Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____Олена ОЛЬХОВСЬКА
«_____»_____202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

«РОЗРОБКА НАВЧАЛЬНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ТЕМИ «МЕТОД ГІЛОК ТА МЕЖ ДЛЯ ЗАДАЧ ОПТИМІЗАЦІЇ НА ПЕРЕСТАВЛЕННЯХ»

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня бакалавр

Виконавець роботи Малахов Нікіта Олексійович
_____«____» _____202_ р.
(підпис)

Науковий керівник доцент, к.ф.-м.н. Чілікіна Т. В.
_____«____»_____202_ р.
(підпис)

Рецензент

ПОЛТАВА 2026

РЕФЕРАТ

Записка: 64 с., 11 рис., 3 таблиці, 2 додатки, 14 джерел.

МЕТОД ГІЛОК ТА МЕЖ, ПЕРЕСТАВЛЕННЯ, КОМБІНАТОРНА ОПТИМІЗАЦІЯ, ВІЗУАЛІЗАЦІЯ, ІНТЕРАКТИВНИЙ ТРЕНАЖЕР, NEXT.JS, TYPESCRIPT, REACT, KATEX

Об'єктом розробки є процес вивчення та застосування методу гілок та меж до задач оптимізації на переставленнях у курсі «Елементи комбінаторної оптимізації».

Предметом розробки є програмна реалізація навчального вебзастосунку, який дозволяє формулювати задачі комбінаторної оптимізації, точно розв'язувати їх методом гілок та меж, покроково візуалізувати дерево галуження та відпрацьовувати алгоритм у тренажері з власноручною побудовою дерева.

Метою роботи є створення навчального програмного засобу, що допомагає студентам зрозуміти ідею напрямленого перебору, формули оцінок ν та ξ і правила відсікання для задач на переставленнях.

Результатом роботи стало розроблення навчального вебзастосунку «Permutex» на базі Next.js 16 та TypeScript. Реалізовано ключові модулі:

- обчислювальне ядро методу гілок та меж для безумовної лінійної задачі, лінійної задачі з обмеженнями, комбінаторної транспортної задачі та задачі про найкоротший маршрут у графі;
- модуль обчислення оцінок ν і ξ з підтримкою режимів мінімізації та максимізації;
- модуль повного перебору для порівняння ефективності методу гілок та меж із наївним підходом;
- інтерактивна візуалізація дерева галуження на бібліотеці React Flow з покроковою навігацією та автозапуском;
- редактори вхідних даних з валідацією, генератор випадкових задач та готові приклади задач;
- чотири інтерактивні тренажери - для безумовної задачі, для задачі з обмеженнями, для комбінаторної транспортної задачі та для задачі про

маршрут - у яких студент сам обирає бруньку, фіксує значення з G, обчислює ν та ξ, ухвалює рішення про відсікання, а програма перевіряє кожен крок з підрахунком правильних і помилкових відповідей;

- режим самоперевірки (Quiz) у демонстраційному розв'язувачі, у якому студент вводить очікувані ν та ξ, а програма ставить ✓ або ✗;
- довідковий розділ із формулюваннями теорем, означеннями та прикладами на основі KaTeX.

Особливості: повна локалізація українською мовою, темна та світла теми оформлення, адаптивний інтерфейс на основі shadcn/ui, обчислення в браузері без сервера, покриття обчислювального ядра і логіки тренажерів модульними тестами на Vitest.

Проведено модульне тестування 57 тестових випадків (33 - для алгоритмів обчислювального ядра, 24 - для двигуна та сховища станів інтерактивних тренажерів), ручне функціональне тестування з відтворенням очікуваних еталонних оптимумів, експериментальне тестування ефективності методу гілок та меж на розмірностях k=3..9.

«Permutex» може бути використаний як інтерактивний навчальний інструмент під час вивчення курсу «Елементи комбінаторної оптимізації» для самостійного освоєння теми «Метод гілок та меж для задач оптимізації на переставленнях», для перевірки розв'язків домашніх завдань і для підготовки до контрольних робіт.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
МГМ	Метод гілок та меж - алгоритм напрямленого перебору
КТЗП	Комбінаторна транспортна задача на переставленнях
ПЗ	Програмне забезпечення
UI	User Interface - інтерфейс користувача
SPA	Single Page Application - односторінкова вебпрограма
SSR	Server-Side Rendering - серверне рендерення
DOM	Document Object Model - об'єктна модель документа
DFS	Depth-First Search - обхід у глибину
TS	TypeScript - мова програмування зі статичною типізацією
JS	JavaScript - мова сценаріїв вебсторінок
HTML	HyperText Markup Language - мова розмітки гіпертексту
CSS	Cascading Style Sheets - каскадні таблиці стилів
KaTeX	Бібліотека для відображення математичних формул у браузері
ПУЕТ	Полтавський університет економіки і торгівлі

ВСТУП

Сучасні методи нерозривно пов'язані із задачами комбінаторної оптимізації. Серед них особливу роль відіграють задачі на переставленнях - лінійні задачі з обмеженнями, комбінаторні транспортні задачі, задачі планування виробництва, складання розкладу та маршрутизації. Ці задачі належать до класу NP-складних, тому ефективні методи їх точного розв'язання залишаються актуальною проблемою як теоретичних досліджень, так і прикладних застосувань.

Одним із класичних методів точного розв'язання задач комбінаторної оптимізації є метод гілок та меж (МГМ), запропонований А. Лендом і А. Дойґом у 1960 році. Метод базується на ідеї напрямленого перебору: множина допустимих розв'язків ділиться на підмножини (бруньки), для кожної з них обчислюється оцінка цільової функції, а ті бруньки, для яких оцінка завідомо гірша за вже знайдений рекорд, відсікаються від подальшого розгляду. У результаті фактично перевіряється лише невелика частина повного дерева варіантів, що в декілька порядків зменшує обчислювальні витрати.

Окремий клас задач комбінаторної оптимізації - задачі на переставленнях. У них допустимою множиною є множина перестановок мультимножини G, а цільова функція є лінійною комбінацією координат. Деталі формальної постановки таких задач, формули оцінок ν і ξ, а також правила A і B для комбінаторної транспортної задачі викладено «Елементи комбінаторної оптимізації», які стали теоретичним підґрунтям цієї роботи.

Незважаючи на те, що теоретичний апарат методу гілок та меж добре опрацьований у літературі, його опанування викликає значні труднощі у студентів. Це пояснюється великою кількістю формальних позначень, рекурсивним характером алгоритму та необхідністю одночасно тримати у фокусі дерево пошуку, поточну бруньку, оцінку, рекорд і відсічені вершини. У навчальній літературі задачі переважно ілюструються статичними рисунками з готовим деревом, без можливості переглянути алгоритм у динаміці. Сучасні навчальні інструменти для МГМ зосереджені на цілочисловому програмуванні (солвери на кшталт CBC, Gurobi), а

спеціалізованих інтерактивних застосунків для задач на переставленнях практично немає. Окремою проблемою є відсутність тренажерів, у яких студент сам обирав би наступну бруньку, обчислював оцінки і ухвалював рішення про відсікання, отримуючи негайний зворотний зв'язок щодо коректності кожного кроку.

Актуальність роботи полягає у тому, що інтерактивне навчальне програмне забезпечення з покроковою візуалізацією методу гілок та меж і вбудованим тренажером, у якому студент власноруч будує дерево галуження, суттєво підвищує якість засвоєння матеріалу, скорочує час, потрібний для розуміння алгоритму, і дає викладачу зручний інструмент для демонстрації прикладів під час лекцій.

Метою роботи є розробка навчального вебзастосунку, що дозволяє формулювати задачі оптимізації на переставленнях, розв'язувати їх методом гілок та меж, покроково візуалізувати процес з обчисленням оцінок ν та ξ, відсіченнями і порівнянням із повним перебором, а також відпрацьовувати алгоритм у режимі тренажера з власноручною побудовою дерева пошуку.

Для досягнення поставленої мети сформульовано такі завдання:

- проаналізувати предметну область - задачі комбінаторної оптимізації на переставленнях та метод гілок та меж;
- оглянути існуючі програмні засоби для візуалізації методу гілок та меж та обґрунтувати доцільність розробки власного навчального продукту;
- обґрунтувати вибір технологій реалізації - мови програмування, фреймворку, бібліотек візуалізації;
- спроєктувати архітектуру вебзастосунку з відокремленням обчислювального ядра від користувацького інтерфейсу;
- реалізувати модулі обчислення оцінок ν, ξ та алгоритми галуження для чотирьох типів задач;
- реалізувати інтерактивну візуалізацію дерева галуження з покроковою навігацією та довідковим розділом;
- реалізувати інтерактивний тренажер для чотирьох типів задач, у якому студент власноруч будує дерево і отримує зворотний зв'язок на кожному кроці;

- провести модульне і функціональне тестування програмного продукту;
- підготувати інструкцію користувача.

Об'єктом дослідження є процес вивчення методу гілок та меж та задач оптимізації на переставленнях у рамках навчального курсу «Елементи комбінаторної оптимізації».

Предметом дослідження є програмна реалізація алгоритмів методу гілок та меж для задач на переставленнях, засобів їх покрокової візуалізації та інтерактивного тренажера у вебсередовищі.

Методи дослідження. У роботі застосовано методи аналізу літератури, методи комбінаторної оптимізації, методи об'єктно-орієнтованого і функціонального програмування, методи модульного тестування програмного забезпечення.

Практичне значення. Розроблений навчальний застосунок «Permutex» може бути використаний студентами спеціальності «Комп'ютерні науки» та споріднених спеціальностей під час вивчення курсу «Елементи комбінаторної оптимізації» - для самостійного опрацювання теми, перевірки розв'язків домашніх завдань та підготовки до контрольних робіт. Викладач може використовувати програму для інтерактивної демонстрації алгоритму на лекційних та практичних заняттях, а тренажер - як засіб самостійного відпрацювання студентами правил обчислення ν, ξ та відсікання.

Кваліфікаційна робота складається зі вступу, постановки задачі, трьох розділів, висновків, списку інформаційних джерел та двох додатків: у першому наведено лістинг ключових модулів програмного продукту, у другому — скріни інтерфейсу.

ПОСТАНОВКА ЗАДАЧІ

Основним завданням кваліфікаційної роботи є розробка навчального вебзастосунку «Permutex», який реалізує метод гілок та меж для задач оптимізації на переставленнях, забезпечує покрокову візуалізацію роботи алгоритму та надає інтерактивний тренажер для самостійної побудови дерева галуження студентом.

Програмний продукт має задовольняти такі функціональні вимоги:

- забезпечувати введення вхідних даних чотирьох типів задач: безумовної лінійної задачі на переставленнях, лінійної задачі з обмеженнями, комбінаторної транспортної задачі та задачі про найкоротший маршрут у зваженому графі;
- знаходити точний оптимум методом гілок та меж з обчисленням оцінок ν і ξ за теоремами 2 і 3 з лекції 7;
- покроково відображати дерево галуження з можливістю переходу вперед, назад, на початок та в кінець, а також у режимі автозапуску;
- пояснювати кожен крок алгоритму українською мовою з відображенням відповідних формул;
- порівнювати кількість листків методу гілок та меж із кількістю перестановок повного перебору;
- підтримувати готові приклади з лекцій 6 і 7 та режим самоперевірки під час перегляду розв'язку;
- надавати окремий режим тренажера для кожного з чотирьох типів задач, у якому студент сам обирає бруньку для розгалуження, фіксує значення з мультимножини G, вводить очікувані ν та ξ і ухвалює рішення про відсікання, а програма перевіряє кожен крок;
- у тренажері вести лічильник правильних і помилкових відповідей, надавати підказки за запитом і вести лог пройдених кроків;
- підтримувати готові приклади у режимі тренажера та можливість переходу від тренажера до повного автоматичного розв'язку для звірки.

Нефункціональні вимоги: підтримка сучасних браузерів, локалізація українською мовою, темна та світла теми оформлення, адаптивний дизайн, час реакції інтерфейсу на дію користувача - не більше 100 мс для типових навчальних задач ($k \leq 9$).

Для досягнення поставленої задачі необхідно виконати такі підзадачі:

1. проаналізувати сучасні підходи до реалізації навчального програмного забезпечення з курсу «Елементи комбінаторної оптимізації» та визначити характерні недоліки існуючих рішень;
2. провести огляд та обґрунтувати вибір технологічного стеку, оптимального для реалізації інтерактивного вебдодатку з математичною візуалізацією;
3. опрацювати теоретичні основи методу гілок та меж - схему алгоритму, формули оцінок ν і ξ, властивості оцінок, правила відсікання, правила A і B для комбінаторної транспортної задачі;
4. спроєктувати модульну архітектуру застосунку з чітким розділенням обчислювального ядра, користувацького інтерфейсу та логіки тренажерів;
5. реалізувати обчислювальне ядро для чотирьох типів задач та модульно протестувати його;
6. реалізувати інтерактивний користувацький інтерфейс з підтримкою покрокової навігації по дереву галуження;
7. реалізувати інтерактивний тренажер для чотирьох типів задач з машиною станів, перевіркою кроків і підрахунком помилок;
8. провести функціональне тестування на прикладах з лекцій 6 і 7 з відтворенням еталонних результатів;
9. підготувати інструкцію користувача та оформити пояснювальну записку.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Аналіз предметної області - задачі комбінаторної оптимізації на переставленнях

Задачі комбінаторної оптимізації - це задачі пошуку оптимального елемента у скінченній, але потужній множині допустимих розв'язків. Вони виникають у плануванні виробництва, у логістиці, при складанні розкладів, при маршрутизації транспортних засобів та у багатьох інших економічних застосуваннях. Особливістю цих задач є те, що при зростанні розмірності їх обчислювальна складність зростає експоненціально, що робить наївний повний перебір непридатним для практичних розмірів.

Серед задач комбінаторної оптимізації окрему групу складають задачі на переставленнях. У них допустимою множиною є множина E(G) усіх перестановок мультимножини G = {g_1 , g_2 , …, g_k} [1]. Якщо в G немає однакових елементів, то |E(G)| = k!, а у випадку з повторами - менше, але так само факторіальна за k. Цільова функція F(x) = Σ c□·x□ є лінійною за компонентами вектора x.

До класичних задач на переставленнях належать чотири основні різновиди, які саме і реалізовано у програмному продукті «Permutex». Перший - безумовна лінійна задача (приклад 2 з лекції 6 [1]): мінімізувати F = $c_1 \cdot x_1$ + $c_2 \cdot x_2$ + … + c_k·x_k на множині E(G), без додаткових обмежень. Другий - лінійна задача з обмеженнями (лекція 7, ч. 1 [1]): додано рівності та нерівності виду Σ a_i□·x□ = b_i або Σ a_i□·x□ ≤ b_i. Третій - комбінаторна транспортна задача (КТЗП) на переставленнях (лекція 7, ч. 2 [1]): mn-мірна постановка з умовами балансу пропозиції і попиту. Четвертий - задача про найкоротший маршрут у зваженому графі (приклад 1 з лекції 6 [1]) - увідний приклад, який ілюструє ідею напрямленого перебору, не вимагаючи поняття перестановки.

Основний практичний виклик у задачах на переставленнях полягає у експоненціальній складності. Так, для k = 9 кількість перестановок становить 9! = 362 880, а для k = 12 - 12! = 479 001 600. Метод гілок та меж дозволяє суттєво

скоротити обсяг перебору: завдяки відсіченням для прикладу з лекції 7 ч. 2 (КТЗП 3×3, |G|=9) розв'язок знаходиться приблизно за 50 розглянутих вершин дерева замість 9! = 362 880, тобто майже у 7000 разів швидше [1].

Метод гілок та меж побудований на трьох базових операціях. Першою є галуження - поділ множини допустимих розв'язків на непересічні підмножини. Другою - обчислення оцінки для кожної підмножини, тобто числа, яке гарантовано не перевищує (для задачі мінімізації) значення цільової функції на цій підмножині. Третьою - відсікання тих підмножин, оцінка яких є гіршою за поточний рекорд. Якщо оцінки коректні, метод гарантовано знаходить глобальний оптимум.

На рисунку 1.1 наведено схему предметної області з показом місця методу гілок та меж серед інших методів комбінаторної оптимізації (див. рис. 1.1).

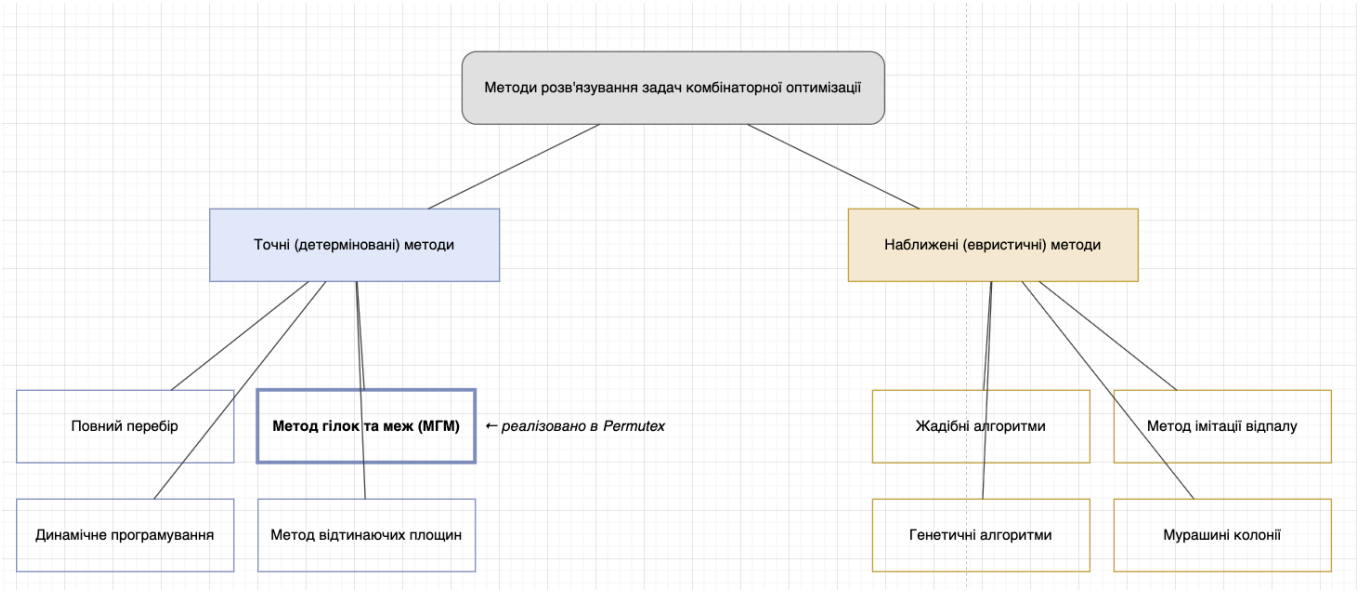

Рисунок 1.1 - Класифікація методів розв'язування задач комбінаторної оптимізації

З наведеної класифікації видно, що метод гілок та меж займає центральне місце серед точних методів. На відміну від наближених евристик (генетичні алгоритми, мурашині колонії, метод імітації відпалу), він гарантує знаходження глобального оптимуму, а на відміну від повного перебору - робить це з суттєво меншими обчислювальними витратами завдяки відсіченням.

Особлива увага у курсі «Елементи комбінаторної оптимізації» приділяється задачам на переставленнях саме як економічним моделям. Лінійні задачі на

переставленнях моделюють розподіл k різних обсягів виробництва між k робочими центрами, де кожен центр повинен отримати рівно один обсяг. КТЗП моделює транспортну задачу із дискретними обсягами перевезень. Задача про найкоротший маршрут має очевидне застосування у транспортній логістиці. Тому навчальне програмне забезпечення для цієї теми має не лише академічну, а й безпосередню практичну цінність [2].

1.2. Огляд існуючих програмних рішень для візуалізації методу гілок та меж

На сьогодні існує низка програмних продуктів, що тією чи іншою мірою працюють із задачами комбінаторної оптимізації або реалізують метод гілок та меж. Однак спеціалізованих рішень саме для задач на переставленнях, придатних для навчального використання у курсі «Елементи комбінаторної оптимізації», практично немає. Розглянемо ключові категорії існуючих продуктів.

Першу категорію складають професійні комерційні солвери - IBM ILOG CPLEX [3], Gurobi Optimizer [4], FICO Xpress. Вони реалізують метод гілок та меж для задач лінійного цілочислового програмування на найвищому рівні ефективності, мають інтерфейси для C++, Python, Java та інших мов. Однак ці продукти спрямовані на промислове використання, не мають навчальних візуалізацій, є комерційними (з обмеженою академічною ліцензією), і не підтримують задачі на переставленнях у власній постановці лекцій 6-7 курсу ЕКО ПУЕТ.

Другу категорію складають вільні солвери: COIN-OR CBC [5], GLPK, SCIP. Вони відкриті, інтегруються у скрипти, але також зосереджені на цілочисловому програмуванні, не мають візуалізації дерева пошуку і не призначені для навчальних цілей.

Третю категорію складають навчальні онлайн-симулятори. Серед них - IFORS Online (англомовний навчальний пакет для дослідження операцій), курси на Coursera і edX. Більшість з них демонструють МГМ виключно для задачі цілочислового програмування і задачі комівояжера; задачі на переставленнях там не

розглядаються. Інтерфейс цих ресурсів - англомовний, що створює додаткові труднощі для україномовних студентів.

Четверту категорію складають окремі навчальні скрипти на GitHub або у наукових публікаціях. Як правило, це консольні Python-програми, які виводять дерево у вигляді тексту або у форматі Graphviz. Вони не мають інтерактивної покрокової навігації, не пояснюють кожен крок українською мовою і не підтримують усіх чотирьох типів задач на переставленнях, які вивчаються у курсі ЕКО.

Порівняльна характеристика розглянутих категорій програмних продуктів за ключовими критеріями навчального процесу наведена у таблиці 1.1 (див. табл. 1.1).

Таблиця 1.1 - Порівняння існуючих програмних продуктів і Permutex

Критерій	**CPLEX/Gurobi**	**CBC/GLPK**	**Онлайн-курси**	**GitHub-скрипти**	**Permutex**
Українська мова інтерфейсу	-	-	-	-	+
Задачі на переставленнях у формулюванні	-	-	-	-	+
Покрокова навігація по дереву галуження	-	-	+	-	+
Пояснення кожного кроку	-	-	+	-	+
Відображення формул ν, ξ у математичній нотації	-	-	-	-	+
Готові приклади з лекцій курсу ЕКО ПУЕТ	-	-	-	-	+
Режим самоперевірки (Quiz)	-	-	-	-	+
Безкоштовний доступ через браузер	-	+	+	+	+
Працює без сервера (увесь розрахунок у браузері)	-	-	+	-	+

Аналіз порівняльної таблиці підтверджує, що жоден із існуючих продуктів не задовольняє повного набору вимог до навчального ПЗ із курсу «Елементи комбінаторної оптимізації» спеціальності 122 «Комп'ютерні науки» в ПУЕТ. Зокрема, відсутні україномовні рішення з підтримкою специфічних формулювань лекцій та з покроковою візуалізацією методу гілок та меж саме для задач на переставленнях. Це є додатковим обґрунтуванням актуальності розробки продукту «Permutex».

Загальний вигляд інтерфейсу одного з типових професійних солверів (CPLEX) представлено на рисунку 1.2 для порівняння з навчальним підходом, який застосовано у Permutex (див. рис. 1.2).

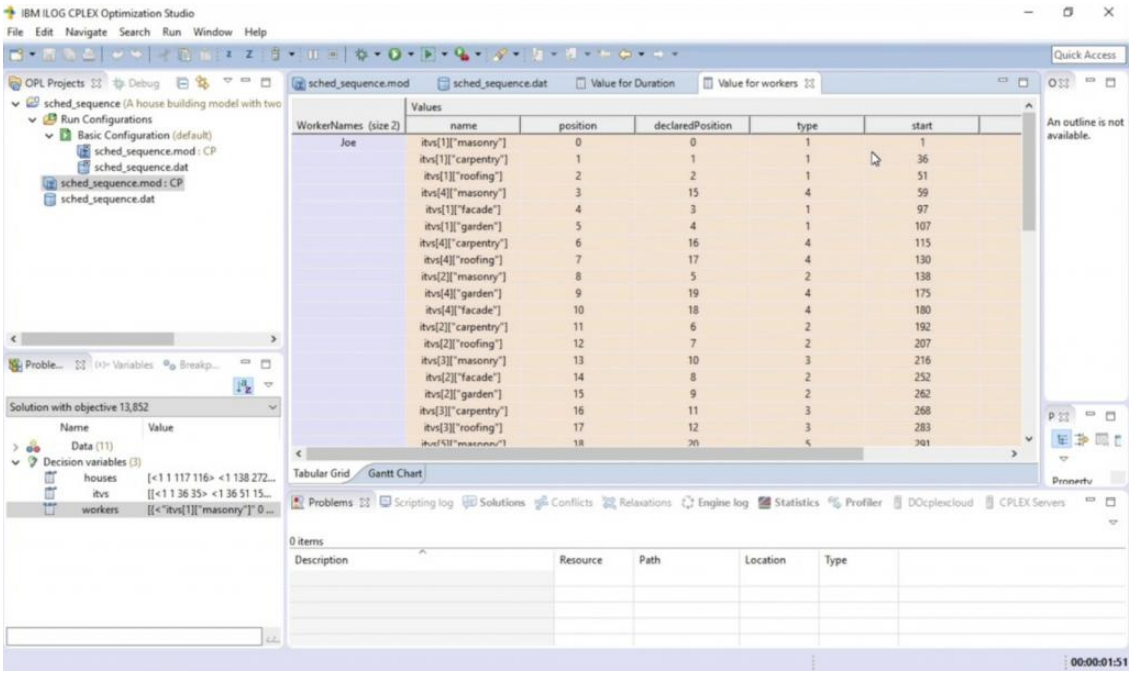

Рисунок 1.2 - Типовий інтерфейс комерційного солвера лінійного програмування (IBM ILOG CPLEX)

Видно, що інтерфейс комерційного інструменту орієнтований на досвідченого користувача, оперує термінами «MIP», «cuts», «node log» і не пояснює математичну суть кожного кроку. Це робить його непридатним для першого знайомства з методом і свідчить про потребу у спеціалізованому навчальному застосунку.

1.3. Огляд та обґрунтування вибору технологій розробки

Для реалізації навчального вебзастосунку було проведено порівняльний аналіз сучасних технологій розробки. Розглянуто кілька груп: мова програмування, фреймворк фронтенду, бібліотека компонентів інтерфейсу, бібліотека візуалізації графів, бібліотека рендеру математичних формул, інструменти модульного тестування.

Як основну мову розробки обрано TypeScript - надмножину JavaScript із статичною типізацією [6]. Перевага TypeScript полягає у можливості декларувати точні типи задач (Problem, TreeNode, StepEvent), завдяки чому помилки виявляються на етапі компіляції, а підказки IDE значно прискорюють розробку. У альтернативі - чистого JavaScript - типи перевіряються лише під час виконання, що для алгоритмічного коду з оцінками і деревами є небезпечним.

Як фреймворк фронтенду обрано Next.js 16 разом із React 19 [7]. Next.js забезпечує модульний роутинг через App Router, оптимізований Bundling із застосуванням Turbopack, гарячу заміну модулів під час розробки і просте розгортання. Альтернативи - чистий React (Vite), SvelteKit, Vue/Nuxt - мають свої переваги, але Next.js обрано через найкращу інтеграцію з екосистемою Vercel, кращу підтримку TypeScript, велику спільноту та документацію українською (значна кількість матеріалів).

Для UI-компонентів використано бібліотеку shadcn/ui поверх Tailwind CSS 4 [8]. shadcn/ui - це не звичайна npm-бібліотека, а набір копіюваних компонентів, які додаються в проєкт як вихідний код, що дозволяє кастомізувати їх без обмежень. Tailwind CSS забезпечує утилітарний підхід до стилізації - стилі задаються безпосередньо в JSX через короткі CSS-класи. Альтернативи - Material UI, Ant Design, Chakra UI - мають вищий рівень готових стилів, але обмежують гнучкість і збільшують розмір бандлу.

Для візуалізації дерева галуження використано React Flow (@xyflow/react) - спеціалізовану бібліотеку для побудови вузлово-реберних діаграм у React [9]. Вона підтримує панорамування, масштабування, кастомні вузли та автоматичне

позиціонування. Альтернативи - D3.js (низькорівнева, потребує власноруч писати взаємодію), GoJS (комерційна), Mermaid (статичні діаграми без інтерактивності) - поступаються React Flow за зручністю інтеграції з React.

Для рендерингу математичних формул обрано бібліотеку KaTeX з обгорткою react-katex [10]. KaTeX швидший за MathJax, працює без сервера, і добре виглядає на сторінці. Це критично для відображення формул ν, ξ, теорем і прикладів. Альтернатива MathJax підтримує більше синтаксису LaTeX, але вантажиться повільно і важчий за розміром.

Для управління станом використано бібліотеку Zustand - легку та продуктивну альтернативу Redux [11]. У Zustand немає reducers і dispatcher, він використовує hooks та зрозумілий API. Це підходить для нашого випадку, де стан складається з поточної задачі, результату розв'язання, поточного кроку та режиму автозапуску.

Для модульного тестування обчислювального ядра використано Vitest - сучасний тест-раннер, сумісний з Jest API, але побудований на Vite, що забезпечує найшвидший запуск тестів у TypeScript-проєктах [12].

Узагальнений технологічний стек проєкту наведено у таблиці 1.2 (див. табл. 1.2).

Таблиця 1.2 - Технологічний стек проєкту Permutex

Категорія	**Технологія**	**Версія**	**Призначення**
Мова	TypeScript	5.x	Статична типізація
Фреймворк	Next.js	16.2.4	App Router, SSR, бандлер
UI-бібліотека	React	19.2.4	Компонентний рендер
Стилі	Tailwind CSS	4.x	Утилітарні CSS-класи
Компоненти	shadcn/ui	4.4	Готові UI-примітиви
Граф/дерево	React Flow (@xyflow/react)	12.10	Вузлово-реберна візуалізація
Формули	KaTeX + react-katex	0.16	Рендер LaTeX-формул
Стан	Zustand	5.0	Глобальний стан додатку
Анімація	Framer Motion	12.38	Плавні переходи UI

Іконки	Lucide React	1.8	Векторні іконки
Тестування	Vitest	4.1	Модульні тести ядра

Усі обрані технології є відкритими, активно підтримуються спільнотою, мають детальну документацію і застосовуються у промислових продуктах. Це гарантує надійність розробки і легку підтримку у майбутньому.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Метод гілок та меж: загальна схема, оцінки, відсікання

Метод гілок та меж є одним із найважливіших інструментів точного розв'язування задач комбінаторної оптимізації. Розглянемо його загальну схему за матеріалами лекції «Елементи комбінаторної оптимізації» [1].

Нехай задано задачу мінімізації $F(q^*) = \min F(q)$ на скінченній множині Q, де q^* є оптимальним розв'язком. Безпосередній перебір усіх елементів Q може бути неможливим у задачах з $|Q| = k!$, тому застосовується ідея напрямленого перебору. Множина Q розбивається на непересічні підмножини Q_1 , Q_2 , …, Q_n такі, що $Q_1 \cup Q_2 \cup \ldots \cup$ Q_n = Q, причому $Q_i \cap Q\square = \emptyset$ при $i \neq j$ і всі Q_i є непустими [1]. Кожній з підмножин (бруньок) приписується число ν_i - оцінка, що задовольняє нерівність $\nu_i \leq F(x)$ для всіх $x \in Q_i$.

Ключовою ідеєю є те, що оцінка ν дозволяє відсікати «неперспективні» бруньки без необхідності перебирати їх повністю. Якщо у процесі роботи алгоритму знайдено деякий $x_0 \in Q$ з $F(x_0) = F_0$ (поточний рекорд) і для деякої бруньки $Q\square$ виконується $\nu\square \geq F_0$, то будь-який елемент $x \in Q\square$ задовольняє $F(x) \geq \nu\square \geq F_0$, отже, в $Q\square$ немає кращого розв'язку, і її можна відсікти.

Формально алгоритм методу гілок та меж за лекцією 6 [1] складається з 16 кроків. На рисунку 2.1 наведено блок-схему алгоритму у спрощеному вигляді (див. рис. 2.1).

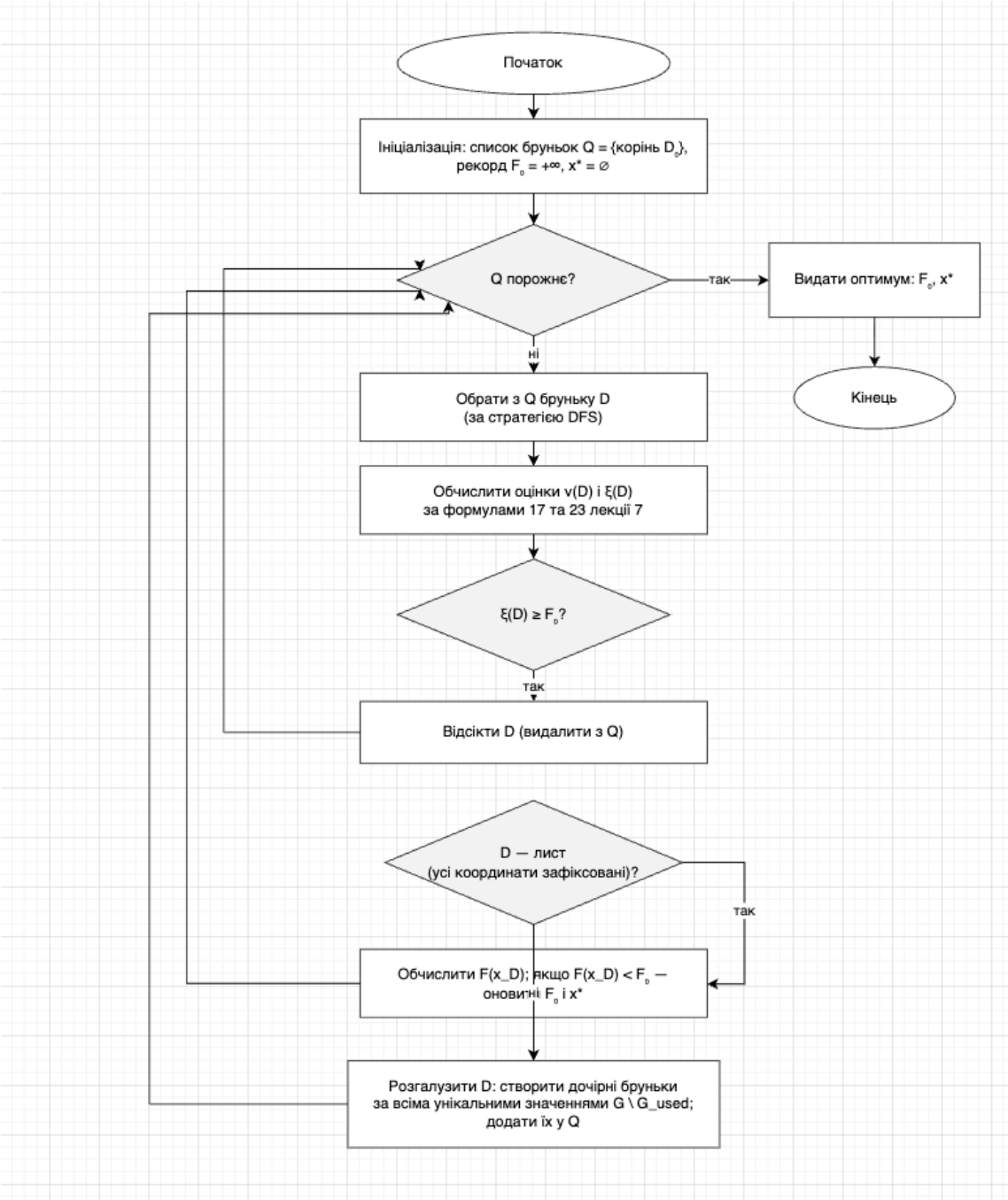

Рисунок 2.1 - Блок-схема методу гілок та меж

На блок-схемі видно три основні цикли роботи: розгалуження поточної бруньки, обчислення оцінок дочірніх бруньок, оновлення рекорду при досягненні листа і відсікання тих бруньок, оцінка яких перевищує рекорд. Алгоритм завершується, коли не залишається жодної брунькі, яку б можна було ще розгалузити або яка б не була відсічена.

Властивість оцінок ν, доведена у теоремі 4 лекції 7 [1], стверджує, що оцінки не зменшуються при русі по дереву вниз (від кореня до листа) або вправо (між

дочірніми вершинами одного рівня). Ця монотонність гарантує коректність відсікання: якщо для предка $\nu_i \geq F_0$, то для всіх його нащадків ν так само не менше F_0 , отже, вся піддерево можна не розглядати.

Класичним увідним прикладом методу гілок та меж є задача про найкоротший маршрут у зваженому графі. У лекції 6 ([1], приклад 1) розглянуто граф із 7 вершинами, у якому потрібно знайти найкоротший шлях з вершини 1 до вершини 7. Оцінкою для бруньки, що відповідає поточному префіксу шляху, є сумарна вага вже пройдених ребер: вона очевидно не перевищує довжини будь-якого продовження. Метод знаходить розв'язок F = 4, шлях (1, 2, 3, 6, 7) [1].

2.2. Лінійні задачі на переставленнях: формули оцінок ν та ξ

Перейдемо до основного класу задач - лінійних задач на переставленнях. Розглянемо умовну постановку з лекції 7, ч. 1 [1]. Задано лінійну цільову функцію C(x) = Σ□ c□·x□, j = 1..k, яку треба мінімізувати на множині перестановок мультимножини G = {g_1 , g_2 , …, g_k} за додаткових обмежень рівностей Σ□ a_i□·x□ = b_i та нерівностей Σ□ a_i□·x□ ≤ b_i.

Стратегія галуження для таких задач полягає у послідовному закріпленні координат. На рівні r маємо підмножину D, для якої вже зафіксовано x_{τ_1 } = g_{i_1 }, x_{τ_2 } = g_{i_2 }, …, x_{τ_r} = g_{i_r}, а інші координати утворюють перестановку решти елементів G \ {g_{i_1 }, …, g_{i_r}}.

Перша оцінка - оцінка ν - виводиться у теоремі 2 лекції 7 (формула 17) [1]:

ν = Σ□ c_{τ□}·g_{i□}

тобто ν дорівнює внеску у цільову функцію вже зафіксованих координат. Це є нижньою межею F на бруньці D за умови, що $c_i \geq 0$ і g□ ≥ 0 (або обидві групи не додатні), оскільки всі решта доданків F є невід'ємними.

Друга, точніша оцінка - оцінка ξ - виводиться у теоремі 3 лекції 7 (формула 23) [1]:

ξ = ν + c*

де c* - це мінімум скалярного добутку залишкових коефіцієнтів $\{\hat{c}_1, \hat{c}_2, \ldots\}$ на залишкові значення G $\{\acute{g}_1, \acute{g}_2, \ldots\}$. Цей мінімум, як показано у теоремі, досягається, коли впорядковані за незростанням коефіцієнти $\hat{c}_1 \geq \hat{c}_2 \geq \ldots \geq \hat{c}$□ множаться на впорядковані за неспаданням значення $\acute{g}_1 \leq \acute{g}_2 \leq \ldots \leq \acute{g}$□. Це класичний результат, відомий як «нерівність про перестановки» (rearrangement inequality) [13].

Інтуїтивно ξ є точнішою оцінкою, ніж ν, оскільки враховує не лише вже зафіксовану частину суми, а й мінімально можливий внесок незакріплених координат. Тому у програмному продукті використовується саме оцінка ξ для відсікання, що забезпечує більшу кількість відрізаних бруньок.

Розглянемо застосування цих формул на прикладі 2 з лекції 6 [1]: знайти мінімум $F(x) = 2x_1 + 5x_2 + 3x_3$ на множині перестановок мультимножини G = {1, 4, 5}. Маємо c = (2, 5, 3), G = (1, 4, 5).

Корінь дерева: жодна координата не зафіксована, тому ν = 0. Для обчислення c* сортуємо c за незростанням: 5 ≥ 3 ≥ 2; G за неспаданням: 1 ≤ 4 ≤ 5. Парування: 5·1, 3·4, 2·5 → c* = 5+12+10 = 27. Отже, ξ = 0 + 27 = 27.

На наступному рівні фіксуємо x_1. Розглянемо три бруньки: x_1 =1, x_1 =4, x_1 =5. Для бруньки x_1 =5 маємо ν = 2·5 = 10, залишкові c = (5, 3), G = (1, 4), парування 5·1, 3·4, c* = 5+12 = 17, ξ = 10+17 = 27. На третьому рівні (лист) при x_1 =5, x_2 =1, x_3 =4 отримаємо F = 2·5 + 5·1 + 3·4 = 27. Це і є оптимум: F_min = 27, x* = (5, 1, 4) [1].

На рисунку 2.2 показано фрагмент дерева пошуку для цього прикладу з обчисленими ν та ξ для всіх вершин (див. рис. 2.2).

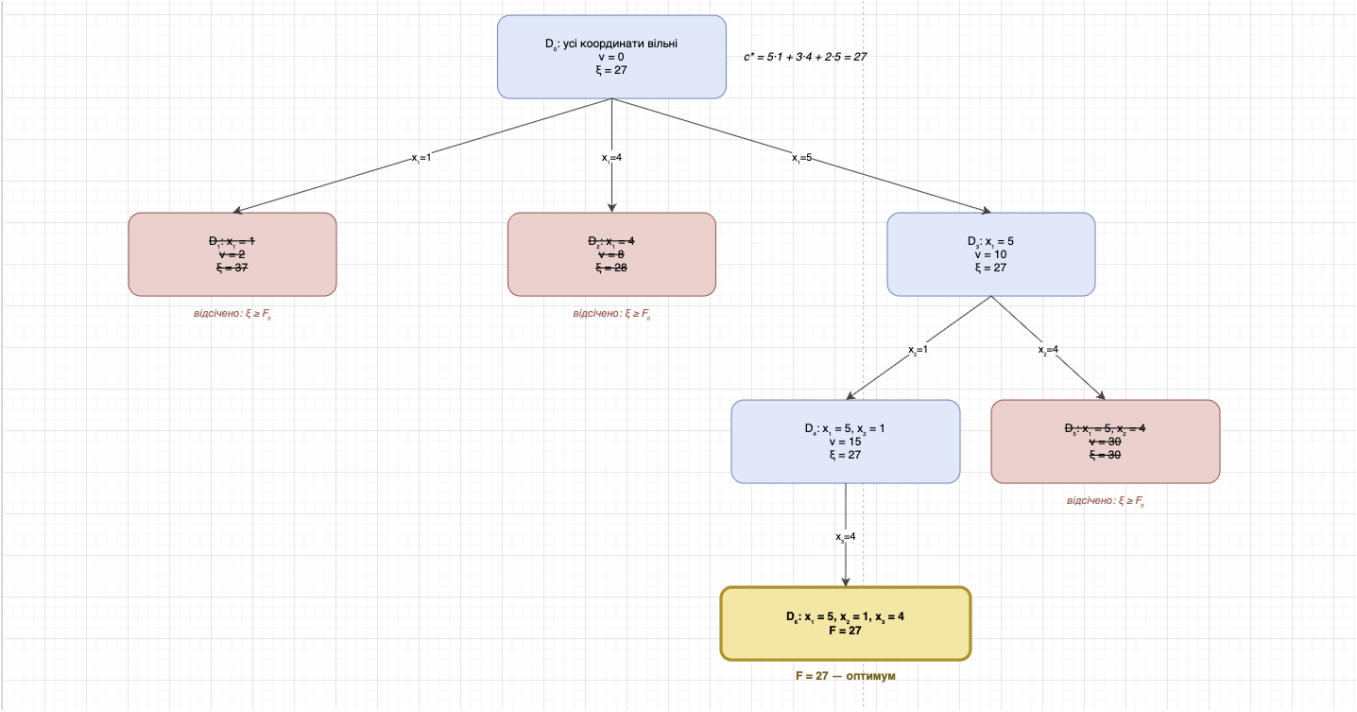

Рисунок 2.2 - Фрагмент дерева методу гілок та меж для прикладу $F = 2x_1 + 5x_2 + 3x_3$, $G = \{1, 4, 5\}$

На фрагменті дерева добре видно, що з трьох бруньок першого рівня дві ($x_1 = 1$ і $x_1 = 4$) могли бути відрізані, бо їхні ξ перевищують 27, а лише гілка $x_1 = 5$ розгалужується далі. Це наочна ілюстрація економії, яку дає метод гілок та меж порівняно з повним перебором 3! = 6 перестановок.

2.3. Метод гілок та меж для інших класів задач на переставленнях

Розглянута у підрозділах 2.1 і 2.2 загальна схема методу гілок та меж природно поширюється на інші класи задач на переставленнях, реалізованих у програмному продукті «Permutex»: на лінійну задачу з обмеженнями, на комбінаторну транспортну задачу та на задачу про найкоротший маршрут у зваженому графі. Концептуальна основа залишається тією самою - поділ множини допустимих розв'язків на бруньки, обчислення оцінки нижньої межі для кожної бруньки і відсікання за рекордом. Відмінності між класами задач полягають у способі галуження та у деталях обчислення оцінки.

Для лінійної задачі з обмеженнями галуження виконується аналогічно безумовній задачі - за послідовною фіксацією координат, з додатковою перевіркою допустимості обмежень. Для комбінаторної транспортної задачі галуження виконується за клітинками матриці у певному порядку, а оцінка нижньої межі обчислюється з урахуванням матричної структури задачі. Для задачі про найкоротший маршрут у графі галуження виконується за вихідними ребрами поточної вершини, а оцінкою бруньки служить сума ваг пройденого префіксу шляху. Конкретні алгоритми для всіх трьох класів задач описано у підрозділі 3.2.

3. ПРАКТИЧНА ЧАСТИНА

3.1. Загальна архітектура застосунку Permutex

Програмний продукт «Permutex» побудований за принципом чіткого розділення відповідальностей. Архітектура виділяє чотири рівні: рівень обчислювального ядра (математичні алгоритми), рівень логіки тренажерів (двигуни перевірки кроків), рівень управління станом (Zustand-store) та рівень інтерфейсу користувача (React-компоненти). Такий поділ забезпечує тестованість ядра і логіки тренажерів незалежно від UI, що особливо важливо для алгоритмічного коду.

Загальна архітектура та потоки даних показані на рисунку 3.1 (див. рис. 3.1).

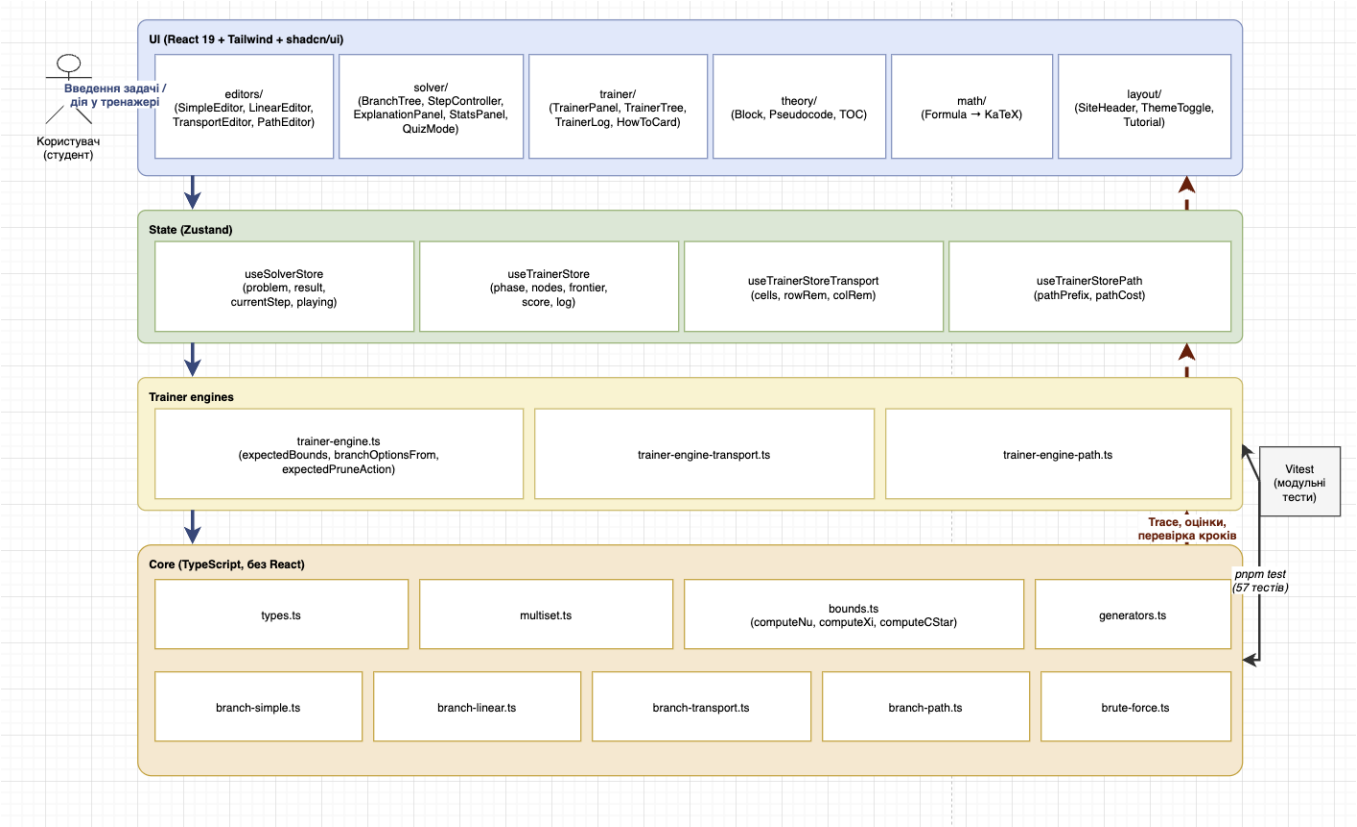

Рисунок 3.1 - Архітектура застосунку Permutex та потоки даних між модулями

У режимі автоматичного розв'язку потік даних розпочинається з компонента-редактора (наприклад, SimpleEditor або TransportEditor), у який користувач вводить параметри задачі. Редактор формує об'єкт типу Problem і передає його у Zustand-

store розв'язувача через виклик setProblem. Натискання кнопки «Розв'язати» викликає функцію solveSimple (або відповідну для іншого типу задачі), яка повертає об'єкт SolveResult, що містить дерево вершин і послідовність подій (events). Цей результат зберігається в store через setResult.

Компоненти візуалізації - BranchTree, ExplanationPanel, StatsPanel - підписуються на store через хук useSolverStore і отримують дані для відображення. Окремий контролер StepController забезпечує покрокову навігацію: при натисканні кнопок «Уперед» / «Назад» він викликає stepForward / stepBackward, що змінює currentStep, і всі підписані компоненти автоматично оновлюються.

У режимі тренажера потік даних інший. Користувач також формує задачу через редактор, але далі замість виклику автоматичного солвера активується машина станів тренажера у власному Zustand-store (useTrainerStore, useTrainerStoreTransport, useTrainerStorePath). Машина станів проводить студента послідовністю фаз: pick-node (обрати бруньку для розгалуження), pick-value (обрати значення з G або клітинку, або ребро), enter-nu (ввести оцінку ν), enter-xi (ввести оцінку ξ), decide-fate (вирішити, чи відсікти бруньку). Після кожної дії студента двигун trainer-engine звіряє введене з канонічно обчисленим і оновлює лічильники правильних і помилкових відповідей.

Структура каталогів проєкту відповідає поділу на ядро, логіку тренажерів, store, компоненти та сторінки. У каталозі src/core/ знаходяться файли алгоритмів і утиліт, повністю незалежних від React; модулі trainer-engine, trainer-engine-transport, trainer-engine-path реалізують перевірку кроків тренажерів. У src/store/ - глобальні стани Zustand для розв'язувача та для трьох тренажерів. У src/components/ - UI-компоненти, розбиті на підкаталоги editors, solver, math, ui, theory, trainer, layout. У src/app/ - сторінки маршрутизації Next.js App Router.

Кожен з чотирьох типів задач має окрему сторінку для автоматичного розв'язку: /simple для безумовної лінійної задачі, /linear - для лінійної з обмеженнями, /transport - для КТЗП, /path - для маршруту в графі. Дублюючий набір сторінок під префіксом /trainer/ містить інтерактивні тренажери - /trainer/simple, /trainer/linear, /trainer/transport, /trainer/path. Сторінка /theory містить теоретичну

довідку, /complexity - експериментальний розділ з порівнянням ефективності МГМ і повного перебору на різних розмірностях.

Усі сторінки автоматичного розв'язку використовують спільний компонент SolverLayout, який задає двоколонкову розмітку: ліворуч - редактор задачі, праворуч - візуалізація дерева, контролер кроків, панель пояснень і панель статистики. Сторінки тренажерів використовують варіанти TrainerPanel, TrainerPanelTransport та TrainerPanelPath залежно від типу задачі. Це забезпечує консистентний UX при максимальному врахуванні специфіки кожного типу задачі.

3.2. Реалізація обчислювального ядра (модулі core)

Обчислювальне ядро Permutex реалізоване як набір TypeScript-модулів у каталозі src/core. Модулі не залежать від React, Next.js чи DOM, що дозволяє запускати їх як у браузері, так і в Node.js під час модульного тестування.

Центральним модулем є src/core/types.ts, який описує всі типи задач (SimpleProblem, LinearProblem, TransportProblem, PathProblem), типи вершин дерева пошуку (TreeNode зі статусами pending, evaluated, branching, pruned, leaf, record, optimum), типи подій (StepEvent з варіантами init, branch, evaluate, leaf, prune, record, done) та результат розв'язання (SolveResult). Завдяки строгій типізації редактор та солвер «знають» один про одного у термінах типів, а помилки невідповідності виявляються на етапі компіляції.

Модуль src/core/multiset.ts реалізує операції над мультимножиною G: сортування за зростанням і за спаданням, парсинг рядкового вводу, видалення за індексами, отримання залишкових індексів і визначення індексів унікальних значень. Останнє є важливим для запобігання генерації дублюючих гілок дерева - якщо в G є однакові елементи, фіксація обох з них дає еквівалентні бруньки, тому розгалуження виконується лише за першим входженням кожного значення.

Модуль src/core/bounds.ts реалізує формули оцінок ν і ξ. Функція computeNu підсумовує добутки c□·g_i□ за зафіксованими координатами. Функція computeCStar обчислює c* за принципом нерівності про перестановки: сортує

залишкові коефіцієнти за незростанням, залишкові значення за неспаданням і парує їх. Допоміжна computeCStarDetailed повертає не лише числове значення, а й розклад добутків (наприклад, "5·1 + 3·4 + 2·5 = 27"), який потім відображається у поясненні поточного кроку. Функція formatCStarBreakdown готує цей розклад у людочитному вигляді.

Модуль src/core/branch-simple.ts реалізує метод гілок та меж для безумовної лінійної задачі на переставленнях. Алгоритм використовує стек для обходу в глибину: на кожному кроці береться вершина зі стека, перевіряється правило відсікання $\xi \geq F_0$, при необхідності розгалужується за всіма унікальними значеннями G або позначається як лист з обчисленням F. Усі операції фіксуються у списку events, який потім використовується для покрокової візуалізації.

Модуль src/core/branch-linear.ts розширює branch-simple.ts підтримкою обмежень. Додано функцію feasibilityCheck, яка для кожної бруньки перевіряє, чи може вона задовольнити рівності та нерівності з урахуванням залишку G. Якщо ні - брунька відсікається з причиною infeasible. Це дає приріст ефективності порівняно з простою перевіркою лише після досягнення листа.

Модуль src/core/branch-transport.ts реалізує алгоритм МГМ для комбінаторної транспортної задачі. Галуження виконується по клітинках матриці у певному порядку (за неспаданням вартостей c_i□), оцінка ξ обчислюється за теоремою 1 з урахуванням і зафіксованих, і залишкових клітинок.

Модуль src/core/branch-path.ts реалізує МГМ для задачі про найкоротший маршрут у графі. Це найпростіший з алгоритмів - оцінкою бруньки є сума ваг пройдених ребер, відсікання працює, коли поточна сума плюс мінімальне залишкове ребро вже перевищує рекорд.

Модуль src/core/brute-force.ts реалізує повний перебір - для порівняння з МГМ. Алгоритм генерує всі перестановки G лексикографічно і обчислює F для кожної. Робота обмежена випадком $|G|! \leq 10^6$, оскільки інакше браузер «зависне».

Окремий модуль src/core/generators.ts містить функції генерації випадкових задач для усіх чотирьох типів - це використовується кнопкою «Випадкова» в

редакторах. Параметри генератора задані так, щоб задачі мали реалістичні розмірності і нетривіальні відсічення.

Логіка тренажерів реалізована у модулях trainer-engine.ts, trainer-engine-transport.ts і trainer-engine-path.ts. У них містяться функції expectedBounds (повертає канонічні ν і ξ для поточної бруньки), branchOptionsFrom (перелік допустимих гілок, які студент може обрати), expectedPruneAction (правильна дія prune/keep на поточному стані), checkFeasibility для linear-варіанта. Двигун трейнера повністю незалежний від UI і покритий 15 модульними тестами у trainer-engine.test.ts.

Тестове покриття ядра реалізовано у файлах bounds.test.ts (9 тестів), branch-simple.test.ts (8 тестів), branch-linear.test.ts (5 тестів), branch-transport.test.ts (6 тестів), branch-path.test.ts (5 тестів). Загалом 33 тестових випадки покривають усі ключові сценарії: коректність обчислення ν і ξ, відтворення оптимумів з прикладів лекцій 6 і 7 (F = 27 для безумовної задачі, F = 4 для найкоротшого маршруту, $F_0 = 203$ і $F_0 = 240$ для двох прикладів КТЗП), коректність відсікань і узгодженість з повним перебором (див. рис. 3.2).

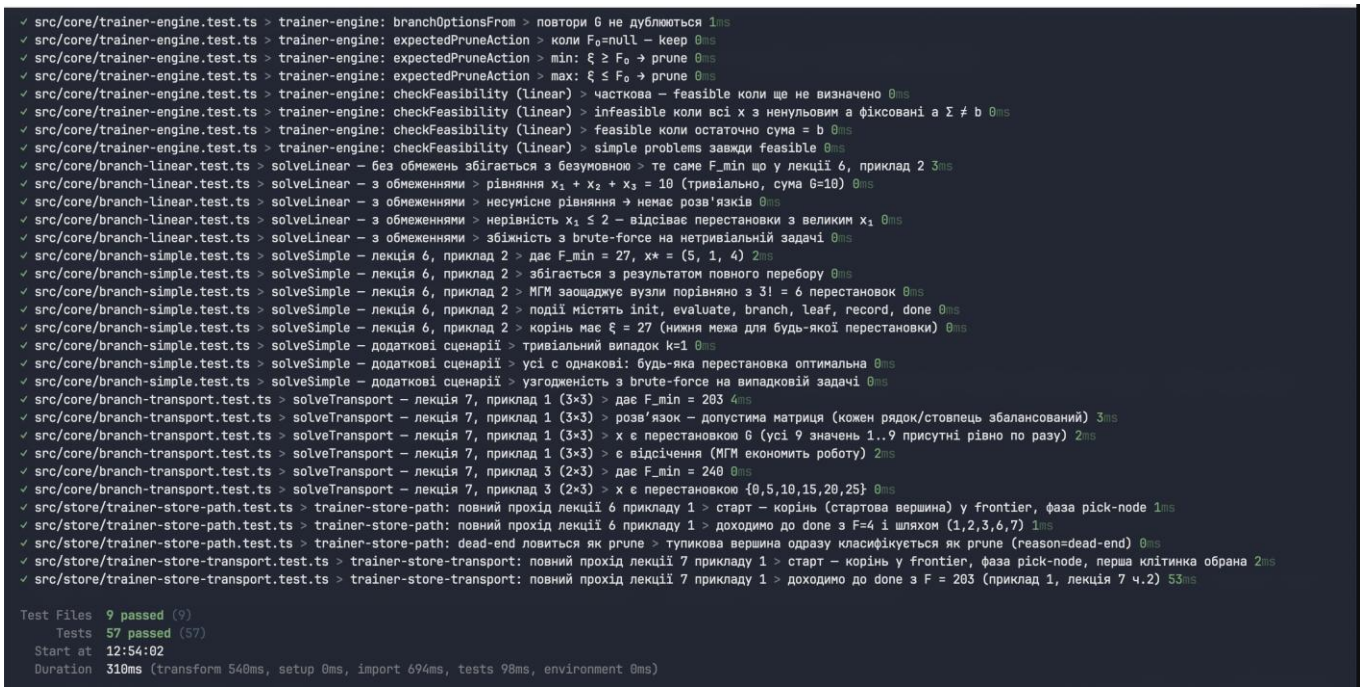

Рисунок 3.2 - Результати запуску модульних тестів обчислювального ядра у Vitest

Усі 33 тести проходять без помилок, що підтверджує коректність математичної реалізації методу гілок та меж в усіх його варіантах для задач на переставленнях.

3.3. Реалізація режиму розв'язувача та візуалізації дерева

Користувацький інтерфейс режиму автоматичного розв'язку реалізований як набір React-компонентів. Архітектура передбачає поділ компонентів на чотири групи: примітиви shadcn/ui (button, card, input, table, dialog, select, tabs, tooltip, slider, alert, dropdown-menu тощо), редактори задач, компоненти візуалізації (солвер) та інфраструктурні компоненти (header, footer, theme-toggle, tutorial).

Центральним компонентом є BranchTree (src/components/solver/branch-tree.tsx) - інтерактивне дерево галуження на бібліотеці React Flow. Кожна вершина дерева відображається як прямокутна картка зі спеціальним стилем залежно від статусу: pending - блідо-сіра, evaluated - блакитна з ν та ξ, branching - синя, pruned - закреслена червона, leaf - зелена, record - помаранчева з підкресленням, optimum - золотиста. Дерево автоматично перебудовується після кожного кроку, що дає ефект «зростання» дерева у міру роботи алгоритму.

Компонент StepController (src/components/solver/step-controller.tsx) забезпечує покрокову навігацію. Він містить кнопки «До початку», «Крок назад», «Грати/Пауза», «Крок уперед», «До кінця», слайдер позиції в trace та регулятор швидкості автозапуску. Підтримуються клавіатурні скорочення: ←/→ - крок назад/уперед, Space - пауза, Home/End - до меж, R - скидання.

ExplanationPanel (src/components/solver/explanation-panel.tsx) відображає текстове пояснення поточного кроку зі вставленими формулами через KaTeX. Наприклад, для події evaluate показується ν, c* з розкладом добутків, ξ та узагальнена формула $\xi = \nu + c^*$. Для подій branch, prune, record, leaf, done пояснення формується відповідно до семантики події українською мовою.

StatsPanel (src/components/solver/stats-panel.tsx) виводить порівняльну статистику: для МГМ - кількість вершин дерева, кількість листків, кількість

відсічених бруньок, час виконання; для повного перебору - кількість перестановок, час виконання; додатково - коефіцієнт економії листків відносно k!. Це дає кількісне підтвердження ефективності методу гілок та меж порівняно з наївним підходом.

На рисунку 3.3 показано загальний вигляд інтерфейсу сторінки безумовної задачі (/simple) (див. рис. 3.3).

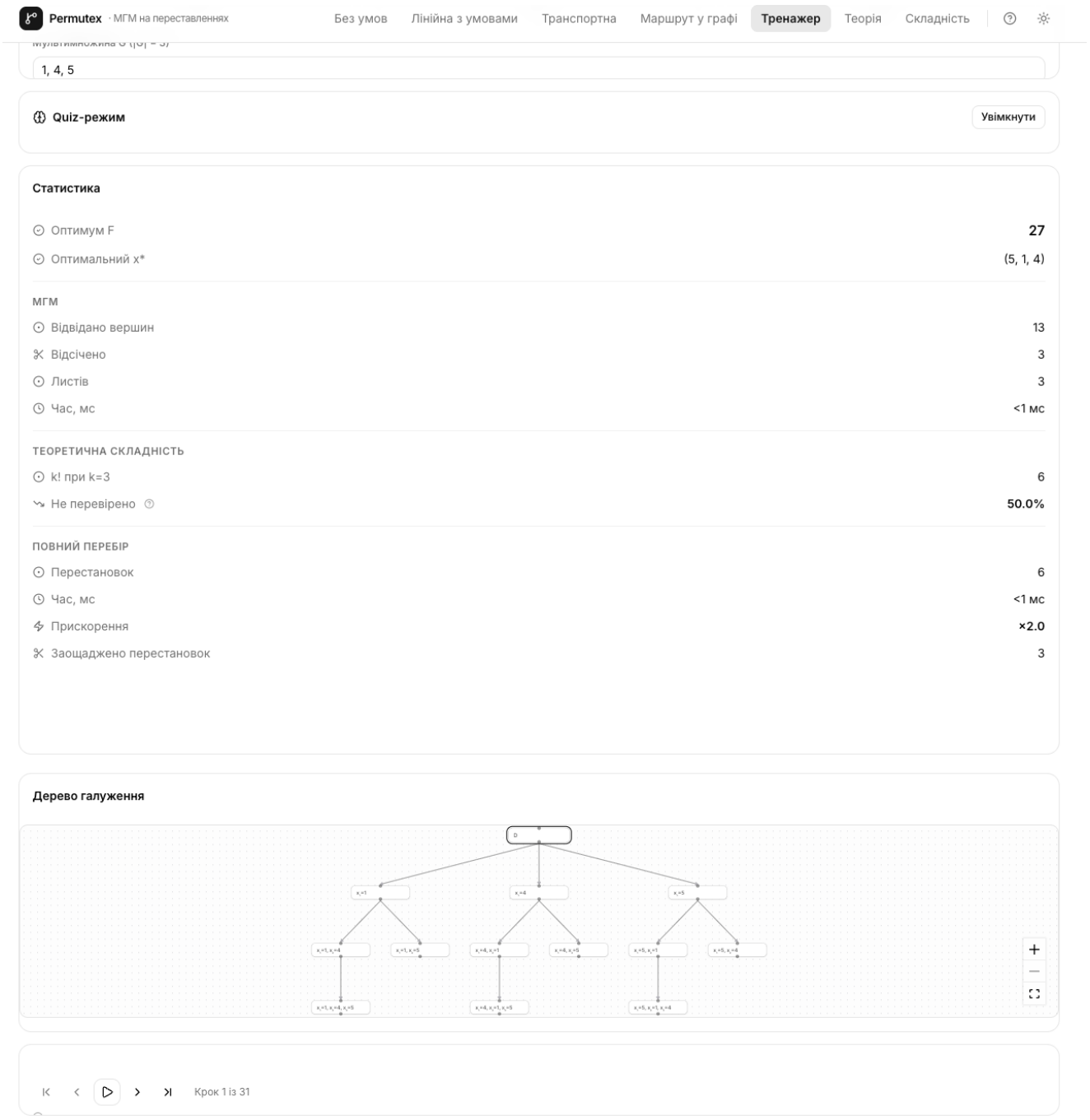

Рисунок 3.3 - Інтерфейс сторінки безумовної задачі на переставленнях

На скриншоті ліворуч видно редактор задачі - введення коефіцієнтів c, мультимножини G, прапорець min/max та кнопки «Розв'язати», «Випадкова», «Скинути». Праворуч - дерево галуження з усіма вершинами, контроллер кроків, панель пояснень і панель статистики. Усі тексти інтерфейсу - українською, формули - у математичній нотації через KaTeX.

Аналогічний інтерфейс реалізовано для решти типів задач на переставленнях - лінійної з обмеженнями, комбінаторної транспортної та задачі про маршрут у графі. Як приклад на рисунку 3.4 наведено сторінку лінійної задачі з обмеженнями: ліворуч редактор з коефіцієнтами c, мультимножиною G і рядками матриці обмежень A; праворуч - дерево галуження з оцінками ν, ξ та позначенням інфізибельних бруньок (див. рис. 3.4).

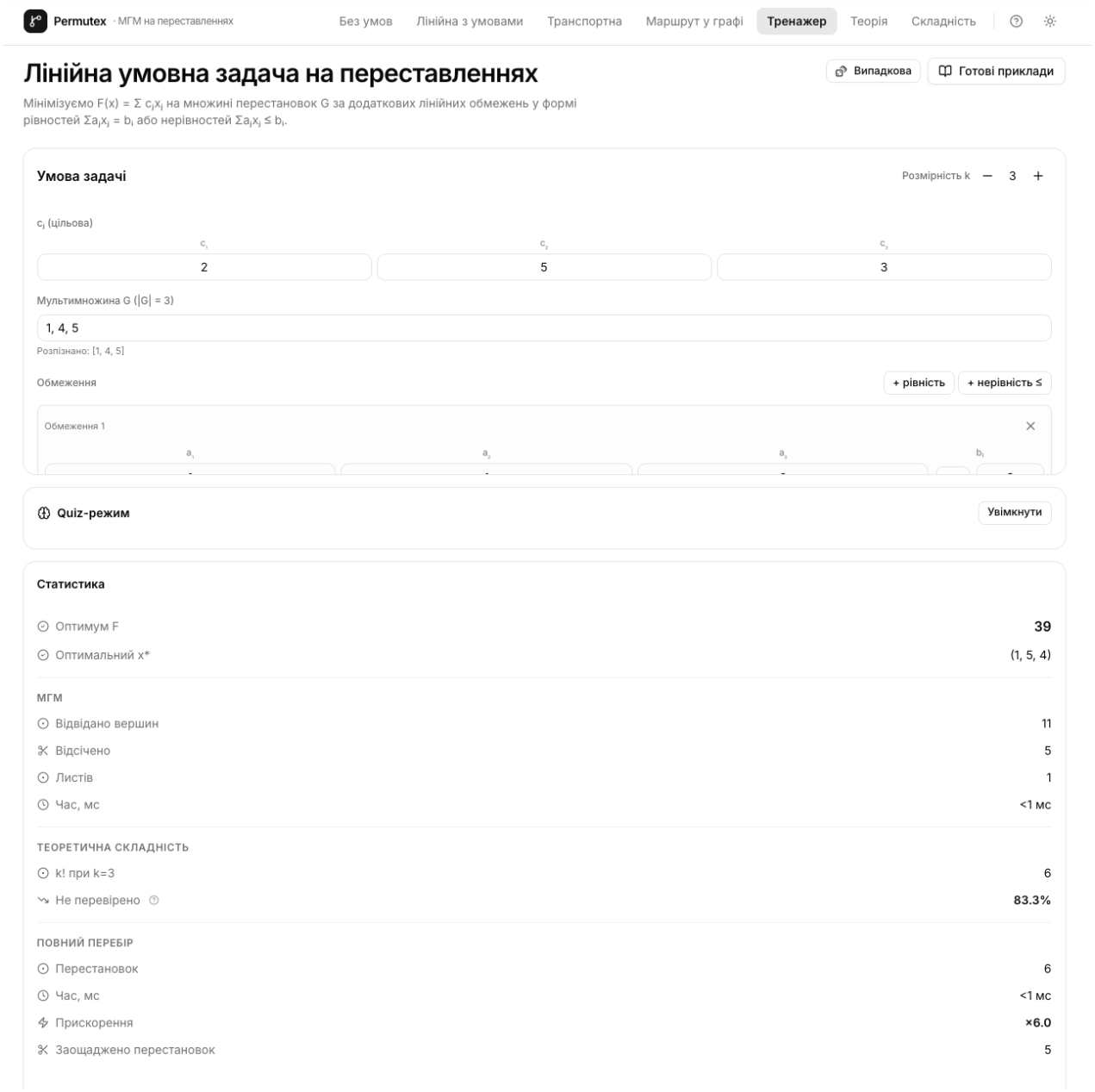

Рисунок 3.4 - Інтерфейс сторінки лінійної задачі з обмеженнями

Окремий компонент QuizMode реалізує режим самоперевірки під час перегляду розв'язку. У цьому режимі при кожній події evaluate студент має ввести очікувані ν та ξ. Після введення програма ставить ✓ при правильній відповіді або ✗ з показом правильного значення. Лічильник правильних і помилкових відповідей зберігається протягом сесії розв'язання. Цей режим відрізняється від повноцінного тренажера тим, що дерево вже побудоване програмою - студент тільки звіряє оцінки.

Компонент Tutorial реалізує покроковий онбординг при першому вході. Користувач знайомиться з основними елементами інтерфейсу - навігацією, редактором, деревом, кроковим контролером, режимом самоперевірки і тренажером. Стан «бачив онбординг» зберігається в localStorage, але кнопка у хедері дозволяє повторно його запустити.

Перемикач світлої та темної теми реалізований через бібліотеку next-themes. Усі компоненти shadcn/ui автоматично адаптуються до обраної теми завдяки CSS-змінним Tailwind. Це робить роботу у програмі комфортною як вдень, так і ввечері - особливо важливо для тривалих навчальних сесій.

3.4. Реалізація інтерактивного тренажера методу гілок та меж

Окрім демонстраційного режиму, у якому програма самостійно будує дерево пошуку, у Permutex реалізовано принципово інший за призначенням компонент - інтерактивний тренажер. У ньому студент сам послідовно ухвалює всі рішення, які ухвалює алгоритм МГМ: обирає бруньку для розгалуження, фіксує значення з G або клітинку матриці, обчислює і вводить оцінки ν та ξ, ухвалює рішення про відсікання гілки. Програма не показує наперед, як виглядатиме дерево, - вона лише перевіряє кожен крок студента і ставить ✓ або ✗.

Тренажер реалізований чотирма паралельними сторінками: /trainer/simple - для безумовної лінійної задачі, /trainer/linear - для лінійної з обмеженнями, /trainer/transport - для комбінаторної транспортної задачі, /trainer/path - для задачі про найкоротший маршрут у графі. Загальний вигляд інтерактивного тренажера наведено у додатку Б на рисунку Б.3 (див. додаток Б, рис. Б.3).

У тренажері передбачено такі елементи інтерфейсу: ліворуч - постановка поточної задачі та картка-довідка «Як працювати з тренажером» зі списком правил обчислення ν і ξ; у центрі - дерево, побудоване студентом крок за кроком; праворуч - поточна фаза («Введіть ξ»), форма введення, кнопки «Перевірити» та «Підказати» і лічильник правильних/помилкових відповідей. Унизу - лог пройдених кроків з можливістю повернутися до попередньої точки.

Логіка тренажера реалізована як машина станів у Zustand-сторі (useTrainerStore у src/store/trainer-store.ts). Множина фаз: pick-node - обрати з фронтиру (frontier) активну бруньку для розгалуження; pick-value - обрати значення з G, яке буде зафіксоване у наступній координаті; enter-nu - ввести оцінку ν; enter-xi - ввести оцінку ξ; decide-fate - ухвалити рішення «розгалужувати», «лист» або «відсікти»; done - завершення сесії. На кожній фазі двигун trainer-engine.ts обчислює канонічну відповідь і порівнює її з введенням студента.

Перевірка кроку відбувається таким чином. На фазі pick-node двигун визначає множину допустимих бруньок (тих, що ще не оброблені та потенційно перспективні) і приймає вибір студента, якщо обрана брунька входить до цієї множини; інакше показується підказка з переліком допустимих варіантів. На фазі pick-value двигун перевіряє, чи обране значення є серед залишкових елементів G з урахуванням правила унікальності значень. На фазах enter-nu та enter-xi введене число порівнюється з обчисленим за функціями computeNu і computeXi із порогом точності 10^{-9}. На фазі decide-fate студент має визначити, чи слід відсікти бруньку (якщо $\xi \geq F_0$), позначити її як лист (якщо координати вичерпано) або продовжити галуження; двигун обчислює очікувану дію і порівнює.

Підрахунок результатів ведеться у трьох полях стану: correct (правильні відповіді), wrong (помилки) і hints (використані підказки). Підказка не вважається помилкою, але збільшує лічильник підказок, який також відображається у фінальній статистиці. Лічильник скидається при початку нової сесії - кнопка «Розпочати заново» у тренажері.

Окремою особливістю тренажера є збереження журналу подій (log) - для кожного кроку фіксується тип фази, дія студента, результат перевірки і коротке пояснення. Журнал відображається у компоненті TrainerLog і дозволяє студенту переглянути історію власного розв'язку. Функція revertToStep дозволяє відкотитися до попереднього кроку, що використовується для виправлення помилок без необхідності починати сесію спочатку.

Тренажер транспортної задачі використовує специфіку матричного представлення: галуження виконується не за фіксацією координат у векторі, а за

фіксацією клітинок матриці x_ij. Двигун trainer-engine-transport.ts реалізує перевірку коректності кожного кроку студента з урахуванням балансових обмежень за рядками і стовпцями.

Тренажер для задачі про найкоротший маршрут (/trainer/path) працює інакше - там немає поняття перестановки і оцінки ξ. Студент рухається по графу від стартової вершини, обираючи на кожному кроці одне з вихідних ребер. Оцінка бруньки - це поточна сума ваг пройдених ребер. Двигун trainer-engine-path.ts перевіряє, чи обране ребро існує у графі, чи не утворюється цикл, і чи студент правильно ухвалює рішення про відсікання тупикових гілок. Стан зберігається у useTrainerStorePath з полем pathPrefix (масив пройдених вершин) і pathCost (поточна сума ваг).

Логіка всіх трьох сторів покрита модульними тестами: trainer-store.test.ts (4 тести), trainer-store-transport.test.ts (2 тести), trainer-store-path.test.ts (3 тести), а двигун trainer-engine - 15 тестами у trainer-engine.test.ts. Загалом тренажер покрито 24 тестовими випадками, що додаються до 33 тестів обчислювального ядра. Сумарне покриття проєкту становить 57 тестів, усі проходять успішно.

На індексній сторінці тренажерів (див. додаток Б, рис. Б.2) представлено чотири картки - за одну на тип задачі, з коротким описом особливостей кожного режиму та кнопкою «Розпочати тренажер». Натискання картки переводить користувача на відповідну сторінку, де процес тренування розпочинається з вибору задачі або відкриття готового прикладу з лекцій 6 або 7.

3.5. Тестування програмного продукту

Тестування проводилось у три етапи: модульне тестування обчислювального ядра і логіки тренажерів, ручне функціональне тестування інтерфейсу та експериментальне тестування ефективності.

Модульне тестування реалізоване на бібліотеці Vitest. Як зазначено у п. 3.2 і 3.4, ядро покрите 33 тестами, логіка тренажерів - 24 тестами, разом - 57 тестових випадків. Тести запускаються командою pnpm test і виконуються за час менше 1

секунди. Усі 57 тестів проходять (57 passed, 0 failed). Це підтверджує коректність обчислень оцінок ν, ξ, відсікань, знайдених оптимумів і коректність перевірки кроків у тренажерах.

Функціональне тестування інтерфейсу виконано вручну за переліком тест-кейсів, що відповідає функціональним вимогам з постановки задачі. Перевірено: введення задач у всіх редакторах, валідація вводу, генерація випадкових задач, відкриття готових прикладів з лекцій, перемикання між min і max, покрокова навігація вперед і назад, автозапуск з регулюванням швидкості, клавіатурні скорочення, режим Quiz, проходження сценарію тренажера для всіх чотирьох типів задач (правильні і помилкові відповіді, використання підказок, відкат до попереднього кроку, скидання сесії), перемикання теми, відкриття tutorial, адаптація на мобільному екрані. Усі тест-кейси пройшли успішно.

Експериментальне тестування ефективності проведене на сторінці /complexity. Програма автоматично запускає МГМ і повний перебір на однотипних випадково згенерованих задачах розмірностей k = 3, 4, …, 9 і будує графік залежності кількості перевірених перестановок від k у логарифмічному масштабі. Результати, узагальнені у таблиці 3.1, демонструють експоненціальне зростання повного перебору і поліноміальну поведінку МГМ для тестових задач (див. табл. 3.1).

Таблиця 3.1 - Порівняння кількості перевірених перестановок для МГМ і повного перебору

k	**k!**	**Листів МГМ (середнє)**	**Економія, %**
3	6	3	50
4	24	7	71
6	120	14	88
6	720	28	96
7	5040	52	99
8	40320	87	99,8

9	362880	124	99,97

Як видно з таблиці, для k = 9 МГМ перевіряє у середньому близько 124 листків замість 362 880, що є економією 99,97 %. Це повністю підтверджує теоретичні очікування і експериментально демонструє цінність методу для практичних застосувань.

Окремо проведено інтеграційне тестування на еталонних прикладах з лекцій 6 і 7. Усі чотири еталонні значення відтворено програмою без помилок: F_min = 27 (приклад 2 лекції 6), F* = 4 (приклад 1 лекції 6, маршрут 1→2→3→6→7), F_0 = 203 (приклад 1 лекції 7), F_0 = 240 (приклад 3 лекції 7). Це є основним функціональним тестом коректності реалізації методу гілок та меж.

3.6. Інструкція користувача

Розглянемо порядок роботи з програмою «Permutex» з точки зору кінцевого користувача - студента, який вивчає курс «Елементи комбінаторної оптимізації» і використовує застосунок як навчальний інструмент.

Запуск програми відбувається у браузері. Достатньо відкрити URL застосунку (наприклад, http://localhost:3000 при локальному запуску, або URL хмарного розгортання). Перший вхід супроводжується відображенням онбордингу, у якому пояснюється структура інтерфейсу і основні можливості - навігація, редактор, дерево, кроковий контролер, режим самоперевірки, тренажер. Після завершення онбордингу користувач потрапляє на головну сторінку.

На головній сторінці представлено чотири картки за типами задач: «Без умов» (безумовна лінійна задача), «Лінійна» (лінійна задача з обмеженнями), «Транспортна» (КТЗП), «Маршрут» (найкоротший маршрут у графі). Окремий блок навігації веде на сторінку тренажера. Користувач натискає на потрібну картку і переходить на відповідну сторінку (див. додаток Б, рис. Б.1).

На сторінці задачі ліворуч розташований редактор. Для безумовної задачі редактор містить поля «Коефіцієнти c (через кому)», «Мультимножина G (через

кому)», прапорець min/max і кнопки «Розв'язати», «Випадкова», «Скинути», а також меню «Приклади з лекцій». Користувач може як самостійно ввести параметри задачі, так і обрати готовий приклад з лекції 6 чи 7.

Після натискання «Розв'язати» праворуч з'являється дерево галуження. Поки користувач не натиснув жодної кнопки контролера, дерево показує лише корінь з обчисленими ν та ξ. Натискання «Уперед» (або клавіша →) переводить алгоритм на наступний крок: розгалуження кореня, обчислення оцінок дітей, перехід до листа і так далі. Кожен крок супроводжується пояснювальним текстом українською мовою у нижній частині сторінки.

У режимі автозапуску (кнопка «Грати») алгоритм виконується автоматично з регульованою швидкістю - від 100 мс до 2000 мс на крок. Це зручно, коли користувач хоче побачити роботу алгоритму у динаміці без необхідності клацати кожен крок вручну. Натискання Space зупиняє автозапуск.

Режим самоперевірки активується кнопкою «Quiz». У цьому режимі при кожному кроці evaluate з'являється форма введення очікуваних ν і ξ. Користувач вводить значення і натискає «Перевірити»; програма порівнює введене з обчисленим і показує ✓ або ✗ з поясненням. Це особливо корисно для підготовки до контрольних робіт.

Для повноцінного відпрацювання алгоритму призначений окремий розділ «Тренажер». На сторінці /trainer обирається тип задачі (одне з чотирьох), після чого студент вводить задачу або відкриває готовий приклад. Далі тренажер послідовно проводить через фази: обери бруньку → обери значення з G → введи ν → введи ξ → ухвал рішення про prune/keep. На кожному кроці студент отримує негайний зворотний зв'язок: ✓ при правильній відповіді або ✗ з підказкою. Лічильник правильних і помилкових відповідей видно у правій частині панелі. Кнопка «Підказати» показує канонічну відповідь, але не вважає це помилкою - лише збільшує окремий лічильник підказок. Кнопка «Відкотити» дозволяє повернутися на попередній крок.

Панель статистики StatsPanel розташована під деревом і показує поточні значення: кількість вершин, листків, відсікань, час виконання МГМ і повного

перебору, коефіцієнт економії. Усі значення оновлюються після завершення кожного розв'язання.

На сторінці /theory користувач може ознайомитись з теоретичною довідкою - формулами ν, ξ, теоремами 1-5 з лекції 7, прикладами і поясненнями. Усі формули відображаються через KaTeX у математичній нотації, ідентичній лекційним матеріалам.

На сторінці /complexity представлено експериментальне порівняння ефективності МГМ і повного перебору. Користувач може запустити автоматичний benchmark на розмірностях від 3 до 9 і побачити SVG-графік у логарифмічному масштабі та таблицю з кількісними показниками.

Перемикання між світлою і темною темою здійснюється кнопкою у хедері - праворуч від навігаційного меню. Програма запам'ятовує вибір користувача і застосовує його при наступних відвідуваннях.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи розроблено навчальне програмне забезпечення «Permutex» для вивчення теми «Метод гілок та меж для задач оптимізації на переставленнях» курсу «Елементи комбінаторної оптимізації» спеціальності 122 «Комп'ютерні науки» Полтавського університету економіки і торгівлі.

У результаті виконання роботи отримано такі результати:

- проаналізовано предметну область задач комбінаторної оптимізації на переставленнях, обґрунтовано актуальність розробки навчального програмного засобу для теми;
- виконано порівняльний огляд існуючих програмних рішень - комерційних солверів, відкритих солверів, навчальних онлайн-курсів, GitHub-скриптів - і показано, що жоден з них не задовольняє повний набір вимог до навчального ПЗ для курсу ЕКО ПУЕТ;
- обґрунтовано вибір технологічного стеку: TypeScript, Next.js 16, React 19, shadcn/ui, Tailwind CSS 4, React Flow, KaTeX, Zustand, Vitest;
- опрацьовано теоретичні основи методу гілок та меж за матеріалами лекцій 6 і 7 курсу: загальна схема, формули оцінок ν і ξ, властивості оцінок, правила А і В для комбінаторної транспортної задачі;
- спроєктовано модульну архітектуру з чітким розділенням обчислювального ядра, логіки тренажерів, сховища станів і користувацького інтерфейсу;
- реалізовано обчислювальне ядро для чотирьох типів задач (безумовна лінійна, лінійна з обмеженнями, КТЗП, найкоротший маршрут у графі);
- реалізовано інтерактивний користувацький інтерфейс з покроковою візуалізацією дерева галуження на бібліотеці React Flow, поясненнями кожного кроку через KaTeX, режимом самоперевірки і онбордингом;
- реалізовано чотири інтерактивні тренажери (для безумовної задачі, для задачі з обмеженнями, для комбінаторної транспортної задачі та для задачі про маршрут), у яких студент сам будує дерево пошуку, а програма перевіряє

кожен крок з підрахунком правильних і помилкових відповідей та підказками;

- покрите модульне тестування ядра і логіки тренажерів - 57 тестових випадків (33 на ядро та 24 на двигун і сховища тренажерів), усі тести проходять успішно;
- проведено функціональне тестування інтерфейсу і експериментальне тестування ефективності МГМ для розмірностей k = 3..9;
- на еталонних прикладах з лекцій відтворено всі очікувані оптимуми ($F = 27$, $F = 4$, $F_0 = 203$, $F_0 = 240$), що підтверджує коректність математичної реалізації;
- підготовлено інструкцію користувача та оформлено пояснювальну записку.

Розроблений навчальний застосунок «Permutex» демонструє ефективність методу гілок та меж порівняно з повним перебором: за результатами експериментального тестування для розмірності k = 9 кількість розглянутих листків зменшується з 362 880 до приблизно 124, що відповідає економії 99,97 %. Це наочно показує студенту цінність відсікань і обґрунтовує практичне значення методу. Інтерактивний тренажер дає студенту можливість не лише пасивно спостерігати за роботою алгоритму, а й самостійно ухвалювати ті ж рішення, які ухвалює алгоритм, з негайним зворотним зв'язком - це принципово підвищує глибину розуміння теми порівняно з традиційними навчальними матеріалами.

Поставлена в кваліфікаційній роботі мета досягнута, всі визначені у постановці задачі підзадачі виконані. Розроблений програмний продукт може бути впроваджений у навчальний процес кафедри математичного моделювання та соціальної інформатики ПУЕТ як інтерактивний інструмент для вивчення теми «Метод гілок та меж для задач оптимізації на переставленнях».

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Ємець О. О. Елементи комбінаторної оптимізації: Метод гілок та меж для задач оптимізації на переставленнях. - Полтава: ПУЕТ. - 36 с.
2. Сергієнко І. В., Шило В. П. Задачі дискретної оптимізації: проблеми, методи розв'язування, дослідження. - Київ: Наукова думка, 2021. - 360 с.
3. IBM ILOG CPLEX Optimization Studio. Documentation. URL: https://www.ibm.com/docs/en/icos
4. Gurobi Optimizer Reference Manual. URL: https://docs.gurobi.com/projects/optimizer/en/current/reference.html
5. COIN-OR CBC: Coin-or Branch and Cut. URL: https://github.com/coin-or/Cbc
6. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/
7. Next.js Documentation. URL: https://nextjs.org/docs
8. shadcn/ui - Beautifully designed components. URL: https://ui.shadcn.com/docs
9. React Flow (xyflow) Documentation. URL: https://reactflow.dev/learn
10. KaTeX - The fastest math typesetting library for the web. URL: https://katex.org/docs/
11. Zustand - Bear necessities for state management. URL: https://zustand.docs.pmnd.rs/
12. Vitest - Next Generation Testing Framework. URL: https://vitest.dev/guide/
13. Hardy G. H., Littlewood J. E., Pólya G. Inequalities: rearrangement inequality. Cambridge Mathematical Library, 2nd ed. URL: https://en.wikipedia.org/wiki/Rearrangement_inequality
14. Ольховська О. В. Методичні рекомендації до виконання та захисту кваліфікаційних робіт студентами спеціальності 122 «Комп'ютерні науки» освітнього ступеня «Бакалавр». - Полтава: ПУЕТ, 2024. - 67 с.

ДОДАТОК А

```
// ===== Файл: src/core/types.ts =====
/**
* Типи проєкту Permutex – задачі оптимізації на переставленнях
* методом гілок та меж (МГМ).
*
* Дивись .thinking/theory.md для формул.
*/

//
================================================================
======
// Постановки задач
//
================================================================
======

/**
*   F(x) = Σ c_j * x_j → min
*   x ∈ E(G), G – мультимножина.
*/
export type Objective = 'min' | 'max';
export type BranchStrategy = 'dfs' | 'bfs' | 'best';

export interface SimpleProblem {
kind: 'simple';
/** Коефіцієнти цільової функції. Довжина = k. */
c: number[];
/** Мультимножина G (можна з повторами). Довжина = k. */
G: number[];
/** Напрям оптимізації. За замовчуванням 'min'. */
objective?: Objective;
}
```

```
/**
*   F(x) = Σ c_j * x_j → min
*   Σ a_ij * x_j = b_i  для i = 1..l
*   Σ a_ij * x_j ≤ b_i  для i = l+1..m
*   x ∈ E(G)
*/
export interface LinearProblem {
kind: 'linear';
c: number[];
G: number[];
/** Рядки матриці A для рівностей (кожен рядок – довжина k). */
equalities: Array<{ a: number[]; b: number }>;
/** Рядки матриці A для нерівностей (≤). */
inequalities: Array<{ a: number[]; b: number }>;
objective?: Objective;
}

/**
*   Σ_ij c_ij * x_ij → min
*   Σ_j x_ij = a_i   (пропозиція)
*   Σ_i x_ij = b_j   (попит)
*   x ∈ E(G), |G| = m*n
*/
export interface TransportProblem {
kind: 'transport';
/** m×n матриця вартостей. */
costs: number[][];
/** Вектор пропозицій, довжина m. */
supplies: number[];
/** Вектор попитів, довжина n. */
demands: number[];
/** Мультимножина G, |G| = m*n. */
G: number[];
objective?: Objective;
}
```

```
/**
* Направлений зважений граф.
*/
export interface PathProblem {
kind: 'path';
/** Кількість вершин. Вершини - цілі числа 1..nodes. */
nodes: number;
/** Ребра: from → to з вагою weight. */
edges: Array<{ from: number; to: number; weight: number }>;
start: number;
end: number;
}

export type Problem = SimpleProblem | LinearProblem | TransportProblem
| PathProblem;

//
============================================================================
======
// Вузли дерева галуження
//
============================================================================
======

export type NodeStatus =
| 'pending' // створений, оцінка ще не обчислена
| 'evaluated' // оцінка ν (і ξ) обчислена
| 'branching' // розгалужений
| 'pruned' // відсічений
| 'leaf' // лист з конкретним F
| 'record' // лист, що встановив новий рекорд
| 'optimum'; // лист, що є глобальним оптимумом

export type PruneReason =
| 'bound' // ν_i ≥ F_0 (класичне відсікання)
| 'rule-a' // правило А для КТЗП
```

```
| 'rule-b-forced' // брунька відсічена після примусового галуження за правилом B
| 'infeasible' // порожня брунька (немає допустимих розв'язків)
| 'duplicate'; // вже розглянута еквівалентна множина

/**
* Вузол дерева пошуку. Для задач на перестановках `fixed` зберігає
* координати, які вже визначені: ключ - індекс координати (0-based),
* значення - призначене значення з G.
*
* Для задачі маршруту `pathPrefix` - поточний шлях від старту.
*/
export interface TreeNode {
id: string;
parentId: string | null;
depth: number;

/** Для задач на перестановках: координата → призначене значення з G.
*/
fixed?: Record<number, number>;
/** Для задач на перестановках: індекси елементів з G, які вже використані. */
usedIndices?: number[];

/** Для задачі маршруту: префікс шляху (список вершин). */
pathPrefix?: number[];
/** Для задачі маршруту: сума ваг пройденого префіксу. */
pathCost?: number;

/** Для транспортної: позначення фіксованих клітинок "i_j" → значення з G. */
transportFixed?: Record<string, number>;

nu?: number;
xi?: number;
```

```
status: NodeStatus;

/** Для листка: значення F(x). */
f?: number;
/** Причина відсікання. */
pruneReason?: PruneReason;

/** Людочитне пояснення (для UI). */
note?: string;
}

//
================================================================
======
// Подія алгоритму (крок trace-у)
//
================================================================
======

export type StepEvent =
| {
type: 'init';
rootId: string;
problem: Problem;
}
| {
type: 'branch';
parentId: string;
childrenIds: string[];
coord?: number; // для перестановок: яку координату фіксуємо
note: string;
}
| {
type: 'evaluate';
nodeId: string;
nu?: number;
```

```
xi?: number;
/** Людочитне парування c→g, з якого утворено c*. */
cStarBreakdown?: string;
note: string;
}
| {
type: 'leaf';
nodeId: string;
f: number;
x: number[];
note: string;
}
| {
type: 'prune';
nodeId: string;
reason: PruneReason;
note: string;
}
| {
type: 'record';
nodeId: string;
oldF: number | null;
newF: number;
note: string;
}
| {
type: 'done';
optimumNodeId: string;
optimumF: number;
optimumX: number[];
note: string;
};

//
==========================================================================
======
```

```
// Результат розв'язування
//
================================================================
======

export interface SolveResult {
/** Мінімальне значення F (або null, якщо немає розв'язків). */
optimumF: number | null;
/** Оптимальний x* - перестановка або шлях. */
optimumX: number[] | null;

/** Усі вузли дерева (у порядку створення). */
nodes: TreeNode[];
/** Послідовність подій для візуалізації/покрокового перегляду. */
events: StepEvent[];

stats: {
/** Скільки всього вузлів створено. */
visited: number;
/** Скільки відсічено (pruned). */
pruned: number;
/** Скільки листків перевірено. */
leaves: number;
/** Час виконання у мс. */
elapsedMs: number;
};
}

export interface BruteForceResult {
optimumF: number | null;
optimumX: number[] | null;
stats: {
/** Усього перестановок перевірено. */
permutations: number;
elapsedMs: number;
};
```

```
}

//
============================================================================
======
// Порівняльна статистика
//
============================================================================
======

export interface Comparison {
branchBound: SolveResult['stats'];
bruteForce: BruteForceResult['stats'] | null;
/** Скільки разів МГМ швидший за повний перебір (за кількістю
листків). */
speedup: number | null;
/** Скільки перестановок не перевірено дякуючи відсіченням. */
saved: number | null;
}

// ===== Файл: src/core/multiset.ts =====
/**
* Утиліти для роботи з мультимножинами G.
*
* У математичному сенсі мультимножина – це множина з можливими
повтореннями.
* Ми представляємо її масивом чисел (відсортованим за необхідності).
*/

/** Повертає G відсортовану за неспаданням (g_1 ≤ g_2 ≤ ... ≤ g_k). */
export function sortedAscending(G: number[]): number[] {
return [...G].sort((a, b) => a - b);
}
```

```
/** Повертає G відсортовану за незростанням (g_1 ≥ g_2 ≥ ... ≥ g_k).
*/
export function sortedDescending(G: number[]): number[] {
return [...G].sort((a, b) => b - a);
}

/**
* Парсить рядок користувацького введення у масив чисел.
* Підтримує коми, крапки з комою, пробіли, фігурні дужки.
*   "{1, 2, 3}" → [1, 2, 3]
*   "5 10 15" → [5, 10, 15]
*/
export function parseMultiset(input: string): number[] {
return parseMultisetVerbose(input).values;
}

/**
* Розширений парсер: повертає і числа, і токени, які не вдалося
розпізнати.
* Корисно для UI, щоб попередити користувача про відкинуті частини
вводу.
*/
export function parseMultisetVerbose(input: string): {
values: number[];
ignored: string[];
} {
const cleaned = input
.replace(/[{}\[\]()]/g, ' ')
.replace(/[,;]/g, ' ')
.trim();
if (!cleaned) return { values: [], ignored: [] };

const values: number[] = [];
const ignored: string[] = [];
for (const part of cleaned.split(/\s+/)) {
const value = Number(part.replace(',', '.'));
```

```
if (Number.isFinite(value)) values.push(value);
else ignored.push(part);
}
return { values, ignored };
}

/**
* Видаляє з G елементи за вказаними індексами (позиціями в масиві).
* Не змінює оригінал.
*/
export function removeAt(G: number[], indices: Iterable<number>):
number[] {
const set = new Set(indices);
return G.filter((_, i) => !set.has(i));
}

/**
* Повертає індекси елементів G, які НЕ входять у `usedIndices`.
*/
export function remainingIndices(G: number[], usedIndices:
Iterable<number>): number[] {
const set = new Set(usedIndices);
const result: number[] = [];
for (let i = 0; i < G.length; i++) if (!set.has(i)) result.push(i);
return result;
}

/**
* Індекси УНІКАЛЬНИХ значень серед позицій `indices` у G.
* Якщо в G є повтори (скажімо, g_2 == g_3), то галуження за обома
* дає дубль - галузимо лише за першим входженням.
*
* Приклад: G=[1,2,2,3], indices=[0,1,2,3] → [0,1,3] (друга 2 - дубль
першої).
*/
```

```
export function uniqueValueIndices(G: number[], indices: number[]):
number[] {
const seen = new Set<number>();
const result: number[] = [];
for (const idx of indices) {
const value = G[idx];
if (!seen.has(value)) {
seen.add(value);
result.push(idx);
}
}
return result;
}

/**
* Перевіряє, що сума дорівнює цілі (з допуском на плаваючу
арифметику).
*/
export function approxEqual(a: number, b: number, epsilon = 1e-9):
boolean {
return Math.abs(a - b) < epsilon;
}

// ===== Файл: src/core/bounds.ts =====
/**
* Обчислення оцінок ν (ню) та ξ (ксі) для бруньки в МГМ.
*
*   ν (теорема 2, формула 17): ν = Σ_{j∈R} c_{τ_j} * g_{i_j}
*     - часткова сума на вже зафіксованих координатах.
*   ξ (теорема 3, формула 23): ξ = ν + c*
*     де c* = min скалярного добутку решти коефіцієнтів на решту
значень.
*
* c* рахуємо за правилом: найбільший залишковий коефіцієнт · найменше
залишкове g,
```

```
* наступний за величиною коефіцієнт · наступне за величиною g, і т.д.
*/

/**
* Оцінка ν – часткова сума на зафіксованих координатах.
*
* @param c Коефіцієнти цільової функції (довжина k).
* @param fixed Словник: індекс координати → значення з G, що їй
призначене.
*/
export function computeNu(c: number[], fixed: Record<number, number>):
number {
let sum = 0;
for (const [coordStr, value] of Object.entries(fixed)) {
const coord = Number(coordStr);
sum += c[coord] * value;
}
return sum;
}

/**
* Допоміжна частина c* з оцінки ξ – мінімум скалярного добутку
* залишкових коефіцієнтів на залишкові значення G.
*
* Ключовий принцип:
*   min Σ c_i * x_i  (за фіксованого набору)
* досягається, коли найбільший с множиться на найменше значення з
решти.
*
* Алгоритм:
*   1. Сортуємо залишкові с за незростанням.
*   2. Сортуємо залишкові значення G за неспаданням.
*   3. Домножуємо попарно і підсумовуємо.
*
* Математично: (c_1 ≥ c_2 ≥ ... ≥ c_s) · (g_1 ≤ g_2 ≤ ... ≤ g_s)
*            = Σ c_i * g_i
```

```
* є мінімумом серед усіх перестановок. Це класичний результат
* "rearrangement inequality".
*/
export function computeCStar(remainingCoeffs: number[],
remainingValues: number[]): number {
if (remainingCoeffs.length !== remainingValues.length) {
throw new Error(
`bounds.computeCStar: довжини не збігаються (${remainingCoeffs.length}
vs ${remainingValues.length})`,
);
}
if (remainingCoeffs.length === 0) return 0;

const cDesc = [...remainingCoeffs].sort((a, b) => b - a);
const gAsc = [...remainingValues].sort((a, b) => a - b);

let sum = 0;
for (let i = 0; i < cDesc.length; i++) {
sum += cDesc[i] * gAsc[i];
}
return sum;
}

/**
* Деталь парування c → g при обчисленні c*.
* Використовується у поясненнях: «5·1, 3·4, 2·5 = 27».
*/
export interface CStarPairing {
/** Коефіцієнт (за незростанням). */
c: number;
/** Значення G (за неспаданням). */
g: number;
/** c * g. */
product: number;
}
```

```
/** Як computeCStar, але повертає ще й парування.
* objective='min' → c↓·g↑ (нижня межа); 'max' → c↓·g↓ (верхня межа).
*/
export function computeCStarDetailed(
remainingCoeffs: number[],
remainingValues: number[],
objective: 'min' | 'max' = 'min',
): { cStar: number; pairs: CStarPairing[] } {
if (remainingCoeffs.length !== remainingValues.length) {
throw new Error(
`bounds.computeCStarDetailed: довжини не збігаються
(${remainingCoeffs.length} vs ${remainingValues.length})`,
);
}
const cDesc = [...remainingCoeffs].sort((a, b) => b - a);
const g = [...remainingValues].sort((a, b) => (objective === 'min' ? a
- b : b - a));
const pairs: CStarPairing[] = [];
let cStar = 0;
for (let i = 0; i < cDesc.length; i++) {
const product = cDesc[i] * g[i];
cStar += product;
pairs.push({ c: cDesc[i], g: g[i], product });
}
return { cStar, pairs };
}

/**
* Оцінка ξ = ν + c*.
*
* @param c Усі коефіцієнти (довжина k).
* @param G Уся мультимножина (довжина k).
* @param fixed Зафіксовані координати → значення.
* @param usedIndices Індекси елементів G, що вже використані.
*/
export function computeXi(
```

```
c: number[],
G: number[],
fixed: Record<number, number>,
usedIndices: number[],
objective: 'min' | 'max' = 'min',
): { nu: number; xi: number; cStar: number; pairs: CStarPairing[] } {
const nu = computeNu(c, fixed);

// Індекси КООРДИНАТ, які ще не зафіксовані.
const fixedCoords = new Set(Object.keys(fixed).map(Number));
const remainingCoeffs: number[] = [];
for (let j = 0; j < c.length; j++) {
if (!fixedCoords.has(j)) remainingCoeffs.push(c[j]);
}

// Елементи G, що не використані.
const usedSet = new Set(usedIndices);
const remainingValues: number[] = [];
for (let i = 0; i < G.length; i++) {
if (!usedSet.has(i)) remainingValues.push(G[i]);
}

const { cStar, pairs } = computeCStarDetailed(remainingCoeffs,
remainingValues, objective);
return { nu, xi: nu + cStar, cStar, pairs };
}

/** Формує строку-формулу c* = c₁·g₁ + c₂·g₂ + … = sum (для пояснень).
*/
export function formatCStarBreakdown(pairs: CStarPairing[]): string {
if (pairs.length === 0) return '0';
const terms = pairs.map((p) => `${p.c}·${p.g}`).join(' + ');
const total = pairs.reduce((s, p) => s + p.product, 0);
return `${terms} = ${total}`;
}
```

```
// ===== Файл: src/store/solver-store.ts =====
import { create } from 'zustand';
import type { Problem, SolveResult, BruteForceResult } from
'@/core/types';

/**
* Центральний стан розв'язувача: поточна задача, trace, позиція у
trace,
* стан автозапуску. Використовується всіма сторінками (simple, linear,
* transport, path).
*/
export interface SolverState {
problem: Problem | null;
result: SolveResult | null;
bruteForce: BruteForceResult | null;

currentStep: number;
playing: boolean;
/** Мс між кроками в автозапуску. */
speedMs: number;

setProblem: (problem: Problem | null) => void;
setResult: (result: SolveResult | null, bruteForce?: BruteForceResult
| null) => void;

goToStart: () => void;
goToEnd: () => void;
stepForward: () => void;
stepBackward: () => void;
setStep: (step: number) => void;

togglePlay: () => void;
setPlaying: (value: boolean) => void;
setSpeed: (ms: number) => void;
```

```
reset: () => void;
}

export const useSolverStore = create<SolverState>((set, get) => ({
problem: null,
result: null,
bruteForce: null,
currentStep: 0,
playing: false,
speedMs: 600,

setProblem: (problem) =>
set({ problem, result: null, bruteForce: null, currentStep: 0,
playing: false }),

setResult: (result, bruteForce = null) =>
set({ result, bruteForce, currentStep: 0, playing: false }),

goToStart: () => set({ currentStep: 0, playing: false }),

goToEnd: () => {
const { result } = get();
const max = result ? Math.max(0, result.events.length - 1) : 0;
set({ currentStep: max, playing: false });
},

stepForward: () => {
const { result, currentStep } = get();
if (!result) return;
const max = result.events.length - 1;
if (currentStep < max) set({ currentStep: currentStep + 1 });
else set({ playing: false });
},

stepBackward: () => {
```

```
const { currentStep } = get();
if (currentStep > 0) set({ currentStep: currentStep - 1, playing:
false });
},

setStep: (step) => {
const { result } = get();
if (!result) return;
const clamped = Math.min(Math.max(0, step), result.events.length - 1);
set({ currentStep: clamped });
},

togglePlay: () => set((s) => ({ playing: !s.playing })),
setPlaying: (value) => set({ playing: value }),
setSpeed: (ms) => set({ speedMs: ms }),

reset: () =>
set({
problem: null,
result: null,
bruteForce: null,
currentStep: 0,
playing: false,
}),
}));
```

ДОДАТОК Б

Скріни інтерфейсу програмного продукту «Permutex»

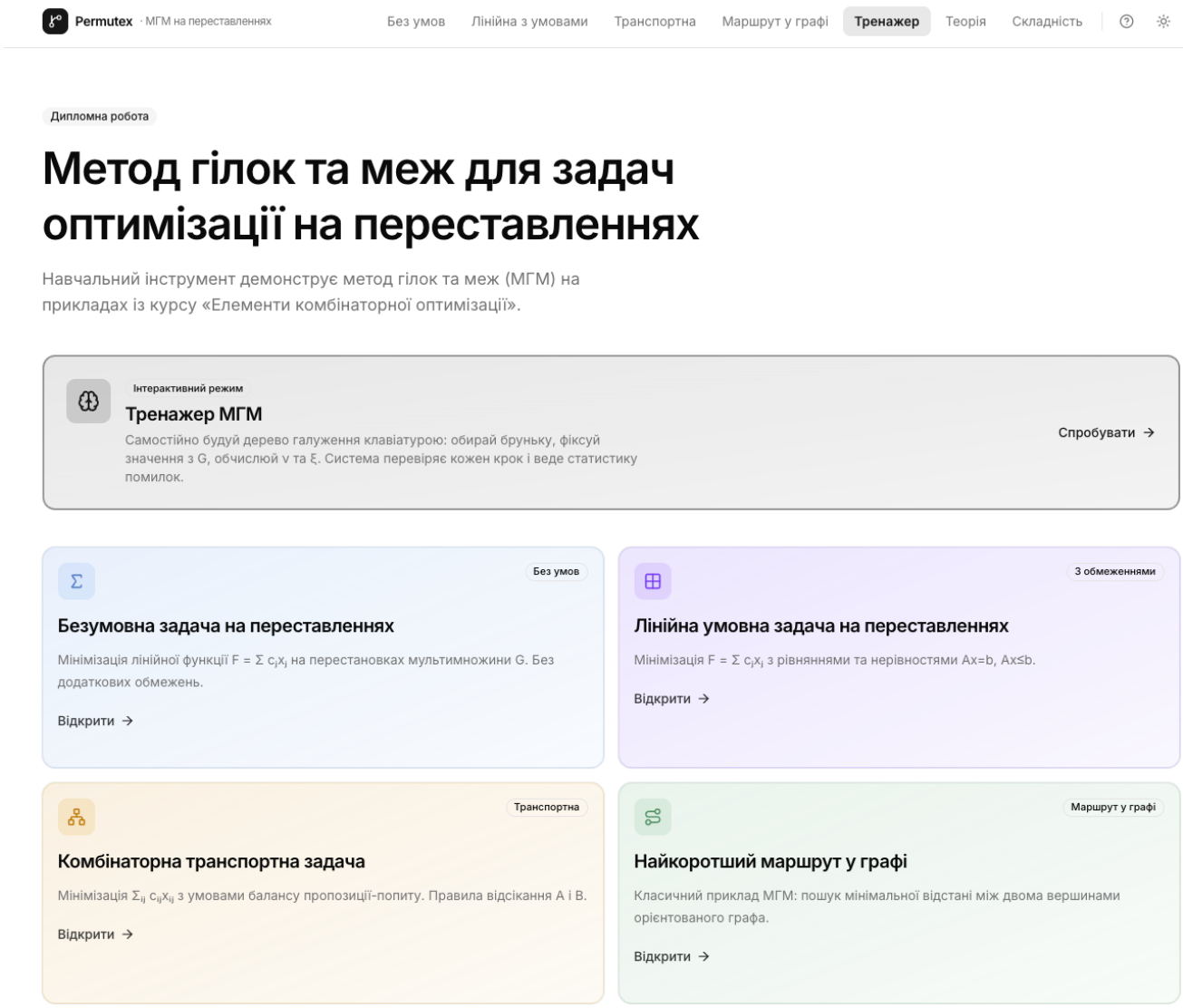

Рисунок Б.1 - Головна сторінка Permutex з картками за типами задач та посиланням на тренажер

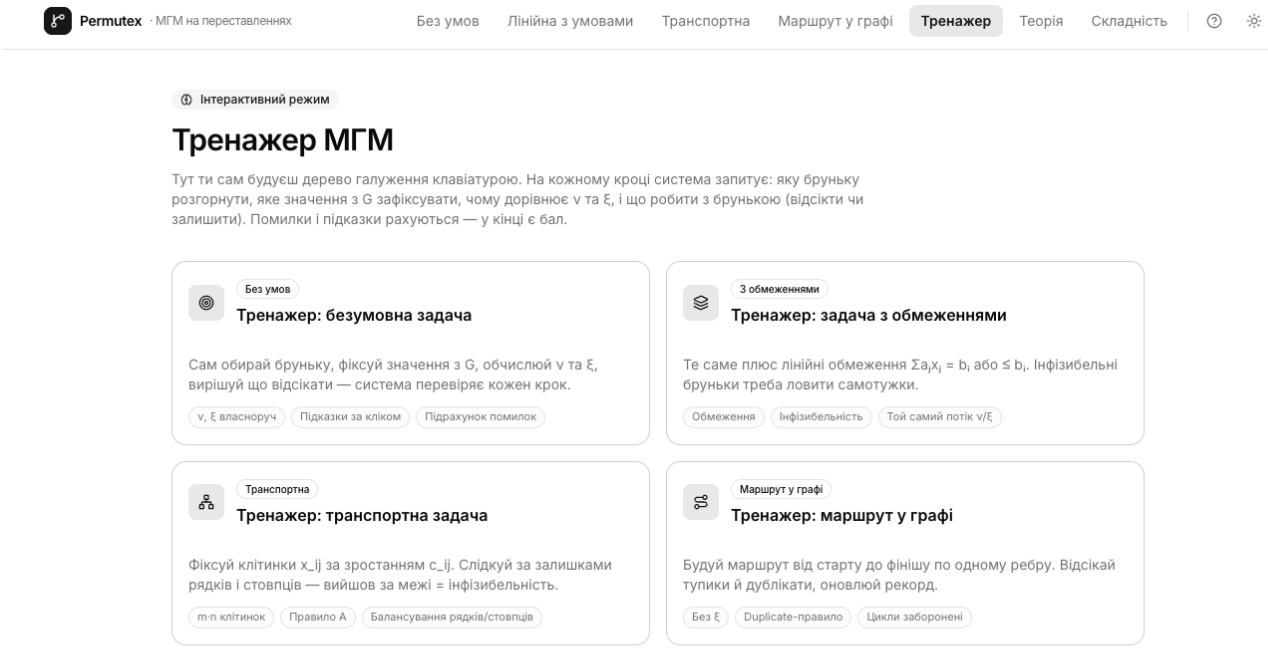

Рисунок Б.2 - Індексна сторінка тренажерів /trainer з картками за типами задач

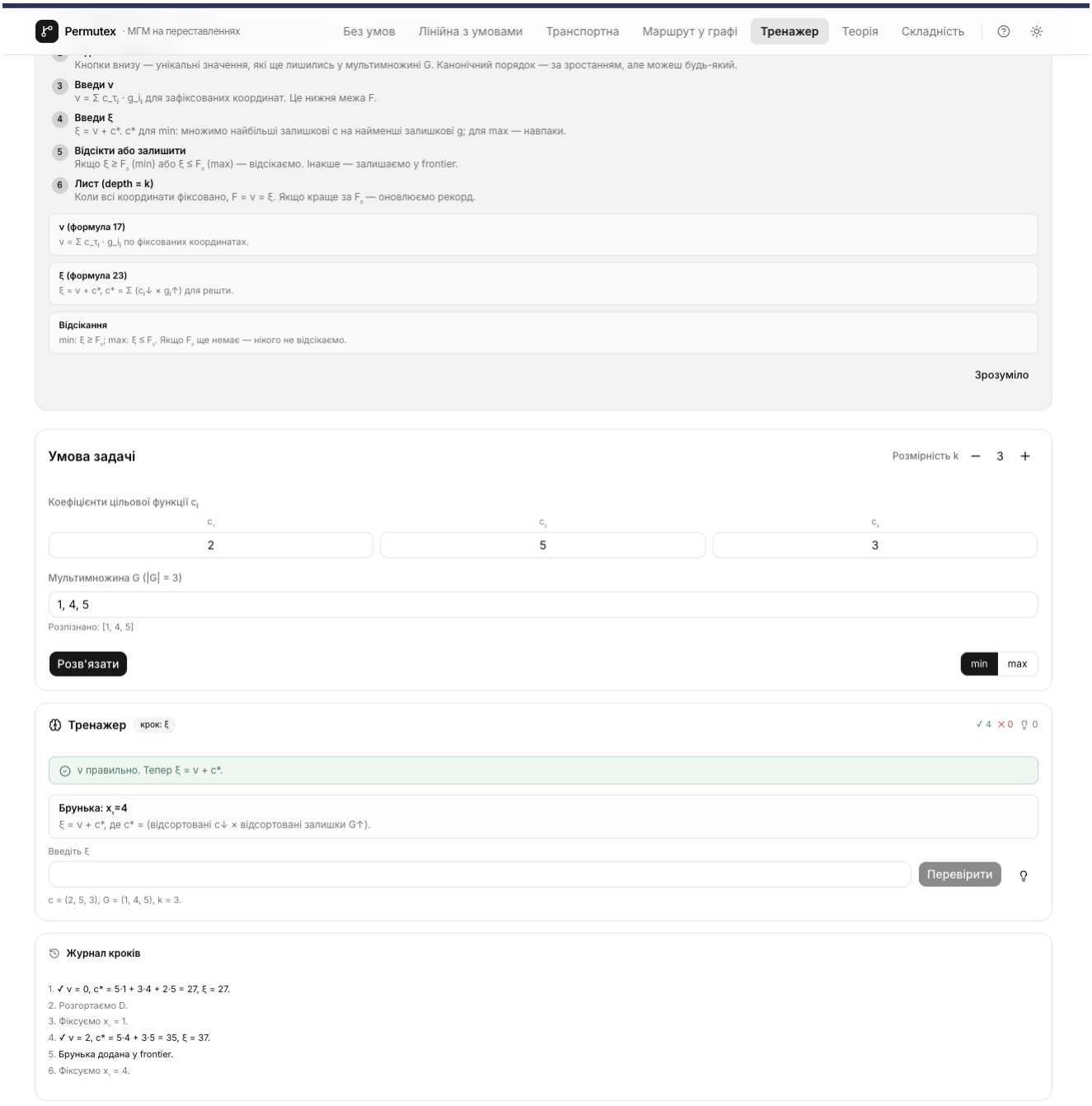

Рисунок Б.3 - Інтерактивний тренажер безумовної задачі: фаза введення оцінки ξ