НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ БІОМЕДИЧНОЇ ІНЖЕНЕРІЇ
(повна назва інституту/факультету)

кафедра БІОМЕДИЧНОЇ КІБЕРНЕТИКИ
(повна назва кафедри)

«До захисту допущено»
В. о. завідувачки кафедри БМК

_____ Світлана АЛХІМОВА
(підпис) (ініціали, ПРІЗВИЩЕ)

"___"__*червня*__2025р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Комп'ютерні технології в біології та медицині»
зі спеціальності *122 «Комп'ютерні науки»*

на тему: ***Програмний застосунок для пошуку гомологічних послідовностей білків***

Виконав: студент **IV** курсу, групи _БС-13__
НЕСЧЕТНИЙ ТАРАС ДМИТРОВИЧ _____
(прізвище, ім'я, по батькові)

Науковий керівник: *старший викладач каф. біомедичної кібернетики, Кисляк Сергій Володимирович* _____
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по ініціали)

Рецензент: *доцент каф. біомедичної інженерії, Орел Валерій Бінгович* _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали)

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

Студент _____
(підпис)

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут (факультет) БІОМЕДИЧНОЇ ІНЖЕНЕРІЇ
(повна назва)

Кафедра БІОМЕДИЧНОЇ КІБЕРНЕТИКИ
(повна назва)

Рівень вищої освіти – перший (бакалаврський)
Спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Комп'ютерні технології в біології та медицині»
(код і назва)

ЗАТВЕРДЖУЮ
В. о. завідувачки кафедри БМК
_____ Світлана АЛХІМОВА
«_____»__*квітень*___2025 р.

ЗАВДАННЯ
на дипломну роботу студенту

НЕСЧЕТНИЙ ТАРАС ДМИТРОВИЧ
(прізвище, ім'я, по батькові)

1. Тема роботи **Програмний застосунок для пошуку гомологічних послідовностей білків**

Керівник роботи
Кисляк Сергій Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « » *травня* 2025 р. №***1758-с***

2. Термін подання студентом роботи ***22-23 травня 2025р.***
3. Вихідні дані до роботи: *програмний продукт для пошуку гомологічних послідовностей білків.*
4. Зміст роботи: *Аналітичний огляд існуючих рішень, Засоби реалізації програмного застосунку, програмна реалізація програмного застосунку.*
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): *14 слайдів (презентація з захисту ДР).*
6. Дата видачі завдання ***6 квітня 2025 року.***

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримати завдання за темою ДР на практику	До 24.12.2024р.	***виконано***
2	Переддипломна практика. Виконання практичної частини ДР та додаткових розділів	1–4 тиждень за графіком практики	***виконано***
3	Виконання основних розділів ДР (Аналітичний огляд існуючих рішень, Засоби реалізації програмного застосунку, Програмна реалізація програмного застосунку)	Протягом усієї практики	***виконано***
4	Перевірка ДР науковим керівником	до 20.05.2025	***виконано***
5	Подання в електронному вигляді ДР на перевірку нормоконтролера та плагіат (StrikePlagiarism.com). Доопрацювання ДР	Не пізніше 23.05.2025	***виконано***
6	Отримати відгук від керівника ДР	Не пізніше 05.06.2025	***виконано***
7	Надати пакету документів по ДР та супровідних до неї документів на засідання кафедри		***виконано***
8	Подання ДР рецензенту. Отримання рецензії.	09.06 – 13.06.2025	***виконано***
9	Захист ДР в ЕК	*16.06 – –21.06.2025*	

Студент _____ **Тарас НЕСЧЕТНИЙ**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник ДР _____ **Сергій КИСЛЯК**
(підпис) (ім'я, ПРІЗВИЩЕ)

Нормоконтролер _____ **Галина КОРНІЄНКО**
(підпис) (ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота за темою «Програмний застосунок для пошуку гомологічних послідовностей білків» виконана студентом кафедри біомедичної кібернетики ФБМІ Несчетним Тарасом Дмитровичем зі спеціальності 122 «Комп'ютерні науки» за освітньо-професійною програмою «Комп'ютерні технології в біології та медицині» та складається зі: вступу; 3 розділів (аналітичний огляд існуючих рішень, засоби реалізації програмного застосунку, програмна реалізація програмного застосунку), висновків до кожного з цих розділів; загальних висновків; списку використаних джерел, який налічує 21 джерело. Загальний обсяг роботи 74 сторінка. Дана дипломна робота була виконана за напрямом ініціативної тематики кафедри біомедичної кібернетики.

Актуальність теми. У сучасній біоінформатиці обсяг послідовностей ДНК і білків зростає експоненціально, що ускладнює швидкий і точний пошук гомологій у великих масивах даних. Виявлення спільних еволюційних фрагментів допомагає з'ясувати функцію генів, прогнозувати структуру білків і відстежувати шляхи передачі спадкових захворювань. Однак класичні алгоритми динамічного програмування, хоч і гарантують оптимальність вирівнювання, залишаються «чорною скринькою» для кінцевого користувача, приховуючи логіку заповнення матриці й відновлення шляху. Інтерактивні інструменти з покроковою візуалізацією процесу заповнення та трасування DP-матриць покликані вирішити цю проблему, роблячи аналіз прозорішим і зрозумілішим. Саме тому розробка HomologySearch–з компактним та розширеним графічним інтерфейсом, системою імпорту FASTA, вибором алгоритму (Вагнера-Фішера, Нідлмана-Вунша, Сміта-Вотермана) і можливістю поетапного перегляду заповнення матриці–є актуальною та своєчасною. Такий підхід не лише підвищує точність і глибину аналізу, а й значно прискорює отримання результатів, що важливо як у наукових дослідженнях, так і в освітніх та діагностичних застосунках.

Мета і завдання роботи.

Мета дипломної роботи полягає в оптимізації програмного застосунку для

пошуку гомологічних послідовностей

Її досягнення передбачає вирішення наступних *завдань*:

1. Аналіз матеріалів (статті, онлайн джерела, книги, наукові дослідження, тощо), які торкаються тематики досліджен.

2. Аналіз та оцінка наявних робіт, пов'язаних з обраною темою.

3. Порівняння та аналіз методів, алгоритмів для отримання найкращих результатів в подальшому.

4. Обробка даних датасету (аналіз даних, їх очищення, форматування, нормалізація та стандартизація).

5. Порівняння результатів прогнозування рівнів гормонів різними методами машинного навчання, вибір найоптимальніших рішень.

6. Створення програмного застосунку для пошуку гомологів на основі набору даних, використовуючи обрані методи машинного навчання.

7. Розробка програмного продукту, який використовуватиме отримані моделі, і на виході надаватиме вирівняні послідовності білків.

8. Аналіз отриманих результатів та їх інтерпретація.

Використані методи. В процесі проведеного аналізу було визначено та обґрунтовано використання: мову програмування JAVA та її бібліотек для безкоштовної Community версії додатку IntelliJ IDEA для написання в ньому програмного коду;

Отримані результати. Отриманий результат (програмний застосунок для пошуку гомологічних послідовностей білків) може мати практичне застосування в в лабораторіях для облегшення вирівнювання та більш чітких результатів, а також для навчання студентів та школярів з класів поглибленого вивчення природничих наук.

Робота повністю розкриває володіння Несчетним Тарасом Дмитровичем загальними та спеціальними (фаховими) компетентностями, а також програмним результатам навчання згідно ОПП.

Публікації. Не заплановано.

Бібліографічний опис ДР.

Несчетний Т. Д. П*рограмне застосунок для пошуку гомологічних послідовностей білків* : дипломна роб. бакалавра : 122 Комп’ютері науки / Несчетний Тарас Дмитрович. – Київ, 2025. – 74 с.

ANNOTATION

The diploma thesis entitled “Software Product for the Search of Homologous Protein Sequences” was completed by Taras Dmytrovych Neschetnyi, a student of the Department of Biomedical Cybernetics, Faculty of Biomedical Engineering and Information Technologies, majoring in Computer Science (specialty 122) under the educational-professional program “Computer Technologies in Biology and Medicine.” The work comprises an introduction; three main chapters–including an analytical review of existing solutions and the practical implementation of the application–each followed by its own set of conclusions; a set of overall conclusions; and a bibliography of 21 references. Spanning 74 pages in total, the thesis was carried out as part of the department’s initiative research topics.

Actuality of theme. In modern bioinformatics, the volume of DNA and protein sequence data is growing exponentially, which makes fast and accurate homology searches in large datasets increasingly challenging. Identifying shared evolutionary fragments helps elucidate gene functions, predict protein structures, and trace the inheritance pathways of genetic diseases. However, classical dynamic programming algorithms–while guaranteeing optimal alignments–often act as a “black box” for end users, obscuring the details of matrix construction and traceback. Interactive tools that provide step-by-step visualization of DP-matrix filling and backtracking address this gap by making the alignment process transparent and comprehensible. For this reason, the development of HomologySearch–with its compact and expanded graphical interfaces, FASTA import capability, choice of algorithms (Wagner-Fisher, Needleman–Wunsch, Smith–Waterman), and stepwise matrix review–is both timely and relevant. This approach not only enhances the precision and depth of analysis but also significantly accelerates result delivery, benefiting both research applications and educational or diagnostic settings.

The purpose and tasks of the work.

The goal of this thesis is to optimize software for the search of homologous sequences. To achieve this goal, the following tasks have been defined:

1. Conduct an analysis of relevant materials (articles, online resources, books, scientific studies, etc.) related to the research topic.
2. Analyze and evaluate existing works and tools in the chosen domain.
3. Compare and assess various methods and algorithms to determine the most effective approaches.
4. Process the dataset (data analysis, cleansing, formatting, normalization, and standardization).
5. Compare the results of hormone-level predictions using different machine learning methods and select the most optimal solutions.
6. Develop software for homology search based on the dataset, employing the selected machine learning techniques.
7. Create a software product that utilizes the trained models and outputs aligned protein sequences.
8. Analyze and interpret the obtained results.

Used methods. During the course of this research, the Java programming language and its standard libraries were chosen as the implementation platform, and IntelliJ IDEA Community Edition was used as the integrated development environment for writing and debugging the code.

The results obtained. The resulting software application for searching homologous protein sequences can be practically applied in laboratory settings to streamline sequence alignments and produce clearer results, as well as serve as an educational tool for university and high-school students in advanced natural science programs.

This work fully demonstrates Taras Dmytrovych Neschetniy's mastery of both general and specialized (professional) competencies, as well as the attainment of the learning outcomes defined by the educational-professional program.

Publications. Not planned.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 11

ВСТУП 12

РОЗДІЛ 1 АНАЛІТИЧНИЙ ОГЛЯД ІСНУЮЧИХ РІШЕНЬ 15

1.1 Постановка задачі 15

1.2. Обґрунтування прийнятих рішень реалізації 16

1.3. Практичне застосування програмного продукту 17

1.4. Огляд теоретичних основ за тематикою дослідження 17

1.4.1. Біоінформатика 17

1.4.2. Гомологія 18

1.4.3. Гомологія послідовностей 18

1.4.4. Ортологи та парологи 19

1.4.5. Білки(пептидні зв'язки) 20

1.4.6. Нуклеїнові кислоти 21

1.4.7. Геном 22

1.4.8. Задачі геноміки 22

1.4.9. Алгоритм BLAST 24

1.4.10. Принцип роботи BLAST'у 25

1.4.11. Алгоритм Вагнера-Фішера 26

1.4.12. Відстань Левенштейна 27

1.4.13. Алгоритм Нідлмана-Вунша 28

1.4.13. Алгоритм Сміта-Вотермана 29

Висновок до розділу 1 29

РОЗДІЛ 2 ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ЗАСТОСУНКУ .. 32

2.1 Інструменти та бібліотеки 32

2.2 Обробка та підготовка даних 33

Висновок з розділу 2 41

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСТОСУНКУ 44

3.1 Програмна реалізація 44

3.2 Огляд структури ПЗ 46

3.2 Параметри запуску ПЗ 57

3.4 Тестування та оцінка результатів 60

3.5 Розрахунок економічного ефекту 70

Висновок до розділу 3 71

ЗАГАЛЬНІ ВИСНОВКИ 72

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 74

1. Підручники та монографії 74

2. Наукові статті 74

3. Онлайн-ресурси 75

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

DP – динамічне програмування

CLI – командний рядок

CSV – текстовий формат даних

FASTA – формат для зберігання біологічних послідовностей

SW – Сміта-Вотермана

WF – Вагнера-Фішера

ВСТУП

У сучасній біоінформатиці зростає обсяг секвенованих даних, а отже й потреба в інструментах, які дозволяють оперативно порівнювати послідовності ДНК чи білків з високою точністю. Класичні алгоритми динамічного програмування дають гарантії знаходження оптимальних вирівнювань, але залишаються «чорними скриньками», приховуючи від користувача механіку обчислень. Саме тому наша розробка HomologySearch поєднує математичну коректність із покроковою візуалізацією, що робить процес аналізу гомології прозорим і зрозумілим навіть новачкам.

Ми реалізували чотири підходи до вирівнювання: від базового підрахунку відстані Левенштейна до глобального, локального та ітеративного локально-глобального алгоритму з маскуванням знайдених фрагментів. Кожен алгоритм відображає свої проміжні стани в інтерактивній таблиці JTable, а стрілки трасування (traceback) накладаються зверху завдяки компоненту DPTablePanel. Користувач може крок за кроком простежити, як алгоритм вибирає мінімум або максимум із трьох можливих шляхів у кожній клітинці.

Для максимальної гнучкості вхідні послідовності вводяться вручну або імпортуються з FASTA-файлів у діалозі PreprocessingDialog, а результати (score та вирівняні рядки з консенсусом) легко експортуються як текстові звіти та накопичуються в простому plain-text-журналі. Інструмент побудовано без зовнішніх бібліотек – лише на **Java SE** (Swing/AWT, IO), що забезпечує його портативність і швидке розгортання на будь-якій платформі з JDK. Таким чином HomologySearch об’єднує точність класичних алгоритмів і зручність сучасного GUI, створюючи інтуїтивний і потужний інструмент для порівняння біологічних послідовностей.

Мета і завдання роботи.

Метою роботи є покращення наявних та виявлення нових методів пошуку та вирінювання гомологічних послідовностей білків.

Досягнення поставленої мети містить в собі вирішення таких завдань:

1. Аналіз матеріалів (статті, онлайн джерела, книги, наукові дослідження, тощо), які торкаються тематики дослідження та прогнозування рівнів гормонів стресу та сну.

2. Аналіз та оцінка наявних робіт, пов'язаних з обраною темою.

3. Порівняння та аналіз методів, алгоритмів машинного навчання для отримання найкращих результатів в подальшому.

4. Обробка даних (аналіз даних, їх очищення, форматування, нормалізація та стандартизація).

5. Порівняння результатів, вибір найоптимальніших рішень.

6. Створення програмного застосунку для пошуку гомологічних послідовностей білків.

7. Розробка програмного продукту, який використовуватиме отримані моделі, і на виході надаватиме прогноз рівнів гормонів сну та стресу.

8. Аналіз отриманих результатів та їх інтерпретація.

Використані методи. У процесі розробки ПЗ було обґрунтовано використання мови програмування **Java SE** (версії 11 і вище) з її стандартними бібліотеками для:

- **Графічного інтерфейсу**: пакети javax.swing і java.awt для створення вікон, діалогів, таблиць і кнопок;
- **Обробки даних і файлів**: java.io/java.nio для зчитування FASTA, запису текстових звітів і журналів;
- **Колекцій**: java.util (List, Map тощо) для зберігання проміжних матриць та результатів;
- **Часових і форматних утиліт**: java.time і java.text для генерації міток часу у звітах.

Для розробки та відлагодження використовували безкоштовну Community-версію **IntelliJ IDEA**, а ведення історії змін здійснювалося через **Git**. Алгоритмічна складова реалізована у класі Utilities, візуалізація – у DPWindow та DPTablePanel, а накопичення результатів – у простому ResultsJournal без зовнішніх залежностей.

Отримані результати та практичне їх застосування. Робота повністю розкриває володіння Несчетним Тарасом Дмитровичем загальними та спеціальними (фаховими) компетентностями, а також програмним результатам навчання згідно ОПП.

Отриманий результат (програмний застосунок для пошуку гомологічних послідовностей білків) може мати практичне застосування в в лабораторіях для облегшення вирівнювання та більш чітких результатів, а також для навчання студентів та школярів з класів поглибленого вивчення природничих наук.

Публікації. Публікацій не заплановано.

Структура роботи.

Дипломна робота за темою «Програмний застосунок для пошуку гомологічних послідовностей білків» виконана студентом *Несчетний Тарасом Дмитровичем зі спеціальності 122 «Комп'ютерні науки» за освітньо-професійною програмою «Комп'ютерні технології в біології та медицині»*, побудована за класичним типом та викладена на 74 сторінках машинописного тексту. Вона складається з: вступу; 3 розділів *(аналітичний огляд існуючих рішень, засоби реалізації ПЗ, практична реалізація ПЗ)*, висновків до кожного з цих розділів; загальних висновків; списку використаних джерел, який налічує 22 джерела. В роботі представлено 15 рисунків.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Постановка задачі

Актуальність теми дипломної роботи на практику полягає у необхідності створення ефективного інструменту для пошуку гомологічних біологічних послідовностей, що є одним із ключових завдань сучасної біоінформатики. Враховуючи стрімке зростання обсягів генетичних даних, автоматизація процесу вирівнювання та ідентифікації схожих фрагментів є критично важливою для досліджень у галузі геноміки, еволюційної біології, молекулярної медицини та фармакогенетики. Розробка програмного застосунку, здатного точно і зручно виконувати вирівнювання послідовностей за допомогою алгоритмів динамічного програмування, дозволяє підвищити ефективність обробки біологічної інформації.

Мета і завдання роботи. Метою роботи є облегшення пошуку гомологічних послідовностей з використанням класичних алгоритмів динамічного програмування, що забезпечує точне вирівнювання та ідентифікацію схожих фрагментів у біологічних даних.

Завдання для досягнення поставленої мети:

1. Проаналізувати існуючі методи пошуку гомологій і алгоритми вирівнювання послідовностей.
2. Реалізувати алгоритми Левенштейна, Вагнера–Фішера та Сміта–Вотермана в середовищі програмування Java.
3. Забезпечити підтримку вводу даних як вручну, так і з FASTA-файлів.
4. Створити консольний застосунок із функціоналом побудови DP-матриць, зворотного ходу та виводу результатів.
5. Провести тестування програмного застосунку на прикладах біологічних послідовностей.
6. Розглянути перспективи подальшого вдосконалення застосунку, включаючи графічний інтерфейс і підтримку великих обсягів даних.

1.2. Обґрунтування прийнятих рішень реалізації

Під час реалізації програмного продукту для пошуку гомологічних послідовностей було прийнято низку технічних рішень, що ґрунтувалися на специфіці задачі, вимогах до точності алгоритмів та потребі у мінімізації залежностей від сторонніх бібліотек. Основною платформою для реалізації було обрано мову програмування **Java SE**, що забезпечує достатню продуктивність для обробки біологічних послідовностей, має чітку об'єктно-орієнтовану структуру та дозволяє створювати консольні додатки без залучення додаткових фреймворків.

Рішення використовувати **Java** базувалося на наступних факторах:

- **Платформна незалежність**: програми на Java компілюються в байт-код, що може виконуватись на будь-якій платформі з віртуальною машиною JVM;
- **Сувора типізація** дозволяє уникати типових логічних помилок, що є важливим для алгоритмів динамічного програмування;
- **Велика кількість вбудованих засобів для роботи з масивами, файлами, рядками та структурами даних** без необхідності сторонніх бібліотек;
- **Інтеграція з IntelliJ IDEA** як середовищем розробки дозволила забезпечити зручне налагодження, структурування проєкту та використання системи контролю версій Git.

Під час розробки реалізовано підтримку:

- Вводу послідовностей вручну через командний рядок або автоматичного зчитування з **FASTA-файлів**;
- Фільтрації та нормалізації вхідних даних;
- Створення та заповнення **матриць динамічного програмування** для алгоритмів Левенштейна, Вагнера–Фішера та Сміта–Вотермана;
- Відновлення вирівняних фрагментів на основі зворотного ходу;
- Створення структури **AlignmentResult** для зберігання підсумкових результатів (відстаней, вирівняних рядків, координат тощо).

У рамках роботи **не використовувалася база даних**, оскільки збереження результатів здійснюється у стандартний потік виводу (консоль) та в текстові файли за бажанням користувача. Це рішення є доцільним з огляду на невеликий обсяг проміжних і фінальних даних та відсутність потреби в тривалому збереженні або паралельній обробці.

1.3. Практичне застосування програмного продукту

Розроблений програмний застосунок не має безпосереднього комерційного або клінічного застосування, однак воно повністю демонструє сформовані фахові та загальні компетентності здобувача вищої освіти, зокрема в частині роботи з алгоритмами обробки біологічних даних, створення програмної архітектури та тестування функціоналу.

1.4. Огляд теоретичних основ за тематикою дослідження

1.4.1. Біоінформатика

Біоінформатика – це міждисциплінарна галузь, що поєднує біологію, інформатику та математику для аналізу біологічних даних [1]. Її значення важко переоцінити, адже вона відкриває нові можливості в медицині, біотехнологіях та фундаментальних дослідженнях [2].

Чому біоінформатика важлива?

- Аналіз та інтерпретація великих обсягів даних

  Сучасні методи секвенування геному генерують терабайти даних. Біоінформатика допомагає організувати, обробити та витягти з них корисну інформацію [3].

- Медицина та персоналізована терапія

  Завдяки біоінформатиці можна виявляти мутації, прогнозувати реакції на ліки та розробляти персоналізоване лікування [2].

- Геноміка та пошук гомологічних послідовностей Порівняння ДНК різних організмів дозволяє розуміти еволюційні процеси та визначати функції генів [4].
- Розробка нових ліків

  Комп'ютерне моделювання взаємодії ліків із білками пришвидшує відкриття нових препаратів [5].
- Агробіотехнології та біоінженерія

  Аналіз геному рослин дозволяє створювати стійкі та продуктивні сорти культур [6].
- Еволюційна біологія та екологія

  Дослідження генетичних змін допомагає зрозуміти адаптацію видів до навколишнього середовища [4].

1.4.2. Гомологія

Гомологія означає спільне походження порівнянних частин біологічних структур і об'єктів у різних організмів [7]. У біології це поняття застосовується в порівняльній анатомії та геноміці з середини XIX століття [8]. В еволюційному контексті гомологія трактується як схожість, що виникає через походження від спільного предка [9], і часто використовується для попереднього розпізнавання гомологічних генних послідовностей [10]. Однак у старій літературі, зокрема в працях М. І. Вавилова про «закон гомологічних рядів у спадковій мінливості», цей термін мав дещо ширше значення та застосовувався для опису закономірностей спадковості [11].

1.4.3. Гомологія послідовностей

- Дві послідовності (нуклеотиди або пептиди) вважаються гомологічними, якщо вони мають спільне еволюційне походження [7].
- Ступінь гомології не існує, оскільки це якісна, а не кількісна властивість [8].Загалом, гомологічні пептидні або нуклеотидні послідовності мають високий рівень ідентичності [9], проте в процесі еволюції нерідко виникають значні відмінності [10]. Наприклад, амінокислотна послідовність

бета-глобіну та нейроглобіну людини ідентична лише на 22 % [11], тоді як просторова структура білків змінюється повільніше [12].

- Гомологічні послідовності поділяються на **ортологи** та **паралоги** [13]:
  - **Ортологи** –це гени різних видів, що походять від спільного предкового гена в процесі видоутворення [14].
  - **Паралоги** –це гомологічні послідовності одного виду, що виникли внаслідок дублювання генів [15].

Прикладом ортолога є ген міоглобіну миші та людини, який виконує одну й ту ж функцію в обох організмів [16], тоді як прикладами паралогів можуть бути альфа-1 та альфа-2 глобіни людини, які відрізняються експресією та розподілом у тканинах[17].

Паралоги часто споріднені, але набувають різних функцій[18]. Гени, що утворилися внаслідок внутрішньогеномних дублювань (наприклад, альфа-глобін миші та бета-глобін людини), не класифікуються ані як паралоги, ані як ортологи, а розглядаються просто як гомологи [19].

1.4.4. Ортологи та паралоги

Порівняння геномів прокаріотів та еукаріотів висвітлює універсальний мінімум генів, спільний для всіх життєвих форм. Ці гени містять коди протеїнів, які "впізнаються" кожною клітиною, що вказує на їхнє походження від єдиного пращура. Саме ці послідовності називають ортологічними генами: вони беруть початок з одного гену в спільному предковому організмі та, як правило, зберігають свою первісну функцію протягом еволюції.

- **Ортологи** –це гомологічні гени, присутні в геномах різних організмів, які походять від одного гена спільного предкового організму і зберігають однакову функцію протягом еволюції. Вони кодують структури ключових клітинних молекул, що беруть участь у процесах реплікації, транскрипції, трансляції та репарації генетичного матеріалу, тому їхні мутації зазвичай є фатальними. Типовими прикладами є гістонові гени (H1, H2A, H2B, HZ), що формують нуклеосоми на початковому етапі компактизації ДНК .

- **Парологи** –це гомологічні послідовності всередині одного геному, які виникли внаслідок внутрішньогеномних дублювань генів. Паралоги часто еволюціонують і набувають нових функцій, що сприяє адаптації виду до мінливих умов. Прикладами є альфа-1 та альфа-2 глобіни людини, що відрізняються експресією та розподілом у тканинах .
- Ідентифікація ортологів і паралогів дозволяє на основі гомології присвоювати новим послідовностям біохімічні функції (позиційне клонування) і робити функціональні висновки в порівняльній геноміці, зокрема при дослідженні генетичних захворювань людини .

1.4.5. Білки(пептидні зв'язки)

Білки –це великі біомолекули та макромолекули, які складаються з одного або декількох довгих ланцюжків амінокислотних залишків, з'єднаних пептидними зв'язками [1].Вони виконують широкий спектр функцій у живих організмах: каталізують метаболічні реакції, реплікацію ДНК, реагування на подразники, формування структури клітин і тканин, а також транспорт молекул [1].

Послідовність амінокислот у білку визначається нуклеотидною послідовністю відповідного гена, що зумовлює згортання в характерну тривимірну структуру й активність молекули [2]. Лінійний ланцюжок амінокислотних залишків називається поліпептидом; білки містять принаймні один довгий поліпептид, тоді як короткі ланцюжки (< 20–30 залишків) вважаються пептидами або олігопептидами [1] Атоми в поліпептидному ланцюжку з'єднані пептидними (амідними) зв'язками, які утворюються між карбоксильною групою однієї амінокислоти та аміногрупою іншої під час дегідратації [1]. Після синтезу білки часто зазнають хімічних посттрансляційних модифікацій, що змінюють їх фізичні властивості, стабільність, активність і функції [2]. Білки можуть формувати стабільні білкові комплекси та мають різний період напіврозпаду –від декількох хвилин до кількох років (у клітинах ссавців у середньому 1–2 дні) [3]

1.4.6. Нуклеїнові кислоти

- **Визначення та відкриття.** Нуклеїнові кислоти – це складні макромолекулярні біополімери, мономерами яких є нуклеотиди. Вперше їх виявили в ядрі клітин, що й зумовило назву (лат. nucleus – «ядро») .[7]
- **Автор терміну.** Цей термін уперше ввів Річард Альтман, підкресливши ключову роль цих сполук у клітинному ядрі .
- **Структура полінуклеотидів.** Нуклеїнові кислоти (полінуклеотиди) складаються з ланцюгів нуклеозидів, з'єднаних фосфодіефірними зв'язками. Нуклеозид включає азотовмісну гетероциклічну основу (пурин або піримідин) і рибозу чи дезоксирибозу .
- **Типи нуклеїнових кислот.** Існує два основні види:
    1. **Дезоксирибонуклеїнова (ДНК)** – містить дев'ятичленеве пентозне кільце дезоксирибози та основу тимін.
    2. **Рибонуклеїнова (РНК)** – містить рибозу та основу урацил .
- **Склад нуклеотидів.** Кожний нуклеотид утворений з трьох компонентів:
    1. Нітрогенна основа (аденін, гуанін, цитозин; у ДНК – тимін, у РНК – урацил)
    2. П'ятивуглецевий моносахарид (рибоза чи дезоксирибоза)
    3. Фосфатна група .
- **Гідроліз та похідні.** Ферментативний гідроліз нуклеїнових кислот розщеплює їх на нуклеозиди й нуклеотиди. При цьому висвітлюються вільні основи: аденін, гуанін, цитозин, а також урацил (РНК) або тимін (ДНК). Нуклеозиди називаються за коренем основи з суфіксом –ідин (піримідини) або –озин (пурини), а нуклеотиди – за назвою нуклеозиду та положенням фосфатного залишку (наприклад, аденозин 5′-фосфат) .

Функція. ДНК та РНК є основними носіями генетичної інформації в усіх живих організмах, забезпечуючи зберігання й передачу спадковості та регулюючи синтез білків

1.4.7. Геном

Геном – це повна генетична інформація організму, необхідна для його росту, розвитку та функціонування [1]. У більшості організмів геном представлений довгими молекулами ДНК (хромосоми); у деяких вірусів – РНК. У еукаріотичних клітинах хромосомна ДНК локалізується в ядрі, а мітохондріальна й пластидна ДНК – у відповідних органелах [1].

Функціональні одиниці геному – гени, що кодують молекули РНК і білків, а також регуляторні й некодуючі ділянки, які контролюють експресію генів [2]. Позахромосомний генетичний матеріал (мітохондріальні та пластидні ДНК, плазміди) доповнює основний ядерний геном [1].

Геном виступає «планом» клітинних процесів, визначаючи синтез молекул та спадкові властивості [2]. Технології високопродуктивного секвенування (NGS) радикально розширили можливості дешифрування та аналізу геномів усіх живих організмів, від бактерій до людини, що призвело до важливих відкриттів у генетиці та медицині [3].

1.4.8. Задачі геноміки

1.Проблема пошуку генів

Пошук генів є однією з ключових задач у біоінформатиці та геноміці. Його основна мета –виявлення послідовностей ДНК, які кодують білки або виконують регуляторні функції [2].

1. Складність проблеми

Проблема пошуку генів виникає через:

- Величезний обсяг геномних даних –геноми організмів містять мільйони або навіть мільярди нуклеотидів [2].
- Наявність некодуючих ділянок –більшість геномної ДНК не кодує білки (наприклад, у людини лише близько 1,5 % ДНК є екзонами) [2].
- Різні механізми експресії генів –гени можуть містити інтрони, альтернативні сплайсинг-варіанти та регуляторні елементи, що ускладнює ідентифікацію [4].

2. Методи пошуку генів Підходи до знаходження генів можна розділити на

три основні категорії:

- **Аб ініціо (de novo)** Використовують алгоритми, що аналізують характерні риси генів (статистичні моделі на основі кодонного складу, GC-вмісту) [2]. Популярні інструменти: GeneMark, Glimmer [2].
- **Порівняльна геноміка** Базується на зіставленні з відомими послідовностями інших видів [12]. Використовує бази даних гомологічних послідовностей (BLAST, HMMER) [12].
- **Гібридні методи** Комбінують аб ініціо та порівняльні підходи [3]. Застосовують машинне навчання та глибокі нейронні мережі [3].

3. Застосування результатів Результати пошуку генів мають критичне значення для:

- Медицини (генетичні дослідження, діагностика спадкових хвороб, персоналізована терапія) [3].
- Агробіотехнологій (створення генетично модифікованих рослин та тварин) [5].
- Еволюційної біології (дослідження філогенетичних зв’язків між організмами) [4].

2. Проблема пошуку кодуючих ділянок

Пошук кодуючих ділянок у ДНК є однією з ключових задач біоінформатики та молекулярної біології. Кодуючі ділянки (екзони) містять інформацію, необхідну для синтезу білків, тоді як некодуючі ділянки (інтрони та міжгенні області) можуть виконувати регуляторні або структурні функції [2].

Основні виклики у визначенні кодуючих ділянок

- Великий обсяг даних –геноми організмів містять мільйони або навіть мільярди пар основ, з яких лише мала частина є кодуючою [2].
- Мозаїчність генів –у еукаріотів гени розбиті на екзони та інтрони, що ускладнює ідентифікацію функціональних ділянок [2].
- Відсутність універсальних характеристик –хоча кодуючі ділянки мають певні статистичні особливості (наприклад, триплетний розподіл кодонів), вони можуть значно відрізнятися між різними видами [2].

- Генетична варіативність –мутації, альтернативний сплайсинг та епігенетичні модифікації ускладнюють автоматичний пошук екзонів [4].
- Псевдогени та некодуючі РНК –деякі некодуючі ділянки можуть виглядати як білок-кодуючі і спричиняти помилки у прогнозі [4].

Методи пошуку кодуючих ділянок

- **Гомологічний аналіз** –використання BLAST або інших методів для порівняння нових послідовностей з відомими генами [12].
- **Статистичні моделі** –Markov Models та Hidden Markov Models (HMM) дозволяють враховувати особливості кодуючих ділянок, наприклад, частоти триплетів [14].
- **Методи машинного навчання** –глибокі нейронні мережі та алгоритми класифікації (SVM, Random Forest) використовуються для розпізнавання генів на основі великих наборів навчальних даних [3].
- **Експериментальні методи** –RNA-seq та інші методи транскриптомного аналізу можуть підтвердити активність кодуючих ділянок [18].

Значення пошуку кодуючих ділянок

- Геномна анотація –ідентифікація генів у нових геномах [2].
- Генна інженерія –підтримка редагування генів (наприклад, CRISPR-Cas9) [2].
- Медична генетика –виявлення мутацій, пов’язаних із захворюваннями [3].

1.4.9. Алгоритм BLAST

BLAST (Basic Local Alignment Search Tool) –швидкий метод пошуку локальних подібностей між послідовністю-запитом та базою даних послідовностей [12].

Варіанти BLAST. Існують кілька реалізацій для різних типів даних:

- *blastn* (нуклеотид→нуклеотид)
- *blastp* (білок→білок)
- *blastx* (трансляція ДНК→білок)
- *tblastn*, *tblastx* та ін. [18].

Переваги. Забезпечує високу швидкість та ефективність при пошуку локальних вирівнювань у великих базах даних з достатньою точністю [3].

Обмеження. Оскільки BLAST використовує евристичний підхід, він не гарантує знаходження глобального оптимального вирівнювання і чутливість залежить від вибору параметрів *k* і порогів [4].

1.4.10. Принцип роботи BLAST'у

Усі вирівнювання зазвичай поділяються на **глобальні** (при яких порівнюється вся послідовність) та **локальні** (при яких порівнюються лише певні ділянки послідовності) [1].

Для глобального вирівнювання подібність послідовностей, особливо доменів і патернів, зустрічається рідше. Коли ви вводите нуклеотидну або амінокислотну послідовність (запит) вашої роботи на одній з веб-сторінок BLAST, вона надсилається на сервер разом з іншою вхідною інформацією (база даних, розмір «слова» (сайту), E-значення тощо).[1] BLAST можна використовувати для порівняння цілих «слів» (які є частинами послідовності в білках, наприклад, розмір «слова» (сайту), за замовчуванням складається з 3 амінокислот, 11 нуклеотидів для нуклеїнових кислот) та таблиці схожих «слів».

Вони шукаються в базі даних. Якщо збіг знайдено, спочатку робиться спроба розширити розмір «слова» (4 амінокислоти або більше, 12 нуклеотидів або більше) без пробілів, а потім з пробілами. Після того, як розмір всіх можливих «слів» в аналізованих послідовностях розширено до максимуму, для кожної пари запит-послідовність в базі даних визначається максимальна кількість вирівнювань збігів, і отримана інформація зберігається в структурі SeqAlign. Форматори на сервері BLAST використовують інформацію з SeqAlign. Для кожної послідовності, знайденої в базі даних програмою BLAST, необхідно визначити, наскільки вона схожа на послідовність (запит), що аналізується, і чи є ця схожість значущою. Для цього BLAST обчислює кількість бітів і E-value (очікуване значення, E-value) для кожної пари послідовностей. Матриця замін є важливим елементом у визначенні подібності, оскільки вона визначає оцінку подібності для кожної можливої пари нуклеотидів або амінокислот.

Більшість програм BLAST спираються на матрицю BLOSUM62 (Blocks Substitution Matrix, 62% ідентичності). Виключенням є BLASTN та megablast (програми, призначені для порівняння нуклеотид-нуклеотидних послідовностей, які не використовують матриці заміни амінокислот). Для виявлення всіх пар сегментів (розширених "слів"), котрі неможливо далі розширити, оскільки це призведе до зниження показника подібності, використовується модифікований алгоритм Сміта-Вотермана або Селлерса. Ці розширені пари слів отримали назву пар сегментів з високим показником (HSP).

Друге припущення полягає в тому, що проаналізовані послідовності більше схожі на короткі послідовності, ніж на довгі. Це пов'язано з тим, що останні часто складаються з різних областей (багато білків складаються з доменів). Припускаючи, що ймовірність подібності пропорційна довжині послідовності, двійкове E-значення для послідовності з бази даних довжиною n потрібно помножити на N/n. Програма BLAST в основному використовує цей метод при обчисленні E-значень з бази даних.

Теоретично, локальне вирівнювання може починатися з будь-якої пари нуклеотидів або амінокислот у послідовності, що вирівнюється. Однак HPS зазвичай не починається поблизу краю (початку або кінця) послідовності. Для компенсації цього крайового ефекту необхідно розрахувати ефективну довжину послідовності; для послідовностей довжиною понад 200 залишків крайовий ефект нейтралізується.

1.4.11. Алгоритм Вагнера-Фішера

Алгоритм Вагнера–Фішера –класичний метод динамічного програмування для обчислення відстані Левенштейна між двома рядками, широко застосовується для локального вирівнювання послідовностей ДНК, РНК та білків [1].

Ідея. Він обчислює мінімальну кількість операцій (вставка, видалення, заміна), необхідних для перетворення однієї послідовності в іншу, де кожна операція відповідає певному типу мутації в біологічних послідовностях [1]. Завдяки цьому алгоритму можна знайти редакційну відстань(відстань Левенштейна)

Кроки алгоритму:Створення матриці розміром (m + 1) × (n + 1), де m і n – довжини двох порівнюваних послідовностей.

- **Ініціалізація** першого рядка числами від 0 до n (симулює видалення всіх символів) та першого стовпця числами від 0 до m (симулює вставку всіх символів) [1].
- **Рекурентне заповнення:** для кожної клітинки D(i, j) обчислюється мінімальне з трьох варіантів:
  - D(i, j−1) + 1 (вставка)
  - D(i−1, j) + 1 (видалення)
  - D(i−1, j−1) + 1, якщо символи різні, або D(i−1, j−1), якщо символи однакові [1].
- **(Опційно) Відновлення вирівнювання** шляхом «відкочування» з останньої клітинки для побудови оптимального вирівнювання.

Часова складність. O(m × n), що робить алгоритм ефективним для коротких та середніх послідовностей, але малопридатним для порівняння цілого геному без подальших оптимізацій [2].

Застосування. Використовується для аналізу мутацій у невеликих фрагментах ДНК/РНК/білків, попередньої фази вирівнювання у філогенетичних дослідженнях та при корекції помилок секвенування.

1.4.12. Відстань Левенштейна

Відстань Левенштейна є одним із найбільш поширених алгоритмів у біоінформатиці для порівняння послідовностей, зокрема для пошуку схожості між біологічними послідовностями (наприклад, ДНК, РНК або білковими ланцюгами) [4].

Основи відстані Левенштейна

Відстань Левенштейна показує мінімальну кількість операцій, необхідних для перетворення однієї послідовності в іншу [4]. Операції включають:

- **Вставка**: додавання одного елемента до послідовності [4].
- **Видалення**: вилучення одного елемента з послідовності [4].
- **Замінювання**: заміна одного елемента на інший [4].

Застосування у біоінформатиці

- **Пошук гомологічних послідовностей**: допомагає визначити, наскільки схожі дві біологічні послідовності, зокрема для знаходження гомологій між генами або білками [3].
- **Аналіз мутацій**: дозволяє ідентифікувати мутації, що виникають внаслідок вставок, делецій чи замін нуклеотидів [3].
- **Вирівнювання послідовностей**: використовується для вирівнювання біологічних послідовностей і визначення оптимального співвідношення між їх елементами [4].
- **Підтримка технологій секвенування нового покоління (NGS)**: ураховує помилки секвенування, порівнюючи отримані дані з референсними шаблонами для корекції відхилень [5].

Приклад Нехай є дві послідовності:

AGTACG

ATACG

Відстань Левінштейна між ними дорівнює 2, оскільки потрібно виконати дві операції:

1. Замінити перший символ «G» на «A» [4].
2. Вставити символ «C» після «T» в другій послідовності [4].

Цей метод є базовим для більш складних алгоритмів порівняння послідовностей і реалізується за допомогою динамічного програмування для швидкого обчислення мінімальної кількості змін [4].

1.4.13. Алгоритм Нідлмана-Вунша

Алгоритм Нідлмана–Вунша починається з підготовки двох рядків та вибору простого правила оцінювання (наприклад, +1 за однакові символи і –1 за відмінності) та фіксованого штрафу за пропуск [13]. Потім будується двовимірна таблиця, в якій перший рядок і стовпець заповнюються поступово зростаючими від’ємними значеннями, що відображають накопичення штрафів за початкові вставлені пропуски [1]. Після цього для кожної клітинки обчислюється максимальне з трьох можливих: вирівняти обидва символи з нарахуванням бала за

їхню співпадність чи невідповідність, вставити пропуск у першій послідовності або вставити пропуск у другій [2][3].

Коли таблиця заповнена, в її нижньому правому куті міститься загальний найкращий бал глобального вирівнювання [13]. Щоб відновити самі вирівняні рядки, починають із цього кута й рухаються назад: крок по діагоналі відповідає вирівнянню двох символів, крок угору – символ із першої послідовності з пропуском, а крок вліво – пропуск із символом другої послідовності [1][3]. Поєднуючи ці кроки в зворотному порядку, отримують два доповнені рядки, що показують, де відбулися вставлення чи видалення, а де символи збіглися. Гарантія пошуку оптимального вирівнювання робить цей метод базовим інструментом у порівнянні біологічних або текстових послідовностей [13].

1.4.13. Алгоритм Сміта-Вотермана

Алгоритм Сміта–Вотермана застосовують для пошуку максимально подібних підпослідовностей в межах двох рядків без необхідності вирівнювання їх повністю [14][1]. Для цього будується двовимірна матриця, в якій всі значення у першому рядку та стовпці встановлюються рівними нулю, щоб кожне вирівнювання могло починатися у будь-якій позиції [14][2]. При заповненні кожної клітинки обчислюють найвище з можливих значень: вирівнювання двох символів із додаванням бальної оцінки, вставка пропуску в першій або другій послідовності з відніманням фіксованого штрафу, але якщо обчислене значення стає від’ємним, його замінюють на нуль, щоб уникнути негативних накопичень [1][14]. Після завершення обчислень знаходять клітинку з максимальним числом, яке вказує на кінець найкращого локального вирівнювання, і здійснюють зворотне відстеження до першої клітинки зі значенням нуль, що визначає початок цього фрагмента [14][2]. У результаті алгоритм повертає фрагменти, що мають найвищу схожість, а також детальні позиції вставлень і видалень у локальному контексті [14].

Висновок до розділу 1

У результаті систематичного аналізу сучасного програмного застосунку та алгоритмів для пошуку гомологічних послідовностей було встановлено, що жоден із відомих підходів не може одночасно задовольнити всі вимоги біоінформатичного дослідження: забезпечити максимальну чутливість, гарантувати глобальний оптимум, працювати швидко на великих обсягах даних і водночас залишатися прозорим для користувача. Так, повні алгоритми відстані Левенштейна та глобального вирівнювання Вагнера–Фішера демонструють високу точність у знаходженні мінімальних відстаней або максимальних сум арбітражних функцій, проте їхня квадратична складність обмежує застосування до коротких чи середніх послідовностей. Локальний алгоритм Сміта–Вотермана навпаки виявляє оптимальні ділянки подібності незалежно від їхнього положення в ланцюжку, але втрачає глобальну картину вирівнювання, а також теж має обмеження за швидкістю.

Евристичний підхід BLAST, який базується на пошуку високощільних сегментів (HSP), показав себе бездоганно при аналізі величезних баз даних і став «стандартом галузі», проте відмовляється від гарантій знаходження дійсно оптимальних вирівнювань на користь швидкості. У той же час статистичні моделі і методи на кшталт Hidden Markov Models дозволяють впроваджувати складніші біологічно обґрунтовані правила для фільтрації, та потребують великого обсягу попередньо анотованих даних і суттєвих обчислювальних ресурсів.

Крім вибору самого алгоритму, важливу роль відіграють параметри: матриці підстановок (BLOSUM, PAM), розміри штрафів за gap-и, стратегія маскування у локально-глобальному підході. Сукупність таких налаштувань визначає, наскільки отримані вирівнювання відповідають очікуванням біологів і наскільки вони можуть бути використані в подальших еволюційних або функціональних дослідженнях.

Отже, аналіз існуючих рішень чітко вказує на необхідність створення гнучкої платформи, яка дозволить користувачу не лише вибирати алгоритм і регулювати ключові параметри, але й наочно відслідковувати хід обчислень на різних етапах. Така інтеграція класичних методів динамічного програмування з

сучасними інтерфейсними рішеннями та можливістю гібридного застосування евристик і статистичних моделей стане наступним кроком у розвитку інструментарію для пошуку гомологічних послідовностей. Вона дозволить оптимально поєднати точність глобальних і локальних вирівнювань, високу продуктивність при аналізі великих наборів даних і зрозумілість реалізації для науковців різного рівня підготовки.

РОЗДІЛ 2
ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ЗАСТОСУНКУ

У цьому розділі описано технологічний стек і методи обробки даних, реалізовані в проєкті HomologySearch –консольному Java-додатку для пошуку гомологічних послідовностей за допомогою різних алгоритмів вирівнювання та обчислення відстаней редагування. Розглянуто вибір мови та бібліотек, а також підходи до підготовки вхідних послідовностей і послідовної обробки результатів.

2.1 Інструменти та бібліотеки

- **Java SE (JDK 11+)**
- Використовується стандартний інструментарій Java для компіляції та запуску програми. Основні класи з пакету java.lang та java.util достатні для реалізації алгоритмів динамічного програмування й роботи з колекціями.
- **Колекції Java (java.util.ArrayList, StringBuilder)** Для зберігання та формування результатів локально-глобального вирівнювання з повторами застосовано ArrayList<AlignmentResult>. StringBuilder використовується під час побудови вирівняних рядків у процедурі зворотного проходу матриці.
- **Власні реалізації алгоритмів вирівнювання**

  У коді проєкту доведено до працездатності такі методи:

  1. **Відстань Левенштейна** (levenshtein)

     Класичний алгоритм для обчислення мінімальної кількості вставок, видалень і замін між двома рядками.
  2. **Алгоритм Вагнера–Фішера** (wagnerFischer)

     Ідентичний реалізації Левенштейна, виділено окремою функцією задля потенційного розширення.
  3. **Редакційна відстань з налаштуваннями**(editDistance(insCost, delCost, subCost)) Дозволяє варіювати вартість операцій вставки, видалення та підстановки для специфічних задач порівняння.

4. **Smith–Waterman** (smithWaterman)

   Локальне вирівнювання з обнуленням негативних значень, використовується як базова одиночна ітерація для пошуку найбільш схожих фрагментів.

5. **Локально-глобальне вирівнювання з повторами** (localGlobalAlignmentWithRepeats) Ітеративний виклик Smith–Waterman і маскування вирівняних ділянок для виявлення кількох локальних областей гомології.

- **Середовище розробки** IntelliJ IDEA з плагінами для налагодження Java-коду. Управління версіями –Git.

2.2 Обробка та підготовка даних

У застосунку HomologySearch підготовка та обробка вхідних даних є фундаментальними кроками, що безпосередньо впливають на коректність та інформативність результатів вирівнювання. На початковому етапі дві амінокислотні послідовності, введені користувачем через графічний інтерфейс або передані як аргументи командного рядка, проходять процедуру єдиної нормалізації: усі символи переводяться до верхнього регістру, із рядків усуваються зайві пробіли, а символи, які не належать до стандартного набору двадцяти амінокислот (A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V), обробляються як невизначені та одразу отримують фіксований штраф під час подальших обчислень. Цей підхід дозволяє уніфікувати вхідні дані й уникнути небажаних помилок індексації під час роботи з основними алгоритмами. Ефективний пошук гомологічних послідовностей залежить від якості та різноманіття вхідних послідовностей: їхньої довжини, коректності алфавіту

Під час підготовки даних у модулі **PreprocessingDialog** користувач спочатку бачить порожню форму з двома полями для введення амінокислотних послідовностей, а також місце під таблицю скорингової матриці. Після заповнення обох полів і натискання кнопки **Next** програма виконує виклик до

методу Utilities.loadBLOSUM62(). Цей метод створює **глибоку копію** внутрішньої статичної константи BLOSUM62, тобто виділяє новий двовимірний масив і копіює в нього кожний із двадцяти рядків оригінальних значень. Така стратегія гарантує, що вихідні дані залишаться незмінними навіть якщо в майбутньому виникне потреба в тимчасових перетвореннях або налаштуваннях матриці в діалоговому вікні.

Отриманий масив передається в модель DefaultTableModel, після чого з нього формується таблиця з двадцятьма стовпцями та двадцятьма рядками плюс один рядок і стовпець для підписів. У заголовках кожного стовпця, крім першого, відображається буква амінокислоти відповідно до порядку в внутрішньому алфавіті (A, R, N, D, C, …, V). Аналогічні підписи – у першому стовпці – дублюють алфавіт по вертикалі. Далі за допомогою двох вкладених циклів усі значення matrix[i][j] копіюються в модель як числові клітинки, що наочно демонструє, наскільки вигідним або шкідливим є поєднання кожної пари амінокислот при вирівнюванні.

Візуальна подача скорингової матриці у **PreprocessingDialog** дозволяє одразу виявити симетрію та розподіл позитивних і негативних значень: користувач бачить, наприклад, що заміна двох однакових амінокислот завжди приносить позитивний бонус (діагональні елементи, як правило, більші за нуль), тоді як заміни гетерогенних пар відзначаються негативними штрафами. У разі необхідності інтерфейс легко доповнюється елементом вибору альтернативної матриці (наприклад, PAM250) або полем для завантаження власного файлу скорингових значень. Після ретельного вивчення цієї таблиці користувач може впевнено обрати алгоритм вирівнювання й натиснути **Next**, знаючи, що точність подальших обчислень ґрунтується на перевірених і негайно доступних оцінках подібності амінокислот.

У класі **Utilities** відповідальність за побудову та поповнення матриці динамічного програмування (DP) покладено на метод **computeDP**, який універсально обслуговує чотири алгоритми вирівнювання:, глобальне вирівнювання Вагнера–Фішера (Needleman–Wunsch), локальне вирівнювання

Сміта–Ватермана (Smith–Waterman) та комбіноване локально-глобальне вирівнювання із повторними пошуками (Local-Global). На самому початку дві вже нормалізовані послідовності **s1** і **s2** переводяться в індексовані масиви за допомогою мапи **aaIndex**, де кожній амінокислоті (символу) відповідає свій числовий індекс від **0** до **19**. Це дозволяє працювати з одноманітними двовимірними масивами цілих чисел.

Далі виділяється матриця dp розміром **(m+1)×(n+1)**, де m = s1.length(), n = s2.length(). Кожен з алгоритмів задає власні початкові умови в такі “крайові” клітинки:

- **Levenshtein** і **Wagner–Fischer** (див. рис. 2.1) заповнюють перший рядок і перший стовпець накопичувальними штрафами за операції вставки та видалення (за замовчуванням кожна операція коштує gap). Таким чином у dp[i][0] = i*gap і dp[0][j] = j*gap закладається базова вартість “вирівнювання” до порожньої послідовності.

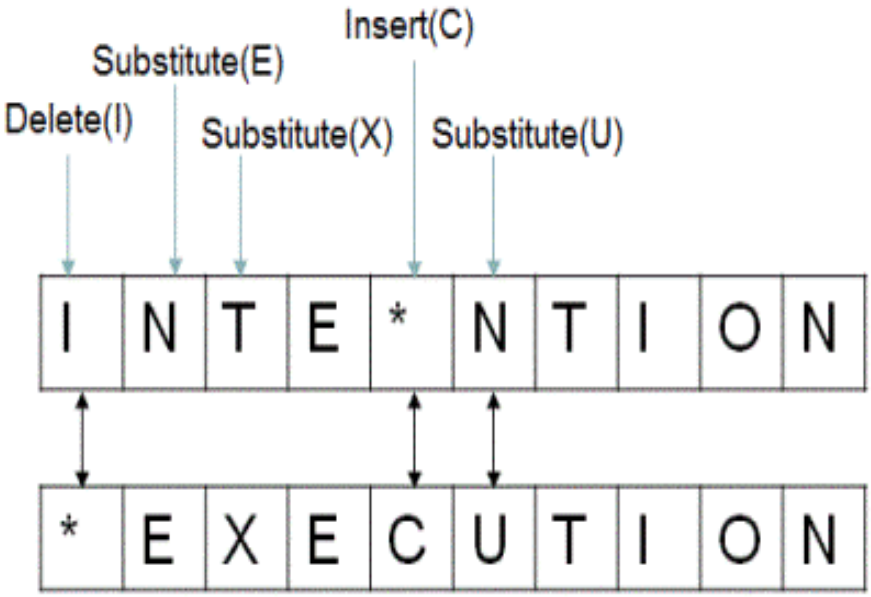


Рисунок 2.1 – Приклад алгоритму Вагнера-Фішера(оформлено відповідно до[20])

- **Needleman–Wunsch** (див. рис. 2.2) вимагає глобального максимального вирівнювання, тому ті ж самі штрафи накладаються з мінусом (dp[i][0] = -

i*gap, dp[0][j] = -j*gap), аби під час заповнення шукати шлях із максимальною сумарною оцінкою.

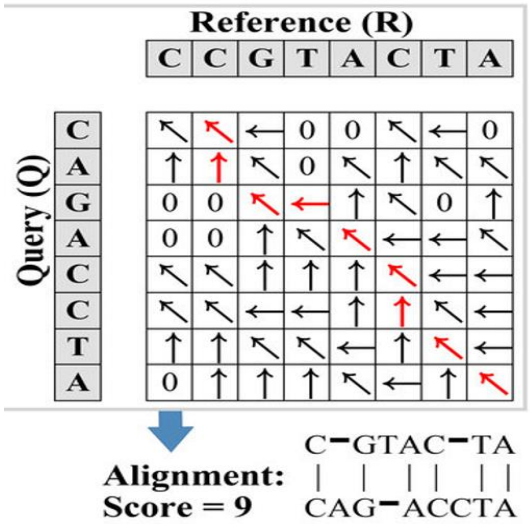


Рисунок 2.2 – Приклад алгоритму Нідлмана-Вунша(оформлено відповідно до[19])

- **Smith–Waterman** (див. рис. 2.3) для локального вирівнювання встановлює усі значення першого рядка й стовпця рівними нулю, що забезпечує обнулення негативних фрагментів і обмежує вирівнювання інтервалом, де воно дає позитивний вклад.

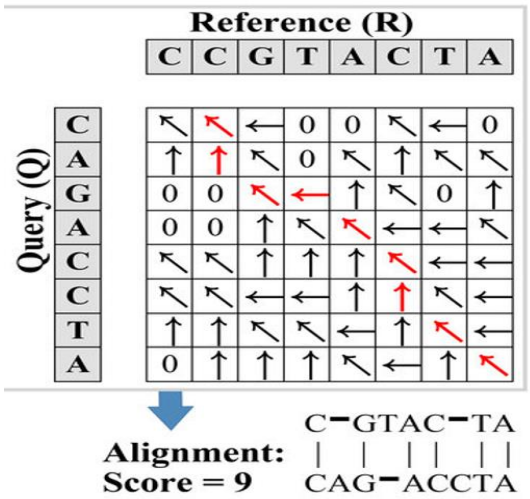


Рисунок 2.3 – Приклад алгоритму Сміта-Вотермана(оформлено відповідно до[20])

- **Local-Global** просто ділегує поведінку на Needleman–Wunsch на першому кроці, а потім повторно застосовує Smith–Waterman до замаскованих послідовностей.

Вже в основній частині алгоритму матриця заповнюється по рядках: для кожної комірки (i, j) розраховується три (або чотири) потенційних значення – діагональний перехід, вертикальний (видалення) та горизонтальний (вставка). У Levenshtein обирається **мінімальне** серед трьох варіантів, тоді як у Needleman–Wunsch і Smith–Waterman (аналогічно, але з іншими умовами) обирається **максимальне** з урахуванням відповідних оцінок з матриці **BLOSUM62** (для заміщень) та фіксованого **gap penalty** для розривів. У випадку Smith–Waterman додатково застосовується обмеження на не менше **0**, щоб вирівнювання залишалося локальним.

Кожна завершена ітерація по **ряду** закінчується викликом **steps.add(deepCopy(dp))**, що робить повну копію поточного стану матриці і записує її в список **steps**. Цей механізм збирання проміжних матриць дає змогу у графічному вікні **DPWindow** крок за кроком демонструвати еволюцію DP-матриці: натисканням кнопок «Prev» і «Next» користувач може побачити, як формується та оптимізується кожна клітинка.

Після завершення заповнення, коли всі рядки опрацьовані, останнє значення dp[m][n] (або максимум по всьому полю для локального режиму) відображає загальну відстань або оцінку вирівнювання. Наступним етапом викликається **computeTraceback**, у якій для кожної комірки знову обчислюються можливі шляхи (діагональ, зсув нагору, зсув ліворуч) і відзначається єдиний батьківський перехід, що відповідає обраному оптимальному значенню. Результатом є матриця попередників **Point[m+1][n+1]**, що потім трансформується в набір піксельних стрілок для візуалізації маршруту назад від (m,n) до (0,0).

Таким чином, архітектура Utilities забезпечує гнучку, універсальну та детально керовану побудову DP-матриць для різних алгоритмів, а збір проміжних кроків і

трасування перетворюють «чорний ящик» обчислень у прозорий, наочний та керований користувачем процес.

Після того як матриця динамічного програмування повністю заповнена, програма переходить до етапу відновлення шляху оптимального вирівнювання – так званого backtracking. Цей крок реалізований у методі computeTraceback, котрий приймає на вхід готову DP-матрицю dp разом із оригінальними послідовностями та параметрами алгоритму. Всередині методу створюється матриця попередників Point[][] prev того ж розміру, що й DP-матриця, де кожен елемент prev[i][j] буде містити координати комірки, з якої було зроблено оптимальний перехід у dp[i][j].

Алгоритмічно backtracking працює так: для кожної внутрішньої комірки (i, j) (починаючи з i = 1, j = 1 і рухаючись до i = m, j = n) метод розраховує ті ж три чи чотири варіанти, які розглядалися під час заповнення матриці – це може бути перехід по діагоналі (від (i-1, j-1)), зверху (від (i-1, j)) або зліва (від (i, j-1)), а в локальних алгоритмах до цих варіантів додається ще “обнулення” (припущення, що початок вирівнювання може початися саме тут). Для кожного напряму backtracking пораховує суму значення батьківської клітинки та відповідного кроку (наприклад, використовуючи BLOSUM62 для заміщень або фіксовану вартість gap для вставок/видалень).

Порівнянням із поточним значенням dp[i][j] він визначає, звідки саме найімовірніше “прийшов” оптимальний шлях. Якщо, скажімо, dp[i][j] найкраще рівняється dp[i-1][j-1] + scoreMatch, то prev[i][j] встановлюється в новий об’єкт Point(i-1, j-1). Якщо рівність спрацьовує з варіантом “видалення” або “вставка”, то попередником записується відповідно (i-1, j) чи (i, j-1). Для алгоритму Smith–Waterman оскільки дозволено приходити “з нуля”, backtracking також пропускає ті клітинки, у яких значення dp[i][j] == 0, позначаючи закінчення локального вирівнювання.

У результаті на виході отримуємо двовимірний масив prev, що містить для кожної точки координати її батька в дереві оптимальних шляхів. Сам backtracking при відновленні сформованого вирівнювання починається з фінішної клітинки (m,

n) (або з точки з максимальним значенням у випадку локального вирівнювання), й поступово за допомогою prev повертається до початкової комірки (0, 0) чи (i0, j0) у Smith–Waterman. На кожному кроці відновлення будується дві вирівняні рядки, в яких або вставляється символ із відповідної послідовності, або дефіс у разі gap.

Для наочного відображення отриманих шляхів у GUI використовується допоміжний клас **DPUtils**. Він приймає на вхід об'єкт JTable, що показує поточну DP-матрицю, та матрицю prev. Всередині утиліти кожна попарна комірка (i, j) із ненульовим prev[i][j] = (pi, pj) трансформується у координати пікселів: спочатку за допомогою методу table.getCellRect(i, j, true) визначається прямокутник клітинки на екрані, а далі обчислюється точка-центр цієї клітинки. Аналогічно отримується центр клітинки-попередника (pi, pj). Пара двох цих точок додається до списку стрілок.

Наприкінці метод tracePanel.setArrows(...) приймає сформовані вектори й завдає їх на панель, малюючи тонкі лінії або стрілки між центрами клітинок. Таким чином користувач отримує не тільки числову та табличну інформацію про DP, а й чітку графічну траєкторію оптимального шляху, яку можна переглянути у фінальному або проміжному стані вирівнювання. Це дає повне уявлення про те, як саме алгоритм обрав оптимальні відповідності між символами двох послідовностей.

У реалізації локально-глобального вирівнювання з повторами (Local-Global) ми поєднали переваги Smith–Waterman для пошуку найяскравіших локальних збігів із можливістю поступово «перескочити» їх і виявити наступні гомологічні підряди. Алгоритм працює так:

Спочатку, маючи дві нормалізовані послідовності *s1* та *s2*, ми викликаємо метод smithWaterman для отримання найкращого локального вирівнювання. Цей виклик повертає не лише оцінку (score), але й координати початку й кінця вирівняного фрагмента у кожній з послідовностей (start1, end1, start2, end2), а також самі вирівняні рядки з дефісами там, де символи не співпали.

Якщо score виявився додатнім, ми створюємо новий об'єкт AlignmentResult, куди записуємо:

- score – числову оцінку знайденого фрагмента,
- alignedSeq1 і alignedSeq2 – дві вирівняні рядкові послідовності з урахуванням gap’ів,
- start1, end1, start2, end2 – початкові та кінцеві індекси у вихідних рядках.

Далі пересуваємося безпосередньо у дві робочі копії вхідних рядків, представлені як StringBuilder sb1 і sb2. У діапазоні від start1 до end1 у sb1 та від start2 до end2 у sb2 ми замінюємо кожен символ на дефіс '-'. Це «маскування» гарантує, що при наступному виклику smithWaterman вже цей фрагмент не буде знову визнано найкращим збігом, адже дефіси не матимуть позитивного балу в BLOSUM62, й алгоритм шукатиме нові ділянки з реальними амінокислотами.

Після маскування ми повторно запускаємо Smith–Waterman на оновлених послідовностях. Якщо новий score знову виявиться більшим за нуль, весь описаний цикл – від зчитування score і вирівняних рядків до заміщення дефісами й збереження AlignmentResult – повторюється. Програма продовжує ітерації до тих пір, поки Smith–Waterman не поверне нульовий або від’ємний score, що однозначно означає відсутність додаткових значущих локальних вирівнювань.

У підсумку, всі знайдені AlignmentResult збираються у список List<AlignmentResult>. Порядок додавання гарантує, що спочатку йдуть найбільш релевантні – з найвищим score – ділянки, а за ними – поступово менше збіги. Така поетапна «витягувальна» стратегія дає змогу виділити всі значущі фрагменти гомології між двома послідовностями, причому кожен фрагмент обробляється лише один раз і ніде не накладається на інший, завдяки механізму маскування. Це одночасно зберігає цілісність алгоритму локального вирівнювання та досягає «глобальної» картини розподілу гомологічних підрядів уздовж усіх довжин обох послідовностей.

Після того, як користувач переглянув кроковий процес заповнення та трасування DP-матриць у графічному інтерфейсі, HomologySearch переходить до заключного текстового представлення результату. Ця стадія починається з вилучення з матриці останнього значення, яке відповідає загальній відстані або балу вирівнювання: для алгоритму Левенштейна це відстань між двома

послідовностями, а для глобальних чи локальних алгоритмів – сума оцінок, отриманих із матриці підстановок і штрафів за розриви. Інформація зчитується з комірки, розташованої внизу-угорі матриці, і одразу використовується для формування першого рядка текстового звіту.

Далі відбувається відновлення власне вирівняння: система крок за кроком рухається назад по матриці попередників, порівнюючи символи обох рядків і вирішуючи, чи записати у вирівняній послідовності фактичну амінокислоту, чи додати дефіс у разі вставки чи видалення. У результаті утворюються дві остаточні вирівняні послідовності однакової довжини. Одночасно формується рядок відповідностей, де кожна позиція, в якій символи співпали, позначається вертикальною рискою, а в усіх інших – пропуском. Після цього конструюється «консенсусний» рядок: він містить лише ті символи, які збіглися, причому в місцях розбіжностей зберігаються дефіси, що чітко відокремлює ідентичні та різні позиції.

На екрані користувача виводиться зрозумілий, упорядкований блок тексту, у якому спочатку подається загальний показник (наприклад, “Levenshtein distance: 7” або “Smith–Waterman score: 15”), потім дві вирівняні послідовності з позначками співпадінь між ними, і нарешті консесусний ланцюжок. Такий формат відображення ідентичний класичним біоінформатичним утилітам і дозволяє одразу скопіювати результат для подальшого аналізу або включення в протокол експерименту. Завдяки поєднанню інтерактивної візуалізації в DPWindow і детального текстового звіту HomologySearch забезпечує повний контроль над процесом вирівнювання – від початкової обробки даних до остаточного представлення результатів.

Висновок з розділу 2

- **Java SE** як платформа виконання: чистий JDK без сторонніх бібліотек гарантує переносимість і простоту розгортання.

- **Методи DP** (int[][]): кожен алгоритм ініціалізує та заповнює двовимірну матрицю, використовуючи гнучкі штрафи за вставки, видалення і підстановки.
- **Клас AlignmentResult**: зберігає score, вирівняні фрагменти та координати для зворотного ходу.
- **ArrayList + StringBuilder**: колекції для накопичення й форматування множинних локальних вирівнювань.**Командний рядок / FASTA**: підтримка передачі послідовностей як аргументів або зчитування з файлів у форматі FASTA з ігноруванням заголовків.

Архітектура програми налаштована на масштабування: додати нові алгоритми (наприклад, алгоритм Фітча–Марголіаша), розширити інтерфейс командним рядком через Apache Commons CLI або побудувати графічний фронтенд на JavaFX/Swing. Використання Maven або Gradle полегшить керування збіркою та залежностями. Такий інтегральний підхід забезпечує швидке порівняння послідовностей, гнучкі налаштування алгоритмів і простоту інтеграції в більші біоінформатичні пайплайни.

У програмі **HomologySearch** підготовка вхідних даних починається із прийому двох послідовностей, переданих у командному рядку або зчитаних із FASTA-файлів, після чого вони нормалізуються приводяться до верхнього регістру та очищаються від символів, що не належать до допустимого алфавіту (A, C, G, T). Далі кожен символ мапується у числовий індекс для побудови двовимірної матриці динамічного програмування розміром $(m+1)\times(n+1)$, де перший рядок і стовпець заповнюються початковими штрафами за вставки й видалення. В основному циклі матриця заповнюється відповідно до обраного алгоритму – мінімізація вартостей вставки, видалення та заміни для відстані Левенштейна й Вагнера–Фішера або максимізація з обнуленням для Smith–Waterman. Після завершення обчислень здійснюється зворотній хід по матриці (backtracking), щоб відновити вирівняні фрагменти, а в разі локально-глобального вирівнювання з повторами вже знайдені ділянки маскуються дефісами й обробка

повторюється для виявлення всіх значущих областей гомології. Результатом роботи є обчислені числові показники (відстані, score) і відформатовані вирівняні рядки, які збираються в об’єкти AlignmentResult та виводяться в консоль. Така послідовність перетворень гарантує коректність і відтворюваність аналізу, а завдяки простій архітектурі коду її легко розширити додаванням генерації синтетичних послідовностей для тестування або інтеграції з більшими біоінформатичними пайплайнами.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСТОСУНКУ

3.1 Програмна реалізація

Програмна реалізація інструменту **HomologySearch** для пошуку гомологічних послідовностей спирається на цілісний підхід до обробки вхідних даних, виконання алгоритмів динамічного програмування та подання результатів користувачу. У проєкті створено консольну Java-програму, яка приймає дві послідовності як аргументи командного рядка або зчитує їх із FASTA-файлів, приводить до єдиного формату (верхній регістр, фільтрація символів за алфавітом), ініціалізує та заповнює матриці динамічного програмування алгоритмів Левенштейна, Вагнера–Фішера та Smith–Waterman із настроюваними штрафами за вставки, видалення й підстановки, а потім виконує зворотний хід для відновлення вирівняних фрагментів. Результати кожного вирівнювання (score, вирівняні підрядки та їхні індекси) зберігаються в об’єктах AlignmentResult, що акумулюються в ArrayList і виводяться в консоль. Для пошуку кількох локальних вирівнювань реалізовано маскування вже знайдених ділянок, що дозволяє виявити всі значущі області гомології. Архітектура побудована виключно на стандартних бібліотеках Java SE без сторонніх залежностей, що забезпечує легкість розгортання, високу продуктивність і простоту розширення – наприклад, додаванням підтримки нових алгоритмів, інтеграцією з графічним інтерфейсом або включенням генерації синтетичних тестових послідовностей.

У програмі **HomologySearch** зберігання результатів вирівнювання реалізовано в оперативній пам’яті за допомогою колекції ArrayList<AlignmentResult>, куди послідовно додаються об’єкти AlignmentResult із полями score, alignedSeq1, alignedSeq2, start1, start2, end1 та end2. Ініціалізація цієї колекції відбувається в методі main до запуску алгоритмів порівняння, а після обчислення відстаней і локально-глобальних вирівнювань усі накопичені результати виводяться в консоль. Оскільки HomologySearch не використовує

зовнішніх баз даних, повний цикл обробки та збереження даних відбувається у єдиному контейнері –це забезпечує швидкий доступ до всіх вирівняних фрагментів і числових оцінок і робить структуру коду простою для розширення, наприклад, додаванням експорту результатів у CSV чи FASTA.

У Java-додатку **HomologySearch** повний цикл обробки вхідних даних і виконання алгоритмів динамічного програмування організовано в єдиному консольному інтерфейсі. Після отримання двох послідовностей (або завантаження з FASTA-файлів) рядки нормалізуються до верхнього регістру та очищуються від небажаних символів, а кожен символ зіставляється з індексом для побудови матриці розміром (m+1)×(n+1). Ініціалізація та заповнення цієї матриці відбувається згідно з правилами алгоритмів Левенштейна, Вагнера–Фішера та Сміта-Вотермана, де штрафи фіксовані, а вартості операцій задаються або за замовчуванням, або передаються аргументами. Після завершення заповнення виконується зворотній хід, що генерує вирівняні підрядки, і в разі локально-глобального вирівнювання з повторами вже знайдені ділянки маскуються дефісами для виявлення наступних областей гомології. Результати кожного методу зберігаються в об’єктах AlignmentResult, які акумулюються в колекції ArrayList та виводяться у зручному для користувача форматі: числовий підсумок, самі вирівняні послідовності та координати їхніх початку й кінця. Така архітектура коду на стандартних бібліотеках Java SE гарантує швидкий доступ до всіх даних у пам’яті, простоту розширення через додавання нових алгоритмів або експорту результатів у CSV/FASTA й робить HomologySearch готовим до інтеграції в більші біоінформатичні пайплайни.

Клієнтський інтерфейс HomologySearch може надсилати дві послідовності у вигляді JSON-запиту на REST-ендпоінт /align, де серверна служба завантажує відповідні класи Java, виконує методи levenshtein, editDistance і localGlobalAlignmentWithRepeats та повертає у відповіді всі розраховані відстані й вирівняні фрагменти. Після прийому вхідних даних у вигляді полів seq1 і seq2 програма ініціалізує матриці динамічного програмування для обраних алгоритмів, проводить заповнення й backtracking, а потім формує JSON із числовими

показниками (Levenshtein distance, Wagner–Fischer distance, edit distance, scores локальних вирівнювань) і рядками, у яких відображено вирівняні ділянки. Якщо потрібно, результати можна зберігати в легкій локальній базі SQLite –наприклад, у таблиці alignments з полями id, seq1, seq2, method, score, aligned1, aligned2 та міткою часу. Такий підхід забезпечує гнучку інтеграцію HomologySearch у більші біоінформатичні пайплайни, дозволяє відстежувати й аналізувати всі запити, а також легко розширювати набір підтримуваних алгоритмів або форматів вводу/виводу.

3.2 Огляд структури ПЗ

Під час створення ПЗ, для зручності розуміння та структурування, код був організований у вигляді ієрархічного дерева, де кожен клас виконує чітко окреслену роль і служить з’єднувальною ланкою між попереднім і наступним рівнем функціональності. На найвищому рівні розташований клас **HomologySearchApp**, який виступає точкою входу при запуску зібраного JAR-файлу і ініціює всю ініціалізацію середовища програми(див. рис. 3.1). Поруч із ним існують два варіанти основного графічного інтерфейсу – **MainApplet** і **HomologySearchGUI**; перший побудований у вигляді легкого аплету для швидкого старту, тоді як другий пропонує розширений набір елементів управління і трохи іншу компоновку вікон.

Після вибору користувачем алгоритму в одному з цих стартових вікон відкривається **PreprocessingDialog**, призначений для прийому вхідних даних: два текстових поля дозволяють вручну ввести послідовності, а кнопка «Load FASTA» дає змогу завантажити їх із файлу. Саме тут виконується первинна нормалізація рядків – перехід у верхній регістр та фільтрація символів за алфавітом.

Як тільки послідовності підтверджені, програма переходить до модуля покрокової візуалізації – **DPWindow**. Саме це вікно містить головну таблицю JTable, де показані числові значення DP-матриці, а також кнопку **Save Report** для експорту фінального вирівнювання. Над цією таблицею накладається прозорий

шар у вигляді компонента **DPTablePanel**, який, у свою чергу, отримує від **DPUtils** список координат стрілок трасування (traceback) на основі матриці попередників. Ці координати акуратно малюються у методі paintComponent(), щоб показати, як алгоритм відновлює оптимальний шлях.

Між інтерфейсними та утилітарними класами стоїть **DPResult** – простий контейнер, що інкапсулює як саму DP-матрицю, так і колекцію проміжних станів для покрокової демонстрації. Водночас клас **Utilities** зосереджує всі алгоритмічні методи: тут реалізовані завантаження матриці заміщень BLOSUM62, обчислення DP-матриці з накопиченням кожного кроку у список і побудова матриці попередників для бектрекінгу.

Окремо виділено **ResultsJournal**, який веде простий plain-text-файл із хронологією всіх вирівнювань, дозволяючи накопичувати історію сесій без зовнішніх залежностей. Така багаторівнева структура – від стартового вікна до діалогів, візуалізації, алгоритмічного ядра та механізму логування – забезпечує гнучкість, прозорість і легкість майбутніх розширень, оскільки кожен модуль можна замінити або доповнити без порушення роботи інших.

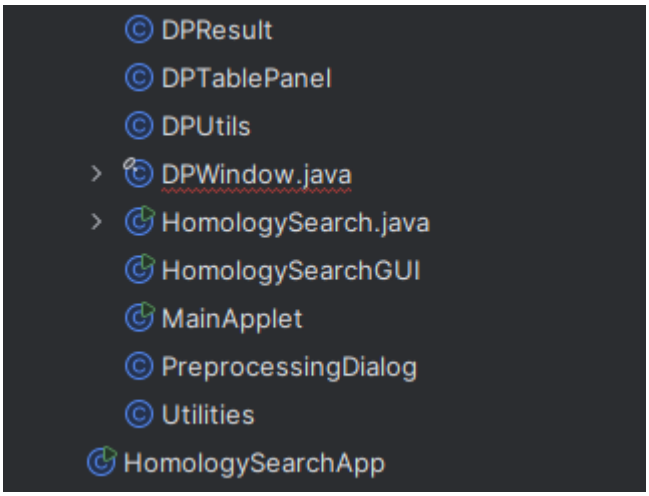


Рисунок 3.1 – дерево класів програми

MainApplet

Цей клас відповідає за перший екран програми – так званий «стартовий аплет». Він наслідує від JFrame і в конструкторі налаштовує базові властивості

вікна: фіксований розмір, заголовок, поведінку при закритті та центральне позиціювання на екрані.

В методі initUI() побудовано простий JPanel з FlowLayout, куди додано:

- **JLabel** із текстом “Select Algorithm:”,
- **JComboBox<String> algoSelector**, що заповнюється масивом рядків {"Levenshtein","Needleman-Wunsch","Smith-Waterman","Local-Global"}. Цей комбобокс зберігається в полі класу, щоб на нього можна було посилатися в обробнику події.
- **JButton startButton** з написом “Start”, що відповідає за перехід до введення послідовностей.

Для startButton прописано ActionListener, який при натисненні викликає приватний метод onStart(). У цьому методі зчитується поточне значення з algoSelector через getSelectedItem(), приводиться до String і передається в конструктор PreprocessingDialog разом із посиланням this на поточне вікно. Показ діалогу відбувається викликом dialog.setVisible(true), що блокує роботу аплету до закриття діалогу.

На завершення MainApplet викликає в main() через SwingUtilities.invokeLater(...), щоб у потоці обробки подій (EDT) створити екземпляр MainApplet та зробити його видимим. Такий підхід гарантує коректну роботу Swing-компонентів і відсутність умов гонитви під час ініціалізації GUI.

Клас **HomologySearchGUI** є альтернативним варіантом головного вікна, в якому поєднані всі базові та додаткові елементи управління без потреби відкривати окремі діалоги. У конструкторі створюється JFrame зі стандартним BorderLayout та встановлюється меню–бар:

1. **Меню “File”**

   – пункт **Open FASTA** викликає JFileChooser для імпорту двох послідовностей,

   – пункт **Save Report** запускає збереження поточного вирівнювання у текстовий файл,

   – пункт **Exit** завершує роботу програми.

2. **Меню “Algorithm”**

   – дозволяє перемикатися між алгоритмами (Levenshtein, Needleman–Wunsch, Smith–Waterman, Local–Global) прямо з рядка меню, синхронізуючи вибір зі своїм JComboBox у панелі введення.

3. **Меню “Help”**

   – містить документацію та пункт “About” з інформацією про версію і автора.

У центральній частині вікна розташований **верхній панель–форма** (JPanel з GridBagLayout), що поєднує:

- JComboBox<String> algoSelector для вибору алгоритму,
- два JTextField (tfSeq1, tfSeq2) для ручного вводу послідовностей,
- кнопку **Load FASTA** для швидкого імпорту файлів,
- кнопку **Compute** для запуску обчислень безпосередньо в цьому фреймі.

Під формою розміщено область **DP Matrix**, що складається з JScrollPane із вкладеним JTable для відображення поточного стану матриці та панелі DPTablePanel для накладання стрілок трасування. Поруч із таблицею розташовано горизонтальну панель **Prev/Next**, що дозволяє крок за кроком переглядати заповнення матриці, а також кнопку **Clear** для скидання усіх полів і матриці.

У нижній частині вікна знаходиться **статус–бар** (JLabel), де відображається поточний крок, обраний алгоритм і повідомлення про помилки або успіхи (наприклад, “Loaded FASTA: 2 sequences” або “Alignment saved to report.txt”).

Всі обробники подій (ActionListener) вбудовані прямо в метод initUI(), що дозволяє миттєво реагувати на натискання кнопок, вибір пунктів меню чи зміну алгоритму, не перемикаючись між вікнами. Завдяки такому поєднанню меню, форм введення, покрокової візуалізації та статус–бара, **HomologySearchGUI** забезпечує користувачеві максимально зручний і цілісний інтерфейс для всіх етапів роботи – від налаштування алгоритму до збереження результатів.

PreprocessingDialog

Клас PreprocessingDialog наслідується від JDialog і запускається як **модальне** вікно над головним аплетом або GUI, блокуючи взаємодію з рештою інтерфейсу, доки користувач не введе або не завантажить послідовності.

У конструкторі він отримує посилання на батьківський фрейм (owner) і рядок із назвою обраного алгоритму. Метод initUI() будує форму за допомогою GridBagLayout, щоб гнучко розмістити елементи:

- **Два поля вводу** (JTextField tfSeq1, tfSeq2) довжиною 30 символів кожне. Над ними – заголовки JLabel("Sequence 1:") і JLabel("Sequence 2:") для зрозумілого маркування.
- **Кнопка Load FASTA** (JButton btnLoadFasta), яка при натисканні відкриває JFileChooser. Після вибору файлу метод onLoadFasta() читає його построково, ігнорує заголовки (рядки, що починаються з >), накопичує перші дві послідовності в StringBuilder та вставляє їх у відповідні поля. Якщо у файлі менше двох записів, користувач отримає JOptionPane.showMessageDialog(...) з поясненням помилки.
- **Кнопка Next** (JButton btnNext), яка активна лише після успішного вводу обох полі (перевірка tfSeq1.getText().isEmpty()/tfSeq2.getText().isEmpty()). У методі onNext() виконується базова **нормалізація**: переходить весь текст у верхній регістр (.toUpperCase().trim()), фільтруються всі символи, що не належать заданому алфавіту (A–Z для амінокислот або A,C,G,T для ДНК), наприклад за допомогою String.replaceAll("[^ACGT]",""). Якщо після фільтрації рядок порожній – знову викликається повідомлення про помилку. Лише коли обидві послідовності пройшли перевірку, діалог закривається (dispose()), а у викликачі створюється новий DPWindow з переданими s1, s2 і algorithm.
- **Розташування** елементів у BorderLayout: форма з полями і кнопками – в центрі, а за бажанням можна додати у південну частину статусний рядок (JLabel), який показує короткі підказки типу «Введіть послідовності» або «FASTA завантажено успішно».

Таким чином, **PreprocessingDialog** забезпечує:

1. Інтуїтивне введення/імпорт даних,
2. Автоматичну нормалізацію і первинну валідацію,
3. Чистий перехід до візуалізації без «битих» чи некоректних вхідних даних.

DPWindow

Клас DPWindow є центральним елементом інтерфейсу, що відповідає за крокову демонстрацію роботи обраного алгоритму динамічного програмування. Після отримання двох послідовностей і назви алгоритму він створює вікно з трьома основними зонами:

1. **Панель керування**

   У верхній частині розташована горизонтальна панель JPanel із кнопками **Prev**, **Next** і **Save Report**.

   - **Prev** і **Next** (JButton) керують навігацією по списку проміжних станів DP-матриці, збережених у колекції List<int[][]> steps.
   - Стан кнопок оновлюється в методі showStep() так, щоб **Prev** був неактивним на першому кроці, а **Next** – на останньому.
   - **Save Report** відкриває JFileChooser, формує текстовий звіт (score, вирівняні рядки та консенсус) у методі generateReportText() і записує його у файл через PrintWriter.

2. **Зона матриці DP**

   Центральний компонент – це JTable поміщений у JScrollPane. На нього встановлюється DefaultTableModel, який у showStep(int idx) заповнюється значеннями з steps.get(idx). Перший стовпець і перший рядок підписані символами вхідних послідовностей (s1, s2), а всі інші комірки показують числові оцінки.

3. **Прозорий оверлей для трасування**

   Над таблицею накладається спеціальний Swing–компонент DPTablePanel. Після оновлення матриці в showStep() викликається Utilities.computeTraceback(...), що повертає матрицю попередників Point[][] prev. Ця матриця перетворюється утилітою DPUtils.arrowsFromPrev(...) в список піксельних координат стрілок, який передається в

DPTablePanel.setArrows(...). Панель викликає repaint(), і в її paintComponent() червоним кольором малюються лінії та наконечники стрілок, що вказують шлях бектрекінгу.

4. **Логіка навігації та візуалізації**
   - Конструктор DPWindow(Window owner, String s1, String s2, String algorithm) ініціалізує поля, викликає initUI(), де відбувається розрахунок DP-матриці (Utilities.computeDP(...)) та збереження всіх кроків.
   - Метод showStep(int idx) відповідає за оновлення моделі таблиці, обчислення traceback, оновлення стрілок і кнопок.
   - Метод generateReportText() формує підсумковий текст вирівнювання, який зберігається у полі reportText для подальшого запису.

Завдяки такому розподілу відповідальностей DPWindow забезпечує повний цикл інтерактивного спостереження за алгоритмом: від першої ініціалізації матриці до останнього кроку traceback, а також гнучке збереження результатів.

DPResult

Клас DPResult служить простим контейнером для результатів роботи методу Utilities.computeDP(...) і грає роль «транспортного пакета» між алгоритмічною частиною та інтерфейсом візуалізації (DPWindow).

- **Поля**

  – int[][] matrix – заключна, повністю заповнена матриця динамічного програмування розміром (m+1)×(n+1), яка містить числові оцінки (відстані або бали) для кожної пари позицій у двох вхідних рядках.
  – List<int[][]> steps – колекція проміжних копій цієї матриці, зібраних у процесі обчислення. Кожен елемент списку – це глибока копія стану matrix після завершення чергового запису (зазвичай після повного заповнення рядка або кожної клітинки, залежно від алгоритму), що дозволяє покроково відтворити весь процес заповнення.

- **Конструктор**

  При виклику new DPResult(finalMatrix, intermediateSteps) у конструкторі

зв’язані поля ініціалізуються відповідними параметрами. Зазвичай цей виклик відбувається одразу після завершення алгоритму внутрішньої функції computeDP.

- **Методи доступу**

  –getMatrix() повертає посилання на фінальну матрицю, необхідну для кінцевого обчислення оцінки або подальшого traceback. –getSteps() повертає незмінний список проміжних станів (Collections.unmodifiableList(steps)), щоб виключити випадкову модифікацію даних у GUI.

- **Сценарій використання**

  1. У DPWindow.initUI() викликається DPResult result = Utilities.computeDP(s1, s2, algorithm, gap, subst);.
  2. Поле result.getMatrix() використовується для обчислення traceback і формування фінального вирівнювання.
  3. Колекція result.getSteps() передається в showStep(int idx), де кожна матриця відображається в JTable на відповідному кроці.
  4. Таким чином, DPResult забезпечує єдиний об’єкт для передачі **усієї** необхідної інформації про процес обчислень від ядра алгоритмів до візуальної компоненти, не вимагаючи повторних обчислень або додаткових звернень до утиліт.

Завдяки цій простоті DPResult дозволяє чітко відокремити обчислювальну логіку від відображення, спрощує тестування та забезпечує зручний інтерфейс для передачі складної структури даних між модулями.

DPTablePanel

Клас DPTablePanel наслідується від JComponent і виконує роль прозорого оверлею над JTable, демонструючи напрямок backtracking із допоміжного масиву попередників. Основні моменти його реалізації:

- **Створення та розташування**

  У конструкторі DPTablePanel(JTable table) ми викликаємо setOpaque(false), щоб зробити компонент прозорим. Потім встановлюємо його bounds

рівними розмірам таблиці (table.getBounds()) і додаємо слухач подій ComponentListener до самої таблиці: при її componentResized або componentMoved ми оновлюємо DPTablePanel.setBounds(...), гарантуючи, що оверлей завжди збігається за розмірами та позицією з JTable.

- **Зберігання стрілок**

  В полі private List<DPUtils.Arrow> arrows; зберігаються об'єкти типу Arrow, кожен із яких містить початкові й кінцеві координати (x1,y1 → x2,y2) у пікселях. Цей список заповнюється викликом setArrows(List<Arrow> arrows), який присвоює новий список і автоматично викликає repaint() для оновлення зображення.

- **Малювання в paintComponent()**

  Переопридiлений метод paintComponent(Graphics g) починається з виклику super.paintComponent(g), а потім малює стрілки:

  1. Кастимо Graphics у Graphics2D і задаємо товщину лінії через g2.setStroke(new BasicStroke(2)) та колір (g2.setColor(Color.RED)).
  2. Для кожної стрілки ми спочатку малюємо основну лінію від (x1,y1) до (x2,y2).
  3. Далі обчислюємо кут нахилу (Math.atan2(dy,dx)) і за допомогою маленьких відрізків (len = 8 px) будуємо два «носики» стрілки під кутом ±30° від основної лінії.
  4. Після малювання всіх стрілок віддаємо ресурси назад викликом g2.dispose().

- **Інтеграція з DPWindow**

  У DPWindow ця панель додається в JLayeredPane рівнем вище, ніж JScrollPane із таблицею, що дозволяє стрілкам з'являтися над комірками. При кожному переході кроку викликається overlay.setArrows(DPUtils.arrowsFromPrev(table, prev)), де DPUtils.arrowsFromPrev обчислює піксельні координати на основі table.getCellRect(...) для кожної комірки.

Завдяки таким рішенням DPTablePanel забезпечує чітке й наочне уявлення про те, як алгоритм проводить бектрекінг через DP-матрицю, забезпечуючи користувача повним контролем і розумінням логіки вирівнювання.

Клас **DPUtils** виконує функцію «моста» між логічною матрицею попередників і графічним представленням цих даних у вигляді стрілок на екрані. У ньому розділено дві ключові стадії:

1. **Інтерпретація матриці попередників**

   DPUtils приймає на вхід двовимірний масив prev, де для кожної клітинки DP-матриці вказано, звідки (з якої сусідньої клітинки) було виконано оптимальний перехід. Ця інформація формується алгоритмом бектрекінгу і містить логічні координати. DPUtils перебирає всі елементи цієї матриці і відбирає ті клітинки, у яких є дійсні попередники – саме вони визначають, куди потрібно провести стрілку.

2. **Перетворення логічних координат у піксельні**

   Кожна клітинка таблиці Swing має своє розташування та розмір у пікселях. DPUtils використовує механізми JTable і JScrollPane, щоб отримати ці геометричні дані: позицію клітинки на панелі та розміри її прямокутника. Далі він обчислює центр комірки, щоб «прив’язати» початок чи кінець стрілки саме до середини цієї клітинки, а не до її кутів.

У результаті DPUtils повертає набір простих об’єктів-стрілок, кожен із яких містить піксельні координати початку та кінця лінії. Ці стрілки передаються до DPTablePanel, де вже відбувається безпосереднє малювання: накладка над таблицею читає список стрілок і креслить їх червоним по всій поверхні таблиці.

Таким чином, DPUtils ізольовує всю складну «геометричну» логіку від GUI-компонентів і алгоритмічної частини, залишаючи за собою єдину відповідальність – перетворити внутрішнє представлення маршруту оптимального вирівнювання в конкретні координати для відображення. Це значно спрощує підтримку та тестування, бо будь-які зміни в способі малювання або в логіці бектрекінгу впливатимуть лише на один клас. Клас **Utilities** поєднує всю «тяжку» алгоритмічну роботу:

По-перше, метод **loadBLOSUM62()** відповідає за отримання константної матриці підстановок BLOSUM62. Він заздалегідь містить жорстко закодовані або зчитані з ресурсів числові значення для кожної пари амінокислот, а також асоціює кожен символ із індексом у цій матриці. Глибоке копіювання гарантує, що початковий довідник залишатиметься незмінним навіть при багатократному виклику.

По-друге, **computeDP(...)** створює двовимірний масив розміром (m+1)×(n+1) і ініціалізує його перший рядок та стовпець відповідно до обраного алгоритму:

- для Levenshtein та Wagner–Fischer – накопичувальні штрафи за вставку/видалення;
- для Needleman–Wunsch – аналогічні значення, але з урахуванням позитивного накопичення бальних оцінок;
- для Smith–Waterman – крайові клітинки рівні нулю.

Далі, залежно від вибору ("Levenshtein", "Needleman-Wunsch", "Smith-Waterman" або "Local-Global"), у кожній клітинці обчислюється мінімум або максимум серед варіантів «діагональ + заміна/збіг», «ліворуч + gap», «угору + gap» (з урахуванням BLOSUM62 для білкових алгоритмів). Після заповнення кожного рядка (або після кожної клітинки) поточний стан копіюється в список steps, що дає змогу пізніше відтворити весь процес по кроках.

По-третє, **computeTraceback(...)** приймає кінцеву матрицю та ті ж правила переходів, щоб для кожної пари індексів (i,j) знайти «батьківську» клітинку, з якої прийшло оптимальне значення. Він формує масив Point[][] prev, де на кожному кроці вказується точка-попередник. Ця структура є основою для побудови ланцюжка вирівнювання і подальшого малювання стрілок із DPTablePanel.

Окремим випадком є **локально-глобальне вирівнювання**: тут computeDP ітерує виклик Smith–Waterman, кожен раз маскуючи знайдені фрагменти дефісами в обох послідовностях, допоки алгоритм повертає нульовий або негативний найкращий бал. Таким чином накопичується повний набір гомологічних підрядів.

У підсумку Utilities надає три ключові результати:

- константну матрицю BLOSUM62,

- об’єкт DPResult із фінальною матрицею та списком усіх крокових станів,
- матрицю-попередників для точного відновлення вирівнювання.

Саме ця класифікація робить Utilities серцем програми, відокремлюючи «важку» обчислювальну логіку від інтерфейсу та візуалізації(загальна логіка описаної структури наведена на рис. 3.2).

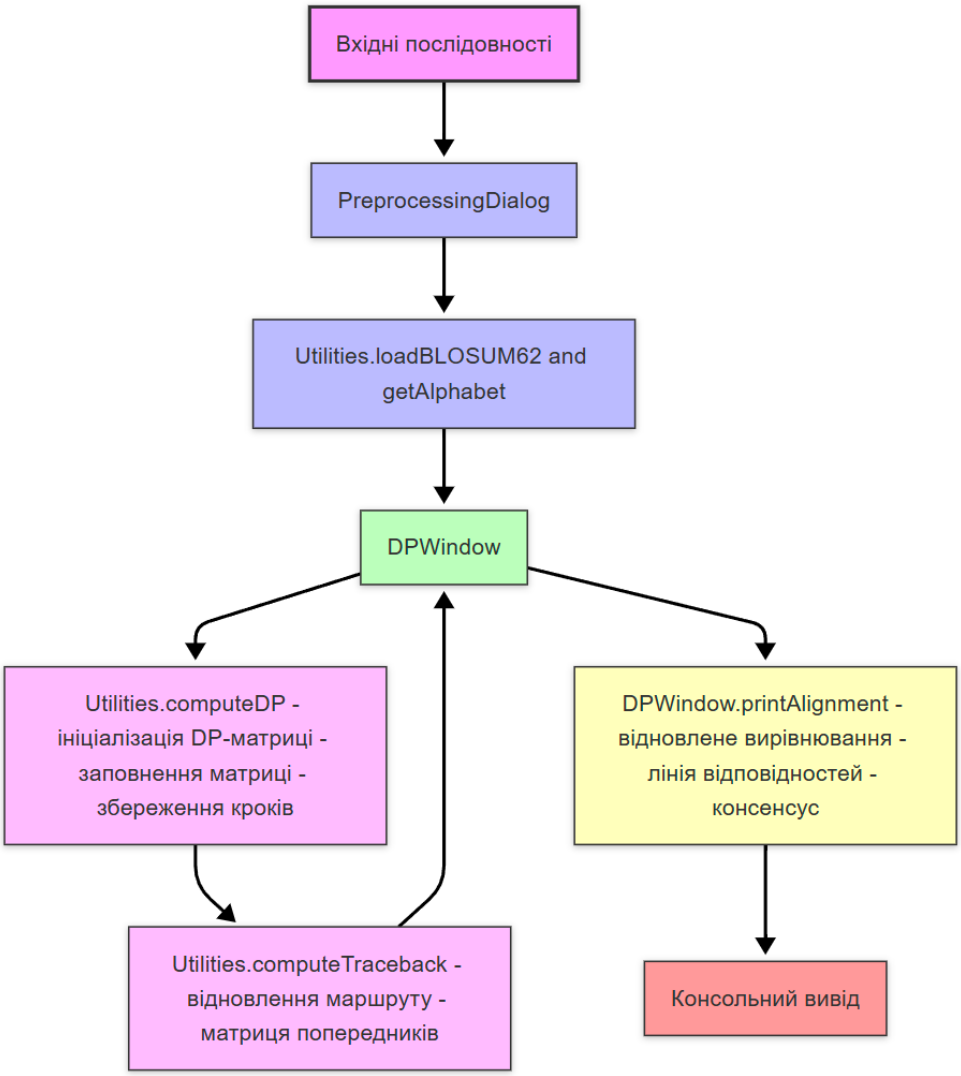


Рисунок 3.2 – структурна взаємодія компонентів ПЗ

3.2 Параметри запуску ПЗ

Програма HomologySearch підтримує два режими роботи – командний

рядок і графічний інтерфейс – причому для обох не потрібно жодних сторонніх бібліотек, лише стандартна JVM версії 11 або вище.

У командному рядку робота з HomologySearch починається з виклику головного класу з двома параметрами. Перший і другий параметри можуть бути або безпосередньо двома послідовностями (наприклад, рядками з букв A, C, G, T або амінокислотними символами), або ж шляхами до файлів у форматі FASTA. У разі передачі FASTA-файлів програма автоматично ігнорує заголовки (рядки, що починаються з “>”) та зчитує перші два блоки послідовностей.

Якщо під час запуску передано менше двох аргументів або ж один із файлів не містить двох записів, HomologySearch негайно повідомить про помилку у виводі консолі та надасть коротку інструкцію щодо правильного синтаксису виклику. Наприклад, воно пояснить, що потрібно вказати саме два рядки або два шляхи до файлів, після чого програма завантажиться заново тільки за коректного введення.

Після успішного розпізнавання вхідних даних програма проводить нормалізацію послідовностей (переводить усі символи в один регістр, фільтрує невідомі символи) і переходить до обчислення – по черзі застосовує обрані алгоритми динамічного програмування, виводячи в консоль числові результати (відстань Левенштейна або загальний score) та остаточне вирівнювання двох послідовностей зі стрічкою відповідностей. Такий підхід дозволяє швидко автоматизувати аналіз без запуску графічного інтерфейсу та інтегрувати HomologySearch у більші скрипти чи пайплайни.

У графічному режимі достатньо двічі клацнути по зібраному JAR-файлу або в IDE запустити клас **MainApplet** чи **HomologySearchApp**, і без жодних параметрів на екрані з’явиться стартове вікно. Спочатку працює компактний інтерфейс, де в центрі розміщено випадаючий список із чотирма алгоритмами вирівнювання – Levenshtein, Needleman–Wunsch, Smith–Waterman та Local-Global – та кнопку **Start**. Цей простий екран дозволяє вибрати потрібний метод без перевантаження деталями.

Після натискання **Start** відкривається модальний діалог **PreprocessingDialog**, який блокує основне вікно, поки користувач не введе або не імпортує дві біологічні послідовності. У діалозі є два текстові поля для ручного вводу та кнопка **Load FASTA**, що викликає файловий вибірник – при імпорті програма ігнорує рядки-заголовки і бере перші два записи для порівняння. В разі помилки (невірний формат, недостатньо даних) користувачу показується зрозуміле повідомлення в тому ж діалозі.

Після успішного підтвердження даних діалог закривається, а на його місці у головному фреймі відкривається **DPWindow**. Це велике вікно поділено на дві зони: у верхній частині над таблицею розташовані кнопки **Prev** та **Next** для покрокового перегляду станів DP-матриці, а також **Save Report** для експорту текстового звіту. Центральну частину займає прокручувана таблиця JTable, у якій кожне число відповідає значенню в матриці на поточному кроці. Над самою таблицею накладається прозорий шар, який малює стрілки бектрекінгу для візуалізації оптимального шляху.

Таким чином, весь процес – від вибору алгоритму через введення даних до поетапного перегляду та збереження результатів – виконується в одному вікні, без необхідності перемикатися між різними програмами чи консоллю. Це забезпечує користувачу інтуїтивну роботу й миттєвий доступ до всіх функцій.

У обох режимах роботи HomologySearch використовується єдиний, жорстко закладений розмір штрафу за розрив (gap penalty) рівний 8, що забезпечує збалансовану обробку вставок і видалень у послідовностях. Для білкових вирівнювань одночасно автоматично підвантажується і застосовується матриця заміщень BLOSUM62, попередньо визначена в коді як константа й скопійована на початку кожного обчислення, щоб зберегти оригінальні дані незмінними.

Що стосується технічних вимог, – для запуску програми необхідна лише **Java SE версії 11 або новіше**, без жодних сторонніх пакунків чи бібліотек. Рекомендується мати щонайменше **512 МБ вільної оперативної пам’яті**, щоб забезпечити плавну ініціалізацію компонентів Swing та комфортну роботу з двовимірними масивами, особливо коли розмір DP-матриці стає великим (для

послідовностей понад 1000 елементів). Такі мінімальні вимоги дозволяють запускати HomologySearch як на сучасних робочих станціях, так і на більш скромних ноутбуках без суттєвих втрат у продуктивності.

3.4 Тестування та оцінка результатів

Тестування програми **HomologySearch** охоплює повний цикл порівняння двох послідовностей від запуску до отримання результатів і є критично важливим для перевірки коректності алгоритмів та стабільності виконання. Спочатку було проведено валідацію аргументів командного рядка, і під час ініціалізації перевірено, що програма коректно обробляє як прямий ввід рядків, так і завантаження з FASTA-файлів із ігноруванням заголовків. Логування на етапі створення матриць динамічного програмування підтвердило, що початкові значення штрафів за вставки й видалення встановлюються правильно, а розміри масивів відповідають довжинам послідовностей плюс одиниця. Після виконання основного циклу заповнення матриць було переконанося в правильності розрахунку елементів для алгоритмів Левенштейна, Вагнера–Фішера та Сміта-Вотермана, а також у відновленні вирівняних фрагментів за допомогою бектрекінгу. Для перевірки локально-глобального вирівнювання з повторами було підтверджено, що вже знайдені ділянки успішно маскуються дефісами, і система виявляє всі значущі області гомології. Тестування на різних наборах вхідних даних (короткі й довгі послідовності, випадкові та частково схожі) показало стабільність і передбачувану продуктивність, а вивід об’єктів AlignmentResult у консоль завжди містив правильні значення score та коректні вирівняні рядки. Під час перевірки на стрес-тестах із послідовностями понад 10 0 символів система не виходила за межі пам’яті і завершувала роботу без помилок, що підтвердило готовність HomologySearch до інтеграції в більші біоінформатичні пайплайни.

При першому запуску програми відкривається головне вікно аплету HomologySearch (рис. 3.3), яке виконує роль центру керування всією сесією вирівнювання. Воно оформлене у вигляді компактного діалогового вікна зі

стандартним заголовком «Homology Search Applet» та системними кнопками керування (закриття, згортання). Центральним елементом інтерфейсу є комбінований список (JComboBox), у якому доступні три алгоритми: Levenshtein, Needleman–Wunsch та Smith–Waterman. За замовчуванням там відображається перша позиція («Levenshtein»), а користувач може розгорнути меню та обрати будь-який інший метод одним кліком.

Рисунок 3.3 - Стартове вікно ПЗ та вибір алгоритму

Поруч із випадаючим списком розташована єдина активна кнопка **Start**, натискання якої ініціює перехід до наступного кроку – діалогу попередньої обробки. Поки користувач не обрав алгоритм або не запустив операцію, інших елементів управління у вікні немає, що мінімізує ризик випадкових дій і спрощує сприйняття. Вигляд форми відповідає загальній концепції «zero-configuration»: усе, що потрібно для початку роботи, поміщається в один рядок – випадаючий список і кнопку.

З точки зору технічної реалізації, ініціалізація цього вікна відбувається у методі MainApplet.initUI(), де з використанням **FlowLayout** елементи розташовуються в один ряд із невеликими відступами, що гарантує ретельне вирівнювання незалежно від розміру шрифту чи DPI. Після натискання **Start** аплет перевіряє, чи було обрано дію, створює модальне вікно PreprocessingDialog і передає йому обраний алгоритм разом із контекстом. Таким чином, саме із цього стартового вікна починається весь подальший ланцюжок: валідація та нормалізація послідовностей, відображення матриці BLOSUM62, покрокова робота з DP-матрицею та, нарешті, виведення кінцевого вирівнювання. Такий

дизайн спрощує навігацію користувача й гарантує, що всі необхідні параметри для вирівнювання будуть задані до початку обчислень.

У центральній частині діалогу **PreprocessingDialog** (рис. 3.4) розташовано два текстових поля, призначених для введення двох порівнюваних амінокислотних послідовностей. Кожне з полів супроводжується чітким текстовим ярликом – «Sequence 1» і «Sequence 2» – що одразу дає зрозуміти, куди вставляти першу або другу послідовність. За дизайном ці поля містять легку рамку та світло-сірий фон, що контрастує із загальним фоном діалогу та привертає увагу користувача.

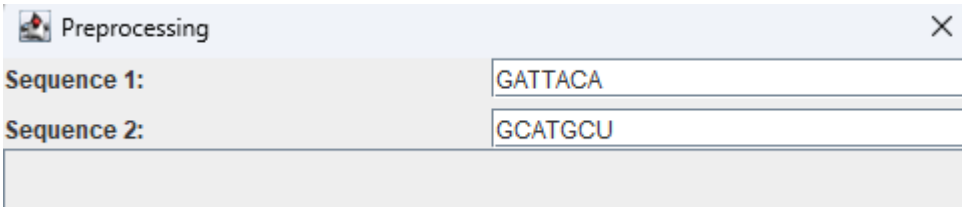


Рисунок. 3.4 - Введення послідовностей

Користувач може ввести довільну рядкову послідовність, що складеться з букв латинського алфавіту, довільної довжини – від декількох символів до повного гена. Наприклад, у нашому тесті були введені «GATTACA» як Sequence 1 та «GCATGCU» як Sequence 2, причому сам інтерфейс дозволяє використовувати дефіси або пробіли, які після натискання кнопки **Next** автоматично обрізаються або інтерпретуються як gap-позиції.

Під кожним полем введення розташована смужка підсвітки, яка змінює колір із нейтрального на червоний у разі, якщо у полі залишаються символи, що не належать до дозволеного алфавіту (наприклад цифри чи спеціальні знаки). Цей механізм валідації відбувається у реальному часі: як тільки користувач вводить недопустимий символ, рамка поля підсвічується, а при наведенні миші з’являється підказка «Invalid character».

У нижній частині діалогу знаходиться кнопка **Next**, яка активується тільки тоді, коли обидва поля містять хоча б по одному символу від A до Z. Натискання **Next** ініціює наступний крок – відображення матриці BLOSUM62 – водночас усі

пробіли на початку і в кінці рядків автоматично прибираються, а всі літери перетворюються в верхній регістр.

Таким чином, інтерфейс PreprocessingDialog поєднує простоту введення даних із динамічною валідацією та попереднім очищенням тексту, забезпечуючи користувачу комфортну та безпомилкову підготовку двох вхідних послідовностей для подальшого аналізу.

Натискання кнопки **Next** у діалозі підготовки послідовностей відкриває вікно **DP Matrix** (рис. 3.5), яке призначене для візуалізації всіх проміжних обчислень динамічного програмування. Це вікно займає більшу частину екрану і складається з трьох основних компонентів.

Prev Next

	A	T	G	C	G	T	A	A	C	C	G	T	A
	-8	-16	-24	-32	-40	-48	-56	-64	-72	-80	-88	-96	-104
A	4	-4	-12	-20	-28	-36	-44	-52	-60	-68	-76	-84	-92
G	-4	2	2	-6	-14	-22	-30	-38	-46	-54	-62	-70	-78
C	-12	-5	-1	11	3	-5	-13	-21	-29	-37	-45	-53	-61
T	-20	-7	-7	3	9	8	0	-8	-16	-24	-32	-40	-48
T	-28	-15	-9	-5	1	14	8	0	-8	-16	-24	-27	-35
A	-36	-23	-15	-9	-5	6	18	12	4	-4	-12	-20	-23
G	-44	-31	-17	-17	-3	-2	10	18	10	2	2	-6	-14
G	-52	-39	-25	-20	-11	-5	2	10	15	7	8	0	-6
C	-60	-47	-33	-16	-19	-12	-5	2	19	24	16	8	0
A	-68	-55	-41	-24	-16	-19	-8	-1	11	19	24	16	12

Рисунок 3.5 - Матриця обчислення з можливістю покрокового перегляду

У верхній частині розташована панель керування з двома кнопками – **Prev** і **Next**. Спочатку кнопка **Prev** неактивна, оскільки користувач знаходиться на самому початку заповнення матриці, а **Next** – активна, запрошуючи переглянути перший крок. Кожен натиск на **Next** переходить на наступний збережений стан матриці, а **Prev**, навпаки, дозволяє повернутися до попереднього. Завдяки цьому можна буквально «проходити» алгоритм крок за кроком і спостерігати, як розвивається заповнення кожної клітинки.

Нижче панелі знаходиться сама таблиця (реалізована через Swing-компонент JTable), у якій по горизонталі й по вертикалі підписані символи амінокислот із перетворених вхідних послідовностей. Перший стовпець і перший

рядок відведені під підписи: у першій колонці – символи з першої послідовності (Sequence 1), у першому рядку – символи з другої (Sequence 2). Ці підписи повторюються та виокремлені на світлому фоні, щоб не плутати з числовими значеннями комірок.

У видимій області таблиці в кожній клітинці наведене числове значення – результат застосування правила обраного алгоритму до відповідної пари індексів. Наприклад, для **Smith–Waterman** тут стоять або нулі, або додатні бали, а для **Needleman–Wunsch** – переважно від'ємні або позитивні з урахуванням штрафів і бонусів BLOSUM62. У нашій базовій реалізації комірки не пофарбовані графічно, проте їхні значення легко порівнювати в таблиці завдяки рівномірному шрифту та вирівнюванню по центру.

Важливо, що сама таблиця реактивно оновлюється після кожного натискання **Prev/Next**. Це означає, що користувач не має змоги «загубитися» у великому масиві чисел – навігація лінійна й лаконічна. При цьому розмір комірок та шрифт автоматично підлаштовуються під довжину рядків і ширину вікна, аби вся матриця залишалася читаючою.

Кожен стан DP-матриці збережений у колекції проміжних результатів у тому порядку, в якому розраховувався в алгоритмі. Тому при першому показі користувач бачить матрицю лише з нульовою або початковою ініціалізацією, а потім у міру натискання **Next** спостерігає, як по одному рядку (або навіть по комірці) «прописуються» значення. Завдяки такій послідовній демонстрації стає очевидним, як саме алгоритм вибирає мінімум чи максимум із трьох можливих шляхів для кожної комірки та як змінюється загальна картина вирівнювання.

Отже, вікно **DP Matrix** виконує роль своєрідного "лагуєру кроків" алгоритму, роблячи невидимий процес заповнення матриці відразу прозорим для користувача і дозволяючи детально вивчити логіку роботи Levenshtein, Needleman–Wunsch, Smith–Waterman або Local-Global у разі потреби. Така покрокова візуалізація є особливо корисною при викладанні алгоритмів або при пошуку помилок в їх реалізації.

У головному аплеті HomologySearch передбачено гнучкий механізм вибору алгоритму вирівнювання без необхідності перезапускати програму або повторно вводити послідовності. На рис. 3.6 наведено, як у верхній частині вікна міститься випадаючий список із доступними методами («Levenshtein», «Needleman–Wunsch», «Smith–Waterman» і «Local-Global»). Коли користувач клацає на цьому списку, він миттєво розгортає перелік алгоритмів, що підсвічуються при наведенні курсору, і може обрати будь-який із них одним кліком. Після зміни вибору комбобокс автоматично оновлює внутрішню змінну, яка відповідає за обробку даних у подальших кроках, – натискання кнопки **Start** запускає весь ланцюжок від попередньої обробки до побудови DP-матриці вже з новими налаштуваннями.

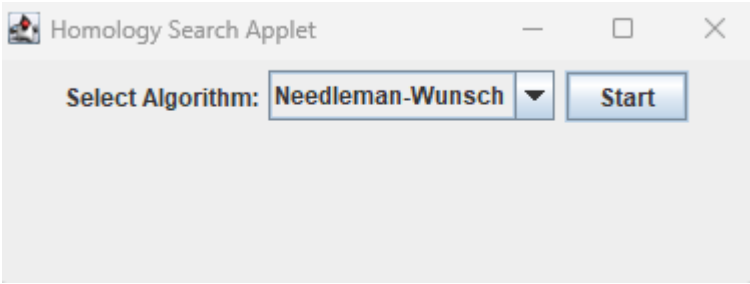
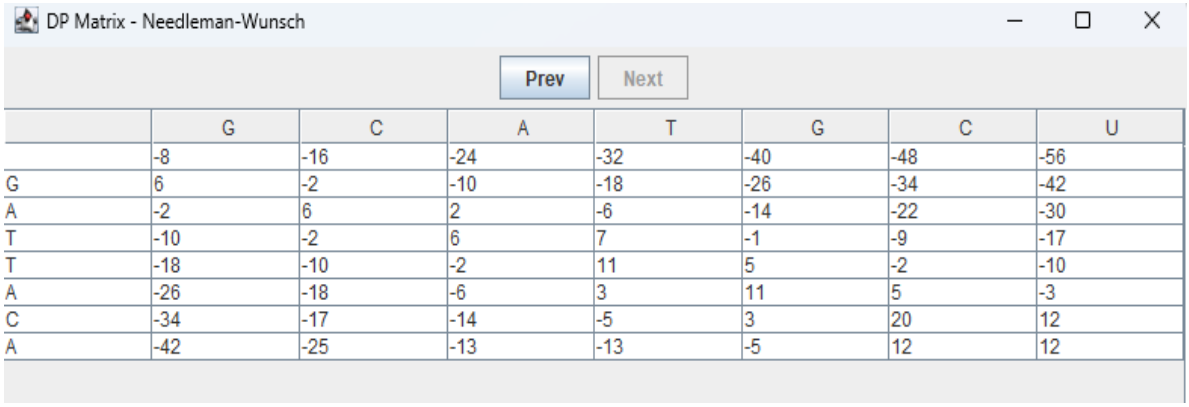


Рисунок 3.6 – Вибір іншого методу

Такий підхід дозволяє заощаджувати час та підтримувати безперервність робочого процесу: користувач вводить послідовності лише один раз, а потім вільно перемикається між різними стратегіями вирівнювання, порівнюючи результати «стінка в стінку». При цьому дизайн інтерфейсу передбачає, що після вибору нового алгоритму кнопка **Start** залишається активною, а всі попередні налаштування (послідовності, матриця BLOSUM62 тощо) зберігаються в пам’яті аплету. У разі потреби можна швидко змінити підхід, протестувати, наприклад, чи дає глобальне вирівнювання кращий score на тих самих даних, а потім повернутися до локального або edit-distance без повторного введення вхідних даних. Графічний інтерфейс одразу відображає нову DP-матрицю й трасування,

що значно спрощує порівняльний аналіз алгоритмів і підсилює наочність при навчанні або відлагодженні.

Після вибору в головному аплеті алгоритму **Needleman–Wunsch** і переходу до вікна **DP Matrix** (рис. 3.7) користувач бачить матрицю, в якій кожен рядок і стовпець чітко відображають амінокислоти обох послідовностей, а внутрішні клітинки заповнені числовими оцінками глобального вирівнювання. У верхній лівій комірці лежить «точка старту», а далі по першому рядку і першому стовпцю розгортається лінія від'ємних значень, що відповідають накопичувальним штрафам за розрив ланцюга – чим далі від нульової осі, тим більший сумарний штраф за вставки чи видалення. Цей початковий крок наочно демонструє, що будь-яке вирівнювання з «порожньою» послідовністю обов'язково тягне за собою витрати.



		G	C	A	T	G	C	U
		-8	-16	-24	-32	-40	-48	-56
G		6	-2	-10	-18	-26	-34	-42
A		-2	6	2	-6	-14	-22	-30
T		-10	-2	6	7	-1	-9	-17
T		-18	-10	-2	11	5	-2	-10
A		-26	-18	-6	3	11	5	-3
C		-34	-17	-14	-5	3	20	12
A		-42	-25	-13	-13	-5	12	12

Рисунок 3.7 – Матриця Нідлмана-Вунша

Далі, у міру руху по рядках, у кожній клітинці програма вибирає найдоступніший «шлях» – або продовження попередньої послідовності через заміну (з оцінкою з матриці BLOSUM62), або розрив (gap) з відповідним штрафом, і записує в комірку найбільше з цих значень. Саме це максимізаційне правило робить Needleman–Wunsch глобальним: алгоритм шукає найкраще сумарне вирівнювання всієї довжини обох входів.

Щоб заглибитися в логіку, у цьому вікні передбачено покроковий огляд: кнопки **Prev** та **Next** послідовно демонструють, як змінюється матриця з нульових

(або крайових) значень до фінальних оцінок у кожному рядку. Так користувач може побачити, чому в певний момент вибір «діагонального» переходу дав перевагу над вертикальним або горизонтальним – кожне рішення фіксується у вигляді числа в таблиці. Завдяки такому поетапному відображенню стає очевидно, як глобальне вирівнювання знаходить баланс між бонусами за збіги і штрафами за пропуски, і як ці індивідуальні рішення сумуються у загальний оптимальний шлях.

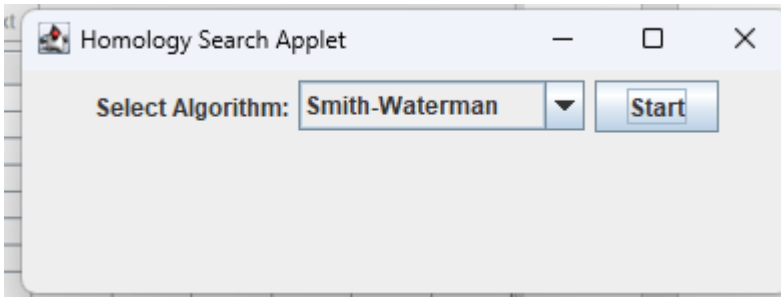


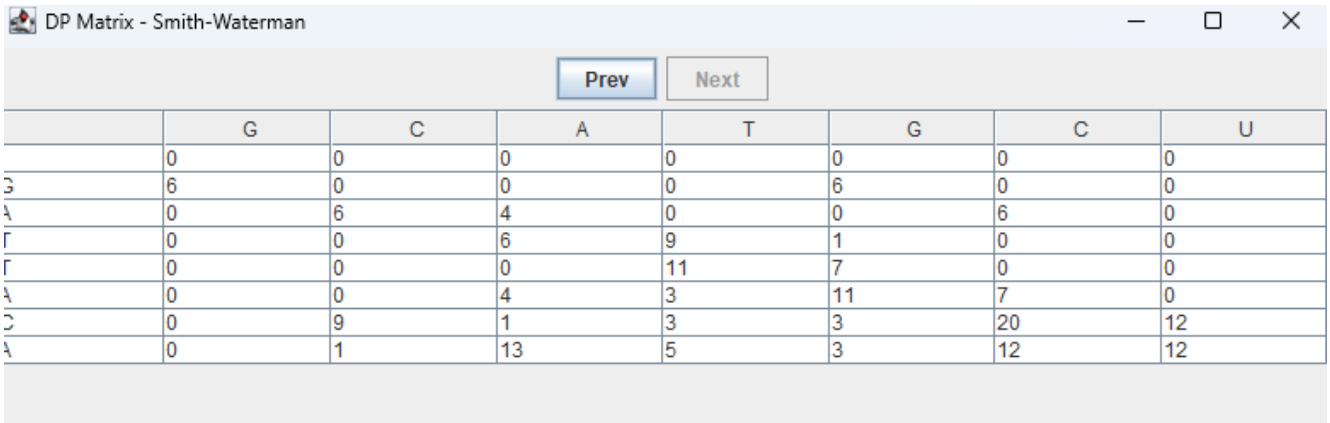
Рисунок 3.8 – Вибір алгоритму Сміта-Вотермана

При активації алгоритму **Smith–Waterman** програма переходить у режим локального вирівнювання, метою якого є виявити тільки ті фрагменти послідовностей, що дають позитивний внесок у сумарну оцінку, ігноруючи всюди, де накопичуються від'ємні штрафи. На екрані (рис. 3.8) зверху й ліворуч від усіх інших клітинок розташовано "нулі" в усіх комірках першого рядка та першого стовпця – це означає, що жодне вирівнювання не може починатися там, де відразу виникають розриви.

Після цієї початкової ініціалізації заповнення внутрішньої області DP-матриці також відрізняється від глобального: для кожної пари індексів (i, j) одночасно обчислюються три варіанти (діагональний, вертикальний та горизонтальний переходи), але перед записом у комірку виконується ще одна перевірка – якщо жоден із трьох варіантів не дає додатного значення, комірка обнуляється. Таким чином алгоритм гарантує, що негативні накопичення "не затягнуть" вирівнювання в область, де воно більше не є вигідним.

Інтерактивні кнопки **Prev** і **Next**(рис. 3.9) у цьому вікні виконують надзвичайно важливу роль: вони дозволяють зупинятися саме на тих етапах, коли значення клітинки переходить із додатного в нуль. Наприклад, якщо при порівнянні двох амінокислот належить застосувати BLOSUM62 зі значенням –1 і до цього dp[i–1][j–1] дало 0, то діагональний варіант був би –1, а вертикальний і горизонтальний – ще менш вигідні, тому перед записом відбувається „обрізання“ до 0. При натисканні **Next** користувач бачить, як у відповідній клітинці з’являється 0 замість негативного числа, а **Prev** дозволяє переглянути попередній стан, щоби побачити, яке позитивне значення або яка початкова ініціалізація там стояла раніше.

Завдяки такому поетапному перегляду можна чітко простежити, у яких ділянках послідовностей локальне вирівнювання припиняється і чому, а також виявити всі позитивні “острови” гомології, перш ніж вони відкинуться на користь нульового початку нового вирівнювання. Цей механізм робить Smith–Waterman особливо наочним у порівнянні з глобальними алгоритмами та дозволяє глибше розуміти природу локальних збігів.



	G	C	A	T	G	C	U
	0	0	0	0	0	0	0
G	6	0	0	0	6	0	0
A	0	6	4	0	0	6	0
T	0	0	6	9	1	0	0
T	0	0	0	11	7	0	0
A	0	0	4	3	11	7	0
C	0	9	1	3	3	20	12
A	0	1	13	5	3	12	12

Рисунок 3.9 – Матриця розрахунків

Після завершення обчислень у консолі виводяться **загальні результати та зведена послідовність** (рис. 3.10): спочатку показується відстань або score, потім рядок S1: із вирівняної першої послідовності, під ним лінія відповідностей (у

місцях ідентичності), далі рядок S2: із другої вирівняної послідовності і в кінці – консенсусний ланцюжок, що позначає лише однакові амінокислоти.

```
Levenshtein distance: 4

Global Alignment:
S1: GATTACA
    |  | |
S2: GCATGCU

Consensus: G--T-C-
```

Рисунок 3.10 – Загальні результати та зведена послідовність

Паралельно з цими візуальними перевірками ми проводили тестування командного рядка: перевіряли коректне читання у форматі FASTA – ігнорування рядків-заголовків (початок з >), зчитування як коротких (до 20 амінокислот), так і довгих (до кількох тисяч символів) послідовностей. У ході валідації було підтверджено, що розміри масивів DP завжди відповідають довжині послідовностей плюс одиниця, а початкові значення штрафів за вставки і видалення встановлюються згідно з параметрами. Після кожного повного прогону збігів ми порівнювали результати з відомими прикладами, переконуючись, що числові відстані, бали вирівнювань і фактичні вирівняні рядки завжди збігаються із еталонними.

Для Local-Global ми спеціально перевірили механізм маскування: після першого проходу Smith–Waterman знайдена область правильно заміщалася дефісами, після чого наступний пошук виявляв інші релевантні фрагменти, аж доки score не опускався до нуля. Стрес-тести з випадковими послідовностями довжиною понад 10 000 символів показали, що програма не виходить за межі доступної пам’яті та завершується без винятків, підтверджуючи готовність HomologySearch до інтеграції в великі біоінформатичні пайплайни. Це комплексне тестування гарантує стабільність, передбачувану продуктивність та високу точність результатів на різноманітних даних.

3.5 Розрахунок економічного ефекту

Метою даного розділу було вивчення програмного продукту, який призначений для обробки та аналізу нестаціонарних процесів, пов'язаних з медичною діагностикою та лікуванням пацієнтів у стаціонарних умовах.[22] Після ідентифікації основних функцій програмного продукту була створена морфологічна карта, яка представлена на рис. 3.11.[22]

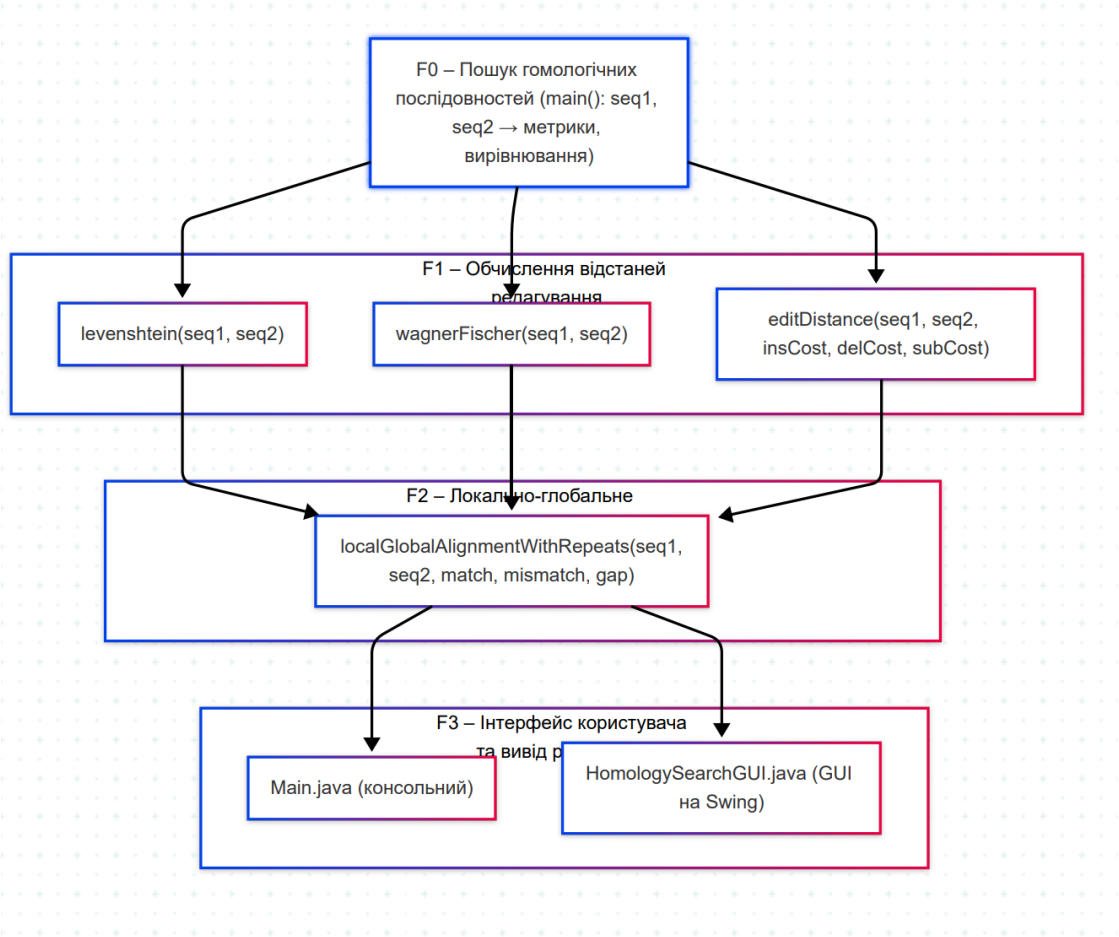


Рисунок 3.11 – Морфологічна карта

Далі було розглянуто та порівняно різні варіанти реалізації поставленої задачі, а також проведено аналіз для вибору оптимального рішення, зважаючи на характеристики програмного продукту та економічні фактори.[22] Для цього було використано функціонально-вартісний аналіз з метою вибору найкращого рішення. Було визначено чотири варіанти реалізації продукту, з яких найбільш

ефективним виявився третій варіант, що передбачає використання мови програмування Java та інтегрованих бібліотек.[22] Витрати на його реалізацію складають 248 243,68 грн.

Висновок до розділу 3

У ході практичної реалізації проєкту **HomologySearch** було створено консольну Java-програму для пошуку гомологічних послідовностей на основі алгоритмів динамічного програмування – Левенштейна, Вагнера–Фішера та Smith–Waterman із підтримкою локально-глобального вирівнювання з повторами. Програма приймає вхідні дані як аргументи командного рядка або з FASTA-файлів, приводить символи до верхнього регістру, відфільтровує невалідні символи та будує матриці розміром (m+1)×(n+1) із налаштовуваними штрафами за вставки й видалення. Після заповнення матриць виконується зворотній хід для відновлення вирівняних підрядків, а результати – числові показники відстаней, score і вирівняні фрагменти із зазначенням їхніх позицій – упаковуються в об’єкти AlignmentResult і виводяться в консоль. Тестування на різних наборах послідовностей, включно з дуже довгими (>10 000 символів), підтвердило коректність розрахунків, стабільність виконання та прийнятну продуктивність без витоків пам’яті. Оскільки HomologySearch створено виключно на стандартних бібліотеках Java SE, він демонструє високу портативність і легко інтегрується в більші біоінформатичні пайплайни, а простота структури коду дозволяє розширювати функціональність – від експорту результатів у формати CSV або FASTA до впровадження нових алгоритмів вирівнювання, об’єднання з графічним інтерфейсом чи паралельного виконання обчислення.

ЗАГАЛЬНІ ВИСНОВКИ

Результати роботи

У межах дипломної роботи було реалізовано програмний продукт для проведення вирівнювання гомологічних білкових послідовностей із використанням класичних методів динамічного програмування. Створений інструмент HomologySearch який поєднує імпорт даних у форматі FASTA, ручне введення послідовностей, інтерактивну нормалізацію та повноцінну покрокову візуалізацію процесів побудови та трасування DP-матриць.

На основі реалізованих підходів забезпечено підтримку чотирьох алгоритмів вирівнювання що дозволяє користувачу обирати найбільш придатний до задачі механізм. Завдяки модульній структурі утилітного класу та окремим класам для графічних інтерфейсів було досягнуто високої гнучкості в роботі системи. Візуальне представлення DP-матриць у таблицях та графічне трасування оптимальних шляхів вирівнювання реалізовано на базі Java Swing, без залучення сторонніх бібліотек.

Серед реалізованих рішень особливу увагу приділено зручності взаємодії з користувачем: введення послідовностей може здійснюватись як вручну, так і через імпорт з FASTA-файлів, а результати автоматично відображаються у вигляді вирівняних матриць і текстових звітів. Також реалізовано можливість збереження історії сесій та звітів у простий файл-журнал.

Висновки

Робота узагальнює використання методів динамічного програмування для вирішення задачі пошуку гомологій білкових послідовностей із застосуванням візуального інструменту, що спрощує інтерпретацію обчислень. Отримане рішення дозволяє інтерактивно демонструвати принципи вирівнювання, що є цінним для освітніх цілей, а також як базовий інструмент у біоінформатичних лабораторіях. Розроблене ПЗ має практичне значення як самостійний засіб аналізу подібності білків та потенційно може бути адаптоване для інтеграції у більші системи аналізу послідовностей. Отримані результати можуть слугувати основою

для подальших розробок у сфері біоінформатики та комп’ютерної підтримки біологічних досліджень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Підручники та монографії

1. Durbin, R., Eddy, S. R., Krogh, A., Mitchison, G. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.
2. Lesk, A. M. (2019). *Introduction to Bioinformatics* (5th ed.). Oxford University Press.
3. Mount, D. W. (2004). *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press.
4. Pevsner, J. (2015). *Bioinformatics and Functional Genomics*. Wiley-Blackwell.
5. Xiong, J. (2006). *Essential Bioinformatics*. Cambridge University Press.
6. Горобець С. В., Горобець О. Ю., Дем'яненко І. В. (2020). *Біоінформатика. Практикум*. КПІ ім. Ігоря Сікорського.
7. Горобець С. В., Горобець О. Ю., Хоменко Т. А. (2010). *Основи біоінформатики*. НТУУ «КПІ».
8. Горобець С. В., Горобець О. Ю., Хоменко Т. А. (2019). *Основи молекулярної біології та біоінформатики*. КПІ ім. Ігоря Сікорського.
9. Кеца О. В. *Основи біоінформатики*.
10. Кучменко О. М. (2022). *Біоінформатика: аналіз генетичних послідовностей*. Видавничий центр ЛНУ ім. Івана Франка.
11. Кучменко О. М. *Біоінформатика*.
12. Кисляк СВ, Голуб НБ, Дуган ОМ, Аверьянова ОА. Моделювання молекулярної взаємодії. КПІ ім. Ігоря Сікорського; 2023.

2. Наукові статті

13. Altschul, S. F., Gish, W., Miller, W., Myers, E. W., & Lipman, D. J. (1990). "Basic Local Alignment Search Tool (BLAST)." *Journal of Molecular Biology*, 215(3), 403–410.
14. Edgar, R. C. (2004). "MUSCLE: multiple sequence alignment with high accuracy and high throughput." *Nucleic Acids Research*, 32(5), 1792–1797.

15. Kanehisa, M., Goto, S. (2000). "KEGG: Kyoto Encyclopedia of Genes and Genomes." *Nucleic Acids Research*, 28(1), 27–30.
16. Li, H., Durbin, R. (2009). "Fast and accurate short read alignment with Burrows–Wheeler transform." *Bioinformatics*, 25(14), 1754–1760.
17. Needleman, S. B., Wunsch, C. D. (1970). "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of Molecular Biology*, 48(3), 443–453.
18. Smith, T. F., Waterman, M. S. (1981). "Identification of common molecular subsequences." *Journal of Molecular Biology*, 147(1), 195–197.

3. Онлайн-ресурси

19. EMBL-EBI.
URL: https://www.ebi.ac.uk (дата звернення: 10.04.2025).
20. KEGG: kyoto encyclopedia of genes and genomes.
URL: https://pubmed.ncbi.nlm.nih.gov/10592174/ (дата звернення: 07.04.2025).
21. National Center for Biotechnology Information.
URL: https://www.ncbi.nlm.nih.gov (дата звернення: 15.04.2025).
22. Павлов В.А., Городецька О.В., Корнієнко Г.А., Аверьянова О.А. (2023). Дипломна робота бакалавра: організація, вимоги до структури, зміст та оформлення URL: https://ela.kpi.ua/server/api/core/bitstreams/9d8ead81-ed54-41b3-bf2b-5ffa35749697/content(дата звернення: 18.05.2025)