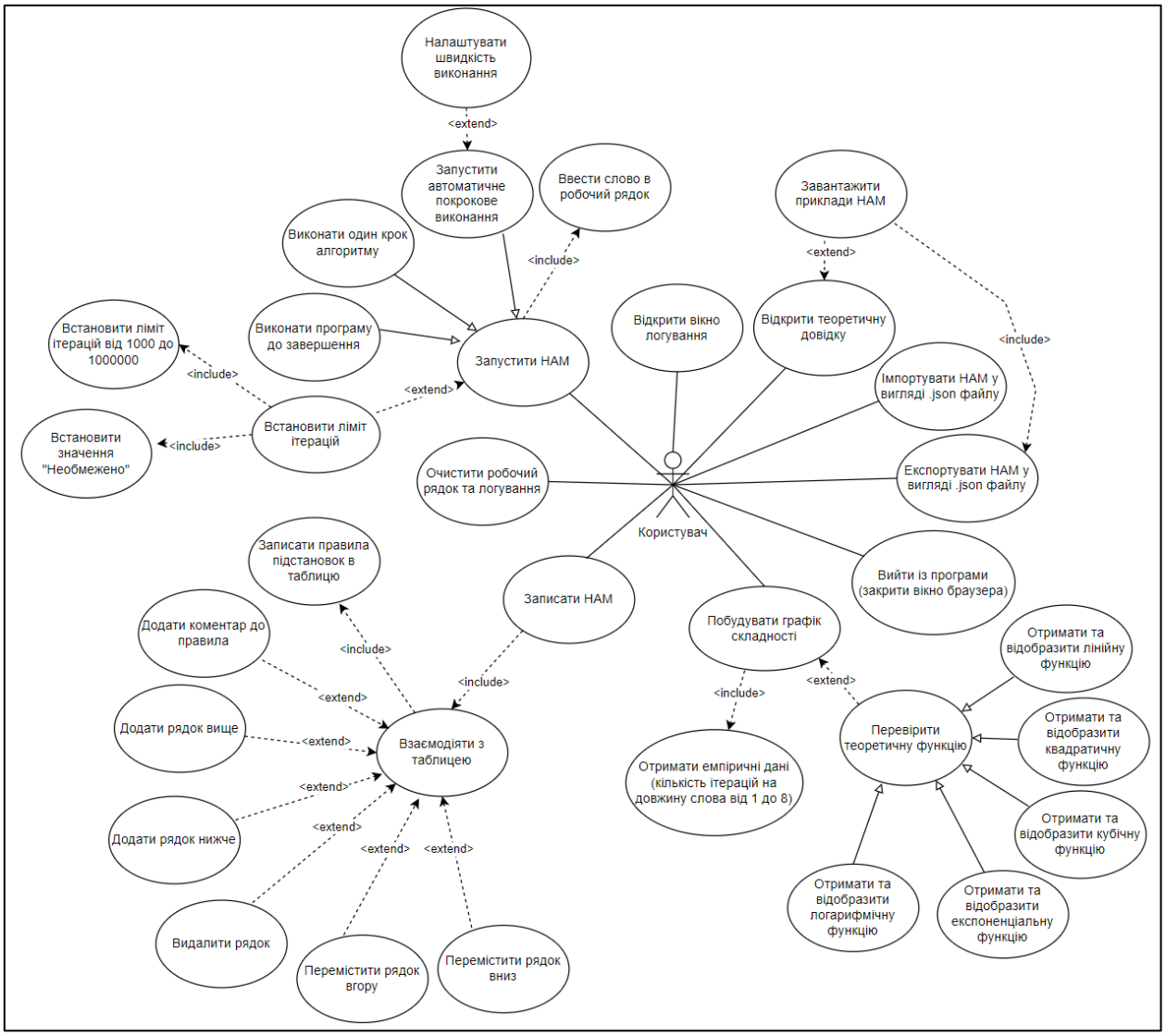

Рисунок 4.1 – Діаграма використання для емулятора НАМ

Діаграма варіантів використання допомагає визначити функціональні вимоги до системи та зрозуміти, які вимоги повинні бути виконані.

Користувач є головним актором у наведеній діаграмі використання, що зображає взаємодію з емулятором нормальних алгоритмів Маркова. Користувач представляє оператора, який використовує програму для запуску та налагодження алгоритмів. Інтерфейс надає можливість користувачеві вводити слова в робочий рядок, записувати правила підстановок в таблицю, а також встановлювати різні параметри для виконання алгоритмів.

Після завантаження емулятора, користувач може вибрати серед декількох функцій, таких як запуск алгоритму, виконання одного кроку алгоритму, або запуск автоматичного покрокового виконання. Ці опції дозволяють користувачеві детально контролювати процес обчислення, встановлювати швидкість виконання та ліміт ітерацій. Серед додаткових можливостей – очищення робочого рядка та логування, завантаження прикладів НАМ, а також імпорт та експорт даних у вигляді .json файлів.

Взаємодія з таблицею дозволяє користувачеві зручним чином додавати, видаляти та переміщувати рядки, а також додавати коментарі до правил підстановок. Ці функції розширюють можливості користувача щодо організації та налаштування алгоритмів, дозволяючи редагувати та модифікувати правила.

У користувача є можливість побудувати графік складності алгоритмів, отримуючи емпіричні дані про кількість ітерацій на різну довжину слова. Після цього він може перевірити теоретичну функцію, вибравши серед різних варіантів (лінійна, квадратична, кубічна, експоненціальна або логарифмічна). Це дозволяє оцінити продуктивність алгоритмів та порівняти результати з теоретичними моделями.

Таким чином, діаграма випадків використання детально ілюструє всі можливості та функції, які надає емулятор нормальних алгоритмів Маркова, показуючи, як користувач може взаємодіяти з програмою для досягнення своїх цілей.

4.2 Інтерактивна таблиця правил підстановок

Основною метою створення інтерактивної таблиці є забезпечення користувача зручним інструментом для роботи з правилами підстановок нормальних алгоритмів Маркова. Це дозволить користувачеві швидко та легко вводити необхідні дані, а також модифікувати їх у процесі роботи. Інтерактивна таблиця сприятиме підвищенню ефективності роботи користувача та полегшить налагодження алгоритмів.

Отже, було розроблено таблицю із можливістю записувати зразок та заміну для правила підстановки, а також коментар до цього правила. Було додано кнопки управління із фунціоналом додавання нового рядку до таблиці зверху або знизу від виділеного (або від першого за замовчуванням), видалення виділеного рядка, а також переміщення виділеного рядка вгору або вниз на один рядок. На даному етапі розробки програма виглядає як на рисунку 4.2.

Рисунок 4.2 – Інтерактивна таблиця

Із рисунка 4.2 можна побачити, що при наведенні курсору на кнопку, його іконка змінюється, а також відображаються пояснення до кнопки, що робить інтерфейс більш інтуїтивно зрозумілим і зручним для користувачів.

Видимість таблиці включає 15 рядків, та після додавання більшої кількості рядків з'являються вертикальні та горизонтальні слайдери, що дозволяють прокручувати таблицю для доступу до всіх її частин. Таблиця розташована в окремому контейнері, який забезпечує незалежну прокрутку як у вертикальному, так і у горизонтальному напрямку (рисунок 4.3).

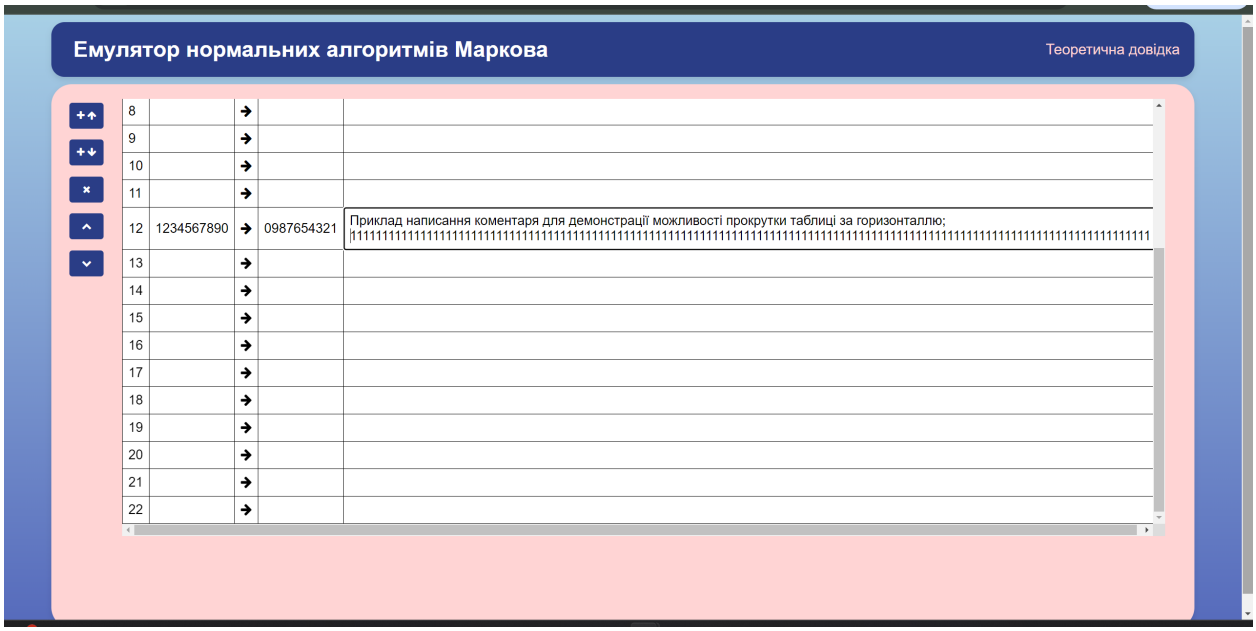

Рисунок 4.3 – Демонстрація прокрутки таблиці

Для зразка та заміни вирівнювання тексту було встановлено по центру, а для коментаря – по лівому краю.

4.3 Робочий рядок та елементи контролю виконання алгоритму

В процесі розробки було додано поле для введення робочого рядка, що дозволяє користувачам вводити текст для обробки алгоритмом.

Додано кнопки для управління виконанням алгоритму, а саме: кнопку запуску алгоритму, кнопку для виконання одного кроку алгоритму, запуску алгоритму до завершення та очищення робочого рядка.

Було додано регулятор швидкості виконання алгоритму, який включає «Швидкість виконання» та повзунок. Це дозволяє користувачам налаштовувати швидкість виконання алгоритму відповідно до їхніх потреб.

Додано випадаючий список для встановлення максимальної кількості ітерацій, який включає мітку «Максимальна кількість ітерацій:» та сам список, що містить варіанти від 1000 до необмеженої кількості ітерацій. Це дозволяє користувачам контролювати обмеження на кількість ітерацій, що запобігає можливим нескінченним циклам і покращує продуктивність алгоритму.

Процес автоматичного покрокового виконання алгоритму програмою зображено на рисунку 4.4.

Рисунок 4.4 – Демонстрація виконання алгоритму

На рисунку 4.4 можна спостерігати, що програма підсвічує зеленим кольором правило підстановки, що виконується в даний момент. Це дозволяє користувачам детальніше відстежувати процес виконання алгоритму.

Автоматичне виконання алгоритму можна призупинити, натиснувши на ту ж саму кнопку, за допомогою якої він був запущений. Іконка на кнопці змінюється на іконку паузи, що дозволяє розпізнати режим зупинки. Після цього користувач може продовжити виконання алгоритму будь-яким з доступних методів. За необхідністю, швидкість виконання можливо змінювати протягом роботи алгоритму.

4.4 Побудова графіка складності алгоритму

Функція "Побудувати графік складності" призначена для візуалізації складності виконання нормальних алгоритмів Маркова. Натиснувши цю кнопку, користувач запускає процес побудови графіка, який включає кілька ключових етапів.

Спочатку перевіряються введені дані. Програма перевіряє наявність робочого рядка та правил підстановки. Якщо робочий рядок не введений, його поле підсвічується червоною рамкою, і користувач отримує повідомлення про необхідність введення даних. Аналогічно, якщо правила підстановки не задані або всі правила порожні, таблиця правил підсвічується червоною рамкою, і виводиться відповідне попередження.

Далі перевіряється, чи не перевищує кількість унікальних символів у робочому рядку допустимий максимум (п'ять символів). Якщо кількість символів перевищує цей ліміт, користувач отримує відповідне повідомлення (див. розділ 3.4).

Після успішного проходження всіх перевірок, програма починає обчислення кількості ітерацій для різних довжин слів. Виконується генерація всіх можливих комбінацій символів заданої довжини, і для кожної комбінації запускається алгоритм Маркова. Програма відслідковує кількість ітерацій, необхідних для завершення роботи алгоритму для кожної комбінації.

Якщо алгоритм досягає встановленого максимального числа ітерацій для певної комбінації символів, виводиться попередження, і побудова графіка зупиняється.

Після обчислення даних для всіх можливих комбінацій, програма будує графік. Графік відображає залежність кількості ітерацій від довжини слова. Для кожної довжини слова на графіку зображується точка, яка показує максимальну кількість ітерацій, що була необхідна для завершення алгоритму.

Графік будується за допомогою бібліотеки Chart.js, що забезпечує якісну та інтерактивну візуалізацію даних [18]. Користувач також має можливість вибрати тип теоретичної функції, що буде порівнюватися з емпіричними

даними, для оцінки складності алгоритму.

Приклад використання такої функції зображено на рисунку 4.5.

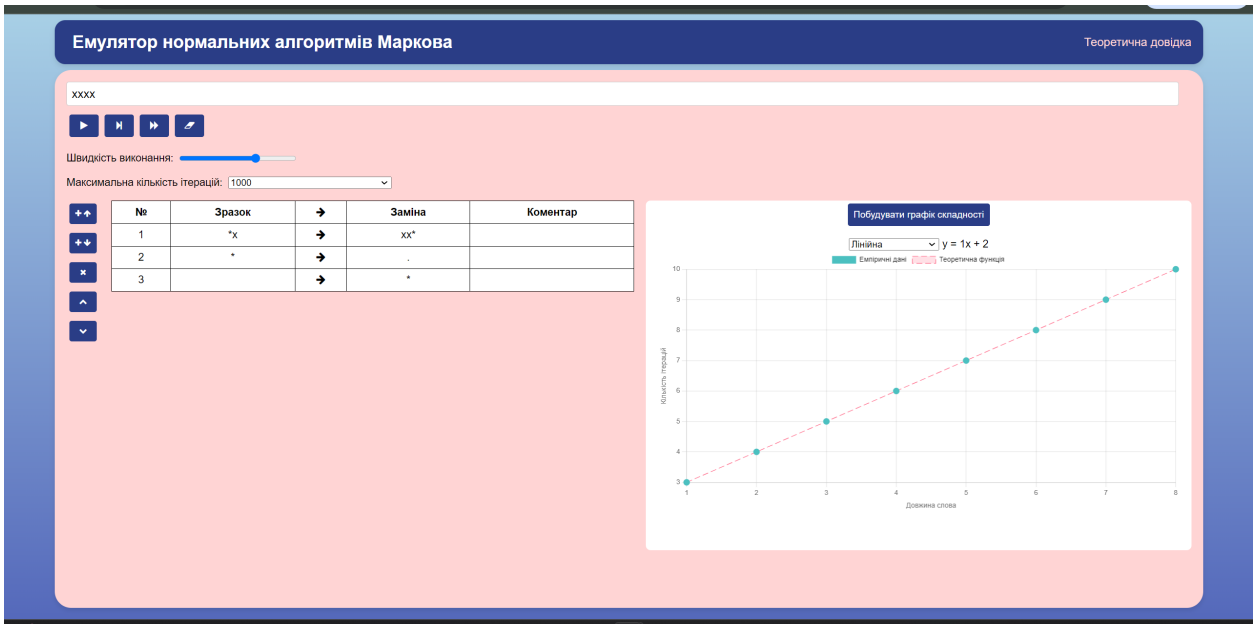

Рисунок 4.5 – Демонстрація побудови графіку

Таким чином, функція "Побудувати графік складності" забезпечує користувачам потужний інструмент для аналізу продуктивності нормальних алгоритмів Маркова, дозволяючи наочно оцінити їх складність та ефективність. Але важливо зауважити, що це не є основним інструментом програми, а лише додатковим. Він може не підходити для специфічних алгоритмів або особливих випадків, коли поведінка алгоритму відрізняється від типових сценаріїв. Тому користувачам слід використовувати цей інструмент з обережністю та враховувати можливі обмеження під час аналізу складності алгоритмів.

4.5 Функції експорту та імпорту файлів для НАМ

Основною метою розробки функцій експорту та імпорту є надання користувачам можливості зберігати поточні правила підстановок та робочий рядок у файл та завантажувати ці дані з файлу у програму. Це забезпечує безперервність роботи та дозволяє користувачам зберігати свої налаштування

для подальшого використання, а також ділитися ними з іншими користувачами або переносити між різними пристроями.

В даній роботі, як формат збереження даних було використано JSON. JSON (JavaScript Object Notation) є текстовим форматом для збереження та обміну даними. Він став стандартом для багатьох веб-додатків через свою простоту та гнучкість [19]. JSON підтримує структури даних, що використовуються в мовах програмування, включаючи масиви та об'єкти, що робить його ідеальним для передачі складних даних між клієнтом і сервером. JSON є компактним форматом, що дозволяє ефективно передавати дані через мережу.

Функція експорту дозволяє користувачам зберігати правила підстановок та робочий рядок у файл формату JSON. Для цього було додано кнопку "Експортувати", яка активує процес створення файлу. При натисканні на цю кнопку збираються всі необхідні дані з таблиці правил та робочого рядка, які потім конвертуються у формат JSON. JSON-об'єкт зберігається як текстовий файл, який користувач може завантажити на свій пристрій.

Функція імпорту дозволяє завантажувати правила підстановок та робочий рядок з файлу формату JSON. Для цього було додано кнопку "Імпортувати", яка відкриває діалогове вікно для вибору файлу. Після вибору файлу дані з нього зчитуються та завантажуються у програму. Це включає оновлення таблиці правил та робочого рядка відповідно до завантажених даних. Використання JSON забезпечує легкість і точність передачі даних між різними сеансами роботи або пристроями.

При імпорті даних з файлу необхідно правильно обробити JSON-об'єкт та оновити інтерфейс програми. Функція завантаження таблиці правил з файлу оновлює дані у відповідних полях інтерфейсу, а робочий рядок оновлюється з JSON-об'єкта. Це забезпечує коректне відображення збережених даних у програмі.

На рисунку 4.6 зображено додані відповідні кнопки, а також поведінку програми при натисканні кнопки «Імпортувати».

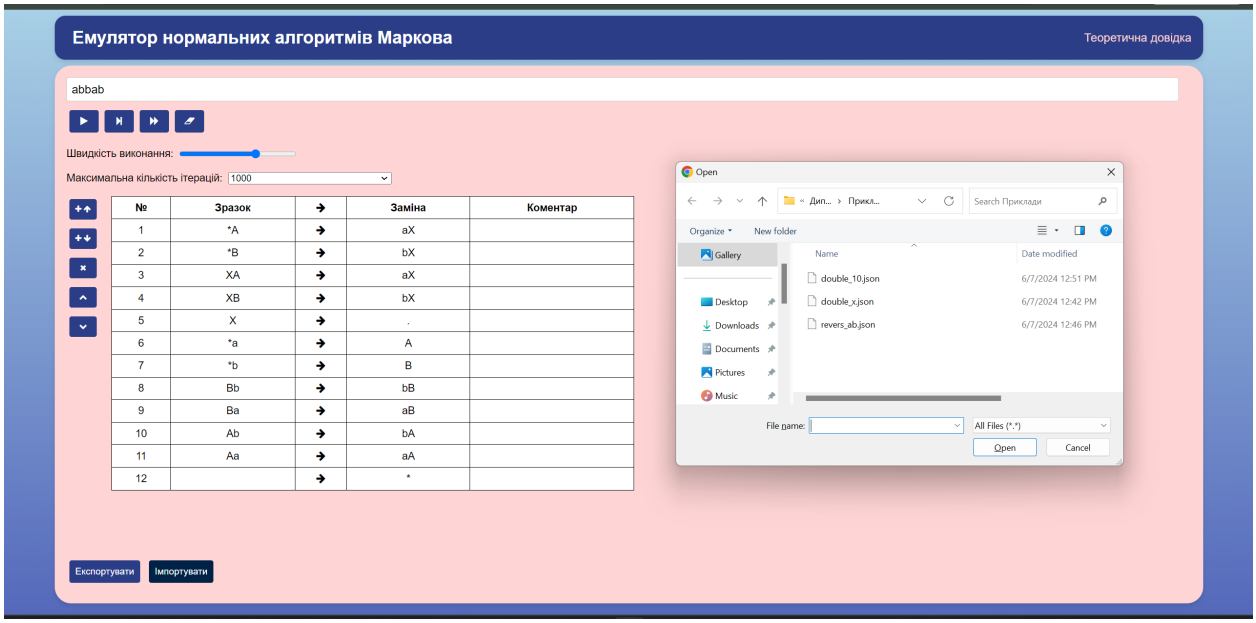

Рисунок 4.6 – Демонстрація імпорту НАМ із файла JSON

Приклад вмісту JSON-файлу зображено на рисунку 4.7.

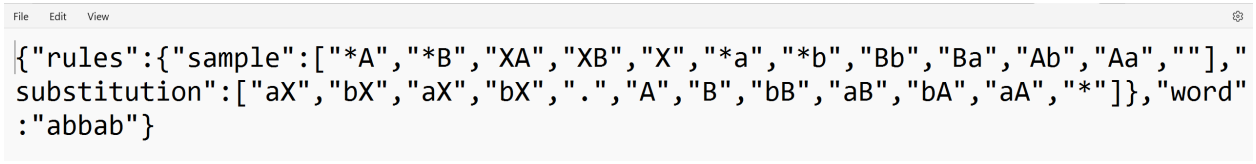

Рисунок 4.7 – Відкритий у блокноті JSON-файл

Таким чином, функції експорту та імпорту забезпечують зручний спосіб збереження та завантаження даних, що дозволяє користувачам зберігати свої налаштування та легко переносити їх між сеансами роботи або пристроями.

4.6 Логування

Основною метою логування є надання користувачам інструменту для відстеження та аналізу кожного кроку виконання алгоритмів. Це включає збереження інформації про застосовані правила підстановок, зміни у робочому рядку та кількість ітерацій. Логи допомагають користувачам розуміти, як алгоритм працює, і знаходити можливі помилки або неефективності.

У процесі виконання алгоритму на кожному кроці збирається інформація про застосоване правило підстановки, попередній та новий стан робочого рядка, а також кількість виконаних ітерацій.

Для відображення логів у програмі було розроблено спеціальний інтерфейс. При натисканні кнопки "Логування" відкривається контейнер для логів, де користувач може побачити всі збережені записи. Кожен запис включає номер ітерації, індекс застосованого правила, саме правило, попередній та новий стан робочого рядка, а також підсвічування змінених символів.

Логи оновлюються в режимі реального часу під час виконання алгоритму. Це забезпечує динамічний та інтерактивний спосіб відстеження виконання алгоритмів.

Приклад логування зображено на рисунку 4.8.

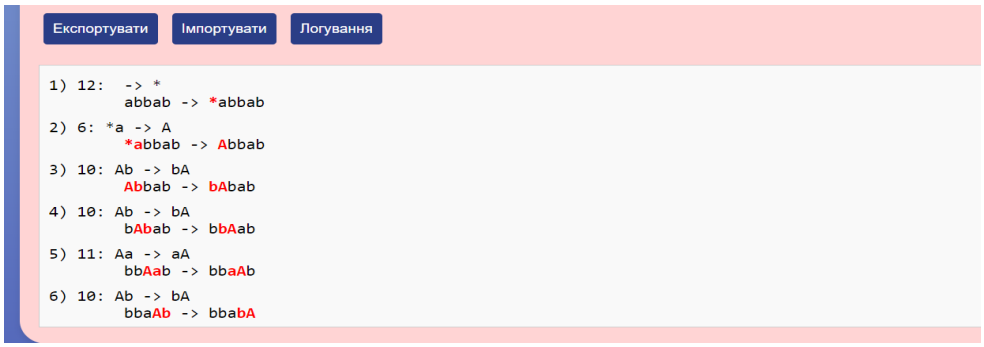

Рисунок 4.8 – Демонстрація вікна логування

Таким чином, функції логування у програмі забезпечують детальний та зручний спосіб відстеження виконання нормальних алгоритмів Маркова. Вони надають користувачам можливість бачити кожен крок обчислень, аналізувати застосовані правила та оптимізувати алгоритми для досягнення кращих результатів.

4.7 Теоретична довідка та інструкція користувача

У процессі розробки було додано сторінку "Теоретична довідка" до

програми для забезпечення користувачів основними теоретичними відомостями про нормальні алгоритми Маркова та інструкцією з використання розробленого емулятора. Ця сторінка служить довідковим матеріалом і допомагає користувачам краще зрозуміти, як працює емулятор та як ефективно ним користуватися.

На рисунку 4.9 зображено сторінку із теоретичною довідкою.

Рисунок 4.9 – Сторінка «Теоретична довідка»

Сторінка "Теоретична довідка" була створена у вигляді HTML-документу під назвою theory.html. Вона містить структуровану інформацію, стилізовану за допомогою CSS, для забезпечення зручного та зрозумілого інтерфейсу. Навігація між головною сторінкою та сторінкою "Теоретична довідка" здійснюється через відповідні посилання у меню навігації.

Додавання цієї сторінки до програми покращує користувацький досвід, забезпечуючи необхідну підтримку та навчальні матеріали, що допомагають користувачам ефективно використовувати емулятор нормальних алгоритмів Маркова.

5 РОЗМІЩЕННЯ ПРОГРАМИ В МЕРЕЖІ ІНТЕРНЕТ

Для забезпечення доступності розробленої програми для широкого кола користувачів було вирішено розмістити її в мережі Інтернет за допомогою платформи GitHub Pages [20]. Ця платформа надає можливість безкоштовного хостингу статичних веб-сторінок безпосередньо з репозиторіїв GitHub, що є прийнятним рішенням для розміщення розробленого емулятора.

На першому етапі необхідно створити новий репозиторій на GitHub. Для цього користувач повинен увійти у свій обліковий запис на GitHub і перейти до створення нового репозиторію, де вказується його назва і надається короткий опис проекту. Вибір опції "Public" забезпечує відкритий доступ до репозиторію, що дозволяє будь-якому користувачу Інтернету переглядати та взаємодіяти з проектом. Завершення створення репозиторію підтверджується натисканням кнопки "Create repository".

Після створення репозиторію необхідно завантажити до нього всі файли проекту. Це включає HTML-файли, файли скриптів та стилів, а також JSON-файли з прикладами. Завантаження файлів може здійснюватися шляхом клонування репозиторію на локальний комп'ютер і подальшого додавання файлів через командний рядок або термінал. При цьому важливо забезпечити, щоб всі необхідні файли були присутні в репозиторії для коректної роботи програми.

Для активації GitHub Pages користувач переходить до налаштувань репозиторію та вибирає відповідну гілку, зазвичай main або master, як джерело для GitHub Pages. Після збереження цих налаштувань, GitHub автоматично створює сайт, який буде доступний за URL, вказаним у розділі "GitHub Pages". Це забезпечує публікацію програми в мережі Інтернет, роблячи її доступною для всіх користувачів.

Переконавшись, що сайт правильно відображається за вказаним URL, необхідно перевірити всі функціональні можливості програми. Це включає

тестування відображення інтерфейсу, коректності роботи скриптів та стилів, а також правильність завантаження прикладів JSON-файлів. Користувачі, відвідуючи сайт, можуть взаємодіяти з програмою, вводити початкові дані, використовувати функції емулятора та завантажувати приклади для тестування.

Для підтримки актуальності розміщеної програми необхідно регулярно оновлювати репозиторій на GitHub. У разі внесення змін до проекту користувач повинен «комітити» та «пушити» ці зміни до репозиторію, щоб вони автоматично відобразилися на сайті. Це дозволяє оперативно вносити виправлення, додавати нові функції та оновлювати вміст програми без необхідності повторного налаштування GitHub Pages.

Здійснивши вищевказані маніпуляції, програму було успішно розміщено на платформі GitHub Pages [21]. На рисунку 5.1 представлено відкритий вебсайт у веб-браузері, який демонструє коректне функціонування та доступність розробленого емулятора для користувачів.

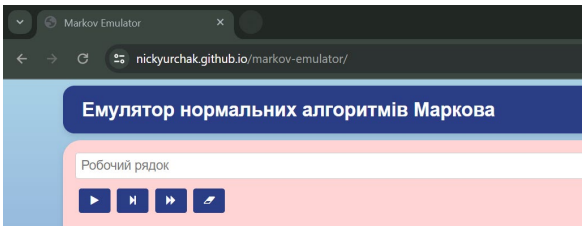

Рисунок 5.1 – Веб-емулятор відкритий у браузері

Розміщення програми на платформі GitHub Pages дозволило забезпечити її доступність для широкого кола користувачів, надавши їм можливість ознайомитися з нормальними алгоритмами Маркова. Це рішення є економічно вигідним та технічно ефективним, оскільки GitHub Pages надає безкоштовний хостинг з простою інтеграцією та автоматичним оновленням сайту при змінах у репозиторії. Завдяки цьому програму можна підтримувати та вдосконалювати, що підвищує її цінність для освітніх та дослідницьких цілей.

6 ПОРІВНЯЛЬНИЙ АНАЛІЗ РОЗРОБЛЕНОГО ЕМУЛЯТОРА З ІСНУЮЧИМИ АНАЛОГАМИ

Після проведеного аналізу існуючих емуляторів нормальних алгоритмів Маркова, викладеного у розділі 2.4, було взято до уваги всі їхні переваги та недоліки з метою розробки покращеного емулятора. Цей розділ містить порівняльний аналіз розробленого емулятора з аналогами, виділення його переваг та недоліків, а також підсумки щодо його ефективності та зручності використання.

Розроблений емулятор має ряд переваг у порівнянні з існуючими аналогами. Перш за все, його доступність та зручність використання. Емулятор реалізовано як веб-додаток, що забезпечує доступність з будь-якого пристрою з веб-браузером. Це дозволяє користувачам працювати з алгоритмами без необхідності встановлення додаткового програмного забезпечення. Інтерфейс емулятора розроблений з урахуванням потреб користувачів, забезпечуючи інтуїтивну зрозумілість та зручність введення правил підстановки та робочого рядка.

Щодо функціональності, емулятор підтримує динамічні таблиці для введення правил підстановки, можливість додавання коментарів до правил, а також функції збереження та експорту алгоритмів у файли. Забезпечено можливість регулювання швидкості виконання алгоритмів, що дозволяє користувачам налаштовувати процес обробки рядків відповідно до своїх потреб. Програма підсвічує зеленим кольором правило підстановки, що виконується в даний момент, дозволяючи користувачам відстежувати процес виконання алгоритму.

Однією з додаткових функцій є можливість побудови графіку складності алгоритмів. Це дозволяє користувачам аналізувати ефективність алгоритмів при різних довжинах вхідних даних. Крім того, забезпечено функцію логування, яка дозволяє зберігати детальну інформацію про кожний крок

виконання алгоритму.

Емулятор має відкритий вихідний код, розміщений на платформі GitHub [22], що забезпечує прозорість та можливість внесення змін та покращень спільнотою розробників. Завдяки тому, що емулятор реалізований повністю на JavaScript, його можна завантажити та зберегти на локальному пристрої для роботи офлайн. Це забезпечує доступ до інструменту навіть без підключення до Інтернету.

Проте, розроблений емулятор має також деякі недоліки. Виконання складних та об'ємних алгоритмів може призвести до перевантаження веб-браузера, що може вплинути на продуктивність роботи емулятора. Хоча емулятор можна використовувати офлайн після завантаження, початковий доступ до нього все ж потребує підключення до Інтернету. Побудова графіку складності алгоритмів може не працювати на деяких специфічних алгоритмах, що обмежує його використання. Крім того, теоретичні функції можуть не завжди давати точний результат, проте варто зазначити, що ця функція є лише додатковим інструментом, а не основним.

У таблиці 6.1 наведено порівняння параметрів емуляторів.

Таблиця 6.1 – Порівняння параметрів розробленого емулятора з існуючими аналогами

Доступні функції	**Розроблений емулятор**	**Веб-емулятор (аналог)**	**Застосунок IOS**	**Десктопний додаток**
Офлайн-робота	Так	Ні	Так	Так
Збереження алгоритму	Так	Ні	Так	Так
Кількість правил	Необмежено	До 50	Необмежено	Необмежено
Кількість циклів	Необмежено	До 500	До 5000	Необмежено
Наявність прикладів	Так	Так	Так	Так

Продовження таблиці 6.1 – Порівняння параметрів розробленого емулятора з існуючими аналогами

Динамічна таблиця	Так	Ні	Ні	Так
Додавання коментарів	Так	Ні	Ні	Так
Регулювання швидкості виконання	Так	Ні	Ні	Так
Миттєве виконання	Так	Так	Так	Ні
Платформа	Браузер	Браузер	iOS	Windows
Детальний звіт	Так	Ні	Так (платно)	Так
Побудова графіку складності	Так	Ні	Ні	Ні
Безкоштовність	Так	Так	Частково	Так
Відкритий вихідний код	Так	Ні	Ні	Ні

Після аналізу існуючих емуляторів нормальних алгоритмів Маркова було враховано всі їхні переваги та недоліки з метою створення покращеного інструменту. Розроблений емулятор поєднує в собі найкращі якості кожного з розглянутих варіантів, забезпечуючи доступність, продуктивність та зручність у використанні. Відкритий вихідний код дозволяє користувачам не лише використовувати емулятор, але й долучатися до його подальшого розвитку та вдосконалення.

Таким чином, розроблений емулятор нормальних алгоритмів Маркова є ефективним та зручним інструментом, який поєднує в собі найкращі якості інших емуляторів і додатково надає можливість офлайн-роботи, побудови графіку складності алгоритмів та логування виконання.

ВИСНОВКИ

У ході виконання випускної кваліфікаційної роботи було досягнуто значних результатів у дослідженні теоретичних основ нормальних алгоритмів Маркова та розробці програми для їхньої емуляції. Було проведено докладне дослідження теоретичних аспектів нормальних алгоритмів Маркова, а також аналіз існуючих емуляторів. Було описано формалізацію правил та визначень для нормальних алгоритмів, а також аналіз тези Маркова та її відношення до машин Тюрінга. Розкрито основні теоретичні принципи, які лежать в основі цих алгоритмів.

У процесі роботи було проведено детальний аналіз існуючих емуляторів нормальних алгоритмів, зокрема веб-емулятора, мобільного застосунку та десктопної програми. Цей аналіз дозволив виявити переваги та недоліки кожного з них і визначити їхні основні характеристики. На основі цього аналізу було зроблено висновок про можливість розробки поліпшеного онлайн-емулятора, який поєднуватиме в собі найкращі характеристики існуючих варіантів.

Було розроблено функціональну частину онлайн-емулятора для нормальних алгоритмів Маркова, який забезпечує зручний та інтуїтивно зрозумілий інтерфейс для введення правил підстановок та робочого рядка. Емулятор підтримує динамічні таблиці для введення правил підстановок, можливість додавання коментарів до правил, а також функції збереження та експорту алгоритмів у файли. Забезпечено можливість регулювання швидкості виконання алгоритмів, що дозволяє користувачам налаштовувати процес обробки рядків відповідно до своїх потреб.

Однією з допоміжних функцій є можливість побудови графіку складності алгоритмів. Це дозволяє користувачам аналізувати ефективність алгоритмів при різних довжинах вхідних даних та робить емулятор корисним інструментом для навчання та досліджень. Крім того, забезпечено функцію

логування, яка дозволяє зберігати детальну інформацію про кожний крок виконання алгоритму. Це робить процес аналізу та налагодження алгоритмів значно зручнішим.

Проведений порівняльний аналіз розробленого емулятора з існуючими аналогами показав, що розроблений емулятор поєднує в собі найкращі якості кожного з розглянутих варіантів, забезпечуючи доступність, продуктивність та зручність у використанні. Відкритий вихідний код, розміщений на GitHub, дозволяє користувачам не лише використовувати емулятор, але й долучатися до його подальшого розвитку та вдосконалення. Завдяки можливості збереження емулятора на локальному пристрої, користувачі можуть працювати з ним навіть в умовах відсутності Інтернету, що підвищує його універсальність.

Недоліками розробленого емулятора є обмеження ресурсів веб-браузера при виконанні складних та об'ємних алгоритмів, що може призвести до перевантаження. Хоча емулятор можна використовувати офлайн після завантаження, початковий доступ до нього все ж потребує підключення до Інтернету. Крім того, побудова графіку складності алгоритмів може не працювати на деяких специфічних алгоритмах, що обмежує його використання. Теоретичні функції можуть не завжди давати точний результат, проте ця функція є лише додатковим інструментом, а не основним.

У підсумку, розробка емулятора нормальних алгоритмів Маркова є важливим внеском у розвиток інструментів для дослідження та навчання формальних мов і алгоритмів. Він забезпечує зручний та ефективний інструмент для роботи з нормальними алгоритмами Маркова, що може бути використаний у навчальних закладах, наукових дослідженнях та практичній діяльності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Гаврилків В.М. Формальні мови та алгоритмічні моделі: навч. посібник. Івано-Франківськ: Голіней. 2023. 180 с.

2. Прийма С.М. Теорія алгоритмів: навч. посібник. Мелітополь: ФОП Однорог Т.В. 2018. 116 с.

3. Щербаков О. В., Фролов О. В. Основи алгоритмізації: методичні рекомендації. Харків: ХНЕУ ім. С. Кузнеця. 2023. 68 с.

4. Нормальні алгоритми Маркова [Електронний ресурс]. URL: https://phm.cuspu.edu.ua/theory_of_algorithms/markov.html.

5. Новотарський М.А. Алгоритми та методи обчислень: навч. посібник. Київ: КПІ ім. Ігоря Сікорського. 2019. 407 с.

6. Стусь О. В. Математична логіка та теорія алгоритмів: лекції. 2017. 150 с.

7. Зацерковний В.І., Гур'єв В.І., Сімакін Ю.С, Фірсова І.В. Алгоритмізація та програмування: конспект лекцій. Чернігів: ЧДЕІУ. 2012. 184 с.

8. Markov Machine Emulator [Електронний ресурс]. URL: https://apps.apple.com/tr/app/markov-algorithm/id1427691412.

9. Konstantin Mishukov AppStore Preview [Електронний ресурс]. URL: https://apps.apple.com/tr/developer/konstantin-mishukov/id1164017853.

10. Емулятор нормальних алгоритмів Маркова [Електронний ресурс]. URL: https://cmcmsu.info/1course/alg.schema.nam.htm.

11. Google Translate [Електронний ресурс]. URL: https://chrome.google.com/webstore/detail/googletranslate/aapbdbdomjkkjkaonfhkkikfgjllcleb.

12. Кучернюк П. В. Технології моніторингу та трафік-інжинірингу в телекомунікаційних мережах. Київ: КПІ ім. Ігоря Сікорського. 2021. 257 с.

13. Учбова модель для вивчення універсального виконавця алгоритмів Маркова [Електронний ресурс]. URL: https://kpolyakov.spb.ru/prog/nma.htm.

14. Іванов Р.О. Веб-застосунок для створення інтерактивних навчальних квестів. Київ: КПІ ім. Ігоря Сікорського. 2023. 142 с.

15. Комар Б.В. Переваги та недоліки використання мобільних додатків в сучасній освіті. Харків: ХНПУ ім. Григорія. Сковороди. 2020. 8 с.

16. Поняття рекурсії [Електронний ресурс]. URL: https://victana.lviv.ua/knyhy/konspekty-lektsii/138-alhorytmizatsiia-ta-prohramuvannia-chastyna-1/660-rozdil-9-vydy-prohramuvannia-2017-r.

17. Ушакова І.О. Проектування інформаційних систем. Харків: ХНЕУ ім. С. Кузнеця. 2015. 236 с.

18. Як використовувати Chart.js [Електронний ресурс]. URL: https://w3schoolsua.github.io/js/js_graphics_chartjs.html#gsc.tab=0.

19. Вступ у JSON [Електронний ресурс]. URL: https://www.json.org/json-uk.html.

20. GitHub Pages Підручник. Швидкий старт [Електронний ресурс]. URL: https://w3schoolsua.github.io/githubpages/index.html#gsc.tab=0.

21. Емулятор нормальних алгоритмів Маркова [Електронний ресурс]. URL: https://nickyurchak.github.io/markov-emulator/index.html.

22. Markov Emulator [Електронний ресурс]. URL: https://github.com/NickYurchak/markov-emulator.

Додаток А

Зауваження нормоконтролера

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження
ПЗ		Змінні в тексті ПЗ та в формулах не однаково оформлено.
С. 62		Продовження таблиці невірно оформлено.

Додаток Б

Лістинг програмного застосунку

Файл «index.html»

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Markov Emulator</title>
<link rel="stylesheet" href="styles.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
</head>
<body>
<header>
  <h1>Емулятор нормальних алгоритмів Маркова</h1>
  <nav>
    <a href="theory.html">Теоретична довідка</a>
  </nav>
</header>
<div id="container">
  <div id="controls">
    <!-- Input field for the initial working string -->
    <input type="text" id="workRow" placeholder="Робочий рядок">
    <div id="buttonContainer">
      <!-- Button to start the algorithm -->
      <button id="playButton" title="Запустити алгоритм"><i class="fa fa-play" aria-hidden="true"></i></button>
      <!-- Button to execute one step of the algorithm -->
      <button id="stepButton" title="Виконати один крок"><i class="fa fa-step-forward" aria-hidden="true"></i></button>
      <!-- Button to run the algorithm to completion -->
      <button id="startButton" title="Запустити алгоритм до завершення"><i class="fa fa-forward" aria-hidden="true"></i></button>
      <!-- Button to clear the working string and table -->
      <button id="clearButton" title="Очистити"><i class="fa fa-eraser" aria-hidden="true"></i></button>
    </div>
    <div id="speedControl">
      <!-- Slider to control the speed of the algorithm execution -->
```

```
          <label for="speedRange">Швидкість виконання:</label>
          <input type="range" id="speedRange" min="500" max="2000" value="1500">
        </div>
        <div id="iterationControl">
          <!-- Dropdown to select the maximum number of iterations -->
          <label for="iterationSelect">Максимальна кількість ітерацій:</label>
          <select id="iterationSelect">
            <option value="1000">1000</option>
            <option value="10000">10000</option>
            <option value="50000">50000</option>
            <option value="100000">100000</option>
            <option value="250000">250000</option>
            <option value="500000">500000</option>
            <option value="1000000">1000000</option>
            <option value="unlimited">Необмежено (Не рекомендовано)</option>
          </select>
        </div>
      </div>
      <div style="display: flex;">
        <div id="tableControls">
          <!-- Buttons to control the rows of the substitution rules table -->
          <button id="addRowAbove" title="Додати рядок вище"><i class="fa fa-plus" aria-hidden="true"></i> <i class="fa fa-arrow-up" aria-hidden="true"></i></button>
          <button id="addRowBelow" title="Додати рядок нижче"><i class="fa fa-plus" aria-hidden="true"></i> <i class="fa fa-arrow-down" aria-hidden="true"></i></button>
          <button id="deleteRow" title="Видалити рядок"><i class="fa fa-times" aria-hidden="true"></i></button>
          <button id="moveUp" title="Перемістити рядок вгору"><i class="fa fa-chevron-up" aria-hidden="true"></i></button>
          <button id="moveDown" title="Перемістити рядок вниз"><i class="fa fa-chevron-down" aria-hidden="true"></i></button>
        </div>
        <div id="tableContainer">
          <!-- Table for entering substitution rules -->
          <table id="myTable">
            <thead>
              <tr>
                <th>№</th>
                <th>Зразок</th>
                <th><i class="fa fa-arrow-right" aria-hidden="true"></i></th>
                <th>Заміна</th>
                <th>Коментар</th>
```

```
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>1</td>
        <td contenteditable="true"></td>
        <td><i class="fa fa-arrow-right" aria-hidden="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
      </tr>
      <tr>
        <td>2</td>
        <td contenteditable="true"></td>
        <td><i class="fa fa-arrow-right" aria-hidden="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
      </tr>
      <tr>
        <td>3</td>
        <td contenteditable="true"></td>
        <td><i class="fa fa-arrow-right" aria-hidden="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
      </tr>
    </tbody>
  </table>
</div>
    <div id="graphContainer">
  <!-- Button to plot the complexity graph -->
  <button id="plotGraphButton">Побудувати графік складності</button>
  <div id="graphControls" style="display: none;">
    <!-- Dropdown to select the type of function for the graph -->
    <select id="functionSelect">
      <option value="none">Не обрано</option>
      <option value="linear">Лінійна</option>
      <option value="quadratic">Квадратична</option>
      <option value="cubic">Кубічна</option>
      <option value="exponential">Експоненціальна</option>
      <option value="logarithmic">Логарифмічна</option>
    </select>
    <!-- Display the equation of the selected function -->
    <span id="equationText"></span>
  </div>
  <!-- Canvas element to draw the graph -->
  <canvas id="chart"></canvas>
```

```
        </div>
      </div>
      <div id="fileControl">
        <!-- Buttons for exporting, importing, and logging -->
        <button id="exportButton">Експортувати</button>
        <button id="importButton">Імпортувати</button>
        <button id="logButton">Логування</button>
        <!-- Hidden file input for importing files -->
        <input type="file" id="importFile" style="display: none;">
      </div>
      <div id="logContainer" style="display:none;">
        <!-- Container to display logs -->
        <h3>Логи</h3>
        <div id="logContent"></div>
    </div>
    <!-- Including necessary libraries and scripts -->
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/regression/2.0.1/regression.min.js"></sc
ript>
    <script src="main.js"></script>
    <script src="markovAlgorithm.js"></script>
    <script src="graph.js"></script>
    </body>
    </html>
```

Файл «theory.html»

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Теоретична довідка</title>
<link rel="stylesheet" href="styles.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css">
</head>
<body>
<header>
  <h1>Теоретична довідка</h1>
  <nav>
    <a href="index.html">Головна</a>
```

```
  </nav>
 </header>
 <div id="container">
  <section>
   <h2>Основні теоретичні відомості по НАМ</h2>
   <!-- Introduction to the theoretical foundations of NAM -->
   <p>Нормальні алгоритми Маркова (НАМ) є формальною системою перетворення рядків, запропонованою радянським математиком Андрієм Марковим у 1940-х роках. НАМ використовують набір правил підстановок для обробки рядків, де кожне правило складається з зразка та його заміни. Якщо зразок знаходиться у рядку, він замінюється на відповідну заміну. Процес продовжується доки не залишиться правил підстановок, які можна застосувати, або не буде застосовано термінальне правило.</p>
   <p>Термінальні правила вводяться додаванням символу крапки в кінці заміни. Якщо заміна містить крапку в кінці, це означає, що алгоритм зупиняється після застосування цього правила.</p>
   <p>НАМ складається з:</p>
   <ul>
    <li><strong>Алфавіт:</strong> Набір символів, що використовуються у рядках.</li>
    <li><strong>Правила підстановок:</strong> Набір пар "зразок-заміна", які визначають, як перетворювати рядки.</li>
    <li><strong>Робочий рядок:</strong> Початковий рядок, до якого застосовуються правила підстановок.</li>
   </ul>
  </section>
  <section>
   <h2>Інструкція з використання розробленого емулятора</h2>
   <!-- Instructions for using the emulator -->
   <ol>
    <li>Введіть початковий рядок у поле "Робочий рядок".</li>
    <li>Введіть правила підстановок у таблицю. Для цього додайте рядки за допомогою кнопок "Додати рядок вище" (<i class="fa fa-plus" aria-hidden="true"></i> <i class="fa fa-arrow-up" aria-hidden="true"></i>) або "Додати рядок нижче" (<i class="fa fa-plus" aria-hidden="true"></i> <i class="fa fa-arrow-down" aria-hidden="true"></i>) і заповніть комірки "Зразок" та "Заміна". Комірки "Коментар" є необов'язковими та можуть використовуватися для пояснень.</li>
    <li>Для запуску алгоритму натисніть кнопку "Запустити алгоритм" (<i class="fa fa-play" aria-hidden="true"></i>). Використовуйте кнопку "Виконати один крок" (<i class="fa fa-step-forward" aria-hidden="true"></i>) для покрокового виконання або кнопку "Запустити алгоритм до завершення" (<i class="fa fa-forward" aria-hidden="true"></i>) для повного виконання.</li>
    <li>При необхідності змініть швидкість виконання алгоритму за допомогою повзунка "Швидкість виконання". Швидкість виконання можна
```

змінювати навіть протягом роботи алгоритму.

Встановіть максимальну кількість ітерацій у випадаючому списку "Максимальна кількість ітерацій".

Для очищення робочого рядка та таблиці правил натисніть кнопку "Очистити" ().

Для експорту введених правил та робочого рядка у файл натисніть кнопку "Експортувати".

Для імпорту правил та робочого рядка з файлу натисніть кнопку "Імпортувати".

При натисканні кнопки "Побудувати графік складності" відбувається перевірка введених даних. Якщо дані коректні, програма обчислює кількість ітерацій для різних довжин слів і будує графік, що відображає складність алгоритму. Якщо введені дані некоректні, користувач отримує відповідні повідомлення про помилки.

Автоматичне виконання алгоритму можна призупинити, натиснувши на ту ж саму кнопку, за допомогою якої він був запущений. Іконка на кнопці змінюється на іконку паузи, а після зупинки можна продовжити виконання алгоритму будь-яким з доступних методів.

Програма підсвічує зеленим кольором правило підстановки, що виконується в даний момент, дозволяючи користувачам легко відстежувати процес виконання алгоритму.

Для відображення процесу виконання алгоритму та його аналізу використовується логування. Логи можна переглядати у спеціальному контейнері, який відкривається натисканням кнопки "Логування".

</section>

<section>

<h2>Приклади</h2>

<!-- Links to example JSON files for testing -->

Ви можете завантажити та використовувати наступні приклади для тестування емулятора:

Приклад "Дублювання бінарного слова"

Приклад "Дублювання слова в алфавіті x"

Приклад "Реверсування слова в алфавіті ab"

Приклад "Віднімання унарного числа"

<a href="https://github.com/NickYurchak/markov-

```
emulator/blob/main/Examples/swap_first0_last1.json">Приклад "Поміняти перший нуль та останню одиницю"</a></li>
            <li><a href="https://github.com/NickYurchak/markov-emulator/raw/main/Examples/examples.zip">Завантажити всі приклади (архів)</a></li>
          </ul>
        </section>
    </div>
    </body>
    </html>
```

Файл «styles.css»

```
body {
    display: flex;
    flex-direction: column;
    align-items: center;
    min-height: 100vh;
    margin: 0;
    font-family: Arial, sans-serif;
    background: linear-gradient(to top,#5569bb, #A8D0E6);
    font-size: 18px;
}
header {
    display: flex;
    justify-content: space-between;
    align-items: center;
    width: 90%;
    background-color: #2a3d86;
    color: #fff;
    padding: 10px 20px;
    border-radius: 18px;
    margin: 10px;
    box-shadow: 0 4px 8px rgba(0,0,0,0.1);
}
header h1 {
    margin: 10px;
    font-size: 30px;
}
header nav a {
    color: #ffd4d4;
    text-decoration: none;
    font-size: 20px;
}
header nav a:hover {
```

```
    text-decoration: underline;
}
#myTable {
    background: #ffffff;
}
#container {
    width: 90%;
    background: #ffd4d4;
    box-shadow: 0 4px 8px rgba(0,0,0,0.1); /* Shadow for container */
    padding: 20px; /* Padding inside container */
    border-radius: 28px; /* Rounded corners */
    min-height: 85vh;
}
#controls {
    margin-bottom: 20px;
}
#workRow {
    width: calc(100% - 22px); /* Adjust width considering padding */
    padding: 8px;
    margin-bottom: 10px;
    box-sizing: border-box;
    border: 1px solid #ccc; /* Border */
    border-radius: 4px; /* Rounded corners */
    font-size: 20px; /* Increase font size for the working row */
}
#buttonContainer {
    display: flex;
    flex-wrap: wrap;
    margin-bottom: 20px;
}
#buttonContainer button {
    width: 50px;
    font-size: 16px;
}
#fileControl {
    margin-top: 20px;
}
#fileControl button {
    font-size: 16px;
}
#plotGraphButton {
    font-size: 16px;
}
#iterationSelect {
    font-size: 16px;
```

```
}
#speedControl {
    margin-bottom: 20px;
    display: flex;
    align-items: center;
}
#speedRange {
    margin-left: 10px;
    width: 200px;
}
#functionSelect {
    font-size: 16px;
}
#tableControls {
    display: flex;
    flex-direction: column;
    margin-right: 20px;
}
#tableControls button {
    margin-bottom: 10px;
    padding: 10px;
    border: none;
    background-color: #2a3d86; /* Blue background for buttons */
    color: #fff; /* White text */
    border-radius: 4px; /* Rounded corners */
    cursor: pointer; /* Pointer cursor */
}
#tableControls button:hover {
    background-color: rgb(0, 34, 71);
}
#tableContainer {
    flex: 1;
    overflow-y: auto;
    height: calc(20px + 18*33px); /* Approximate height to fit 15 rows */
    margin-right: 20px;
}
table {
    border-collapse: collapse;
    width: 100%;
    font-size: 18px; /* Increase font size for table */
}
th, td {
    border: 1px solid black;
    padding: 8px;
    text-align: center;
```

```
}
button {
    margin: 5px;
    padding: 10px;
    border: none;
    background-color: #2a3d86; /* Blue background for buttons */
    color: #fff; /* White text */
    border-radius: 4px; /* Rounded corners */
    cursor: pointer; /* Pointer cursor */
}
button:hover {
    background-color: rgb(0, 34, 71); /* Slightly darker blue on hover */
}
#graphContainer {
    padding-right: 1%;
    padding-left: 1%;
    flex: 1;
    display: flex;
    flex-direction: column;
    align-items: center;
    border-radius: 8px;
}
#graphContainer button {
    margin-bottom: 20px;
}
.white-background {
    background-color: white;
}
#chart {
    width: 100%;
    height: 400px;
}
.highlight {
    background-color: #d4edda; /* Light green color */
}
#iterationControl {
    display: flex;
    align-items: center;
    margin-top: 20px;
}
#iterationControl label {
    margin-right: 10px;
}
#logContainer {
    margin-top: 20px;
```

```
    padding: 10px;
    border: 1px solid #ccc;
    background-color: #f9f9f9;
    max-height: 300px;
    overflow-y: auto;
}
#logContent p {
    font-family: monospace;
    white-space: pre-wrap;
    margin: 0;
    padding: 5px 0;
}
.changed {
    color: red;
    font-weight: bold;
}
```

Файл «main.js»

```
$(document).ready(function() {
  let state = null; // Variable to store the state of the algorithm
  let intervalId = null; // Variable to store the interval ID for automatic execution
  let logs = []; // Array to store logs
  // Event handler for adding a row above the selected row
  $('#addRowAbove').on('click', function() {
    var selectedRow = $('#myTable tbody .selected');
    var newRow = '<tr><td></td><td contenteditable="true"></td><td><i
class="fa fa-arrow-right" aria-hidden="true"></td><td
contenteditable="true"></td><td contenteditable="true"></td></tr>';
    if (selectedRow.length) {
      selectedRow.before(newRow);
    } else {
      $('#myTable tbody').prepend(newRow);
    }
    updateRowNumbers();
  });
  // Event handler for adding a row below the selected row
  $('#addRowBelow').on('click', function() {
    var selectedRow = $('#myTable tbody .selected');
    var newRow = '<tr><td></td><td contenteditable="true"></td><td><i
class="fa fa-arrow-right" aria-hidden="true"></td><td
contenteditable="true"></td><td contenteditable="true"></td></tr>';
    if (selectedRow.length) {
      selectedRow.after(newRow);
```

```
        } else {
          $('#myTable tbody').append(newRow);
        }
        updateRowNumbers();
      });
      // Event handler for deleting the selected row
      $('#deleteRow').on('click', function() {
        var selectedRow = $('#myTable tbody .selected');
        if (selectedRow.length) {
          selectedRow.remove();
          updateRowNumbers();
        }
      });
      // Event handler for moving the selected row up
      $('#moveUp').on('click', function() {
        var selectedRow = $('#myTable tbody .selected');
        if (selectedRow.prev().length && selectedRow.length) {
          selectedRow.insertBefore(selectedRow.prev());
          updateRowNumbers();
        }
      });
      // Event handler for moving the selected row down
      $('#moveDown').on('click', function() {
        var selectedRow = $('#myTable tbody .selected');
        if (selectedRow.next().length && selectedRow.length) {
          selectedRow.insertAfter(selectedRow.next());
          updateRowNumbers();
        }
      });
      // Event handler for selecting a row in the table
      $('#myTable tbody').on('click', 'tr', function() {
        $(this).toggleClass('selected').siblings().removeClass('selected');
        $('#moveUp').prop('disabled', $('#myTable tbody
tr:first').hasClass('selected'));
        $('#moveDown').prop('disabled', $('#myTable tbody
tr:last').hasClass('selected'));
      });
      // Event handler for executing one step of the algorithm
      $('#stepButton').on('click', function() {
        stepAlgorithm();
      });
      // Event handler for running the algorithm to completion
      $('#startButton').on('click', function() {
        const rules = getRulesFromTable();
        let word = $('#workRow').val();
```

```
  state = null; // Reset state before full execution
  logs = []; // Reset logs
  const maxIterations = getMaxIterations();
  let result;
  do {
    result = markovAlgorithmStep(rules, word, state, maxIterations);
    logStep(result, word, rules); // Log each step
    word = result.word;
    state = result.done ? null : result;
  } while (!result.done);
  $('#workRow').val(result.word);
  // Remove highlighting after completion
  $('#myTable tbody tr').removeClass('highlight');
  displayLogs();
});
// Event handler for clearing the working string and table
$('#clearButton').on('click', function() {
  $('#workRow').val('');
  state = null; // Reset state on clear
  $('#myTable tbody tr').removeClass('highlight'); // Remove highlighting
  logs = []; // Reset logs
  displayLogs();
});
// Event handler for playing/pausing the algorithm execution
$('#playButton').on('click', function() {
  if (intervalId) {
    clearInterval(intervalId);
    intervalId = null;
    $('#playButton i').removeClass('fa-pause').addClass('fa-play');
  } else {
    $('#playButton i').removeClass('fa-play').addClass('fa-pause');
    const speed = getSpeed();
    intervalId = setInterval(stepAlgorithm, speed);
  }
});
// Event handler for changing the execution speed
$('#speedRange').on('input', function() {
  if (intervalId) {
    clearInterval(intervalId);
    const speed = getSpeed();
    intervalId = setInterval(stepAlgorithm, speed);
  }
});
// Event handler for exporting rules and working string to a file
$('#exportButton').on('click', function() {
```

```
        const rules = getRulesFromTable();
        const word = $('#workRow').val();
        const data = {
          rules: rules,
          word: word
        };
        const dataStr = "data:text/json;charset=utf-8," +
encodeURIComponent(JSON.stringify(data));
        const downloadAnchorNode = document.createElement('a');
        downloadAnchorNode.setAttribute("href", dataStr);
        downloadAnchorNode.setAttribute("download", "markov_data.json");
        document.body.appendChild(downloadAnchorNode);
        downloadAnchorNode.click();
        downloadAnchorNode.remove();
      });
      // Event handler for importing rules and working string from a file
      $('#importButton').on('click', function() {
        $('#importFile').click();
      });
      // Event handler for processing the imported file
      $('#importFile').on('change', function(event) {
        const file = event.target.files[0];
        if (file) {
          const reader = new FileReader();
          reader.onload = function(e) {
            const contents = e.target.result;
            const data = JSON.parse(contents);
            loadTableData(data.rules);
            $('#workRow').val(data.word);
          };
          reader.readAsText(file);
        }
      });
      // Event handler for toggling the log container
      $('#logButton').on('click', function() {
        $('#logContainer').toggle();
      });
      // Function to load rules into the table
      function loadTableData(rules) {
        const tbody = $('#myTable tbody');
        tbody.empty();
        for (let i = 0; i < rules.sample.length; i++) {
          const row = `<tr>
            <td>${i + 1}</td>
            <td contenteditable="true">${rules.sample[i]}</td>
```

```
            <td><i class="fa fa-arrow-right" aria-hidden="true"></td>
            <td contenteditable="true">${rules.substitution[i]}</td>
            <td contenteditable="true"></td>
          </tr>`;
          tbody.append(row);
        }
      }
      // Function to execute one step of the algorithm
      function stepAlgorithm() {
        const rules = getRulesFromTable();
        const word = $('#workRow').val();
        const maxIterations = getMaxIterations();
        const result = markovAlgorithmStep(rules, word, state, maxIterations);
        logStep(result, word, rules); // Log each step
        $('#workRow').val(result.word);
        state = result.done ? null : result; // Update state or reset if done
        // Highlight the row with the applied rule
        $('#myTable tbody tr').removeClass('highlight');
        if (result.appliedRuleIndex !== -1) {
          $('#myTable tbody tr').eq(result.appliedRuleIndex).addClass('highlight');
        }
        if (result.done) {
          clearInterval(intervalId);
          intervalId = null;
          $('#playButton i').removeClass('fa-pause').addClass('fa-play');
          setTimeout(() => {
            alert("Робота завершена");
          }, 0);
        }
      }
      // Function to get the speed of execution
      function getSpeed() {
        const maxSpeed = 100; // Minimum interval in milliseconds
        const minSpeed = 2000; // Maximum interval in milliseconds
        const speedRangeValue = $('#speedRange').val();
        return minSpeed - ((speedRangeValue - 100) * (minSpeed - maxSpeed) /
(2000 - 100));
      }
      // Function to get the maximum number of iterations
      function getMaxIterations() {
        if ($('#iterationSelect').val() === 'unlimited') {
          return Infinity;
        } else {
          return parseInt($('#iterationSelect').val(), 10);
        }
```

```
        }
        // Function to update the row numbers
        function updateRowNumbers() {
          $('#myTable tbody tr').each(function(index) {
            $(this).find('td:first').text(index + 1);
          });
        }
        // Function to log each step of the algorithm
        function logStep(result, previousWord, rules) {
          const logEntry = {};
          logEntry.iteration = result.count.applied;
          logEntry.ruleIndex = result.appliedRuleIndex + 1;
          logEntry.rule = `${rules.sample[result.appliedRuleIndex]} ->
${rules.substitution[result.appliedRuleIndex]}`;
          logEntry.beforeWord = previousWord;
          logEntry.afterWord = result.word;
          // Determine the changed characters
          const ai = rules.sample[result.appliedRuleIndex];
          const bi = rules.substitution[result.appliedRuleIndex];
          const index = previousWord.indexOf(ai);
          const beforeHighlight = previousWord.slice(0, index) +
                                  previousWord.slice(index, index +
ai.length).split('').map(char => `<span class="changed">${char}</span>`).join('') +
                                  previousWord.slice(index + ai.length);
          const afterHighlight = previousWord.slice(0, index) +
                                 bi.split('').map(char => `<span
class="changed">${char}</span>`).join('') +
                                 previousWord.slice(index + ai.length);

          logEntry.diff = `${beforeHighlight} -> ${afterHighlight}`;

          logs.push(logEntry);
          displayLogs();
        }
        // Function to display logs in the log container
        function displayLogs() {
          const logContainer = $('#logContent');
          logContainer.empty();
          logs.forEach(log => {
            logContainer.append(`<p>${log.iteration}) ${log.ruleIndex}:
${log.rule}\n\t${log.diff}</p>`);
          });
        }
        // Function to get rules from the table
        window.getRulesFromTable = function() {
```

```
    const rules = { sample: [], substitution: [] };
    $('#myTable tbody tr').each(function(index) {
      const sample_1 = $(this).find('td:nth-child(2)').text().trim();
      const substitution_1 = $(this).find('td:nth-child(4)').text().trim();
      rules.sample.push(sample_1);
      rules.substitution.push(substitution_1);
    });
    return rules;
  }
});
```

Файл «markovAlgorithm.js»

```
// Function to execute one step of the Markov algorithm
function markovAlgorithmStep(rules, word, state, maxIterations = 1000) {
    // Initialize state if not provided
    if (!state) {
        state = {
            index: -1,
            count: { applied: 0 },
            word: word,
            appliedRuleIndex: -1,
            hasReachedMaxIterations: false // Added to track if max iterations reached
        };
    }
    // Check if the maximum number of iterations has been reached
    if (state.count.applied >= maxIterations) {
        if (!state.hasReachedMaxIterations) {
            state.hasReachedMaxIterations = true; // Mark that max iterations have been reached
        }
        return { word: state.word, count: state.count, done: true, appliedRuleIndex: -1, hasReachedMaxIterations: state.hasReachedMaxIterations };
    }
    // Iterate through the rules to find and apply the first applicable rule
    for (let i = 0; i < rules.sample.length; i++) {
        const ai = rules.sample[i];
        const bi = rules.substitution[i];
        state.index = state.word.indexOf(ai);
        if (state.index !== -1) {
            state.count.applied += 1;
            state.appliedRuleIndex = i;
            // Check if the rule is terminal (ends with a dot)
            if (bi[bi.length - 1] === '.') {
```

```
                state.word = state.word.slice(0, state.index) + bi.slice(0, -1) +
state.word.slice(state.index + ai.length);
                return { word: state.word, count: state.count, done: true,
appliedRuleIndex: i, hasReachedMaxIterations: state.hasReachedMaxIterations };
            } else {
                state.word = state.word.slice(0, state.index) + bi +
state.word.slice(state.index + ai.length);
                return { word: state.word, count: state.count, done: false,
appliedRuleIndex: i, hasReachedMaxIterations: state.hasReachedMaxIterations };
            }
          }
        }
        // If no rules were applied, mark as done
        return { word: state.word, count: state.count, done: true, appliedRuleIndex:
-1, hasReachedMaxIterations: state.hasReachedMaxIterations };
      }
      // Function to run the Markov algorithm to completion
      function markovAlgorithmComplete(rules, word, maxIterations = 1000) {
        let state = null;
        let result;
        do {
          result = markovAlgorithmStep(rules, word, state, maxIterations);
          word = result.word;
          state = result.done ? null : result;
        } while (state && !result.done && !state.hasReachedMaxIterations);
        return result;
      }
```

Файл «graph.js»

```
      // Event handler for plotting the complexity graph
      $('#plotGraphButton').on('click', function() {
        const rules = getRulesFromTable();
        const word = $('#workRow').val();
        const uniqueChars = [...new Set(word.split(''))];
        // Check if the alphabet (working string) is entered
        if (word === '') {
          $('#workRow').css('border', '2px solid red');
          alert('Введіть алфавіт');
          return;
        } else {
          $('#workRow').css('border', '');
        }
        // Check if the rules are entered
        if (rules.sample.length === 0 || rules.sample.every(sample => sample === ''))
```

```
{
  $('#myTable').css('border', '2px solid red');
  alert('Введіть правила');
  return;
 } else {
  $('#myTable').css('border', '');
 }
 // Check if the alphabet has no more than 5 symbols
 if (uniqueChars.length > 5) {
  alert('В алфавіті має бути не більше 5 символів');
  return;
 }
 const iterations = [];
 let stopExecution = false; // Added to stop execution if needed

 // Generate combinations of words of increasing length
 for (let n = 1; n <= 8 && !stopExecution; n++) {
  let maxIterations = 0;
  const generateCombinations = (prefix, length) => {
   if (length === 0) {
    const result = markovAlgorithmComplete(rules, prefix);
    if (result.hasReachedMaxIterations) { // Check if max iterations reached
     alert('Перевірте вхідні дані');
     stopExecution = true; // Stop execution
     return;
    }
    if (result.count.applied > maxIterations) {
     maxIterations = result.count.applied;
    }
    return;
   }
   for (let char of uniqueChars) {
    if (stopExecution) return; // Stop further execution if needed
    generateCombinations(prefix + char, length - 1);
   }
  };
  generateCombinations('', n);
  if (stopExecution) break; // Exit loop if execution stopped
  iterations.push(maxIterations);
 }
 if (stopExecution) return; // Stop plotting if execution stopped
 const ctx = document.getElementById('chart').getContext('2d');
 if (window.chartInstance) {
  window.chartInstance.destroy();
 }
```

```
window.chartInstance = new Chart(ctx, {
  type: 'scatter',
  data: {
    datasets: [
      {
        label: 'Емпіричні дані',
        data: iterations.map((y, i) => ({ x: i + 1, y })),
        borderColor: 'rgba(75, 192, 192, 1)',
        backgroundColor: 'rgba(75, 192, 192, 1)',
        showLine: false,
        pointRadius: 5
      }
    ]
  },
  options: {
    scales: {
      x: {
        title: {
          display: true,
          text: 'Довжина слова'
        },
        ticks: {
          stepSize: 1,
          callback: function(value) {
            return Number.isInteger(value) ? value : '';
          }
        }
      },
      y: {
        title: {
          display: true,
          text: 'Кількість ітерацій'
        },
        ticks: {
          stepSize: 1,
          callback: function(value) {
            return Number.isInteger(value) ? value : '';
          }
        }
      }
    },
    plugins: {
      legend: {
        labels: {
          color: 'black'
```

```
            }
          }
        }
      }
    });
    $('#graphContainer').addClass('white-background');
    $('#graphControls').show();
  });
  // Event handler for selecting the function type
  $('#functionSelect').on('change', function() {
    const selectedFunction = $(this).val();
    if (selectedFunction === "none") {
      if (window.chartInstance.data.datasets.length > 1) {
        window.chartInstance.data.datasets.pop();
        window.chartInstance.update();
      }
      $('#equationText').text('');
      return;
    }
    const iterations = window.chartInstance.data.datasets[0].data.map(point => point.y);
    const theoreticalComplexity = findBestFitFunction(iterations, selectedFunction);
    const equation = getEquation(selectedFunction, theoreticalComplexity);
    const theoreticalData = theoreticalComplexity.map((y, i) => ({ x: i + 1, y }));
    if (window.chartInstance.data.datasets.length > 1) {
      window.chartInstance.data.datasets.pop();
    }
    window.chartInstance.data.datasets.push({
      label: 'Теоретична функція',
      data: theoreticalData,
      borderColor: 'rgba(255, 99, 132, 1)',
      backgroundColor: 'rgba(255, 99, 132, 0.2)',
      borderWidth: 1,
      borderDash: [10, 5],
      fill: false,
      showLine: true,
      pointRadius: 0
    });
    window.chartInstance.update();
    $('#equationText').text(equation);
  });
  // Function to find the best fit function for the data
  function findBestFitFunction(data, selectedFunction) {
    const x = [1, 2, 3, 4, 5, 6, 7, 8];
```

```
  const y = data;
  const input = x.map((xi, i) => [xi, y[i]]);
  let result;
  switch (selectedFunction) {
    case 'linear':
      result = regression.linear(input);
      break;
    case 'quadratic':
      result = regression.polynomial(input, { order: 2 });
      break;
    case 'cubic':
      result = regression.polynomial(input, { order: 3 });
      break;
    case 'exponential':
      result = regression.exponential(input);
      break;
    case 'logarithmic':
      result = regression.logarithmic(input);
      break;
    default:
      result = regression.linear(input);
  }
  return result.points.map(point => point[1]);
}
// Function to get the equation of the best fit function
function getEquation(type, data) {
  const x = [1, 2, 3, 4, 5, 6, 7, 8];
  const input = x.map((xi, i) => [xi, data[i]]);
  let result;
  switch (type) {
    case 'linear':
      result = regression.linear(input);
      break;
    case 'quadratic':
      result = regression.polynomial(input, { order: 2 });
      break;
    case 'cubic':
      result = regression.polynomial(input, { order: 3 });
      break;
    case 'exponential':
      result = regression.exponential(input);
      break;
    case 'logarithmic':
      result = regression.logarithmic(input);
      break;
```

```
      default:
        result = regression.linear(input);
      }
     return result.string;
    }
```

Файл-приклад «double_10.json»

{"rules":{"sample":["*1","*0","*","x1","x0","y1","y0","x#","y#","#",""],"substitution":["1x*","0y*","#","1x","0x","1y","0y","#1","#0","#.","*"]},"word":"101"}

Файл-приклад «double_x.json»

{"rules":{"sample":["*x","*",""],"substitution":["xx*",".","*"]},"word":"xx"}

Файл-приклад «revers_ab.json»

{"rules":{"sample":["*A","*B","XA","XB","X","*a","*b","Bb","Ba","Ab","Aa",""],"substitution":["aX","bX","aX","bX",".","A","B","bB","aB","bA","aA","*"]},"word":"abbab"}

Файл-приклад «sub_unari.json»

{"rules":{"sample":["1-1","-1","-"],"substitution":["-","","."]},"word":"11111-11"}

Файл-приклад «swap_first0_last1.json»

{"rules":{"sample":["*0","*1","#1","#0","#","1%","0%","1*",""],"substitution":["1#","1*","1#","0#","%","0.","%0","1.","*"]},"word":"100011010"}