ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Оптимізація розподілу тестових ресурсів для процесу тестування програмного забезпечення»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Андрій ЩЕРБИНА
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Андрій ЩЕРБИНА

Керівник: Людмила СОЛЯНИК
д-р. хім. наук, доц.

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Щербині Андрію Сергійовичу

1. Тема кваліфікаційної роботи: «Оптимізація розподілу тестових ресурсів для процесу тестування програмного забезпечення»

керівник кваліфікаційної роботи Людмила СОЛЯНИК, д-р. хім. наук, доц.,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, характеристики доступних тестових ресурсів, набір тестових сценаріїв та їх атрибутів, вимоги до якості програмного продукту.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження підходів до оптимізації розподілення тестових ресурсів.
2. Аналіз задач багатокритеріальної оптимізації та методів їх розв’язування
3. Розробка та валідація методу оптимального розподілення тестових ресурсів на основі графової моделі та генетичного алгоритму.

5. Перелік ілюстративного матеріалу: *презентація*

1. Актуальність роботи.
2. Графова модель оптимізаційної задачі розподілу тестових ресурсів.
3. Схема гібридного підходу до рішення багатокритеріальної задачі оптимального розподілу тестових ресурсів.
4. Алгоритм пошуку оптимального рішення багатокритеріальної задачі оптимального розподілу тестових ресурсів.
5. Практичні результати.
6. Результати моделювання.

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз науково-технічної літератури, вивчення сучасних підходів до тестування ПЗ	31.10 – 03.11.2025	
2	Дослідження предметної області, аналіз особливостей процесу тестування	04.11 – 05.11.2025	
3	Огляд моделей та методів багатокритеріальної оптимізації	06.11 – 14.11.2025	
4	Формування графової моделі тестових завдань та ресурсів	15.11 – 30.11.2025	
5	Розробка гібридного методу оптимізації	01.12 – 09.12.2025	
6	Реалізація програмного прототипу та проведення експериментальних досліджень	10.12 – 14.12.2025	
7	Оформлення роботи: вступ, висновки, реферат	15.12 – 17.12.2025	
8	Розробка демонстраційних матеріалів	18.12 – 19.12.2025	

Здобувач вищої освіти _____ Андрій ЩЕРБИНА
(підпис)

Керівник
кваліфікаційної роботи _____ Людмила СОЛЯНИК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: _80_ стор., _11_ табл., _10_ рис., 63 джерела.

Мета роботи – оптимізація розподілу тестових ресурсів для тестування програмного забезпечення за рахунок використання гібридного підходу на основі графової моделі та генетичного алгоритму.

Об'єкт дослідження – розподіл тестових ресурсів для процесу тестування програмного забезпечення.

Предмет дослідження – моделі та методи оптимального розподілу тестових ресурсів для процесу тестування програмного забезпечення.

У роботі використано різноманітні методи математичного моделювання, теорії графів, багатокритеріальної оптимізації та евристичні підходи, зокрема генетичний алгоритм NSGA-II.

Проведено аналіз сучасних методів та підходів до автоматизованого планування тестування, оптимізації ресурсів та штучного інтелекту у сфері забезпечення якості програмного забезпечення.

Розроблено гібридний підхід до оптимізації розподілу тестових ресурсів, що поєднує формалізацію тестових залежностей у вигляді графу та генетичний алгоритм для пошуку найкращих рішень. Запропонований підхід дозволяє будувати розклади для процесу тестування з урахуванням пріоритетів тестів, послідовності їх виконання, обмежень ресурсів персоналу та їх кваліфікації.

Проведено серію експериментів для оцінки ефективності запропонованого методу порівняно з базовими стратегіями ручного сформованого плану тестування програмного забезпечення

КЛЮЧОВІ СЛОВА: РЕСУРС ТЕСТУВАННЯ, МОДЕЛЬ, ГРАФ, БАГАТОКРИТЕРІАЛЬНА ОПТИМІЗАЦІЯ, ГЕНЕТИЧНИЙ АЛГОРИТМ

ABSTRACT

Text part of the master's qualification work: _80_ pages, _10_ pictures, _11__ table, _63_ sources.

The purpose of the work is to optimize the allocation of testing resources for software testing through the use of a hybrid approach based on a graph model and a genetic algorithm.

Object of research is the allocation of testing resources within the software testing process.

Subject of research is the models and methods for optimal allocation of testing resources in the software testing process.

Summary of the work: This work investigates the problem of optimally allocating testing resources within the software testing process under conditions of limited personnel, time, and computational capacity. The study employs methods of mathematical modeling, graph theory, multicriteria optimization, and metaheuristic search, with particular focus on the NSGA-II genetic algorithm. A comprehensive analysis of state-of-the-art approaches in automated test planning, resource optimization, and the application of artificial intelligence in software quality assurance has been conducted.

A hybrid optimization approach is proposed, combining a graph-based formalization of test dependencies with an evolutionary search mechanism to generate near-optimal testing schedules. The developed method accounts for test priorities, dependency constraints, execution order, resource availability, and tester qualifications.

To validate the effectiveness of the proposed approach, a series of experimental studies was performed. The results demonstrate that the hybrid method outperforms baseline manually constructed testing plans in terms of coverage, resource utilization, and schedule efficiency, confirming its applicability for complex real-world testing environments.

KEYWORDS: TESTING RESOURCE, MODEL, GENETIC ALGORITHM, GRAPH, MULTICRITERIA OPTIMIZATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AHP	Analytic Hierarchy Process
AI	Artificial intelligence
BDD	Behavior-Driven Development
CD	Continuous Delivery
Cell-DE	Hybrid Cellular Differential Evolution
CI	Continuous Integration
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DE	Differential Evolution
DEMATEL	Decision Making Trial and Evaluation Laboratory
FDP	Fault Detection Process
FTM	Fault Tolerance Models
GA	Genetic Algorithm
GLSA	Genetic Local Search Algorithm
GP	Gradient Projection
GRG	Generalized Reduced Gradient
HaD	Harmonic Distance Algorithm
HC	Hill Climbing
MEMV-OWA	Maximal Entropy Minimum Variance Ordered Weighted Averaging
ML	Machine learning
MODE	Multi-Objective Differential Evolution
NHPP	Non-Homogeneous Poisson Process
NSGA-II	Non-dominated Sorting Genetic Algorithm II
PAES	Pareto Archived Evolution Strategy
PSO	Particle Swarm Optimization
QA	Quality Assurance
RAP	Redundancy Allocation Problem
RWGA	Random Weight Genetic Algorithm
SA	Simulated Annealing
SCC	Strongly Connected Components
SLP	Sequential Linear Programming
SPEA2	Strength Pareto Evolutionary Algorithm2
SQP	Sequential Quadratic Programming
SRGM	Software Reliability Growth Models
TEF	Testing Effort Functions
TRAP	Testing Resource Allocation Problem
UFM	Usage Factor Model
WNS-MODE	Weighted Normalized Sum-MODE
ІТ	Інформаційні технології
ПЗ	Програмне забезпечення

ВСТУП

Життєвий цикл програмного забезпечення складається з кількох етапів, найбільш вартісним з яких є етап тестування. Він вимагає значних затрат усіх основних тестових ресурсів від часу та спеціалістів до програмних інструментів та технічного обладнання. У сучасних умовах програмні рішення та автоматизовані системи стають більшими та складнішими кожного дня. Однак, цього не можна сказати про ресурси для проведення тестування їх функціональності та безпеки. Оскільки ресурси тестування є обмеженими, їх нестача призводить до затримок у релізах програмних продуктів, збільшення витрат на їх тестування та зниження якості функціонування. Єдиним шляхом забезпечення якості та безпеки програмних продуктів є якомога ефективніше використання тих ресурсів, що є у наявності, та оптимальний їх розподіл у процесі тестування.

Сучасні підходи до оптимального розподілу тестових ресурсів базуються на евристичних та метаевристичних алгоритмах, багатокритеріальних оптимізаційних моделях та автоматизованих системах планування робіт. Дедалі частіше для таких задач використовуються методи та моделі штучного інтелекту. Такі моделі надають більше можливостей в аналізі структур задач та дозволяють обирати найкращі стратегії їх вирішення.

Таким чином, задача розробки підходу до оптимізації розподілу тестових ресурсів при тестуванні програмного забезпечення з метою підвищення ефективності цього процесу є сучасною, актуальною та перспективною.

Мета роботи – оптимізація розподілу тестових ресурсів для тестування програмного забезпечення за рахунок використання гібридного підходу на основі графової моделі та генетичного алгоритму.

Об'єкт дослідження – розподіл тестових ресурсів для процесу тестування програмного забезпечення.

Предмет дослідження – моделі та методи оптимального розподілу тестових ресурсів для процесу тестування програмного забезпечення.

Для досягнення мети вирішувалися наступні завдання.

1. Огляд існуючих підходів до задачі розподілу тестових ресурсів.
2. Розробка математичної моделі оптимізації розподілу тестових ресурсів, визначення та формалізація обмежень на кваліфікацію ресурсів та топологічні залежності тестових випадків.
3. Розробка підходу та алгоритмічного забезпечення пошуку оптимальних рішень розподілу ресурсів.
4. Проектування архітектури програмного забезпечення для розподілу тестових ресурсів, обрання технологій та засобів реалізації компонентів системи
5. Розробка програмного забезпечення для апробації розробленого підходу.
6. Проведення моделювання роботи розробленого підходу та оцінка ефективності отриманих розподілів тестових ресурсів.

Робота складається з вступу, чотирьох розділів, висновку, переліку посилань та додатків. У першому розділі проводиться огляд основних понять та положень процесу тестування та тестових ресурсів, а також огляд актуальних наукових робіт, щодо оптимального розподілу тестових ресурсів. Другий розділ присвячений аналізу підходів, моделей, методів та алгоритмів, що використовуються для пошуку оптимальних розподілів, оглянуто їх переваги та недоліки. Третій розділ присвячений опису розробки та реалізації гібридного підходу. Четвертий розділ присвячений перевірці розробленого підходу, оцінці результатів моделювання та їх аналізу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Тестові ресурси та підходи до їх ефективного використання

Тестові ресурси – це будь-які активи, які необхідні для ефективного планування, виконання та оцінки процесу тестування програмного забезпечення. Ці ресурси є обмеженими і для забезпечення якісного тестування в рамках встановлених термінів та бюджету потребують ефективного управління. До тестових ресурсів належать як матеріальні активи, так і нематеріальні ресурси. До нематеріальних ресурсів відносяться час та персонал, до матеріальних – обладнання та бюджет. Розглянемо тестові ресурси, що вважаються основними.

Час. Найбільш критичний та обмежений ресурс у процесі тестування. Недостатність цього ресурсу може привести до значного скорочення процесу тестування, пропуску критичних дефектів та зниження якості програмного забезпечення. Для ефективного управління часом необхідно конкретне визначення потреби в цьому ресурсі та встановлення конкретних та чітких термінів виконання кожного його етапу.

Персонал. Основний інтелектуальний ресурс у процесі тестування. До складу цього ресурсу відносять усіх співробітників, які володіють необхідним досвідом, знаннями та навичками. Рівень кваліфікації, отриманий впродовж трудової діяльності досвід, а також і мотивація співробітників суттєво впливають на якість проведення тестування. Обмежена кількість або низька кваліфікація персоналу, або його перевантаженість, необхідність виконання рутинних завдань, що легко можуть бути автоматизовані, призводить до низького рівня тестування та пропуску дефектів у програмних продуктах.

Обладнання. Ресурс, що об’єднує як фізичні так і віртуальні ресурси, необхідні для проведення тестування. Наявність необхідного, сучасного та надійного обладнання забезпечує можливість якісного виконання всіх

запланованих видів тестування. Нестача обладнання, його моральна застарілість обмежує можливості та впливає на ефективність процесу.

Бюджет. Становить фінансові ресурси, які доступні у процесі тестування. Він включає усі витрати, починаючи з оплати навчання та роботи персоналу до оплати вартості придбання обладнання, або його оренду.

Також у якості тестових ресурсів нерідко розглядаються специфічні аспекти або категорії в рамках основних ресурсів. Розглянемо найпоширеніші з них.

Тестові випадки. До тестових випадків відносяться набори умов, дій та очікуваних результатів. Вони розробляються для перевірки конкретної функціональності програмного продукту або вимог до нього. Кожен тестовий випадок описують вхідними даними, кроками для виконання та результатом який очікується отримати після виконання цих кроків. Тестові випадки є необхідною базою процесу тестування і забезпечують систематичний та структурований підхід до перевірки програмного забезпечення. Тестові випадки розробляються персоналом (тестувальниками) і слугують планом для використання обладнання, як фізичного, так і віртуального. Розробка тестових випадків та їх виконання також включається до загального бюджету тестування.

Тестові дані. До тестових даних відносять будь-які вхідні дані, які необхідні для виконання тестових випадків. Тестові дані можуть включати як правильні, неправильні значення, граничні значення, значення в межах діапазону, випадкові дані тощо. Важливим є саме різноманітність та якість тестових даних, саме такі дані суттєво допомагають при виявленні дефектів. Якісно підібрані тестові дані дозволяють перевіряти, як саме програмний продукт працює у різних умовах, при різних сценаріях, у стандартних, нестандартних та виключних ситуаціях. Тестові дані, так само як і тестові випадки, є результатом інтелектуальної роботи команди тестувальників, також вимагають часу на розробку, займають ресурси обладнання для зберігання. Їх обробка також включається до загального бюджету тестування.

Тестові середовища. Сукупність апаратного та програмного забезпечення, на якому виконується тестування, які повинні максимально відповідати виробничому середовищу, щоб забезпечити реалістичні умови тестування. Правильно

налаштовані тестові середовища є критично важливими для виявлення проблем, пов'язаних з конфігурацією, сумісністю та інтеграцією. Невідповідність тестового середовища виробничому може призвести до пропуску дефектів, які проявляться лише у користувачів. З точки зору основних ресурсів Тестові середовища є комбінацією обладнання (сервери, мережі, ПЗ) та часу, необхідного для їх створення, налаштування та підтримки. Їх вартість також закладається в бюджеті тестування.

Інструменти та фреймворки. До цієї групи ресурсів відноситься все програмне забезпечення, яким користуються для автоматизації, керування та підтримки процесу тестування. Це можуть бути програми для управління та автоматизації тестування, інструменти тестування швидкодії та безпеки, утиліти контролю покриття коду, тощо. Використання такого роду інструментарію підвищує ефективність та продуктивність процесу тестування, скорочується час виконання тестових випадків, покращується якість звітності та автоматизуються прості та рутинні завдання тестування. Інструменти та фреймворки можна розглядати як ресурс обладнання або як результат роботи персоналу, у випадку, коли інструментарій розроблений власноруч. Вартість їх придбання або розробки включається до загального бюджету на тестування, а інтеграція та навчання персоналу як їх використовувати, включається до загального часу процесу тестування.

Логування. Процес автоматизованого збору та запису інформації, що до проходження певного процесу. Логування процесу тестування передбачає фіксацію інформації про виявлені дефекти: час події, стан системи у цей час, що саме виконувалось, які саме помилки з’являлися та як саме програмна система на них реагувала.

Метрики тестування. Кількісні показники, за допомогою яких проводять оцінювання успішності та ефективності проведення процесу тестування.

Логування та метрики тестування використовуються для прийняття обґрунтованих рішень, щодо подальшого проведення процесу тестування. Вони дозволяють оцінювати якість програмного забезпечення, що проходить процес

тестування. Їх можливо розглядати як результат роботи персоналу – тестувальники залучаються до визначення метрик, налаштування їх збору та аналізу. Для аналізу, інтерпретації та відображення результатів даних може використовуватись обладнання. Витрати на персонал та необхідний інструментарій для обробки результатів логування та метрик тестування також включаються до загального бюджету на тестування.

Управління дефектами. Систематичний процес виявлення та виправлення помилок. Включає їх фіксацію, відстеження, пріоритезацію та повторну перевірку після виправлення. Це дуже важлива частина процесу тестування. Ціль процесу управління дефектами – контроль процесу виправлення помилок, запобігання їх повторному виникненню та, як наслідок, забезпечення якості програмного забезпечення. Управління дефектами виконується персоналом та потребує ресурсу часу. Для управління дефектами також можуть використовуватися спеціальні системи, які вимагають ресурсів бюджету на придбання обладнання, ліцензій або підтримку. Витрати на персонал та необхідний інструментарій для управління дефектами включаються до загального бюджету на тестування.

Розподіл тестових ресурсів – це процес призначення обмежених ресурсів (людей, інструментів, часу, бюджету) для виконання тестових завдань. Процес тестування потребує значних ресурсів, особливо для складних програмних систем. Їх розумний розподіл дозволяє при тестувати ПЗ виявляти більше дефектів, зменшувати витрати, запобігти непотрібним перевіркам. Результатом розподілу тестових ресурсів є чітке визначення: які тести слід проводити, які інструменти для цього застосовувати, скільки часу на це витратити та яких фахівців до цього залучити. Ефективний розподіл ресурсів суттєво впливає на рівень забезпеченні якості тестування, мінімізації витрат та підвищенні ефективності роботи фахівців. Для цього застосовується ряд методів та підходів, які дають змогу краще використовувати доступні можливості та не перевантажувати ні обладнання, ні людей. Розглянемо більш детальніше ці методи.

Пріоритезація тестових випадків (тест-кейсів). Один з найважливіших способів ефективно використовувати обмежений час на тестування, передбачає не

просто виконання всіх доступних тестів, а вибір найбільш значущих та критичних, які забезпечать максимальне покриття функціоналу та виявлять найважливіші дефекти. До пріоритезації відносяться наступні стратегії.

1. Ризик-орієнтований підхід. В першу чергу проводиться тестування тих функцій, які мають найбільший потенційний вплив на систему у разі збою або мають високий рівень ризику виникнення проблем (критично важливі бізнес-процеси, функції безпеки, інтеграції зі сторонніми системами, тощо).

2. Метод Парето (80/20). Ґрунтується на припущені, що значна частина помилок (80%) зосереджена в невеликій частині (20%) функціоналу програмного продукту. Тому пріоритет під час тестування мають отримати саме ці проблемні ділянки. Такий підхід дозволяє за обмежений час виявити максимальну кількість дефектів.

3. Тестування на основі використання. В першу чергу проводять тестування функціоналу, який використовується користувачами найчастіше. Це гарантує, що саме ті сценарії, які є критичними та використовуваними будуть перевірятися в першу чергу. Як наслідок, саме вони й будуть працювати належним чином. Це дозволяє швидко виявити помилки та впливає на задоволеність користувачів.

4. Тестування на основі вимог. Тестові випадки пріоритезуються відповідно до важливості вимог – критично важливі вимоги тестуються в першу чергу.

5. Тестування на основі історії дефектів. Якщо є історія попередніх дефектів, більше уваги приділяється тестуванню модулів, де раніше виявлялося багато помилок.

Планування тестових циклів. Це такий метод планування тестового процесу, при якому виконується структуризація усього тестового процесу та розбиття його на послідовні етапи, або цикли. Кожен такий етап має чітку мету, обсяг тестів, вимоги до ресурсів та часові рамки. Це дозволяє виконувати процес тестування системно, контролювати прогрес та вчасно реагувати на проблеми. Основними етапами цього підходу є наступні:

1) розбиття тестування на ітерації – процес тестування поділяється на короткі часові періоди (ітерації, або спринти), у межах цих періодів команда виконує

певний обсяг тестових робіт та отримує регулярні проміжні результати, дозволяє ефективніше керувати часом;

2) визначення часових рамок – для кожного етапу тестування встановлюються конкретні терміни виконання, що допомагає контролювати час та зменшує ймовірність затримок;

3) визначення цілей тестового циклу – для кожного циклу чітко визначаються завдання (наприклад, тестування нового функціоналу, регресійне тестування, тестування продуктивності) та критерії успіху (наприклад, відсоток пройдених тестів, кількість знайдених критичних дефектів);

4) гнучке коригування – план тестування не є фіксованим, його можна змінювати в залежності від виявлених помилок, змін у вимогах або інших факторів, це дозволяє адаптуватися до реальної ситуації, ефективно використовувати час та перенаправляти ресурси на проблемні області.

Використання автоматизованих інструментів тестування. Автоматизовані інструменти тестування – це спеціалізоване програмне забезпечення, яке дозволяє створювати, виконувати та аналізувати результати тестових сценаріїв повністю автоматично, без участі людини, або з мінімальною її участю. Автоматизація значно зменшує обсяг ручної роботи та прискорює процес тестування. Наведемо нижче види тестування, що можуть бути автоматизовані за допомогою відповідних інструментів.

1. Автоматизоване регресійне тестування: автоматичний запуск набору тестів після кожної зміни коду, які перевіряють, чи не порушили нові зміни існуючий функціонал.

2. Автоматизація повторюваних тестів: автоматизація рутинних та повторюваних тестів (перевірка форм, базової функціональності, інтеграцій, тощо).

3. Тестування продуктивності: емуляція навантаження на систему, для моделювання поведінки великої кількості користувачів.

4. Автоматизація UI-тестування: автоматизація взаємодії з інтерфейсом користувача для тестування наскрізних сценаріїв.

5. Автоматизація API-тестування: автоматизація перевірки функціональності, надійності, продуктивності та безпеки інтерфейсів програмування додатків.

Використання ризик-орієнтованого тестування. Ризик-орієнтоване тестування – це методологія тестування, яка пріоритезує тестування функцій та модулів програмного забезпечення на основі оцінки пов'язаних з ними ризиків. Підхід фокусує зусилля тестування на найбільш ризикових аспектах продукту, що дозволяє оптимально використовувати час, концентруючись на потенційно проблемних областях. Підхід реалізується наступними етапами:

1) аналіз можливих сценаріїв відмови: визначаються потенційні проблеми, які можуть виникнути в роботі ПЗ, та їх можливі наслідки.

2) визначення зон з високою ймовірністю критичних помилок: на основі аналізу ризиків визначаються компоненти або функції, які мають високу ймовірність містити критичні дефекти (модулі, функції або інтеграції);

3) пріоритезація тестування: пріоритет надається найбільш ризикованим компонентам.

Контроль покриття тестування. Процес вимірювання того, яка частина програмного забезпечення була протестована тестовими випадками. Підхід допомагає оцінити, наскільки повно протестована система, і визначити області, які потребують додаткового тестування, що дозволяє оптимізувати зусилля та час. Контроль покриття реалізується наступними інструментами.

1) метрики покриття коду: використовуються для визначення відсотка коду, який був виконаний під час виконання тестів:

- покриття операторів – відсоток виконаних рядків коду;
- покриття гілок – відсоток виконаних гілок умовних операторів (if-else);
- покриття шляхів – відсоток виконаних можливих шляхів виконання коду.

2) аналіз покриття коду: дозволяє виявити області коду, які не були охоплені тестами, що вказує на потенційні ризики пропуску дефектів (чи весь необхідний код був перевірений);

3) трасування вимог (зв'язування тестових випадків з бізнес-вимогами): дозволяє перевірити, чи всі вимоги були протестовані (чи весь необхідний функціонал був перевірений).

Використання хмарних та віртуальних середовищ. Передбачає проведення тестування на інфраструктурі, яка надається хмарними провайдерами або створюється за допомогою технологій віртуалізації. Підхід дозволяє економити час та ресурси, надаючи гнучкі та масштабовані середовища для тестування. Перевагами використання хмарних та віртуальних середовищ можна вказати наступні:

– тестування на різних ОС та браузерах: хмарні сервіси надають доступ до великої кількості операційних систем та браузерів та надають можливість тестувати програмні додатки в різних конфігураціях без необхідності утримувати власну інфраструктуру;

– емуляції пристроїв: віртуальні машини та емулятори дозволяють тестувати програмні додатки на різних пристроях без фізичної наявності кожного з них;

– тестування на великій кількості конфігурацій без додаткових витрат: хмарні та віртуальні середовища дозволяють швидко створювати та конфігурувати необхідні тестові середовища та скорочувати час на їх підготовку та налаштування.

Використання крауд-тестування. Крауд-тестування – це метод тестування, при якому тестування програмного забезпечення передається великій групі незалежних тестувальників або кінцевих користувачів. Це дозволяє залучити велику кількість тестувальників з різними профілями та досвідом, що може допомогти виявити неочевидні дефекти та отримати цінні відгуки. Перевагами використання методу можна вказати наступні:

– можливість виявлення неочевидних дефектів, бо зовнішні тестувальники можуть знайти дефекти, які могли пропустити тестувальники, що вже добре знайомі з системою;

– отримання відгуків про зручність використання, стабільність та загальну якість продукту від реальних та потенційних користувачів;

– масове тестування на великій кількості пристроїв та конфігурацій, особливо метод ефективен при тестуванні мобільних додатків та веб-сервісів.

Інтеграція тестування у CI/CD (DevOps підхід). Інтеграція тестування в процес безперервної інтеграції CI та безперервної доставки CD – це практика автоматичного виконання тестів на кожному етапі процесу розробки програмного забезпечення, від коміту коду до розгортання. Такий підхід значно економить час та підвищує якість продукту. Основними механізмами, за допомогою яких тестування інтегрується в CI/CD є наступні:

– автоматичне тестування під час кожного коміту в репозиторій: при кожній зміні коду в системі контролю версій автоматично запускаються різні види тестів для швидкої перевірки;

– регресійне тестування після кожного оновлення: після кожного оновлення коду автоматично запускається набір регресійних тестів для перевірки, чи не з'явилися нові помилки;

– інтеграція з CI/CD інструментами: використання інструментів (наприклад, Jenkins, GitHub Actions, GitLab CI/CD) для автоматизації процесів збірки, тестування та розгортання.

Загальні тенденції впливу розглянутих підходів на використання ресурсів наведено у таблиці 1.1. У таблиці символом «+» позначено сприяння підходом оптимізації ресурсу, символом «-» не сприяння (потреба додаткової кількості ресурсу), відсутність позначки – відсутність впливу на оптимізацію певного ресурсу.

Таблиця 1.1

Загальні тенденції впливу підходів до оптимізації на тестові ресурси

Підхід/Ресурс	Час	Персонал	Обладнання	Бюджет	Тестові випадки	Тестові дані	Тестові середовища	Інструменти та фреймворки	Логування та метрики	Управління дефектами
Пріоритезація тестових випадків	+	+			+	+			+	
Планування тестових циклів	+	+			+	+	+		+	+
Використання автоматизованих інструментів	+	+	+	-	+	+	+	+	+	+
Використання ризик-орієнтованого тестування	+	+			+	+			+	
Контроль покриття тестування	+	+			+	+			+	
Використання хмарних та віртуальних середовищ	+		+	+			+	-		
Використання крауд-тестування	+	+	+	+	+	+	+	-	+	-
Інтеграція тестування у CI/CD	+	+	+	-	+	+	+	+	+	+

1.2 Огляд актуальних наукових робіт в оптимізації розподілу тестових ресурсів

Тестування програмного забезпечення є одним з найбільш ресурснозатратних етапів життєвого циклу розробки програмних продуктів. Саме він гарантує, що програмний продукт відповідає належному рівню надійності, безпеки та відповідає всім вимогам, що висуваються до нього. У [1,2] тестування розглядається з різних точок зору, включаючи його основні принципи, методології, етапи проведення та підходи до забезпечення ефективності тестових процесів. Процес верифікації програмного забезпечення, який виконується перед проведенням тестування, безпосередньо впливає на подальше тестування програмного забезпечення та може суттєво покращити його ефективність та зекономити ресурси на його проведення. У роботах [3,4] розглядаються сучасні питання процесу верифікації програмного забезпечення та його ефективного проведення для подальшого зменшення навантаження на тестувальників і

скорочення витрат на виправлення помилок при тестування. Обговорення існуючих, а також вдосконалених методів тестування з метою кращого забезпечення якості програмного продукту розглядаються авторами у роботі [5].

Штучний інтелект (AI) та машинне навчання (ML) відіграють все більшу роль у тестуванні програмного забезпечення, пропонуючи нові методи автоматизації, оптимізації та підвищення ефективності тестових процесів, його перспективи та виклики з усіх боків оглянуто авторами у [6]. У роботі [7] представлено методи тестування, що використовують AI, та їхню ефективність у різних сценаріях, а робота [8] розглядає методи оптимізації тестових процесів на основі AI, що демонструють потенціал для зменшення витрат і покращення продуктивності тестування. Робота [9] присвячена підвищенню точності та продуктивності тестування, [10] огляду методів встановлення пріоритетів вимог до програмного забезпечення, а [11] огляду застосування методів визначення пріоритетів тестових випадків.

Розподіл ресурсів тестування є ключовим аспектом забезпечення якості програмного забезпечення. Огляд існуючих досліджень та перспектив у цій галузі представлено в роботах [12,13], де автори аналізують основні тенденції та майбутні напрямки розвитку, крім того у статті [12] розглядається проблематика розподілу тестових ресурсів та систематизуються існуючі підходи та виявляються невирішені питання в даній області.

Архітектурно-орієнтований розподіл тестових ресурсів, що враховує структуру програмного забезпечення, досліджується в працях [14, 15]. Автори пропонують нові підходи та методи, що базуються на аналізі архітектури системи для оптимізації розподілу ресурсів. У роботі [16] автори пропонують модель оптимізації, що базується на архітектурі системи та критеріях надійності.

Проблема розподілу ресурсів для тестування різних версій програмного забезпечення є актуальною в умовах постійного розвитку та оновлення програмних продуктів. Підхід до вирішення цієї задачі представлено в статті [17].

Різні підходи до оптимального розподілу ресурсів тестування представлені в роботах [18-20]. У [18] пропонується інтеграція методів MEMV-OWA та

DEMATEL для ефективного управління тестовими ресурсами. Дослідження [19] розглядає розподіл ресурсів у хмарних середовищах, що є актуальним у контексті сучасних розподілених систем. Робота [20] надає фундаментальний огляд проблем розподілу ресурсів, включаючи комбінаторні підходи та оптимізаційні моделі.

Оптимальні стратегії розподілу ресурсів тестування модульного програмного забезпечення розглядаються в роботах [21-25]. У [21] авторами аналізується вплив розподілу ресурсів на чутливість результатів тестування, [22] на чутливість результатів модульного тестування. Дослідження [23] враховує фактори витрат, надійності та зусиль, необхідних для тестування. У роботі [24] представлено підхід, що використовує операційний профіль для оптимального розподілу ресурсів. Робота [25] присвячена застосуванню методів динамічного програмування для управління ресурсами в модульних системах.

Оптимізація моделей розподілу тестових ресурсів представлена в працях [27-29]. У роботі [27] автори описують підхід, що базується на Інтервальних інтуїціоністичних нечітких множинах та аналітичному ієрархічному процесі AHP, який використовується для прийняття рішень у складних багатокритеріальних ситуаціях з високим рівнем невизначеності, а у роботі [28] розроблено стохастичну оптимізаційну процедуру для оцінки достатності тестування залежно від складності проекту. Дослідження оптимізаційних методів зниження ризиків у великих програмних системах через ефективне управління ресурсами тестування містить робота [29].

Багатоцільова оптимізація розподілу тестових ресурсів є важливим напрямком досліджень, що спрямований на врахування множинних факторів при розподілі ресурсів тестування, таких як надійність, витрати та ефективність тестових процесів. У роботах [30, 31] розглядаються питання багатокритеріальної оптимізації та управління обмеженнями у процесі розподілу ресурсів. Робота [30] присвячена оцінці простору допустимих рішень у задачах багатоцільової оптимізації розподілу тестових ресурсів, що є важливим для розуміння складності задачі та вибору ефективних методів її розв'язання.

У дослідженні [26] автор застосовує та оцінює набір підходів та інструментів для оптимізації виконання тестів інтеграційного тестування. Методи оптимізації, що базуються на ентропійних та ройових алгоритмах, розглядаються в дослідженнях [32, 33]. Ці роботи пропонують вдосконалені алгоритми розподілу тестових ресурсів, що враховують множинні цілі та застосовують підходи багатокритеріального прийняття рішень.

Зокрема, у [33] досліджується застосування багатокритеріального аналізу в задачах тестування програмного забезпечення. У роботі [34] розглядаються комбіновані стратегії розподілу ресурсів, що використовують методи аналітичного ієрархічного процесу для визначення оптимального розподілу ресурсів тестування.

Методи, засновані на генетичних алгоритмах та багатокритеріальній оптимізації широко використовується для оптимізації процесів тестування. У дослідженнях [35] подано огляд та аналіз алгоритму NSGA-II, де він застосовується для виявлення локальних проблем у програмному забезпеченні шляхом багатокритеріальної оптимізації, а автори [36] пропонують управління обмеженнями в алгоритмі, що дозволяє знаходити більш ефективні рішення для оптимального розподілу тестових ресурсів.

Питання ефективної обробки обмежень у багатоцільовій оптимізації розподілу тестових ресурсів, де одним із ключових обмежень є забезпечення необхідного рівня надійності, розглядаються у статті [37]. Автори пропонують удосконалені методи для роботи з такими обмеженнями в еволюційних алгоритмах. У контексті розподілу ресурсів тестування важливим аспектом є моделювання надійності програмного забезпечення.

У роботі [38] автори досліджують моделі зростання надійності програмного забезпечення на основі зусиль тестування, зокрема використовуючи модель Exponentiated-Gompertz, а також розглядають визначення часу випуску програмного продукту. Багатоцільова оптимізація розподілу тестових ресурсів, особливо з урахуванням архітектури програмного забезпечення та критеріїв надійності, є предметом дослідження в [39]. Автори пропонують нові межі, керовані надійністю, для покращення цього процесу.

2 АНАЛІЗ ПІДХОДІВ ДО ОПТИМІЗАЦІЇ РОЗПОДІЛЕННЯ ТЕСТОВИХ РЕСУРСІВ

2.1 Математичні моделі, методи та алгоритми оптимізації розподілення ресурсів

Оптимальний розподіл тестових ресурсів дозволяє підвищити ефективність тестування, зменшити ризики та покращити якість кінцевого програмного продукту.

Задача розподілу тестових ресурсів (TRAP) – це комплексна задача, яка полягає у визначенні оптимального способу розподілу обмежених тестових ресурсів між різними тестовими завданнями таким чином, щоб досягти певних цілей тестування. Наприклад, цілями розподілу можуть бути:

– виявити максимальну кількість дефектів;

– мінімізувати вартість тестування;

– оптимізувати покриття тестами.

Ключовими аспектами задачі є наступні:

– обмеження – наявні ресурси час, персонал, обладнання та бюджет є обмеженими, що повинно бути враховане при їх розподілу;

– цілі – чітко визначені цілі тестування, як то максимізація виявлення дефектів (особливо критичних), мінімізація ризиків, забезпечення певного рівня покриття тестами, мінімізація витрат на тестування або досягнення балансу між цими цілями;

– завдання – існує безліч потенційних тестових завдань, кожне з яких вимагає певних ресурсів та має різну ймовірність виявлення дефектів;

– оптимізація – необхідно знайти оптимальний розподіл ресурсів, який найкращим чином досягає визначених цілей в умовах визначених обмежень.

Оптимізація процесу розподілення ресурсів для тестування програмного забезпечення може мати різні підходи залежно від поставлених цілей. На рисунку

2.1 представлено класифікацію моделей оптимізації розподілення ресурсів за трьома основними критеріями:

– кількість оптимізаційних цілей – чи оптимізується один параметр (наприклад, лише вартість або лише надійність), чи кілька одночасно;
– тип цільової функції – що саме є об'єктом оптимізації: надійність, витрати часу та зусиль на тестування, або загальна вартість тестування;
– урахування архітектури ПЗ – чи враховується структура системи, частота використання модулів або їх взаємозв’язки.

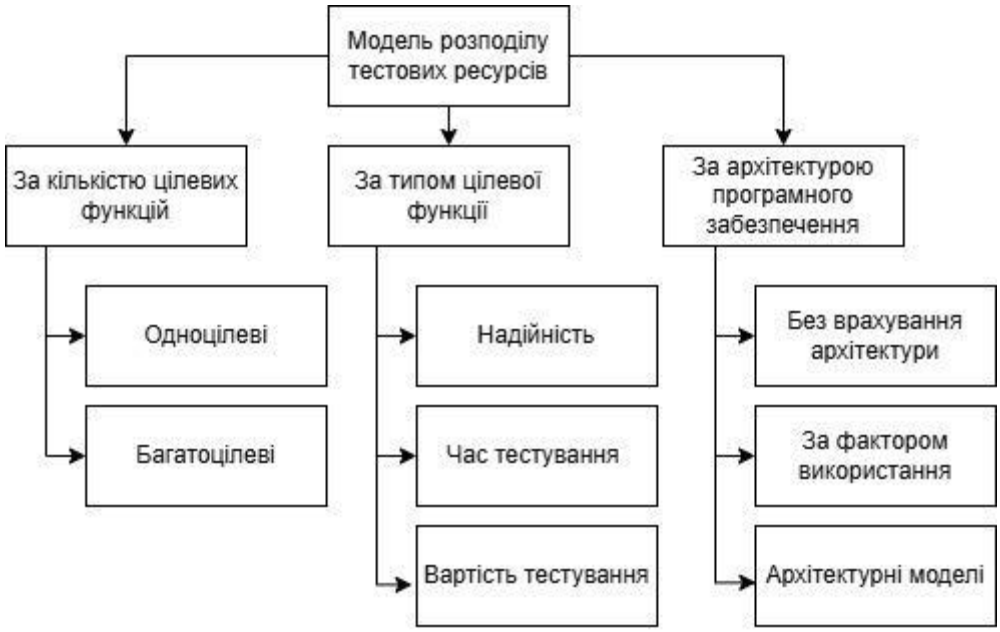


Рис. 2.1 Класифікація моделей оптимізації розподілення ресурсів

Загальними цілями при розгляді задачі розподілу тестових ресурсів вважають: ефективність тестування (надійність, покриття тестами, кількість знайдених дефектів, тощо), вартість тестування та час (зусилля) витрачені на тестування. Вони або протиставляються один одному з метою пошуку відповідних компромісів, або один із них встановлюється як мета, обмежена іншими.

При одноцільовій оптимізації одна з цілей обирається головною, саме вона встановлюється як мета оптимізації, що обмежується іншими. Багатоцільова версія оптимізації розглядає дві або більше цілей разом. Цільові функції є

взаємозалежними. Наприклад, ефективність тестування модуля залежить від часу, спрямованого на його тестування, вартість залежить від часу тестування, але також може враховувати вартість налагодження під час тестування або під час роботи. Таким чином, моделі змінюються залежно від прийнятої цільової функції. Крім того, існують різні способи вираження залежностей між архітектурними модулями. Вони визначають, як цільовий атрибут на рівні системи (наприклад, надійність) обчислюється з того самого атрибута на рівні модуля. Одноцільові моделі розподілу тестових ресурсів виглядають наступним чином.

Модель оптимізації ефективності тестування з обмеженнями на ресурси часу та бюджету (2.1), модель оптимізації часу тестування з обмеженнями на ефективність та бюджет (2.2) та модель оптимізації вартості тестування з обмеженнями на ефективність та час (2.3):

$$max\big(R_s(T_1, T_2, \dots T_n)\big) \text{ за умовою } T \leq T_{max} \text{ та/або } \mathrm{C} \leq \mathrm{C}_{max}, \quad (2.1)$$

$$min(T) \text{ за умовою } R_s(T_1, T_2, \dots T_n) \geq R_{min} \text{ та/або } \mathrm{C} \leq \mathrm{C}_{max}, \quad (2.2)$$

$$min(\mathrm{C}) \text{ за умовою } R_s(T_1, T_2, \dots T_n) \geq R_{min} \text{ та/або } T \leq T_{max}, \quad (2.3)$$

де: R_s – функція ефективності тестування, що залежить від параметрів $T_1, T_2, \dots T_n$;

T – загальний час тестування, $T = \sum_{i=1}^{n} T_i$;

C – загальна вартість тестування, $\mathrm{C} = \sum_{i=1}^{n} C_i$;

T_i – час тестування і-ї компоненти;

C_i – вартість тестування і-ї компоненти;

T_{max} – максимальний час тестування;

C_{max} – максимальна вартість тестування;

R_{min} – мінімальний рівень необхідної ефективності тестування;

n – кількість компонент системи, що тестуються.

Багатоцільові моделі частіш за все намагаються максимізувати ефективність тестування та використовують для цього модель зростання надійності програмного

забезпечення (SRGM), або модель зростання кількості виявлених несправностей. Оптимізація надійності – це підвищення ймовірності безвідмовної роботи ПЗ при мінімальних витратах на тестування та обслуговування. Модель фіксує зв'язок між зростанням надійності (з точки зору процесу виявлення та/або виправлення помилок) і часом (або зусиллями) тестування, витраченими на кожен модуль. SRGM – це широкий клас моделей, що використовують час між відмовами, які спостерігаються під час тестування та налагодження, щоб передбачити наступний час до відмови. Загальна форма SRGM моделі виглядає наступним чином (2.4):

$$m(t) = N\left(1 - e^{-D(t)}\right), \tag{2.4}$$

де: $m(t)$ – кількість виявлених дефектів на момент часу t;

N – загальна (максимальна) кількість дефектів у системі;

$e^{-D(t)}$ – ймовірність того, що дефект не буде виявлений до моменту t;

$D(t)$ – кумулятивна функція виявлення несправностей, яка визначає як змінюється інтенсивність виявлення помилок у часі.

Функція $D(t)$ моделює процес виявлення несправностей (FDP) за допомогою частоти виявлення несправностей на залишкову несправність (або інтенсивність відмови). Форма $D(t)$ визначає конкретну модель зростання надійності.

Розглянемо найчастіше використовувані моделі FDP та випадки їх використання.

Експоненційна модель. Має форму опису $D(t) = \lambda t$ і показує, що дефекти виявляються швидко на початку тестування, потім швидкість їх виявлення знижується. Ця модель підходить для випадків, коли помилки виявляються випадковим чином і незалежно. Модель не враховує навчання тестувальників (покращення їх навичок), або їх адаптацію – тестувальники знаходять помилки з однаковою ймовірністю в будь-який момент часу).

S-подібна модель має форму $D(t) = \frac{at}{b+t}$, яка показує, що спочатку дефекти знаходять повільно, потім швидкість збільшується, а потім знову сповільнюється.

Модель підходить для процесів, де тестувальники з часом стають досвідченішими та використовується, коли виправлення програмного коду підвищує ефективність пошуку нових дефектів. Параметр a контролює темп зростання кількості виявлених дефектів: збільшення значення a призводить до швидшого виявлення дефектів на початкових етапах тестування та визначає крутість S-подібної кривої на початку (тобто, як швидко дефекти будуть знаходитися в першій половині тестування). Параметр b визначає час затримки перед активним зростанням кількості виявлених дефектів: коли тестувальники почнуть знаходити дефекти з більшою швидкістю після початкової фази повільного пошуку та контролює, як швидко тестувальники набирають досвід і починають знаходити помилки з більшою ефективністю. Таким чином, параметри контролюють коли і як тестувальники набираються досвіду.

Логарифмічна модель з формою $D(t) = 1 + \lambda t$ означає, що кількість знайдених дефектів швидко зростає на початку, а потім дуже повільно, Модель ураховує, що з часом залишається все менше нерозкритих помилок, і їх важче знайти та підходить для тестування, коли спочатку знаходяться прості дефекти, а складні виявляються лише пізніше.

Лог-логістична модель має форму $D(t) = \frac{at^b}{1+at^b}$ і є більш гнучкою моделлю S-подібної моделі, що дозволяє моделювати різні сценарії темпів знаходження помилок. Параметр a контролює швидкість зростання кількості виявлених дефектів, збільшення його значення призводить до швидшого початку процесу виявлення дефектів, особливо на початкових етапах тестування та визначає, наскільки швидко відбувається початкове виявлення дефектів до того, як темп виявлення зменшиться. Параметр b визначає плавність кривої виявлення дефектів, його значення впливає на те, як швидко виявлення дефектів починає сповільнюватися після того, як тестувальники набираються досвіду. Чим більший параметр b, тим повільніше зменшується темп виявлення дефектів на пізніх етапах тестування. Таким чином, параметри пов'язані з гнучким контролем швидкості виявлення дефектів, з можливістю моделювати різні сценарії для процесів виявлення дефектів, які змінюються з часом.

Модель Вейбулла. Має форму $D(t) = \lambda t^{\beta}$ та описує процеси виявлення дефектів, де швидкість їх виявлення залежить від часу. Параметр β визначає характер зміни швидкості виявлення дефектів:

– β=1 – модель переходить до експоненційної моделі (стабільний темп виявлення дефектів);

– β>1 – дефекти знаходяться швидше в міру тестування;

– β<1 – дефекти знаходяться повільніше з часом.

Модель використовується для процесів тестування, коли залежно від характеру тестування швидкість виявлення дефектів може прискорюватися або сповільнюватися.

Модель Капура-Гергера. Має форму $D(t) = \frac{1}{1+e^{-(at-b)}}$ та описує такі процеси виявлення дефектів, коли спочатку дефекти знаходяться повільно, потім виявлення пришвидшується до певного моменту, а вже після нього темп знову зменшується. Ця модель дозволяє моделювати процеси виявлення дефектів, коли є фаза навчання, по закінченню якої тестувальники починають працюють більш ефективно.

Модель NHPP. Має форму $D(t) = \int_0^t \lambda(s)ds$. Вона використовує непостійну інтенсивність λ(t), що може змінюватися з часом. Це дозволяє моделювати такі процеси виявлення дефектів, коли ймовірність виявлення дефектів не фіксована, а змінюється та залежить від часу або етапу тестування.

Модель Джелинського-Моранди описується формою $D(t) = \frac{N-m(t)}{N}$. Модель базується на припущенні, що кожен дефект, який знаходять, зменшує кількість можливих залишкових дефектів. Це дозволяє моделювати процеси, де виявлення дефектів з часом стає більш ефективним, оскільки залишкові дефекти стають все важче виявити.

Модель Литтлауда-Веркса. Має форму $D(t) = \lambda t^{\alpha} e^{-\lambda t}$. Модель гнучка та дозволяє моделювати процеси, де ймовірність виявлення дефектів змінюється як від часу, так і від певної форми функції. Вона підходить для процесів з непостійною швидкістю виявлення дефектів.

Загально принципи використання розглянутих моделей можна визначити наступним чином:

– моделі Вейбулла та NHPP краще підходять для опису випадкових процесів, коли залежно від часу змінюється ймовірність виявлення дефектів;

– моделі S-подібна та Kapur-Garg більш детально описують такі процеси, коли швидкість виявлення дефектів змінюється на різних етапах тестування в залежності від досвіду тестувальників і складності дефектів;

– модель Джелінського-Моранди використовує більш детальну дискретизацію і є корисною для імітації циклічних процесів виявлення дефектів;

– моделі Джелінського-Моранди та Литтлауда-Веркса використовуються як для циклічних процесів виявлення дефектів, так і для процесів, коли з часом, через накопичення досвіду, змінюється і ефективність виявлення дефектів;

– моделі Log-Logistic, Logarithmic і Exponential описують різні сценарії зміни темпів виявлення дефектів: від пришвидшеного на початку тестування до сповільненого, в залежності від складності дефектів і досвіду тестувальників.

Ціль оптимізації мінімізації часу або зусиль є менш поширеною у моделях. Така оптимізація частіше виступає як обмеження у багатоцільових моделях. Це обумовлено обмеженістю доступного бюджету (часу, або зусиль), яку неможливо подолати. Оптимізація часу тестування полягає у пошуку балансу між часом, що витрачено на тестування, і якістю яку отримали. Математична модель з цільовою функцією зусиль тестування (TEFs) описує, як змінюється витрачений ресурс (час, людські ресурси, обчислювальні потужності) на тестування програмного забезпечення в часі. Вона дозволяє оцінити залежність між зусиллями, які витрачено на тестування, та ефективністю виявлення дефектів. Вимірюються зусилля тестування у людино-годинах, обчислювальних ресурсах або у вартості витрат. При тестуванні програмних продуктів функції зусиль тестування застосовують у наступних цілях:

– оцінка ефективності тестування –чи достатньо ресурсів витрачається на тестування;

– планування тестування –коли варто збільшити або зменшити витрати на тестування;

– прогнозування залишкових дефектів –чи варто продовжувати тестування або вже досягнуто необхідного рівня якості.

Розглянемо найчастіше використовувані моделі TEFs та випадки їх використання.

Експоненційна модель зусиль тестування використовується для тих процесів, коли зусилля різко падають після початкового етапу активного тестування (2.5):

$$E(t) = E_0\, e^{-\beta t}, \tag{2.5}$$

де: E_0 – початковий рівень зусиль;

β – коефіцієнт, що визначає, як швидко зусилля тестування зменшуються.

t – час, момент часу, для якого оцінюються витрати зусиль на тестування.

Вейбуллівська модель зусиль тестування дозволяє описати складніші зміни зусиль у часі. Вона підходить для процесів тестування, де зусилля можуть спочатку зростати, а потім спадати (2.6):

$$E(t) = \alpha t^{\beta} e^{-\lambda t}, \tag{2.6}$$

де: α – масштабний параметр, що визначає загальний рівень зусиль тестування: чим більше α, тим вищі витрати на тестування;

β – форма кривої, яка керує тим, як зусилля змінюються в часі: при β<1 зусилля зменшуються з часом, при β>1 зусилля спочатку зростають, а потім зменшуються;

λ – швидкість затухання, визначає, наскільки швидко зусилля тестування зменшуються з часом: високе значення λ показує, що тестування швидко згасає після початкової активності, низьке – поступове зниження зусиль тестування;

t – час, момент часу, для якого оцінюються витрати зусиль на тестування.

S-подібна модель (логістична функція зусиль) використовується у випадках, коли тестувальники набувають досвіду. Тоді ефективність тестування змінюється в S-подібній формі (2.7):

$$E(t) = \frac{E_{max}}{1+e^{-\gamma(t-t_0)}}, \tag{2.7}$$

де: E_{max} – максимальні зусилля тестування;

γ – параметр швидкості зміни зусиль;

t_0 – момент часу, коли зусилля досягають середнього рівня;

t – час, момент часу, для якого оцінюються витрати зусиль на тестування.

Логістична модель зусиль тестування описує нерівномірне витрачання зусиль під час процесу тестування. Це відповідає реальному процесу тестування, коли спочатку воно йде повільно, потім стає інтенсивним, а далі поступово знижується (2.8):

$$E(t) = \frac{E_{max}}{1+Ae^{-\alpha ht}}, \tag{2.8}$$

де: E_{max} – максимальні зусилля тестування;

A – константа, яка регулює інтенсивність тестування на початку: чим більше значення, тим менше зусиль на початку;

α – швидкість витрачання зусиль на тестування;

h – індекс структурування процесу, визначає, наскільки добре організований процес розробки.

Похідна від $E(t)$ показує швидкість витрачання зусиль у кожний момент часу:

$$e(t) = \frac{E(t)}{dt} \tag{2.9}$$

Остання ціль оптимізації мінімізувати вартість тестування. Оптимізація вартості тестування уявляє собою пошук компромісу між часом тестування,

витратами та рівнем якості. У цьому випадку, вартість – це міра, яка хоч і пов'язана з витраченими зусиллями, але виходить за її межі. Базовою моделлю функції тестування є модель (2.10):

$$C(t) = C_T(t) + C_F(t) + C_R(t), \tag{2.10}$$

де: $C_T(t)$ – витрати на тестування, що включають оплату тестувальників, використання тестових середовищ, автоматизацію тощо;

$C_F(t)$ – витрати, пов’язані з невиявленими дефектами (вартість виправлення дефектів після випуску, втрати через незадоволення користувачів);

$C_R(t)$ – витрати на виправлення виявлених дефектів у процесі тестування.

Експоненційна модель припускає зростання витрат на виправлення дефектів залежно від часу їх виявлення за експонентою (2.11):

$$C_F(t) = C_0 e^{\lambda(T-t)}, \tag{2.11}$$

де: C_0 – базові витрати на виправлення одного дефекту;

T – загальний час тестування;

λ – параметр зростання витрат у часі.

При використанні функції накопичення виявлених дефектів m(t), загальні витрати тестування можна представити наступним чином (2.12):

$$C_T(t) = c_T m(t), C_R(t) = c_R m(t) \tag{2.12}$$

де: c_T – середні витрати на пошук одного дефекту,

c_R – середні витрати на виправлення одного дефекту.

Наступна модель також вважається загальноприйнятою моделлю оцінки витрат на тестування, що враховує виявлені та невиявлені дефекти, а також загальні зусилля тестування (2.13):

$$C(t) = C_1\left(\frac{\delta}{24}\right)m_c(t) + C_2\left(\frac{\delta}{24}\right)(m_d(\infty) - m_c(t)) + C_3\left(\frac{Y}{24}\right) \qquad (2.13)$$

де: C_1– вартість виявлення дефектів під час тестування;

C_2 – вартість виправлення дефектів, які залишилися після тестування;

C_3 – загальні витрати на тестування, пов’язані з ресурсами;

δ – кількість годин тестування на день;

Y – загальний час тестування в годинах;

$m_c(t)$ – кількість виявлених дефектів до моменту часу t;

$m_d(\infty)$ – загальна кількість дефектів у системі (невідома наперед).

Розглянемо основні типи архітектурних моделей, які використовуються для оптимального розподілу тестових ресурсів між модулями. Ці моделі враховують структуру модулей, їх важливість та частоту використання.

Модель вагового коефіцієнта використання UFM враховує ймовірність використання кожного модуля. Кожному модулю системи призначається коефіцієнт від 0 до 1, який визначає, як часто цей модуль використовується, а модель враховує ймовірність використання модуля у реальній експлуатації. Наприклад, модуль автентифікації та модуль зміни пароля: перший використовується при кожному вході в систему, а другий набагато рідше, тому модулю автентифікації необхідно виділити більше тестових ресурсів, ніж модулю зміни паролю.

Моделі структурної архітектури: моделі Series-Parallel, Bridged та Star. Такі моделі враховують структурні зв’язки між модулями (послідовні, паралельні чи змішані) та забезпечують надійність резервуванням компонентів. Наприклад, у критичних системах можуть бути дубльовані модулі (два незалежні сервери для обробки даних у реальному часі). Ці моделі використовують задачу розподілу

резервування RAP, яка вважається класичною задачею оптимізації. Вона використовується для підвищення надійності складних систем (підвищення стійкості до відмов) за рахунок дублювання (резервування) компонентів.

Архітектурні моделі Маркова. Ці моделі використовують Марковські ланцюги для моделювання переходів між модулями, де кожен модуль розглядається як стан, а взаємодія між ними – як ймовірність переходу між станами. У моделі використовується метрика visit count – середня кількість викликів модуля. Такі моделі найкраще підходять для динамічних систем із випадковими процесами. Прикладом може слугувати веб-сервіс онлайн-магазину, що має маршрутизацію між модулями, а користувач зі сторінки товару може перейти до каталогу або до кошика.

Моделі толерантності до збоїв FTM враховують, що деякі збої можуть бути виправлені або обійдені без критичних наслідків. У моделях використовуються підходи резервування дублюванням модулів, механізми відновлення як перезапуск, повторна спроба, або перемикання на інший модуль та коефіцієнт покриття відмов – параметр значенням з діапазону [0-1] показує наскільки добре модуль справляється з відмовами. Прикладом служить автоматична активація резервного серверу при збої в основному.

Серед розглянутих моделей найбільш повними вважаються архітектурні моделі Маркова та моделі толерантності до збоїв, оскільки вони враховують не тільки зв’язки між модулями, а й їхню поведінку в умовах відмов.

Оскільки моделі задачі розподілу ресурсів тестування є нелінійними оптимізаційними задачами для їх розв'язання використовують наступні основні підходи.

Точні методи. Використовуються в простих однокритеріальних (одноцільових) задачах та погано масштабуються для складних систем. Найпопулярнішим точним методом для таких оптимізаційних задач є метод Лагранжевих множників. Крім того успішно використовується динамічне програмування та такі методи нелінійного програмування, як узагальнений метод

зменшеного градієнта GRG, послідовне квадратичне програмування SQP, метод проекції градієнта GP, або послідовне лінійне програмування SLP.

Метаевристичні методи. Використовуються коли задача розподілу ресурсів вирішується для складної архітектури або містить багатокритеріальну оптимізацію (багатоцільові моделі). Найпоширенішими алгоритмами метаевристик для одноцільових моделей можна вказати наступні:

1) генетичний алгоритм GA – самий популярний для даного типу моделей;
2) генетичний алгоритм з локальним пошуком GLSA – поєднання генетичного алгоритму та локальний пошуку;
3) метод підйому на пагорб HC – покроковий локальний пошук з поліпшенням рішення;
4) метод імітації відпалу SA – метод заснований на процесі кристалізації металів.

В багатокритеріальних задачах (багатоцільові моделі) використовують наступні алгоритми:

1) генетичний алгоритм недомінованого сортування NSGA-II – є найпопулярнішим для даного типу моделей;
2) алгоритм гармонійної відстані HaD;
3) алгоритм диференційної еволюції DE з наступними модифікаціями:
   - багатоцільова диференційна еволюція MODE;
   - зважена нормалізована сума WNS-MODE;
   - гібридна клітинна диференційна еволюція Cell-DE;
4) алгоритм рою частинок PSO;
5) архівована еволюційна стратегія Парето PAES;
6) багатоцільовий еволюційний алгоритм за декомпозицією MOEA/D;
7) ранговий генетичний алгоритм RWGA;
8) покращений алгоритм еволюційного Парето-архіву 2 SPEA2;
9) індикаторний еволюційний алгоритм IBEA;
10) багатоцільовий клітинний еволюційний алгоритм та інші MOCell.

Евристичні, або користувацькі, методи. Такі методи та алгоритми розробляються під конкретну задачу (під користувача). Наступні алгоритми є найбільш популярними при пошуку оптимального розподілу ресурсів:

1) спеціалізовані методи розподілу ресурсів на основі метрик складності;

2) risk-based підхід із матрицею ризиків – метод розподілу ресурсів на основі ризиків із використанням матриці ризиків;

3) fuzzy logic – метод нечіткої логіки для розподілу тест-кейсів на основі операційного профілю та інші.

Машинне навчання та штучний інтелект. До таких алгоритмів відносяться інтелектуальні алгоритми, що будують прогноз для тестування: які тести слід запускати для досягнення максимальної ефективності:

1) кластеризація тест-кейсів – групування схожих тестів та уникнення зайвого дублювання, використовує алгоритми k-means, DBSCAN та подібні;

2) нейронні мережі – передбачають найбільш дефектні частини коду в результаті навчання на історичних даних;

3) підкріплювальне навчання – алгоритм навчається, як оптимально розподіляти ресурси для тестування в реальному часі.

Результати порівняння методів наведено у таблиці 2.1.

Таблиця 2.1

Порівняння методів розв’язання задачі оптимізації розподілу тестових ресурсів

Метод	**Задачі**	**Приклад алгоритмів**	**Переваги**	**Недоліки**
Точні методи	SO	Lagrange multipliers, DP, GRG, SQP	Висока точність	Погано масштабуються
Метаевристики	SO + MO	GA, NSGA-II, MODE, PSO	Гнучкість, масштабованість	Не гарантують точне рішення
Евристичні методи	SO + MO	Risk-based, Fuzzy Logic	Адаптивність	Вимагають налаштування
Машинне навчання та AI	SO + MO	Random Forest, SVM, XGBoost, Reinforcement Learning	Здатність виявляти приховані закономірності, навчання на даних	Потребує великого обсягу якісних даних, складні у інтерпретації

2.2 Сучасні засоби програмної інженерії для оптимізації розподілу тестових ресурсів

Оптимізація розподілу тестових ресурсів є важливою задачею для забезпечення ефективності та якості процесу розробки програмного забезпечення. Сучасні програмні засоби здатні автоматизувати та підвищити ефективність цього процесу. Вони надають інструменти для планування, виконання та аналізу процесу тестування.

Ці засоби можна розділити умовно на кілька категорій залежно від їх основної функціональності:

1) системи управління тестуванням (Test Management Systems, Test Management Tools);
2) інструменти автоматизації тестування (Test Automation Tools);
3) інструменти управління тестовим середовищем (Test Environment Management);
4) інструменти аналізу покриття коду (Code Coverage Tools);
5) інструменти безперервної інтеграції та безперервної доставки (CI/CD);
6) інструменти для управління вимогами (Requirements Management Tools).

Розглянемо існуючі засоби програмної інженерії з точки зору можливості їх використання для певних видів ресурсів та методів їх розподілу.

Системи управління тестуванням – це програмні засоби, які призначені для планування, відстеження та керування тестуванням. Вони допомагають організувати процес тестування, включаючи створення тестових випадків, управління багами, визначення статусу тестів, звітність та документацію. Найпоширенішими системами управління тестуванням є наступні (за типом розгортання) [40,41]:

– хмарні: TestRail, qTest, PractiTest, Testmo, TestMonitor, AIO Tests, Jira Cloud, QA Sphere Cloud, QADeputy;

– серверні: Quality Center, TestLink, SpiraTest;

– інтегровані (плагіни до інших систем): Zephyr (версії Squad та Scale) (для Jira), Xray (для Jira).

Інструменти автоматизованого тестування – це програмні засоби, які призначені для автоматичного виконання тестів. Вони дозволяють запускати тестові сценарії без ручного втручання, автоматизуючи сам процес тестування, та допомагають заощадити час при повторних тестах. Найпоширенішими інструменти автоматизованого тестування можна вказати наступні (за типом тестування) [42,43]:

– веб-тестування: Selenium, Playwright, Cypress, Puppeteer, Taiko, Sahi Pro, HeadSpin (включає можливості веб-тестування);

– мобільне тестування: Appium, Espresso (Android), XCUITest (iOS), HeadSpin (включає можливості мобільного тестування на реальних пристроях);

– API-тестування: JMeter, Postman (використовується для ручного дослідження API), RestAssured (бібліотека для Java);

– GUI (графічний інтерфейс користувача) тестування: QTP (UFT), TestComplete, Ranorex Studio, Tricentis Tosca;

– приймальне тестування BDD: Cucumber, Gauge;

– юніт-тестування (програмні бібліотеки/фреймворки): JUnit (Java), TestNG (Java) .

Інструменти управління тестовим середовищем – це програмні засоби, які призначені для створення, налаштування, підтримки та контроля тестових середовищ, які необхідні для ефективного тестування програмного забезпечення. Вони дозволяють централізувати управління різними аспектами тестових середовищ, забезпечуючи їхню стабільність, доступність та відповідність вимогам тестування. Найпоширеніші інструменти автоматизованого тестування можна вказати наступні (за типом тестування) [44]:

– спеціалізовані платформи: Plutora, Enov8, Apwide Golive (для Jira), ServiceNow Test Management, Quali CloudShell;

– інструменти для автоматизації провізії та конфігурації: Terraform, Ansible, Chef, Puppet;

– платформи контейнеризації та оркестрації: Docker, Kubernetes;

– хмарні сервіси: AWS Cloud Services, Azure Cloud Services, GCP Cloud Services;

– системи управління тестуванням з базовими можливостями управління тестовим середовищем: TestLink, Jira (з використанням плагінів).

Інструменти аналізу покриття коду. Ці програмні засоби призначені для визначення які частини коду були покриті (виконані) під час тестування, а які залишилися непокритими (невиконаними). Вони дозволяють виявляти непротестовані ділянки коду та, так званий, «мертвий» (код, який ніколи не виконується), що може бути ознакою застарілого або непотрібного коду. Також інструменти дозволяють оцінювати ефективність тестів, наскільки добре набір тестів охоплює функціональність програмного забезпечення: низьке покриття коду може бути ознакою недостатньою кількістю або якістю тестів у тестовому наборі. Найпоширеніші інструменти аналізу покриття коду наступні (за типом ліцензування) [45,46]:

– open-source: JaCoCo, Cobertura (Java), Coverlet (C#/.NET), Coverage.py (Python), Istanbul (JavaScript), SimpleCov (Ruby), go cover (Go), gcov(C/C++), llvm-cov (C/C++) , LCOV (C/C++), SonarQube (загального призначення);

– комерційні: Clover (Java), dotCover (C#/.NET), Visual Studio Code Coverage (C#/.NET), BullseyeCoverage (C/C++).

Інструменти безперервної інтеграції та безперервної доставки. Ці програмні засоби призначені для автоматизації етапів розробки, тестування та розгортання програмного забезпечення. Вони дозволяють зробити процес випуску нових версій програмного забезпечення частішим, надійнішим та менш схильним до помилок. Інструменти складові елементи:

1) безперервна інтеграція (Continuous Integration) – практика розробки, яка передбачає часте об'єднання коду від різних розробників в основну гілку репозиторію, яке супроводжується збіркою, тестуванням та перевіркою якості коду, що дозволяє виявляти та виправляти конфлікти інтеграції та помилки на ранніх етапах розробки;

2) безперервна доставка (Continuous Delivery) – продовження CI, яке передбачає автоматизований процес доставки коду до спеціального середовища pre-production, де може бути проведено додаткове тестування;

3) безперервне розгортання (Continuous Deployment) – подальший розвиток Continuous Delivery, який передбачає автоматизований процес розгортання програмного забезпечення в середовищі production.

Найпоширеніші інструменти CI/CD (за типом ліцензування) [47,48]:

– open-source: Jenkins, GitLab CI/CD, GitHub Actions, Travis CI, Argo CD, Spinnaker, GoCD, OpenShift Pipelines, Docker;

– комерційні: Spacelift, CircleCI, Bamboo, TeamCity, Azure DevOps, AWS CodePipeline, Bitbucket Pipelines, Harness, Semaphore, Codefresh, Octopus Deploy, Google Cloud Build, Buddy.

Інструменти для управління вимогами – це програмні засоби, які призначені для організації управління (збір, документування, відстеження змінами, керування) вимогами до програмного забезпечення протягом усього життєвого циклу розробки програмного забезпечення. Найпоширеніші інструменти управління вимогами (за типом інструментарію) [49,50]:

– Jama Software, IBM Engineering Requirements, Innoslate, Polarion Requirements, CodeBeamer, Accompa, ReqView – спеціалізовані інструменти управління вимогами;

– SpiraTeam, Polarion Requirements, CodeBeamer – інструменти життевого циклу ПЗ з модулем управління вимогами;

– Enterprise Architect – інструменти для моделювання та проектування з функціональністю управління вимогами;

– Jira Software, Trello – Інструменти для управління проектами з можливістю базового управління вимогами.

У таблиці 3.2 відображено підтримку категоріями інструментів для тестування програмного забезпечення ключових підходів до оптимізації використання тестових ресурсів.

Таблиця 2.2

Підтримка інструментами тестування програмного забезпечення підходів до оптимізації використання тестових ресурсів.

№	Інструмент	Пріоритезація вимог	Планування тестових циклів	Автоматизоване тестування	Ризик-орієнтоване тестування	Контроль покриття тестування	Хмарні та віртуальні середовища	Інтеграція у CI/CD
1	Системи управління тестуванням	+	+		+			+
2	Інструменти автоматизації тестування		+	+	+	+		+
3	Інструменти управління тестовим середовищем	-		-			+	
4	Інструменти аналізу покриття коду		-		+	+	-	+
5	Інструменти безперервної інтеграції та доставки	+	+	+	+	+	+	+
6	Інструменти для управління вимогами	+		-	+			

3 РОЗРОБКА МЕТОДУ ОПТИМАЛЬНОГО РОЗПОДІЛУ ТЕСТОВИХ РЕСУРСІВ

3.1 Прикладна задача оптимального розподілу тестових ресурсів

Перед QA-відділом IT компанії постала задача планування тестування нового релізу програмного продукту. Реліз охоплює десятки тестових різнотипних тестових сценаріїв від простих рутинних до складних та критичних. Кожний тестовий сценарій описується набором своїх атрибутів: тип складності, базовий час виконання, вимоги, які він покриває, його пріоритет та залежності від інших тестів (наприклад, тести на створення документів треба виконати раніше за тести на формування звітів).

Команда тестувальників, які будуть проводити тестування, складається з обмеженої кількості спеціалістів різної кваліфікації. Кожна кваліфікація має певний профіль компетенцій – перелік типів тестів, які можуть виконувати тестувальникі даної кваліфікації. Одні тестувальники здатні виконувати лише прості рутинні тести, інші кваліфікуються на складних. Також, різні тестувальникі виконують тести з різною швидкістю, в кожного тестувальника ця швидкість індивідуальна. Призначення висококваліфікованого спеціаліста на виконання простого рутинного завдання може бути неефективним. Вартість виконання такого завдання стане невиправдано дорогим, хоча час виконання такого завдання може значно скоротитися.

Вимоги до програмного забезпечення, які перевіряються тестами, мають різний ступінь серйозності. Критичні вимоги повинні бути покрити в першу чергу, менш важливі можуть бути відкладені, або, при жорсткому обмеженні у ресурсах, взагалі виключені.

У такій ситуації QA-відділу треба вирішити комплексну задачу по розподілу ресурсів, а саме: які саме тести, кому з команди тестувальників призначаються та у якій послідовності, щоб досягти одночасно трьох конфліктуючих цілей:

1) покрити якомога більше вимог та перш за все критичних, тобто обрати максимально корисну множину тестів;

2) забезпечити правильний порядок виконання задач;

3) завершити тестування вчасно, дотримуючись дедлайнів релізу;

4) уникати, а за можливості взагалі виключити, ситуації, коли висококваліфікований спеціаліст виконує прості рутинні завдання.

Важливими умовами, які слід враховувати при побудові розкладу тестування є наступні:

1) обмеженість ресурсів: кількість тестувальників певних кваліфікацій обмежена;

2) критичність вимог та необхідність забезпечити їх покриття;

3) залежності між тестовими сценаріями: деякі сценарії повинні бути виконані строго перед іншими;

4) індивідуальна швидкість виконання завдань тестувальниками;

5) набір профільних навичок тестувальника впливає на ефективність призначення .

Таким чином, перед QA-відділом постає складна багатокритеріальна задача оптимізації: сформувати оптимальний план тестування, що буде відповідати технічним, ресурсним і часовим обмеженням релізу. Тобто, необхідно знайти компромісне рішення, що забезпечить належні якість покриття вимог, час проходження тестування та ефективність використання ресурсу тестувальників [51].

Оскільки усі три цілі суперечать одна одній, не існує одного найкращого по усіх критеріях плану. У таких умовах може бути застосовано підхід, при якому будується множина компромісних альтернативних рішень, або фронт Парето. Кожне рішення у ньому є оптимальним, але у сенсі, що поліпшення одного критерію неможливе без погіршення хоча б одного з інших.

Серед найбільш популярних компромісних рішень можна визначити наступні.

1. План «Аврал» орієнтовано на мінімізацію часу виконання тестування за рахунок використання кваліфікованих ресурсів. Кваліфікованих тестувальників залучають до виконання простих рутинних завдань – це дозволяє скоротити загальний час тестування, однак підвищує неефективність використання людських ресурсів. Крім того, частина вимог низького пріоритету можуть залишитися непокритими.

2. План «Ефективність» орієнтовано на досягнення максимальної ефективності використання ресурсів. Кваліфікованих тестувальників долучають лише до складних профільних завдань, а більшість простих та рутинних назначається тестувальникам нижчого рівня кваліфікації. Такий підхід призводить до збільшення загальної тривалості процесу тестування, але мінімізує неефективність використання ресурсу тестувальників.

3. План «Максимальна якість» орієнтовано на максимальне покриття вимог та гарантоване покриття критичних вимог до програмного продукту. Такий підхід підвищує неефективність, бо кваліфіковані тестувальники вимушені працювати над простими та рутинними сценаріями. Загальний час тестування при такому плані збільшується, але виконання його гарантує найменший ризик дефектів у релізі.

4. План «Збалансований» орієнтовано на компроміс між усіма трьома критеріями. Такий підхід забезпечує баланс між часом виконання, ефективністю використання ресурсів та якістю тестування. Кваліфіковані тестувальники можуть бути залучені до простих рутинних завдань у тому випадку, коли це значно скорочує загальний час виконання. У межах плану гарантовано покриваються критичні вимоги, а низько пріоритетні сценарії відкладаються та виконуються за залишковим принципом. В практичних умовах саме такий план розглядяться як найбільш прийнятний.

3.2 Математична модель задачі оптимального розподілу тестових ресурсів

Описану прикладну ситуацію можна формалізувати як задачу оптимального розподілу тестових ресурсів у вигляді орієнтованого графа, що дозволяє врахувати всі ключові аспекти процесу тестування в межах обмежених ресурсів. Завдяки своїй структурі та властивостям, графові моделі здатні ефективно представляти складні системи, забезпечуючи чітке розділення сутностей, відображення взаємозв'язків та залежності між ними [52,53].

Для моделювання задачі ефективного розподілу тестових ресурсів в умовах обмежень пропонується використання орієнтованого графа $G = (V, E)$, де:

1) V – непорожня скінченна множина вершин, яка включає два типи вершин:
   - T множина тестових випадків ($t_i \in T$);
   - R множина тестових ресурсів: ($r_j \in R$);
   - $T \cup R = V, T \cap R = \emptyset$;
2) E – множина орієнтованих ребер, які відображають залежності:
   - $(t_i, r_j) \in E_{res}$ – відображають потребу тестового випадку t_i у ресурсі тестування r_j;
   - $(t_i, t_k) \in E_{dep}, i \neq k$ – відображають залежність тестового випадку t_i від тестового випадку t_k – необхідність виконання t_i до запуску t_k (визначення порядку виконання тестових випадків);
3) атрибути тестових випадків $\forall t_i \in T$:
   - e_{t_i} – тип тестового випадку;
   - $d_{t_i}^{base}$ – базовий час, необхідний для виконання тестового випадку t_i;
   - p_{t_i} – пріоритет тестового випадку t_i (priority, терміновість);
   - c_{t_i} – множина вимог, яка покривається тестовим випадком t_i.
4) атрибути вимог тестових випадків $\forall c \in C$:
   - s_{c_k} – важливість покриття вимоги c_k (severity, критичність);
5) атрибути ресурсів тестових випадків $\forall r_j \in R$:

– e_{r_j} – тип ресурсу r_j;

– q_{r_j} – кількість доступних одиниць ресурсу r_j;

– f_{r_j} – коефіцієнт ефективності, показує, наскільки швидше або повільніше ресурс виконує завдання r_j;

– $K_{r_j}^{poss}$ – множина усіх типів тестових завдань, які може виконувати ресурс r_j ;

– $K_{r_j}^{pref}$ – множина профільних типів тестових завдань, які відповідають основній кваліфікації ресурсу r_j ($K_{r_j}^{pref} \subseteq K_{r_j}^{poss}$);

– W_{r_j} – вартість неефективності, відображає цінність часу ресурсу r_j при призначенні його на непрофільне завдання (штраф);

6) змінні рішення:

– $x_{t_i} \in \{0,1\}$ – бінарна змінна, яка вказує, чи обрано тест на виконання (1, якщо обрано, 0 ні);

– $z_{t_i,r_j} \in \{0,1\}$ – бінарна змінна, що вказує, чи призначено тест t_i ресурсу r_j (1, якщо призначено, 0 ні);

– $y_c \in \{0,1\}$ – бінарна змінна, що показує, чи покрито вимогу c хоча б одним тестовим випадком (1, якщо покрито, 0 ні);

7) цільові функції:

– покриття вимог (якість тестування) (3.1):

$$Cover = \sum_{c \in C} s_c y_c \tag{3.1}$$

– час виконання тестових випадків з урахуванням їх пріоритету (ефективність розкладу) (3.2):

$$Time = \sum_{t_i \in T} \sum_{r_j \in R} (p_{t_i} d_{t_i}^{base} f_{r_i}) \cdot z_{t_i,r_j} \tag{3.2}$$

– ефективність використання ресурсів:

$$Inefficiency = \sum_{t_i \in T} \sum_{r_j \in R} (I_{t_i,r_j}) \cdot z_{t_i,r_j} \quad (3.3)$$

де I_{t_i,r_j} – вартість невідповідності:

$I_{t_i,r_j} = 0$, якщо тип тесту $e_{t_i} \in K_{r_j}^{pref}$ (бажане призначення);

$I_{t_i,r_j} = W_{r_j}$, якщо $e_{t_i} \in (K_{r_j}^{poss} - K_{r_j}^{pref})$ (можливе, але небажане призначення);

8) обмеження на використання ресурсів:

– порядок виконання тестів: тестовий випадок виконується, якщо виконано той, від якого він залежить (3.4):

$$x_{t_k} \leq x_{t_i}, \forall (t_k, t_j) \in E_{dep}; \quad (3.4)$$

– сумісність типу ресурсу: на ресурсі може виконуватися лише сумісний тип тестового випадку (3.5):

$$z_{t_i,r_j} = 0 \text{ , якщо } e_{t_i} \notin K_{r_j}^{poss}; \quad (3.5)$$

– призначення ресурсу: кожному тестовому випадку може бути призначений сумісний ресурс (припускаючи, що $x_{t_i} = 1$) (3.6):

$$\sum_{r_j \in \mathrm{R}} z_{t_i,r_j} = x_{t_i}, \forall t_i \in T; \quad (3.6)$$

– обмеження на кількість ресурсів: загальна кількість одночасних призначень не перевищує кількість доступних одиниць (3.7):

$$\sum_{t_i \in T} z_{t_i,r_j} \leq q_{r_j}, \forall r_j \in R; \quad (3.7)$$

– покриття вимог: вимога c вважається покритою ($y_c = 1$), якщо її покриває хоча б один тест (3.8):

$$\sum_{t_i \in T, c \in c_{t_i}} x_{t_i} \geq y_c. \quad (3.8)$$

Таким чином, багатоцільова задача оптимізації буде полягати в знаходженні компромісного рішення між трьома цілями: $max(Cover)$, $min(Time)$ в та $min(Inefficiency)$ в межах наведених вище обмежень.

Для застосування розробленої математичної моделі задаємо загально прийняті значення деяких параметрів, які буде використано як вхідні дані для побудови оптимального плану тестування.

1. Тип складності тесту.

$$e_t \in \{Routine, Standard, Advanced\},$$

де $Routine$ – рутинні тести низької складності;

$Standard$ – стандартні тести середньої складності;

$Advanced$ – тести високої складності.

2. Пріоритет тесту визначає важливість виявлення дефекту на основі бізнес-потреб (priority, терміновість):

$$p_t \in \{Highest, High, Medium, Low\},$$

де: $Highest$ – найвищий пріоритет (екстрені ситуації, потребують невідкладного виконання);

$High$ – високий пріоритет (потребують виконання якнайшвидше);

$Medium$ – звичайний пріоритет (призначається за замовчанням);

Low – низький пріоритет (виконуються в останню чергу, якщо є ресурси).

3. Важливість покриття вимоги (severity, критичність):

$$s_c \in \{Blocked, Critical, Major, Minor, Trivial\},$$

де $Blocked$ – блокуюча, непокриття робить продукт непридатним для коректного та безпечного використання

$Critical$ – критична, не покриття може спричинити серйозні збої

$Major$ – значна, непокриття впливає на значні частини функціоналу, але не блокує систему

$Minor$ – незначна, непокриття впливає на другорядні функції та не є критичною для бізнес-процесу

$Trivial$ – тривіальна, непокриття має слабкий вплив, за браку часу може бути пропущена.

4. Кваліфікація тестувальника (тип ресурсу)

$$e_r \in \{Junior, Middle, Senior\},$$

де $Junior$ – початковий рівень, виконує прості рутинні тестові сценарії;

$Middle$ – середній рівень кваліфікації, виконує більшість стандартних тестових сценаріїв;

$Senior$ –високий рівень кваліфікації, експерт, виконує складні тести високої критичності.

3.3 Гібридний підхід до оптимального розподілу тестових ресурсів

Для вирішення складної багатокритеріальна задачі оптимізації (п.3.1) було запропоновано гібридний підхід, що поєднує графове моделювання (п.3.2) та еволюційну багатоцільову оптимізацію. Узагальнену блок-схему пропонованого гібридного підходу до вирішення оптимізаційної задачі розподілу тестових ресурсів на графовій моделі наведено на рисунку 3.1 [54].

Рис. 3.1 Гібридний підхід до вирішення оптимізаційної задачі розподілу тестових ресурсів на графовій моделі

Побудова графа. Перший етап пропонованого гібридного підходу до оптимального розподілу тестових ресурсів, основна ціль якого отримати формальний опис системи тестів, ресурсів, залежностей та обмежень.

Відповідно до цілей було обрано формальне уявлення задачі у вигляді орієнтованого графу, що надає наступні переваги:

– графова модель дозволяє чітко відокремити структуру задачі від процесу оптимізації;

– зв’язок між тестом та ресурсом, який його може виконати, зручно уявляти як дводольний граф;

– залежності між тестами зручно уявляти у вигляді орієнтованого ребра графа.

В результаті виконання даного етапу отримуємо уявлення вхідних даних до задачі оптимізації у вигляді графовой структури, що дає наочне формальне маштабоване уявлення всього процесу тестування.

Аналіз та очищення графа. Другий етап пропонованого гібридного підходу до оптимального розподілу тестових ресурсів, основна ціль якого підготувати вихідні дані до подальшого аналізу та ефективного призначення ресурсів за

рахунок усунення неточностей та зайвої інформації, виявлення помилок, розрішення конфліктів.

Цей етап складається з наступних кроків.

1. Перевірка коректності графу. Згідно моделі, граф G є орієнтованим та ациклічним. При залежності тестових випадків один від одного та наявності циклів є неможливим визначити, який з тестових випадків повинен виконуватися першим. Перевірка виконується наступним чином:

   – будуємо підграф $G_T(T, E_T)$, який складається тільки з тестових випадків і залежностей між ними, тобто, ребра (t_i, t_j), які означають, що тестовий випадок t_j залежить від t_i і повинен виконуватись пізніше нього;

   – виконуємо топологічне сортування на графі: якщо сортування виконано успішне, граф є ациклічним и можливо виконати правильний порядок тестових випадків, інакше, якщо виявлено цикл, в логіці послідовності тестових випадків є помилка і неможливо виконати тести без порушення залежностей.

2. Видалення несумісних ребер. Не всі тестові випадки можуть виконуватись на будь-якому ресурсі, тож існування ребер між несумісними тестовим випадком та ресурсом недопустимо. Перевірка виконується наступним чином:

   – будуємо дводольний граф призначень $G_{assign}(T, R, E_{assign})$, у якому ребро (t_i, r_j) існує, якщо тип тесту t_i сумісний з ресурсом r_j: тобто якщо $e_{r_j} \in K_{r_j}$;

   – видаляємо усі несумісні ребра.

3. Виявлення та видалення таких неточностей та аномалій як: некоректність атрибутів тестів, дубльовані та надлишкові тести, тести з некоректними параметрами.

В результаті виконання даного етапу виключаються некоректні сценарії (циклічні залежності) та можливість побудови некоректних початкових рішень, що підвищує якість подальшої оптимізації.

Кластеризація тестових випадків. Третій етап представляє ідею розбиття задачі на підзадачі для зменшення складності основної задачі оптимізації. Мета етапу - поділити велику графову модель на менші, логічно зв'язані частини, з якими легше й ефективніше працювати, особливо в умовах обмежених обчислювальних ресурсів або для паралельної обробки. Виявлення груп схожих тестових випадків покращує поведінку генетичного алгоритму. Кластеризація допомагає формувати кращі початкові рішення за рахунок формування, так званого, кластерного патерна. Також кластеризація дозволяє формувати структуровану хромосому, вдаліше будувати мутації та кросовери, що будуть ефективні для збереження фрагментів вдалих призначень.

Етап передбачає розбиття графу тестових випадків на підграфи (кластери). Кожен кластер містить тільки ті тестові випадки, які залежать лише один від одного та не залежать від інших груп. Етап виконується наступним чином [51]:

– будуємо підграф $G_T(T, E_T)$, який складається тільки з тестових випадків і залежностей між ними, тобто, ребра (t_i, t_j), які означають, що тестовий випадок t_j залежить від t_i і повинен виконуватись пізніше нього;

– проводимо розбиття графа G_T на підграфи (кластери): кожен підграф містить тести, які залежать лише один від одного, але не від інших груп, за допомогою алгоритмів топологічного сортування та SCC-аналізу;

– у кожному підграфі (кластері) G_T обчислюємо найдовший шлях, використовуючи час виконання тестового випадку d_{t_i} – тестові випадки з критичного шляху вважаємо найбільш важливими для функції Time.

В результаті виконання даного етапу простір пошуку скорочується, ефективніше працює генетичний алгоритм пошуку рішення. Це дозволяє підвищити стабільність оптимізації.

Евристичний пошук початкового рішення. Четвертий етап передбачає обрання тестових випадків, що покривають найбільшу кількість ще не покритих вимог, виконання призначення обраним тестовим випадкам доступних ресурсів з урахуванням умов та обмежень та групування обраних тестових випадків по рівням. Мета етапу – побудова якісного початкового варіанту розподілу ресурсів по тестовим випадкам (початкового наближення), який буде стартовою точкою для еволюційного алгоритму. Цей етап складається з наступних кроків.

1. Побудова стартового покриття вимог. Визначення набір тестів $T' \in T$, який максимально покриває множину вимог C. Використовуючи жадібну стратегію, побудова стартового покриття вимог виконується наступним чином:
   - ініціюємо множину покритих вимог $C' = \emptyset$;
   - допоки $C' \neq C$:
   - $\forall t_i \in T \backslash T'$ обчислюємо $\left|\mathrm{c}_{t_i} \cap (C \backslash C')\right|$ – кількість непокритих вимог, що покриває тестовий випадок t_i,
   - обираємо t_{best} з максимальним покриттям;
   - $T' = T' \cup t_{best}$ додаємо тестовий випадок до стартового покриття;
   - $C' = C' \cup \mathrm{c}_{t_{best}}$ додаємо вимоги до множини покритих вимог.

2. Призначення ресурсів. Кожному тесту $\forall t_i \in T'$ призначається один ресурс $r_j \in R$, сумісний з типом e_{t_i}, без перевищення кількості q_{r_j}. Призначення можна виконати двома способами: жадібною стратегією, коли тестовий випадок для призначення обирається у порядку пріоритету або тривалості, або за допомогою алгоритму Хангарі.

3. Визначення рівнів паралельного виконання. Організація виконання тестів у паралельних рівнях, дотримуючись порядку та обмежень графової моделі:
   - топологічним сортуванням визначаємо рівні тестових випадків для $G_T(T, E_T)$;
   - поділяємо тестові випадки на рівні: 0 усі без вхідних дуг (незалежні), 1 - всі залежності яких розміщені в рівні 0, і т.д.

– у межах рівня дозволяємо паралельне виконання, ресурси розподіляємо так, щоб уникати перевищення q_{r_j}в одному рівні.

Виконання даного етапу зменшує час сходимості евристичного алгоритму, бо пошук рішення починається з гарних рішень, які вже можуть бути близькими до компромісних рішень.

Евристична багатоцільова оптимізація. П’ятий етап передбачає генерацію рішень оптимізаційної задачі). Основною метою етапу є формування фронту Парето (набору оптимальних планів), які максимізують покриття вимог *Cover*, мінімізують час виконання тестування *Time* та мінімізує неефективність використання ресурсів *Inefficiency*, враховуючи обмеження (ресурси, залежності, сумісність, покриття вимог). Етап складається з наступних кроків.

1. Генерація рішень евристичним генетичним алгоритмом, де хромосома (рішення) враховує графову структуру задачі.
2. Оцінка рішень. Для кожного можливого рішення в обраній популяції (множині хромосом) оцінюються значення цільових функцій.
3. Якщо не виконуються умови або порушуються обмеження накладається штраф, який враховуються у процесі оптимізації як додаткові «вартісні» елементи, з метою «заохочення» рішень, які задовольняють усім обмеженням та умовам.
4. Обчислення компромісних рішень: пошук компромісу між максимізацією покриття умов *Cover,* мінімізацією часу виконання *Time* та мінімізацією неефективності використання ресурсів *Inefficiency*. Такі рішення можуть бути знайдені пошуком набору Парето-оптимальних рішень, або шляхом розподілу популяції рішень, коли у процесі еволюції отримується множина рішень, з яких вибирається найкраще.

В результаті виконання даного етапу буде формуватися кілька оптимальних планів тестування, враховуючі усі обмеження одночасно, що надасть менеджеру QA-відділу можливість вибору між ними.

Остаточний розподіл ресурсів. Заключний шостий етап підходу передбачає обрання найкращих рішень та їх перевірку на можливість впровадження у реальних умовах. Основними цілями етапу є перетворення хромосом оптимального рішення

у реальні плани тестування, збереження отриманих рішень для подальшого аналізу та візуалізації отриманих результатів.

В результаті виконання даного етапу теоретичне рішення перетворюється у реальний календарний план для тестування, надається можливість виконувати історичний аналіз рішень та підготовка даних для інтерфейсу користувача.

Таким чином, запропонований гібридний підхід дозволяє ефективно вирішувати задачу оптимального розподілу тестових ресурсів з урахуванням обмежень, взаємних залежностей та конфліктних цілей. Він поєднує графовий підхід до моделювання задачі та багатоцільову евристичну оптимізацію. Це надає можливості враховувати одночасно логічну структуру процесу, формувати основу для прийняття обґрунтованих рішень щодо планування проведення процесу тестування. Подальші дослідження можуть бути зосереджені на адаптації моделі до динамічних змін ресурсів під час виконання тестування та динамічне оновлення планів у реальному часі.

3.4 Алгоритм оптимального розподілу тестових ресурсів

Базовим алгоритмом для вирішення багатокритеріальної задачі оптимізації розподілу тестових ресурсів було обрано генетичний алгоритм NSGA-II. Наведемо узагальнений опис алгоритму [35,36].

Позначення:

– P_t – поточна популяція розміром N;

– Q_t – популяція нащадків;

– $R_t = P_t \cup Q_t$ – об’єднана популяція.

Крок 1. Ініціалізація.

1. Задати початкові параметри алгоритму:

– розмір популяції N;

– максимальна кількість поколінь G;

– ймовірність кросовера p_c,

– ймовірність мутації p_m.

2. Згенерувати початкову популяцію P_0 випадковим або евристичним способом.

3. Для кожної хромосоми обчислити значення цільових функцій Cover, Time, $Inefficiency$.

Крок 2. Сортування та оцінка (non-dominated Sorting).

1. Розподілити особин на фронти F_1, F_2, F_3,…за критеріями Cover, Time, $Inefficiency$ (недомінантне сортування):
   - F_1 – усі недоміновані рішення,
   - F_2 – рішення, домінovані лише представниками F_1,
   - і т.д.

2. Для кожної особи з кожного фронту обчислити *crowding distance* – міру «простору», або оцінки «різноманіття» навколо рішення у цільовому просторі.

Крок 3 Генерація популяції нащадків Q_t.

1. Обрати пари хромосом-«батьків» за рангом фронту та оцінкою різноманіття, враховуючи:
   - номер фронту (чим менше, тим краще);
   - crowding distance (чим більше, тим краще).

3. Застосувати кросовер із ймовірністю p_c.

4. Застосувати мутацію з ймовірністю p_m.

5. Для кожного нащадка обчислити значення цільової функції Cover, Time, $Inefficiency$.

Крок 4. Об’єднання популяцій: будуємо розширену популяцію:

$$R_t = P_t \cup Q_t \ , |R_t| = 2N$$

Крок 5. Побудова нової популяції.

1. Виконати недомінантне сортування для всіх особин об’єднаної популяції R_t (розподілити особин на фронти F_1, F_2, F_3,…за критеріями *Cover, Time,* $Inefficiency$)

2. Додати до нової популяції R_{t+1} отримані фронти F_1, F_2, F_3,…допоки сумарний розмір не перевищить N.

3. Якщо усі особи чергового фронту не вміщуються цілком, то:

– виконати сортування елементів фронту за crowding distance;

– вибрати $N - |R_t|$ найкращих за crowding distance.

Крок 6. Формування наступного покоління.

– якщо $t < G$, перейти до кроку 3 (генерація нових нащадків).

– інакше завершити роботу та повернути всю популяцію (фронт Парето).

Як значення фітнес-функції (рівень пристосованості хромосоми) у алгоритмі NSGA-II використовується пара значень (Rank, Distance). Параметр Rank відповідає за те, наскільки це рішення краще за інші, а параметр Distance визначає, наскільки це рішення є унікальним.

У генетичному алгоритмі для розподілу тестових ресурсів пропонується використовувати дворівневу схему застосування операторів кросинговеру та мутації. Така схема буде враховувати наявність кластерної структури тестових випадків та адаптувати генетичні оператори за двома рівнями: рівень кластерів (макро-рівень) та рівень тестових випадків (мікро-рівень). Макро-рівень генетичних операторів буде відповідати за макро-структуру плану тестування – сценарні блоки, мікро-рівень за налаштування розподілу ресурсів та послідовностей тестових випадків у межах кластеру.

Наведемо кодування рішень та генетичні оператори оптимізаційного алгоритму на рівні тестових випадків, що відповідає випадку їх однокластерної структури.

1. Структура хромосоми. Згідно розробленої моделі (п.3.2) багатоцільової оптимізації має три конфліктуючих цільові функції: Time, Cover та Inefficiency. Стандартний підхід до кодування хромосом послідовністю задач (тестових випадків) та перестановкою для них обраних ресурсів (тестувальників) є недостатньо гнучким, так як призводить до утворення інтервального розкладу для тестувальників. Такий розклад не буде ефективним. Якщо ми розуміємо під

ресурсом людину-тестувальника, то час «простою» такого ресурсу в межах стандартного робочого дня, також вартує часу його роботи. Для обходу цієї проблеми кодуванням хромосом генетичного алгоритму (рішень) було обрано підхід «подвійна хромосома». Хромосома складається з 2х частин: порядок задач тестування, у якому вони подаються на розгляд та призначення, що визначає бажаний ресурс для кожної задачі.

Хромосому будемо визначати як впорядковану пару двох векторів цілих чисел довжини $|T|$:

$$H = \langle S, A \rangle, \tag{3.8}$$

де S – вектор послідовності вибору на виконання тестових задач;

A – вектор призначення ресурсів, «бажані» тестувальники на виконання відповідної тестової задачі.

Вектор послідовності визначає чергу, у який задачі будуть розглядатися для формування розкладу:

$$S = \left(s_1, s_2, s_3 \ldots . s_{|T|}\right), \;\; s_k \in \{0..|T|-1\},$$
$$\forall i, j \; \in \{0..|T|-1\}, i \neq j \; \rightarrow s_i \neq \; s_j.$$

Кожен елемент s_k вектору вказує на номер задачі тестування, яку слід розглядати на k-кроці. Вектор S впливає на задоволення обмежень упорядкованості задач тестування та на цільову функцію покриття вимог Cover, обираючи які задачі будуть виконані першочергово.

Вектор призначення ресурсів A показує «бажаний» ресурс для виконання відповідної тестової задачі:

$$A = \left(a_1, a, a_3 \ldots . a_{|T|}\right), \;\; a_i \in \{0..|R|-1\}. \tag{3.9}$$

Кожен елемент a_i є ідентифікатором ресурсу r_{a_i}, який у пріоритеті («бажаний ») до виконання задачі a_i. Вектор A дозволяє мінімізувати цільову функцію неефективності *Inefficiency* так як кодує пари (t_i, r_j) в яких вартість невідповідності прагне до мінімального значення (0 якщо $e_{t_i} \in K_{r_j}^{pref}$).

Таким чином, вектор S хромосоми намагається оптимізувати час та покриття, а вектор A паралельно намагається оптимізувати неефективність використання.

2. Відображення хромосоми у готове рішення. Процес декодування хромосоми опишемо як функцію відображення:

$$\Phi: \langle S, A \rangle \rightarrow \Omega,$$

де: Ω – множина змінних розкладу (start_time, z_{t_i,r_j}).

Алгоритм звертається до задач тестування у порядку визначеному елементами вектору S. Для поточної задачі $t_i = s_k$ алгоритм:

1) перевіряє, чи виконані всі задачі t_l від яких залежить задача t_i: $t_l \in E_{dep}$:

– якщо ні (не усі попередні задачі вже виконано), поточну задачу t пропускаємо;

– якщо так, то обчислює час готовності їх виконання:

$$ReadyTime_i = max\ (\{FinishTime_k | t_k \in Predecessors(t_i)\});$$

2) обирає ресурс r^{fact} між «бажаним» ресурсом з вектору A: $r_{pref} = A[t]$ та альтернативним ресурсом.

$$r^{fact} = \begin{cases} r_{pref}, \text{якщо } FinishTime(t, r_{pref}) \leq (1+\delta) FinishTime(t, r_{alter}) \\ r_{alter}, \text{інакше} \end{cases}$$

де δ – допустимий зазор часу затримки – скільки готові зачекати, поки ресурс r_{pref} завершить попередню задачу;

3) альтернативний ресурс r_{alter} обирається за «жадібною» стратегією як найшвидший за часом завершення роботи:

$$FinishTime(r_{alter}) \quad = \ ((FreeTime_r, ReadyTime_i) + \ d_{t_i}^{base} \cdot f_r \quad).$$

Використання допустимого зазору часу затримки буде дозволяти надавати перевагу мінімізації неефективності використання ресурсу Inefficiency перед мінімізацією загального часу тестування Time. При значенні зазору $\delta = 0$ перевага буде у мінімізації часу, тобто задача буде видаватися наступному за пріоритетом ресурсу. Факт того, що новий обраний ресурс може бути непрофільним для цієї задачі буде ігноруватися, що призведе до штрафування та збільшення значення функції неефективності Inefficiency.

3. Оператори кросинговеру та мутації. Оскільки хромосома алгоритму (рішення) кодується двома частинами, що є різними за природою, оператори кросинговеру та мутації будуть різними для кожної частини хромосоми.

Кросинговер для вектору послідовності розгляду тестових задач S обираємо порядковий кросинговер (two-point order crossover). Такий кросинговер буде підтримувати збереження відносної послідовності задач тестування у хромосомі. Для двох батьків S^A та S^{B}:

1) обираємо дві точки розриву хромосоми $1 \ \leq m_1 \leq m_2 \leq |T|$;

2) елементи між точками розриву «успадковуються» нащадком S^{child1}:

$$\forall i \ \in \{m_1 + 1, \dots m_2\}: \quad S_i^{child1} \leftarrow S_i^A,$$
$$V_{keep} = \{S_i^A | \quad m_1 < i \leq m_2\},$$

де V_{keep} – успадкований сегмент генів;

3) формуємо допоміжну послідовність з батька S^{B} елементи між точками розриву «успадковуються» нащадком S^{child1} – робимо це циклічно, скануємо гени батька S^{B} починаючи з гену $m_2 + 1$.

$$I_{scan} = \langle k_1, k_2 \ldots . k_{|T|} \rangle, k_j = \big((m_2 + j - 1)(mod\ |T|)\big) + 1\ ,$$

$$S_{scan} = \langle S_k^B |\ k\ \in I_{scan} \wedge S_k^B \notin V_{keep}\ \ \rangle,$$

де I_{scan} – послідовність індексів генів для сканування;

S_{scan} – послідовність кандидатів для успадкування за винятком вже успадкованих V_{keep};

4) вставляємо елементи з S_{scan} у пусті позиції нащадка S^{child1}:

$$I_{fill} = \langle l_1, l_2 \ldots . l_{|S_{scan}|} \rangle, l_j = \big((m_2 + j - 1)(mod\ |T|)\big) + 1$$

$$\forall j\ \in \{1, \ldots |S_{scan}|\}: \quad S_{l_j}^{child1} \leftarrow S_{scan_j}$$

де I_{fill} – послідовність індексів генів для вставки.

Для отримання вектору S^{child2} другого нащадка кросинговер виконується симетрично: основу беремо з батька S^{B} по тих самих точках розриву m_1 та m_2, допоміжні послідовності та заповнення виконуються аналогічно, але значення беруться у батька S^{A}.

Використання такого кросинговеру дозволяє зберігати ефективні послідовності, блоки задач.

У якості мутації для вектору послідовності тестових задач S обрано обмінну мутацію. Така мутація дозволить випадково змінити пріоритет виконання тестового завдання. У випадку, коли завдання з високою важливістю вимоги було у кінці вектору S, а завдяки мутації опинилося ближче до началу, це буде підвищувати значення оптимізаційної функції покриття вимог *Cover*.

Випадковим чином обираються 2 позиції генів i та j у векторі S та значення на них обмінюються:

$$s_i' = s_j, \qquad s_j' = s_i\ .$$

Кросинговер для вектору призначення ресурсів A обираємо класичний двоточковий. Гени у векторі призначення ресурсів не залежать один від одного, тож збереження їх послідовності не потребується:

1) обираємо дві точки розриву хромосоми $1 \le m_1 \le m_2 \le |T|$;
2) нащадок 1: $a_i^{child1} = \begin{cases} a_i^A, & \text{для } i < m_1 \text{ або } i > m_2 \\ a_i^B, & \text{для } m_1 < i < m_2 \end{cases}$;
3) нащадок 2: $a_i^{child2} = \begin{cases} a_i^B, & \text{для } i < m_1 \text{ або } i > m_2 \\ a_i^A, & \text{для } m_1 < i < m_2 \end{cases}$.

Двоточковість кросинговеру дозволить комбінувати вдалі стратегії розподілу ресурсів.

У якості мутації для вектору призначення ресурсів A обрано випадкове перепризначення. Вибір гена, який буде мутувати обирається ймовірнісно та незалежно для кожного гену у хромосомі. Якщо ген a_i обрано для мутації, нове значення призначеного ресурсу обирається з множини ресурсів, які здатні виконати задачу $\{R^{poss} = \{r_j \in R | e_{t_i} \in K_{r_j}^{poss}\}$. Така мутація впливає на цільову функцію $Inefficiency$. Якщо поточне призначення на рутинну задачу це дорогий ресурс з високою кваліфікацією ($I > 0$), мутація випадково може змінити призначити на задачу більш дешевий ресурс ($I = 0$, або меньший за W), що вилучить штраф за невідповідне призначення та покращить функцію $Inefficiency$.

4. Фітнес-функція генетичного алгоритму. У розробленому алгоритмі функція пристосування кожної хромосоми (фітнес-функція) оцінюється за допомогою вектору значень цільових функцій оптимізаційної моделі ($Time, Cover, Inefficiency$). Стратегією базового алгоритму NSGA-II є мінімізація фітнес-функції, тому цільову функцію покриття вимог, яка у моделі прагне до максимуму необхідно перетворити у функцію, яка прагне до мінімуму. Це досягається інверсією знаку функції: $(Cover) \to min\,(-Cover)$. Таким чином, фітнес-функція для хромосоми C буде мати вигляд:

$$F(C) = \langle Time(C), -Cover(C), Inefficiency(C) \rangle .$$

Наведемо кодування рішень та генетичні оператори оптимізаційного алгоритму на рівні кластерів тестових випадків.

В результаті розбиття графу тестових випадків на підграфи (кластери) множина тестових випадків T може бути представлена підмножинами $K = \{K_1, K_2, \dots K_k\}$, де кожен кластер K_i містить логічно зв'язані тестові випадки. Кластери будемо використовувати як додатковий механізм збереження структурних шаблонів тестових випадків.

1. Кластерно-орієнтована хромосома. Хромосома визначається аналогічно (3.8):

$$H = \langle S, A \rangle,$$

де S – вектор послідовності вибору на виконання тестових задач;

A – вектор призначення ресурсів, «бажані» тестувальники на виконання відповідної тестової задачі.

Вектор S формується як послідовність кластерів у випадковому порядку:

$$S = \left(K_{p(1)}, K_{p(2)} \ \dots K_{p(k)}\right),$$

де: p – випадкова перестановка номерів кластерів.

Вектор призначення ресурсів A показує «бажаний» ресурс для виконання відповідної тестової задачі, аналогічно до (3.9) :

$$A = \left(a_1, a, a_3 \dots . a_{|T|}\right), \ \ a_i \in \{0..|R| - 1\}.$$

2. Кластерно-орієнтований кросинговер. Глобальним кросинговером обираємо одноточковий порядковий кросинговер (one-point order crossover). Такий кросинговер підтримує механізм унікальності, хромосома не буде містити

повторюванні кластери, та забезпечить достатню різноманітність популяції. Для двох батьків S^A та S^{B}:

1) обираємо точку розриву хромосоми $1 \leq m \leq |K|$;

2) перші m кластерів з S^A «успадковуються» нащадком S^{child1}:

$$\forall i \in \{1, \dots m\}: \quad S_i^{child1} \leftarrow S_i^A,$$
$$V_{keep} = \{S_i^A | \, 1 \leq i \leq m\},$$

де V_{keep} – множина успадкованих кластерів;

3) формуємо допоміжну послідовність з батька S^{B} елементи після точки розриву «успадковуються» нащадком S^{child1} – робимо це циклічно, скануємо гени батька S^{B} починаючи з гену $m+1$.

$$I_{scan} = \langle k_1, k_2 \dots . k_{|K|} \rangle, k_j = \big((m+j-1)(mod\ |K|)\big) + 1\ ,$$
$$S_{scan} = \langle S_k^B | \, k \in I_{scan} \wedge S_k^B \notin V_{keep} \rangle,$$

де I_{scan} – послідовність індексів генів для сканування;

S_{scan} – послідовність кандидатів для успадкування за винятком вже успадкованих V_{keep};

4) вставляємо елементи з S_{scan} у пусті позиції нащадка S^{child1}:

$$I_{fill} = \langle l_1, l_2 \dots . l_{|S_{scan}|} \rangle, l_j = \big((m+j-1)(mod\ |K|)\big) + 1$$
$$\forall j \in \{1, \dots |S_{scan}|\}: \quad S_{l_j}^{child1} \leftarrow S_{scan_j}$$

де I_{fill} – послідовність індексів генів для вставки.

Для отримання вектору S^{child2} другого нащадка кросинговер виконується аналогічно, але з переставленими батьками: основну копію беремо з батька S^{B} по

тій самій точці розриву, допоміжна послідовність та заповнення виконуються аналогічно, але значення беруться у батька S^{A}.

Кросинговер для вектору призначення ресурсів A обираємо класичний одноточковий. Гени у векторі призначення ресурсів не залежать один від одного, тож збереження їх послідовності не потребується

1) обираємо точку розриву хромосоми $1 \leq m \leq |K|$;
2) нащадок 1: $a_i^{child1} = \{a_i^{\mathrm{A}},$ для $i \leq m$ $a_i^B,$ для $i > m$;
3) нащадок 2: $a_i^{child2} = \{a_i^B,$ для $i \leq m$ $a_i^A,$ для $i > m$.

3. Кластерно-орієнтована мутація. У якості глобальної мутації послідовності кластерів S обрано обмінну мутацію. Така мутація змінює пріоритет виконання груп тестів, не руйнуючи внутрішню структуру кластера. Випадковим чином обираються 2 позиції кластерів i та j у векторі S та кластери на них обмінюються:

$$K_i' = K_j, \qquad K_j' = K_i .$$

Процес декодування хромосоми та обчислення фітнес функції не змінюються, оскільки кластеризоване уявлення хромосоми не впливає на механізм отримання рішення. План процесу тестування формується послідовним проходженням тестових випадків у порядку, заданому вектором S, із застосуванням «бажаних» ресурсів згідно вектору A, з дотриманням обмежень залежностей та перевіркою доступності ресурсів.

3.5 Опис розробленого програмного забезпечення

3.5.1 Програмне рішення для реалізації гібридного підходу

Для розробки програмного забезпечення, що реалізує запропонований гібридний підхід до вирішення багатокритеріальної оптимізаційної задачі розподілу тестових ресурсів, було обрано середовище розробки MS Visual Studio 2022, багатошарову архітектуру на базі платформи .NET 8 та мову реалізації C#.

Розроблене програмне забезпечення виконане у вигляді єдиного рішення TestResourceOptimizer за архітектурним патерном Modular Solution (модульний моноліт) [56]. Рішення складається з чотирьох слабозв’язаних проектів:

1. Optimization.Core – призначено для визначення основних сутностей задачі оптимізації(тестові ресурси, тестові випадки, їх атрибути, тощо). Проект реалізує шар домену та не має зовнішніх залежностей, що гарантує незмінність моделі графа незалежно від зміни інтерфейсу чи бази даних.

2. Optimization.Solver – призначено для реалізації процесів пошуку початкового рішення та бакатогритеріальної оптимізації. Проект реалізує шар бізнес-логіки та алгоритмів та інкапсулює складну логіку запропонованого підходу, що дозволяє за необхідності легко змінювати алгоритми пошуку рішення без модернізації інших частин системи.

3. Optimization.Data – призначено для збереження вхідних даних, знайдених рішень та їх історій. Проект реалізує шар доступу до даних та використовує патерн Repository, що дозволяє працювати з об’єктами C# без прив’язки до конкретної СУБД.

4. Optimization.UI – призначено для візуалізації отриманих результатів та взаємодії з користувачем. Проект реалізує шар представлення та використовує патерн MVVM (Model-View-ViewModel). MVVM є стандартним патерном для WPF-додатків, який забезпечує гнучке зв’язування даних з інтерфейсом користувача [57].

Структуру розробленого рішення TestResourceOptimizer наведено на рисунку 3.2.

Інтерфейс користувача розроблено у вигляді головного вікна, яке містить навігаційне меню (ліворуч) та динамічну область контенту (праворуч). Навігаційне меню служить для перемикання між п'ятьма функціональними модулями системи.

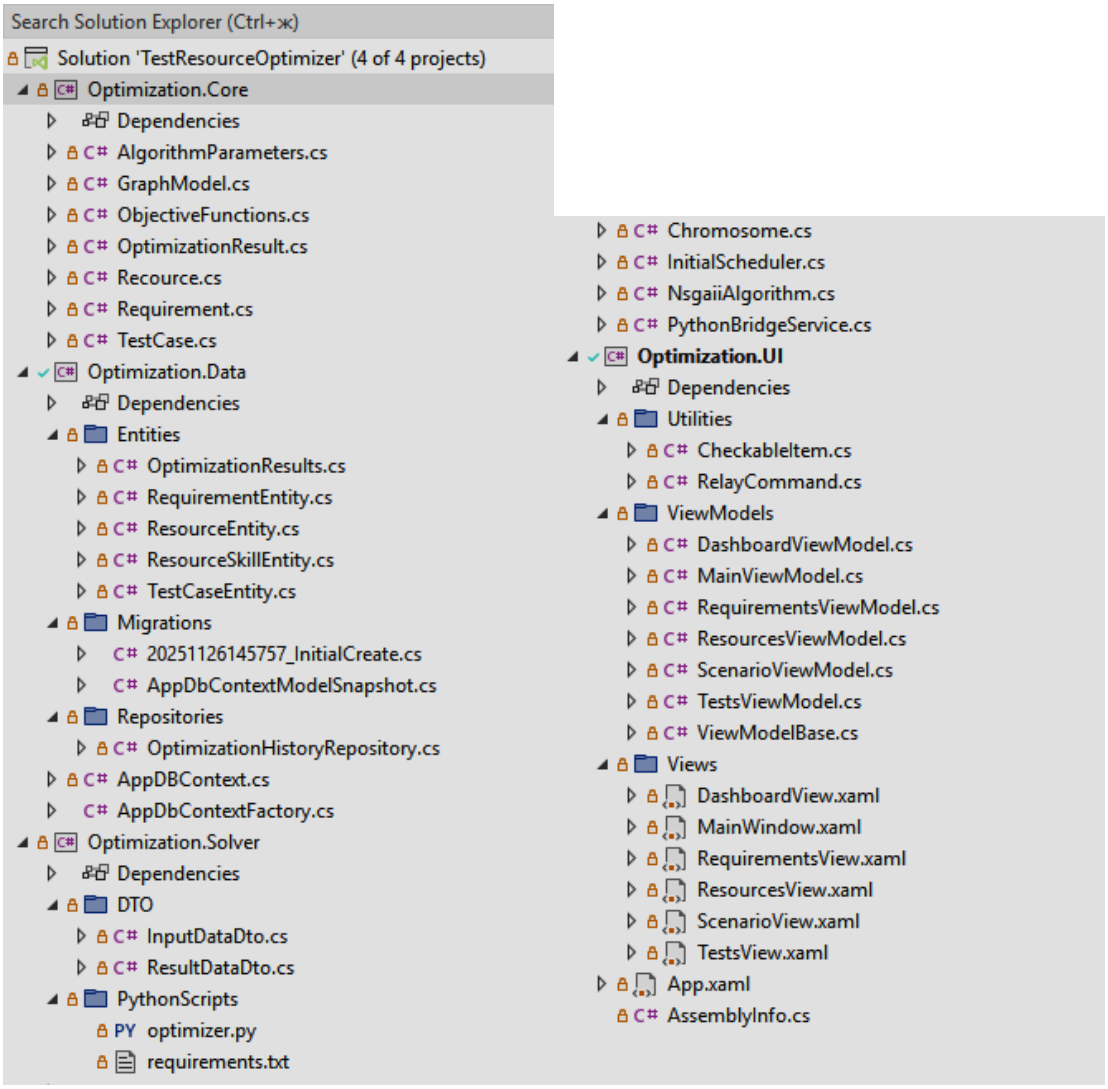


Рис 3.2 Структура розробленого програмного рішення

1. Сторінка «Результати» (рис. 3.3). Модуль призначено для запуску та відображення результатів вирішення задачі багатоцільової оптимізації. Вікно містить область налаштування моделі та аналітичний дашборд, який відображає результати пошуку рішення:

– картки з основними показниками ефективності (числові значення цільових функцій);
– діаграма Парето – графік, що візуалізує найдені кращі рішення;
– діаграма Ганта – розклад для тестувальників з індикацією якості призначення (профільне – зелений, непрофільне - помаранчевий).

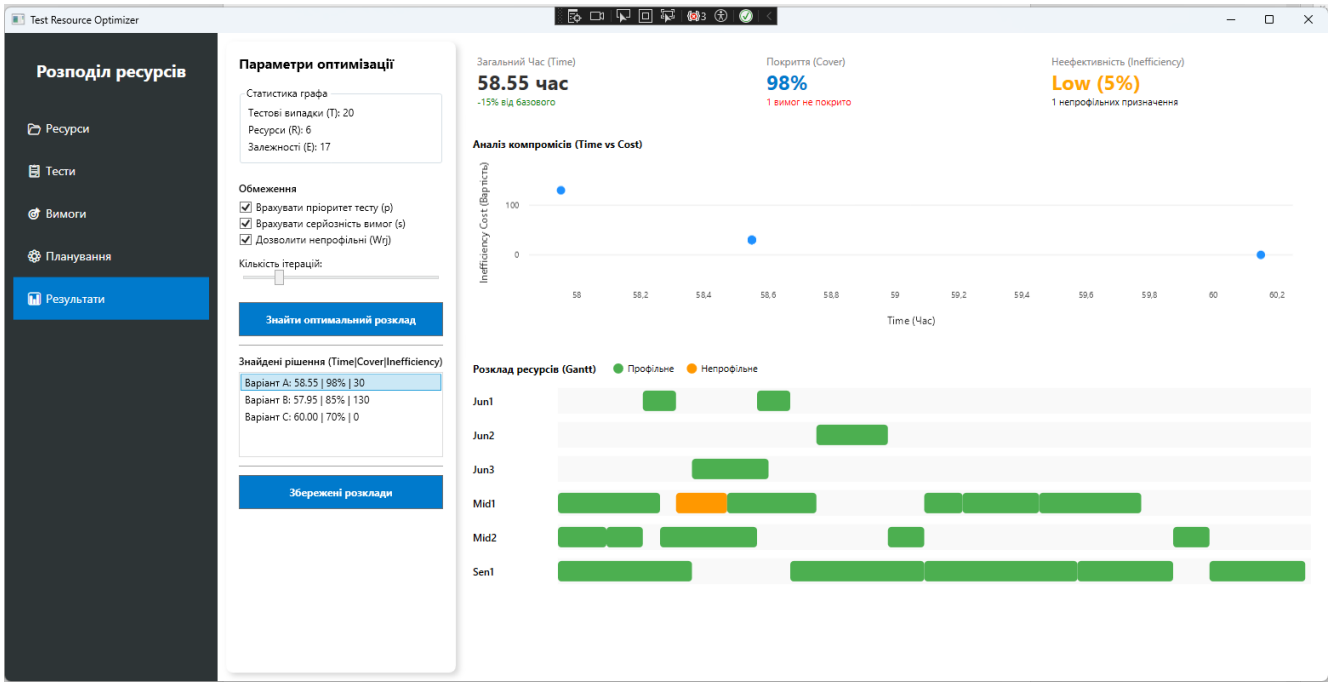


Рис 3.3 Вікно модулю вирішення задачі та відображення результатів

2. Сторінка «Реєстр ресурсів» (рис.В.1). Модуль призначено для відображення та керування множиною ресурсів (R) та їх атрибутами.

3. Сторінка «Репозиторій тестових випадків» (рис. В.2). Модуль призначено для відображення та керування множиною тестових випадків (T) та їх атрибутами.

4. Сторінка «Реєстр вимог» (рис.В.3). Модуль призначено для відображення та керування множиною вимог (C) та їх атрибутами.

5. Сторінка «Налаштування» (рис. В.4). Модуль призначено для налаштування параметрів генетичного алгоритму рішення задачі багатоцільової оптимізації.

Для зберігання даних було розроблено БД TestOptimizationDb, структуру якої наведено на рисунку 3.4.

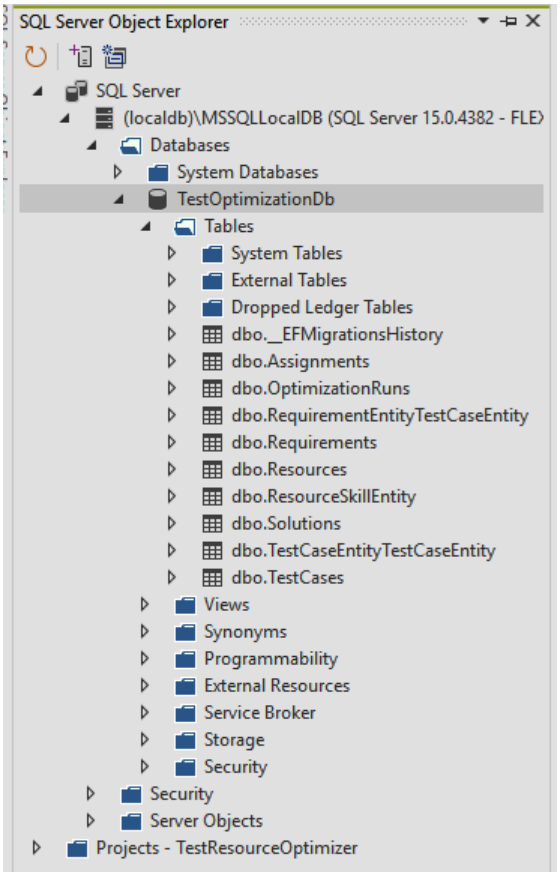


Рис. 3.4 Структура бази даних TestOptimizationDb

3.5.2 Програмна реалізація алгоритму оптимального розподілу тестових ресурсів.

Відповідно до опису алгоритму оптимального розподілу тестових ресурсів було виконано програмну реалізацію алгоритму.

У якості мови реалізації обрано мову Python, що є зручною та широковикористованою мовою для вирішення наукових задач, у тому числі оптимізаційних задач з використанням еволюційних алгоритмів [59]. У якості основного інструменту реалізації було обрано бібліотеку pymoo, що містить функціонал готових та перевірених версій моделей та алгоритмів для вирішення одно та багатокритеріальних задач оптимізації [60]. Особливістю бібліотеки є її гнучкість, щодо адаптації та налаштувань готового функціоналу під особливості та потреби конкретної задачі реалізації. Використання бібліотеки Pymoo дозволило значно спростити процес реалізації алгоритму за рахунок використання готової

версії алгоритму NSGA-II та можливості адаптувати його під розроблену модель багатокритеріальної задачі оптимізації розподілу тестових ресурсів (п.3.2) [62].

На рисунку 3.5 наведено узагальнену блок-схему реалізованого генетичного алгоритму.

Розглянемо основні позначення на блок-схемі:

– PopSize – розмір популяції (кількість особин, хромосом);

– OffspringSize – кількість нащадків, які будуть сформовано;

– Offspring – лічильник нащадків;

– MaxGen – максимальна кількість популяцій;

– Gen – лічильник популяцій;

– MutationRate – вірогідність мутації кожного гену допустимий зазор часу затримки між виконаннм завдань;

– δ – допустимий зазор часу очікування ресурсу між виконанням завдань;

– |T| - потужність множини тестових завдань;

– S – вектор послідовності задач (частина хромосоми);

– A – вектор призначення ресурсів (бажаних) для виконання відповідних задач вектору S (частина хромосоми);

– P0 – початкова популяція;

– Pt – поточна популяція;

– Qt – популяція нащадків;

– Rt – об’єднана популяція батьків та нащадків;

– F1, F2, F3 – множини недомінованих рішень (фронти);

– C1, C2 – хромосоми-батьки, що було обрано для схрещування;

– Child1, Child2 – хромосоми-нащадки, результат схрещування хромосом C1, C2;

– Binary Tournament Selection – метод вибору хромосом-батьків, що використовується у алгоритмі NSGA-II;

– Point Order Crossover – одноточковий кросинговер зі збереженням порядку розташування кластерів та виключенням повторів;

– Point Crossover – класичний одноточковий кросинговер.

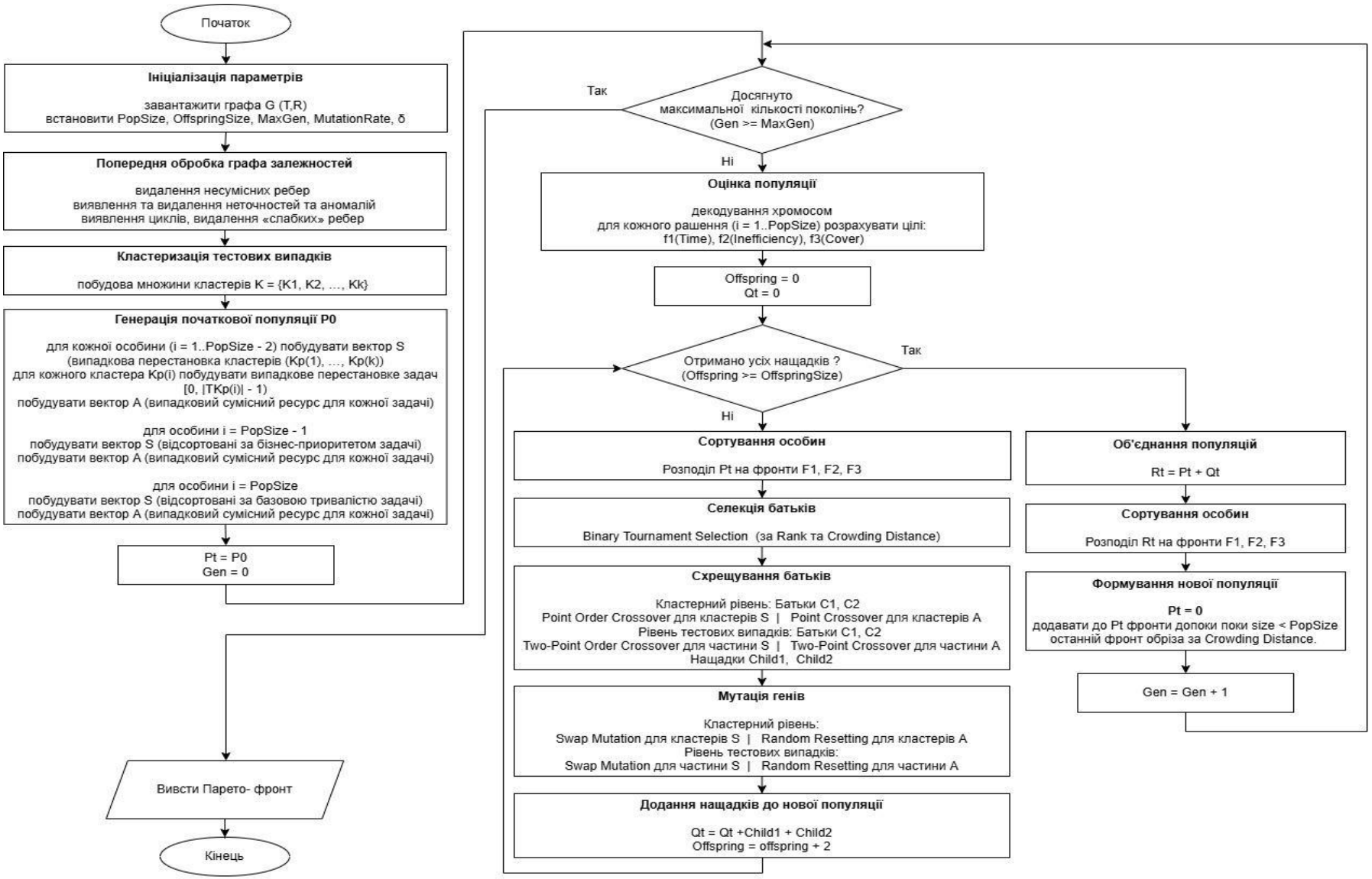


Рис. 3.5 Узагальнена блок-схема алгоритму пошуку оптимальних рішень

– Two-Point Order Crossover – двоточковий кросинговер зі збереженням порядку розташування генів та виключенням повторів;

– Two-Point Crossover – класичний двоточковий кросинговер;

– Swap Mutation – обмінна мутація;

– Random Resetting – мутація випадкового перепризначення;

– f1,f2,f3 – цільові функції (база фітнес значень для алгоритму NSGA-II).

На рисунку 3.6 наведено узагальнену блок-схему алгоритму декодування хромосоми – отримання плану робіт, що закодовано нею.

Розглянемо основні позначання на блок-схемі:

– ResourceFreeTime – список значень часу, коли звільниться кожен з ресурсів (коли тестувальник буде готовий взяти наступну задачу)

– S – вектор послідовності задач;

– A – вектор призначення ресурсів (бажаних) для виконання відповідних задач вектору S;

– r_pref – «бажаний» ресурс для виконання задачі

– r_alter – альтернативний ресурс для виконання задачі (найшвидший що вивільниться);

– Finish() – прогнозований момент часу, коли ресурс завершить виконання задачі

– δ – допустимий зазор часу очікування ресурсу між виконанням завдань;

– Inefficiency – вартість, штраф за непрофільне призначення;

– TaskReadyTime – час готовності задачі, коли саме задача буде готова для виконання (максимум від часу закінчення усіх попередніх залежних задач);

– Start – час, коли ресурс взявся за виконання задачі;

– End – час, коли ресурс закінчив виконання задачі;

– Duration – час, який необхідно для виконання задачі конкретним ресурсом ($d_{t_i}^{base} \cdot f_r$);

– Deadline – глобальне обмеження часу тестування

– ReqCov – множина вже покритих вимог

– Req(t) – множина вимог, які покриваються задачею t;

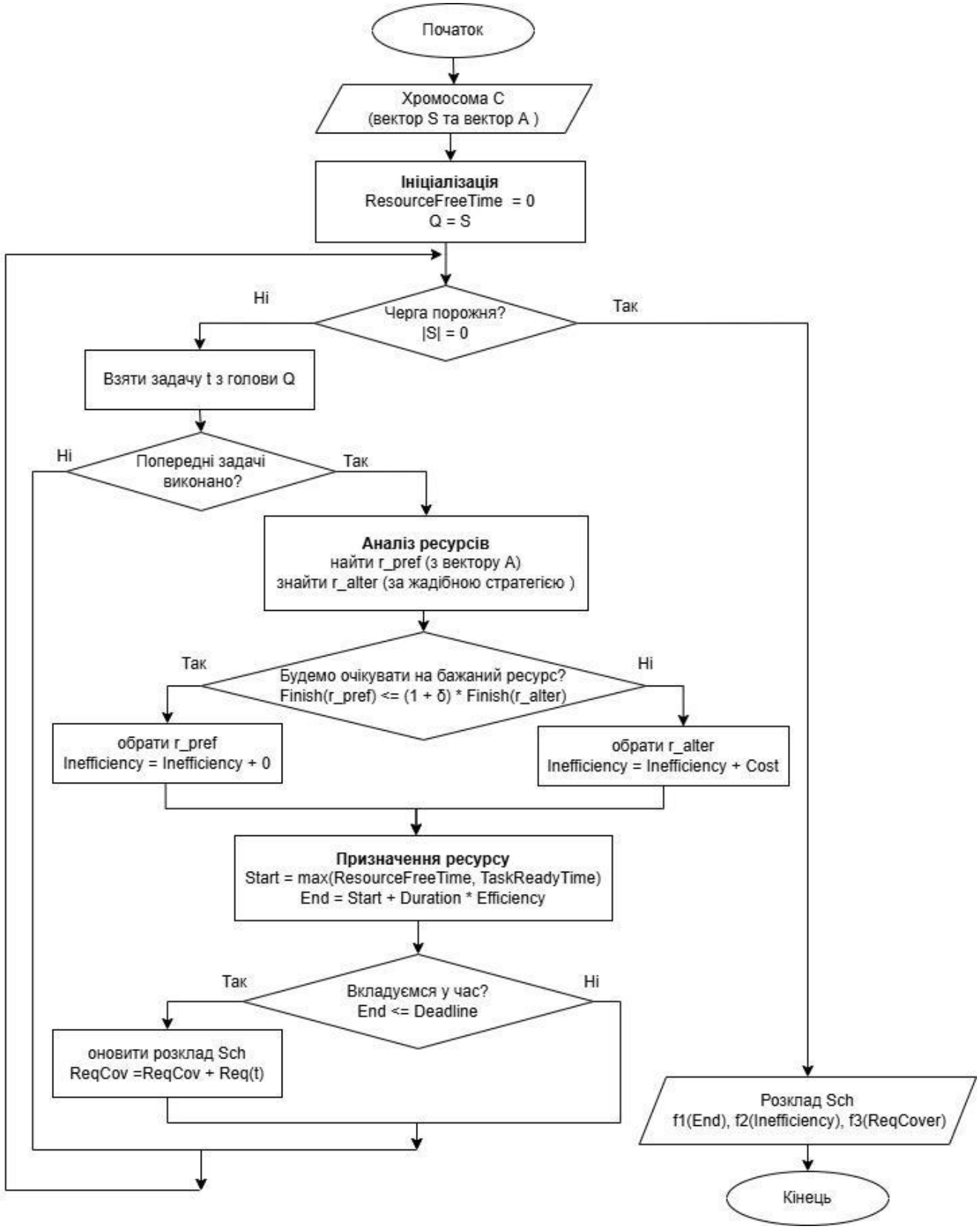


Рис. 3.6 Узагальнена блок-схема декодування рішення

– f1,f2,f3 – цільові функції.

– Sch – план робіт, структура даних (словник), що містить інформацію: хто яку задачу робить (z_{t_i,r_j}), коли починає (Start) та коли закінчує (End).

Таким чином, розроблено програмне забезпечення, що реалізує запропонований гібридний підхід до вирішення багатокритеріальної оптимізаційної задачі розподілу тестових ресурсів. Програмне рішення має графічний інтерфейс користувача, який дозволяє задавати вхідні дані до задачі оптимізації, наглядно демонструвати отриманні результати її вирішення та зберігати їх у базі даних.

4 МОДЕЛЮВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Тестові набори даних для моделювання

Для моделювання роботи запропонованого підходу використовувалися синтезовані тестові дані. Було згенеровано тестові набори даних, що моделюють реальний процес тестування програмного забезпечення. Дані генерувалися за допомогою генератора випадкових чисел генерованих методом random() бібліотеки random мови Python. Для забезпечення відтвореності експериментальних даних використовувалося фіксоване зерно (seed). Фіксоване зерно є початковим значенням для запуску генератора, а його фіксація забезпечує формування однакової послідовності випадкових чисел, незалежно від часу формування та кількості генерацій. У таблиці 4.1 наведено загальні параметри моделі.

Таблиця 4.1

Загальні параметри моделі

Параметр моделі	**Значення та опис**
Кількість тестових випадків (T)	20,120
Кількість вимог (C)	40,240
Кількість тестувальників (R)	6 (3 Juniors, 2 Mids, 1 Senior)
Типи тестових випадків	Routine (40%), Standard (40%), Advanced (20%)
Глибина залежностей тестових випадків	довжина ланцюжків від 5 до 15 тестових випадків
Загальна кількість зв'язків	17

Модель ресурсів (команда тестувальників) складається з 6 ресурсів з індивідуальними коефіцієнтами ефективності роботи. За стандартну ефективність прийнято значення 1, що відповідає базовому часу для виконання тестового випадку:

Таблиця 4.2

Характеристика ресурсів

Ресурс	**Коефіцієнт ефективності (f_{r_j})**	**Вартість неефективності (W_{r_j})**	**Навички ($K_{r_j}^{poss}$)**	**Пріоритет ($K_{r_j}^{pref}$)**
Jun1	1.3 (повільний)	10	Routine	Routine
Jun2	1.4 (повільний)	10	Routine	Routine
Jun3	1.5 (повільний)	10	Routine	Routine
Mid1	1.0 (стандарт)	30	Routine, Standard	Standard
Mid2	0.095 (стандарт)	30	Routine, Standard	Standard
Sen1	0.75 (швидкий)	50	Routine, Standard, Advanced	Advanced

Модель тестових випадків. Генерація тестових випадків виконувалася з урахуванням рівнів складності ($Routine, Standard, Advanced$). Базовий час виконання тестового випадку задається у діапазоні $d_t^{base} \in [1; 16]$ умовних одиниць часу та залежить від його рівня складності:

– $Routine$ – тривалість в межах 1–4 умовних одиниць;

– $Standard$ – в межах 3–8 одиниць;

– $Advanced$ – в межах 6–16 одиниць.

Пріоритетність (бізнес-потреба) тестових випадків визначається за шкалою: $Low = 1, Medium = 20, High = 50, Highest = 100$.

Топологія залежностей між тестовими випадками було сформовано генерацією послідовних ланцюжків з довжиною у діапазоні $L \in [5; 15]$ тестових випадків. Такий підхід дозволяє імітувати проблему блокування ресурсів та їх вимушеного простою.

Модель покриття вимог. Генерація покриття вимог тестовими випадками генерувалася за умовою, що кожен тестовий випадок покриває від 1 до 3 вимог. Вагові коефіцієнти критичності визначалися відповідно до визначених для них типів критичності:

– $Blocked$ – вага 100;

– $Critical$ – вага 50;

– $Major$ – вага 20;

– $Minor$ – вага 5;

– $Trivial$ – вага 1.

Розподіл вимог, які покриваються тестовими випадками, виконується за принципом локальності, тобто, ті тестові випадки, що є близькими за ID частіше покривають сумісні вимоги. Такий підхід до генерації імітує модульну структуру програмного забезпечення, що тестується.

Генерація кількості вимог за кожним типом критичності генерувалась випадковим чином за наступним розподілом ймовірностей:

– $Blocked$ – 5%;

– $Critical$ – 15%;

– $Major$ – 30%;

– $Minor$ – 30%;

– $Trivial$ – 20%.

Такий розподіл ймовірностей моделює ситуацію реальних програмних проектів, де кількість критичних помилок завжди менше, ніж звичайних.

4.2 Оцінка результатів моделювання

З метою визначення якості розподілу тестових ресурсів запропонованим підходом визначимо основні критерії та оцінки [63].

1. Зважене покриття вимог (C_w). Критерій показує ступінь перевірки програмної системи з врахуванням типу критичності кожної вимоги. Значення критерію обчислюємо як відношення суми ваги покритих вимог до суми ваг усіх вимог до програмної системи (4.1):

$$C_w = \frac{\sum_{i \in C_{covered}} s_i}{\sum_{j \in C_{all}} s_j} \tag{4.1}$$

де: $C_{covered}$ – множина покритих вимог;

C_{all} – множина усіх вимог

s_i, s_j – вага важливості (критичності) відповідної вимоги ($Blocked$ = 100, $Critical$ = 50 і т.д.).

2. Загальна тривалість тестування (T). Критерій показує час завершення всього тестового циклу. Він відповідає часу завершення виконання тестовим ресурсом останнього тестового випадку (4.2):

$$T = \max_{r \in R} finish\_time(r), \tag{4.2}$$

де: R – множина доступних ресурсів (тестувальників);

r – ресурс (тестувальник);

$finish\_time(r)$ – час, коли ресурс завершив виконання всіх призначених тестів.

3. Завантаження ресурсів (U_w). Критерій показує середню ступінь використання робочого часу кожного ресурсу (члена команди тестувальників). Він відповідає усередненому значенню коефіцієнтів завантаження кожного ресурсу окремо (4.3):

$$U_w = \frac{1}{|R|} \sum_{r \in R} \frac{T_r^{load}}{T_r^{avail}}, \tag{4.3}$$

де: R – множина доступних ресурсів (тестувальників);

r – ресурс (тестувальник);

$|R|$ – загальна кількість ресурсів;

T_r^{load} – фактичний час, коли ресурс r був зайнятий;

T_r^{avail} – увесь доступний робочий час ресурсу r.

В рамках даного моделювання вважаємо, що доступний фонд часу ресурсу відповідає загальній тривалості тестування проекту. У реальних випадках тестування програмного забезпечення цей час відповідає робочому часу тестувальника та може відрізнятися для різних тестувальників.

4. Частка непрофільних призначень. Критерій показує відповідності тестових випадків спеціалізації ресурсів. Коли тестувальник виконує тестовий випадок, який не входить в перелік його пріоритетних тестових випадків, то це вважається «непрофільним призначенням» та штрафується. Значення критерію відповідає відношенню кількості тестових випадків, на які було призначено тестувальників не за профілем, до загальної кількості тестових випадків (4.4):

$$A = \frac{N_{non\_prof}}{N_{total}}, \tag{4.4}$$

де: N_{non_prof} – кількість призначень для тестувальників не за профілем;

N_{total} – загальна кількість призначень.

5. Інтегральна оцінка $F(S)$. Задача, що вирішується є багатокритеріальною. Кожне її рішення характеризується трьома значеннями різної природи, які не можуть бути порівняні між собою. Для представлення рішення у вигляді одного комплексного показника будемо використовувати інтегральний показник. Формування такого показника будемо виконувати методом адитивної згортки нормованих показників (4.5):

$$F(S) = w_C C^{norm} + w_T T^{norm} + w_U U^{norm} + w_A A^{norm}, \tag{4.5}$$

де w_C, w_T, w_U, w_A – вагові коефіцієнти, що показують важливість певного критерію у єдиному показнику: $w_C + w_T + w_U + w_A = 1$;

C^{norm}, T^{norm}, U^{norm}, A^{norm} - нормовані значення критеріїв.

Такий підхід дозволяє виконувати ранжування отриманих рішень за єдиною шкалою якості.

Проведемо експерименти по розподіленню тестових ресурсів та порівняємо їх з точним рішенням пошуку яке надає бібліотека Google OR-Tools. Оскільки генетичний алгоритм надає Парето-фронт рішень, для порівняння з єдиним рішенням, яке надає OR-Tools оберемо краще за часом рішення, надане генетичним

алгоритмом. Для розрахунку інтегральної оцінки візьмемо наступні значення вагових коефіцієнтів: $w_C = 0.4, w_T = 0.3, w_U = 0.2, w_A = 0.1$.

Перший експеримент проводився для 20 тестових випадків та 40 вимог. Розмір популяції батьків та нащадків дорівнює 50 особин. Результати експерименту для кількості поколінь 25, 50 та 100 наведено у таблицях 4.3 – 4.5 відповідно.

Таблиця 4.3

Результати першого експерименту для кількості поколінь = 25

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	81.8	60.0	58.0	-29.1%	-3.4%
Inefficiency (F2)	510.0	140.0	130.0	-74.5%	-7.1%
Cover (F3)	382	382	382	0.0%	0.0%
C_w	0.430	0.430	0.430	0.0%	0.0%
U_w	21.4%	32.7%	33.6%	+56.7%	+2.7%
A, помилки	0.450 (11)	0.800 (4)	0.850 (3)	-72.7%	-25.0%
F(S)	0.3145	0.4372	0.4502	+43.2%	+3.0%

Таблиця 4.4

Результати першого експерименту для кількості поколінь = 50

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	81.8	60.0	57.5	-29.6%	-4.1%
Inefficiency (F2)	510.0	140.0	180.0	-64.7%	+28.6%
Cover (F3)	382	382	382	0.0%	0.0%
C_w	0.430	0.430	0.430	0.0%	0.0%
U_w	21.4%	32.7%	33.5%	+56.4%	+2.5%
A, помилки	0.450 (11)	0.800 (4)	0.800 (4)	-63.6%	0.0%
F(S)	0.3145	0.4372	0.4462	+41.9%	+2.1%

Таблиця 4.5

Результати першого експерименту для кількості поколінь = 100

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	81.8	60.0	57.4	-29.8%	-4.4%
Inefficiency (F2)	510.0	140.0	200.0	-60.8%	+42.9%
Cover (F3)	382	382	382	+0.0%	+0.0%
C_w	0.430	0.430	0.430	0.0%	0.0%
U_w	21.4%	32.7%	34.3%	+60.2%	+5.0%
A, помилки	0.450 (11)	0.800 (4)	0.800 (4)	-63.6%	+0.0%
F(S)	0.3145	0.4372	0.4484	+42.6%	+2.6%

Другий експеримент проводився для 120 тестових випадків та 240 вимог. Розмір популяцій батьків та нащадків дорівнює 50 особин. Результати експерименту для кількості поколінь 25, 50, 100 та 150 наведено у таблицях 4.6 – 4.9 відповідно.

Таблиця 4.6

Результати другого експерименту для кількості поколінь = 25

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	346.8	231.0	300.1	*-13.5%*	*+29.9%*
Inefficiency (F2)	2100.0	620.0	910.0	-56.7%	+46.8%
Cover (F3)	2402	2458	2458	+2.3%	+0.0%
C_w	0.5120	0.524	0.524	+2.3%	0.0%
U_w	26.4%	45.8%	32.5%	+33.2%	-23.3%
A, помилки	0.589 (46)	0.833 (20)	0.825 (21)	-54.3%	+5.0%
F(S)	0.3193	0.4865	0.4052	+26.9%	-16.7%

Таблиця 4.7

Результати другого експерименту для кількості поколінь = 50

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	346.8	231.0	271.1	*-21.8%*	*+17.4%*
Inefficiency (F2)	2100.0	620.0	640.0	*-69.5%*	*+3.2%*
Cover (F3)	2402	2458	2458	*+2.3%*	*+0.0%*
C_w	0.5120	0.524	0.524	*+2.3%*	*+0.0%*
U_w	26.4%	45.8%	39.4%	*+49.2%*	*-14.0%*
A, помилки	0.589 (46)	0.833 (20)	0.867 (16)	*-65.2%*	*-20.0%*
F(S)	0.3193	0.4865	0.4427	*+38.6%*	*-9.0%*

Таблиця 4.8

Результати другого експерименту для кількості поколінь = 100

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	346.8	231.0	244.0	-29.6%	+5.6%
Inefficiency (F2)	2100.0	620.0	500.0	-76.2%	-19.4%
Cover (F3)	2402	2458	2458	+2.3%	+0.0%
C_w	0.5120	0.524	0.524	+2.3%	+0.0%
U_w	26.4%	45.8%	43.9%	+66.5%	-4.1%
A, помилки	0.589 (46)	0.833 (20)	0.883 (14)	-69.6%	-30.0%
F(S)	0.3193	0.4865	0.4766	+49.3%	-2.0%

Таблиця 4.9

Результати другого експерименту для кількості поколінь = 150

Метрики	**Базовий план**	**OR-Tools**	**Гібридний метод**	**Відносне покращення**	
				Базовий/ Гібридний	**OR/ Гібридний**
Time (F1)	346.8	231.0	241.5	-30.4%	+4.5%
Inefficiency (F2)	2100.0	620.0	350.0	-83.3%	-43.5%
Cover (F3)	2402	2458	2458	+2.3%	+0.0%
C_w	0.5120	0.524	0.524	+2.3%	+0.0%
U_w	26.4%	45.8%	44.8%	+69.5%	-2.3%
A, помилки	0.589 (46)	0.833 (20)	0.925 (9)	-80.4%	-55.0%
F(S)	0.3193	0.4865	0.4846	+51.8%	-0.4%

За результатами проведених експериментів можна зробити наступні висновки:

1) отримані гібридним генетичним алгоритмом результати демонструють позитивний вплив збільшення кількості поколінь пошуку рішення на оцінки критеріїв;

2) зменшення часу проведення тестування збільшує його вартість, що в свою чергу зменшує інтегральну оцінку рішення;

3) зі скороченням часу тестування збільшується завантаженість ресурсів;

4) покриття вимог не змінюється, що обумовлено відсутністю обмеження на час проведення тестування (дедлайни);

5) час пошуку рішення гібридним генетичним алгоритмом значно більший за час пошуку рішення за Базовим планом та інструментарієм OR-Tools.

В порівнянні з базовим рішенням та точним рішенням OR-Tools можна зробити висновок, що обрані найкоротші за часом виконання розподіли мають вищу вартість та, як наслідок, нижчу інтегральну оцінку. Це обумовлено багатокритеріальністю задачі розподілу в порівнянні з однокритеріальністю рішення від OR-Tools або Базового плану. Такі рішення не беруть до уваги ні ступінь покриття ні вартість отриманого плану тестування. Рішення за генетичним алгоритмом, навпаки, намагаючись знайти оптимальне рішення не жертвують вартістю або покриттям заради часу.

На рисунку 4.1 наведено приклад оцінок отриманих рішень за першим експериментом (кількості поколінь = 100) у двох аспектах: Парето-фронт найкращих рішень та найкращі рішення за інтегральною оцінкою.

На рисунку 4.2 наведено склад хромосом отриманих рішень відповідно до оцінок, наведених на рис.1.

На рисунку 4.3 наведено два перші оптимальних плани тестування, отриманих відповідно до Парето-фронту, наведеного на рис. 4.1.

ЕТАП 1: КРАЩІ РІШЕННЯ [10]

ID	Integral	F1(Time)	F2(Ineff)	F3(Cover)	K1 Cov(Raw)	K2 Time(Raw)	K3 Util(Raw)	K4 Prof(Raw)
0	0.4622	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
1	0.4484	57.4	200.0	382	0.43 (382)	0.43 (57.4)	0.34 (34%)	0.80 (4 err)
2	0.4622	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
3	0.4569	58.0	80.0	382	0.43 (382)	0.42 (58.0)	0.34 (34%)	0.90 (2 err)
4	0.4600	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)
5	0.4569	58.0	80.0	382	0.43 (382)	0.42 (58.0)	0.34 (34%)	0.90 (2 err)
6	0.4532	57.5	150.0	382	0.43 (382)	0.42 (57.5)	0.35 (35%)	0.85 (3 err)
7	0.4598	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)
8	0.4532	57.5	150.0	382	0.43 (382)	0.42 (57.5)	0.35 (35%)	0.85 (3 err)
9	0.4598	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)

ЕТАП 2: КРАЩІ РІШЕННЯ [10] ЗА ІНТЕГРАЛЬНОЮ ОЦІНКОЮ

ID	Integral	F1(Time)	F2(Ineff)	F3(Cover)	K1 Cov(Raw)	K2 Time(Raw)	K3 Util(Raw)	K4 Prof(Raw)
0	0.4622	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
2	0.4622	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
44	0.4622	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
47	0.4622	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
11	0.4621	59.4	0.0	382	0.43 (382)	0.41 (59.4)	0.34 (34%)	1.00 (0 err)
4	0.4600	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)
14	0.4600	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)
19	0.4600	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)
43	0.4600	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)
46	0.4600	58.6	30.0	382	0.43 (382)	0.41 (58.6)	0.34 (34%)	0.95 (1 err)

Рис. 4.1 Оптимальні рішення за Парето-фронтом та інтегральною оцінкою.

Chromosome	
[19 13 17 15 12 10 9 5 14 7 6 18 1 3 2 11 8 16 4 0]	[4 0 3 0 3 5 4 5 5 0 5 3 5 5 5 4 4 1 3 4]
[15 9 3 0 17 14 2 10 7 18 13 6 5 1 19 11 8 16 4 12]	[4 0 3 0 3 5 4 5 5 0 5 3 5 5 5 4 5 5 3 4]
[13 5 17 19 0 15 10 9 14 7 6 18 1 3 2 11 8 16 12 4]	[4 0 3 0 3 5 4 5 2 0 5 3 5 5 5 4 4 1 3 4]
[13 5 17 19 3 14 0 7 18 15 6 10 1 9 2 11 8 16 4 12]	[4 3 3 1 5 5 3 5 5 0 5 3 5 5 5 5 5 0 3 4]
[13 10 5 15 17 19 3 14 7 6 18 1 9 11 2 8 16 12 4 0]	[4 5 3 2 3 5 4 5 5 0 5 3 5 5 5 4 4 0 3 4]
[13 18 17 19 3 14 0 7 5 15 6 10 1 9 2 11 8 16 4 12]	[4 3 3 0 3 5 3 5 5 0 5 3 5 5 5 4 5 0 3 4]
[6 15 17 3 0 9 10 2 7 18 13 14 5 1 19 11 8 16 12 4]	[4 0 3 0 3 5 4 5 5 0 5 3 5 5 5 4 5 0 3 4]
[13 10 15 5 17 3 19 14 7 6 0 18 1 9 11 2 8 16 4 12]	[3 0 3 0 3 5 4 5 0 0 5 3 5 5 5 4 4 0 3 4]
[15 3 9 0 17 14 2 10 18 7 13 6 5 1 19 11 8 16 12 4]	[4 0 3 0 3 5 4 5 5 0 5 3 5 5 5 4 5 0 3 4]
[0 13 19 14 17 3 15 7 10 6 18 1 9 5 2 11 8 16 12 4]	[3 0 3 0 4 5 3 4 3 0 5 3 5 5 5 4 4 0 3 4]

Chromosome	
[19 13 17 15 12 10 9 5 14 7 6 18 1 3 2 11 8 16 4 0]	[4 0 3 0 3 5 4 5 5 0 5 3 5 5 5 4 4 1 3 4]
[13 5 17 19 0 15 10 9 14 7 6 18 1 3 2 11 8 16 12 4]	[4 0 3 0 3 5 4 5 2 0 5 3 5 5 5 4 4 1 3 4]
[13 15 10 9 17 19 14 7 6 0 5 18 1 3 2 11 8 16 12 4]	[4 0 4 0 4 5 4 5 3 0 5 3 5 5 5 4 4 1 3 4]
[13 15 10 0 17 14 3 19 7 6 5 18 1 9 2 11 8 16 12 4]	[4 0 4 0 4 5 4 5 3 0 5 3 5 5 5 4 4 1 3 4]
[13 10 5 15 17 3 14 7 6 18 1 9 2 0 19 11 8 16 12 4]	[4 0 3 0 3 5 4 5 5 0 5 3 5 5 5 4 4 0 3 4]
[13 10 5 15 17 19 3 14 7 6 18 1 9 11 2 8 16 12 4 0]	[4 5 3 2 3 5 4 5 5 0 5 3 5 5 5 4 4 0 3 4]
[13 5 17 15 10 19 3 14 7 6 18 1 9 2 11 8 16 12 4 0]	[4 5 3 2 3 5 4 5 5 0 5 3 5 5 5 4 4 0 3 4]
[13 10 5 15 17 19 3 14 7 6 18 1 9 2 11 8 16 12 4 0]	[3 5 3 2 3 5 3 5 5 0 5 3 5 5 5 4 4 0 3 4]
[13 10 5 15 17 19 3 14 7 6 18 1 9 2 11 8 16 12 4 0]	[4 5 3 2 3 5 4 5 5 0 5 3 5 5 5 4 4 0 3 4]
[0 13 17 15 10 19 3 14 7 6 18 1 9 5 2 11 8 16 12 4]	[4 5 3 2 4 5 4 5 5 0 5 3 5 5 5 4 4 0 3 4]

Рис. 4.2 Склад хромосом отриманих рішень

Місце #1 (ID: 0), Integral: 0.4622
  K1(Cov): 0.430 (382), K2(Time): 0.406 (59.4)
  K3(Util): 0.342 (34.2%), K4(Prof): 1.000 (0 err)

Task	Res	Start	End
Verify Catalog_13	Sen1	0.0	10.5
Verify Payment_15	Mid2	0.0	3.8
Audit DB_6	Mid1	0.0	8.0
Test Catalog_7	Mid2	3.8	6.6
Debug API_3	Jun1	6.6	9.2
Test Payment_19	Mid2	8.0	15.6
Debug API_1	Jun1	9.2	14.4
Audit Profile_8	Jun3	10.5	16.5
Test Search_11	Mid1	14.4	21.4
Check UI_17	Jun2	15.6	18.4
Debug Cart_14	Sen1	18.4	28.9
Verify UI_9	Jun1	21.4	26.6
Debug Cart_0	Mid2	26.6	29.5
Verify Search_4	Mid1	28.9	31.9
Validate Catalog_5	Sen1	29.5	41.5
Verify Payment_18	Mid1	31.9	37.9
Debug Payment_2	Mid1	37.9	45.9
Verify Auth_12	Sen1	41.5	49.0
Test Auth_16	Mid2	49.0	51.9
Debug Export_10	Sen1	51.9	59.4

Місце #2 (ID: 1), Integral: 0.4484
  K1(Cov): 0.430 (382), K2(Time): 0.426 (57.4)
  K3(Util): 0.343 (34.3%), K4(Prof): 0.800 (4 err)

Task	Res	Start	End
Verify Payment_15	Sen1	0.0	3.0
Audit DB_6	Mid2	0.0	7.6
Test Catalog_7	Sen1	3.0	5.2
Debug API_3	Jun1	5.2	7.8
Verify Catalog_13	Sen1	5.2	15.8
Test Payment_19	Mid2	7.6	15.2
Debug API_1	Jun1	7.8	13.1
Test Search_11	Mid1	13.1	20.1
Check UI_17	Sen1	15.8	17.2
Audit Profile_8	Jun3	15.8	21.8
Debug Cart_14	Sen1	17.2	27.8
Verify UI_9	Jun1	20.1	25.2
Debug Cart_0	Mid2	25.2	28.1
Verify Search_4	Mid1	27.8	30.8
Validate Catalog_5	Sen1	28.1	40.1
Verify Payment_18	Mid1	30.8	36.8
Debug Payment_2	Mid1	36.8	44.8
Verify Auth_12	Sen1	40.1	47.6
Test Auth_16	Sen1	47.6	49.9
Debug Export_10	Sen1	49.9	57.4

Рис. 4.3 Оптимальні плани за Парето-фронтом

ВИСНОВКИ

1. Проведено огляд існуючих підходів до розподілу тестових ресурсів та методів, що використовуються для вирішення задач оптимізації їх розподілу в умовах багатоцільових обмежень.

2. Розроблено математичну модель оптимізації розподілу тестових ресурсів, яка базується на графовій структурі та формалізує обмеження задачі (кваліфікація фахівців та топологічні залежності тестових випадків).

3. Розроблено гібридний підхід, який поєднує графовий аналіз для попереднього очищення попередніх призначень та врахування обмежень, з методами багатоцільової оптимізації на основі генетичного алгоритму, який дозволяє знайти практичні рішення у множині варіантів призначення ресурсів.

4. Спроектовано архітектуру програмного забезпечення для розподілу тестових ресурсів, обрано технології та засоби реалізації компонентів системи. Для створення програмного рішення обрано середовище розробки MS Visual Studio 2022 та мова програмування C#, для побудови графової моделі задачі – Net бібліотека роботи з графами QuickGraph, для реалізації гібридного алгоритму Python бібліотека Pymoo та технологія WPF для інтерфесу користувача.

5. Розроблено програмне забезпечення гібридного підходу розподілу тестових ресурсів для процесу тестування програмного забезпечення.

6. Проведене моделювання роботи розробленого програмного забезпечення показало, що запропонований гібридний підхід формує кращі плани тестування порівняно з розкладом наданим точним рішенням OR-Tools та евристичним базовим розкладом, дозволяє скоротити час тестування приблизно на 30% порівняно з базовим планом та зменшити кількість непрофільних призначень на 60-80%. Збільшення кількості популяції та збільшення кількості поколінь має позитивний вплив пошуку рішення на оцінки критеріїв. Збільшення популяції у двічі збільшує завантаженість ресурсів у середньому на 6% та додатково покращує інтегральну оцінку рішення у середньому на 3%, однак майже вдвічі збільшує тривалість пошуку рішення. За рахунок рішення задачі багатокритеріальної

оптимізації пропонований підхід може надавати більш вартісні та гірші за інтегральною оцінкою рішення у порівнянні OR-Tools, що не є недоліком, а лише наслідком формування певного фронту рішень. При цьому час пошуку оптимальних рішень значно збільшується, що компенсується наданням не одного, а множини оптимальних рішень.

Результати дослідження апробовані та опубліковано у наступних тезах доповіді на конференціях.

1. Щербина А.С., Золотухіна О.А. Гібридний підхід до багатоцільової оптимізації розподілу тестових ресурсів ІТ-проекту за допомогою графової моделі оптимізації. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.605-607.

2. Щербина А.С., Соляник Л.О., Оптимізаційна графова модель розподілу тестових ресурсів ІТ-проекту II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії», 26 листопада 2025 р, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.255-257.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kulikov S. Software testing. Base course. 3rd edition. Book version 3.2.6. 2024, [Електронний ресурс]. – URL: https://www.linkedin.com/redir/redirect?url=https%3A%2F%2Fsvyatoslav%2Ebiz%2Fsoftware_testing_book_download_en%2F&urlhash=LuXb&trk=article-ssr-frontend-pulse_little-text-block.

2. Glenford J., The art of software testing – 3rd ed. 2012. [Електронний ресурс]. URL: https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf.

3. Ferrara P., Arceri V., Cortesi A. Challenges of software verification: the past, the present, the future. International Journal on Software Tools for Technology Transfer, 2024. [Електронний ресурс]. URL: https://link.springer.com/content/pdf/10.1007/s10009-024-00765-y.pdf.

4. Arceri, V., Negrini, L., Olivieri, L. Challenges of software verification. Int J Softw Tools Technol Transfer, 2024. P.669–672 . [Електронний ресурс]. URL: https://link.springer.com/article/10.1007/s10009-024-00778-7.

5. Arumugam A. Software Testing Techniques New Trends. International Journal of Engineering Research and. V8, 2020. [Електронний ресурс]. URL: https://www.ijert.org/research/software-testing-techniques-new-trends-IJERTV8IS120318.pdf .

6. Khaliq Z. Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect. 2022. [Електронний ресурс]. URL: https://www.researchgate.net/publication/357876318_Artificial_Intelligence_in_Software_Testing_Impact_Problems_Challenges_and_Prospect

7. Данилевич Н., Коповський С. Методи тестування програмного забезпечення штучним інтелектом. Efektyvna ekonomika, 2025. [Електронний ресурс]. URL: https://www.nayka.com.ua/index.php/ee/article/view/5578/5634.

8. Tahvili S., Hatvani L. Artificial Intelligence Methods for Optimization of the Software Testing Process: With Practical Examples and Exercises, 2022. [Електронний

ресурс]. URL: https://www.researchgate.net/publication/362325899_Artificial_Intelligence_Methods_for_Optimization_of_the_Software_Testing_Process_With_Practical_Examples_and_Exercises.

9. Chinamanagonda S. AI-driven Performance Testing AI tools enhancing the accuracy and efficiency of performance testing. Advances in Computer Sciences, 2021. [Електронний ресурс]. URL: https://acadexpinnara.com/index.php/acs/article/view/337/359.

10. Anwar R., Bashir M. B. A Systematic Literature Review of AI-Based Software Requirements Prioritization Techniques. in IEEE Access, vol. 11, 2023. p. 143815-143860. [Електронний ресурс]. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10360116.

11. Khatibsyarbini M. Trend Application of Machine Learning in Test Case Prioritization: A Review on Techniques. in IEEE Access, vol. 9, 2021. p. 166262-166282. [Електронний ресурс]. URL: https://ieeexplore.ieee.org/abstract/ document/9650902

12. 8 Pietrantuono R., On the testing resource allocation problem: Research trends and perspectives, Journal of Systems and Software, vol.161, 2020. [Електронний ресурс]. URL: https://sci-hub.se/https://doi.org/10.1016/j.jss.2019.110462.

13. 9 Nasar M., Johri P., Chanda U., Software Testing Resource Allocation and Release Time Problem: A Review, 2014. [Електронний ресурс]. URL: https://www.mecs-press.org/ijmecs/ijmecs-v6-n2/IJMECS-V6-N2-7.pdf.

14. Zhang G., Li L., Su Z., Shao Z. New Reliability-Driven Bounds for Architecture-Based Multi-Objective Testing Resource Allocation. IEEE Transactions on Software Engineering, 2023. p. 2513-2529. [Електронний ресурс]. URL: https://ieeexplore.ieee.org/document/9960830.

15. Eberhart C., Yamada A., Klikovits Stefan Architecture-Guided Test Resource Allocation Via Logic, 2021. [Електронний ресурс]. URL: https://arxiv.org/pdf/2107.10948.

16. Hiroyuki O., Tadashi D. Optimizing Testing-Resource Allocation Using Architecture-Based Software Reliability Model, Journal of Optimization, vol.2018 ,

2018. pp.1-7. [Електронний ресурс]. URL: https://onlinelibrary.wiley.com/doi/epdf/10.1155/2018/6948656.

17. Anand A., Das S., Kumar V. Resource Allocation Problem for Multi Versions of Software System, 2019. [Електронний ресурс]. URL: https://sci-hub.se/http://dx.doi.org/10.1109/AICAI.2019.8701380.

18. Rubina M., Rajat A., Testing Resource Allocation for Software System: An Approach Integrating MEMV-OWA and DEMATEL, Optimization Models in Software Reliability, 2022. p.215. [Електронний ресурс]. URL: https://ideas.repec.org/h/spr/ssrchp/978-3-030-78919-0_10.html.

19. Kumar M. Management of Optimal Resource Allocation in the Cloud. International Journal of Computer Applications, 2023. p.20-24. [Електронний ресурс]. URL: https://www.ijcaonline.org/archives/volume185/number25/kumar-2023-ijca-923006.pdf.

20. Katoh, N., Shioura, A., Ibaraki, T. Resource Allocation Problems. Handbook of Combinatorial Optimization. Springer, New York, NY, 1998. [Електронний ресурс]. URL: https://sci-hub.se/https://link.springer.com/chapter/10.1007/978-1-4613-0303-9_14.

21. Lo J.-H. Optimal resource allocation and sensitivity analysis for modular software testing, 2023. [Електронний ресурс]. URL: 0https://www.academia.edu/7482768/Optimal_resource_allocation_and_sensitivity_analysis_for_modular_software_testing.

22. Huang C.-Y., Lyu M. Optimal Testing Resource Allocation, and Sensitivity Analysis in Software Development IEEE Transactions on. 54., 2006. p.592 - 603. [Електронний ресурс]. URL: https://www.researchgate.net/profile/Michael-Lyu/publication/3152811_Optimal_Testing_Resource_Allocation_and_Sensitivity_Analysis_in_Software_Development/links/546de3540cf26e95bc3d138c/Optimal-Testing-Resource-Allocation-and-Sensitivity-Analysis-in-Software-Development.pdf.

23. Huang C.-Y., Lo J.-H., Kuo S.-Y., Lyu M. R. Optimal allocation of testing-resource considering cost, reliability, and testing-effort, 2004. [Електронний ресурс]. URL: https://www.cse.cuhk.edu.hk/~lyu/paper_pdf/PRDC_2004_v2.pdf.

24. Xiao X., Dohi T., Okamura H., Optimal software testing-resource allocation with operational profile: computational aspects, Life Cycle Reliability and Safety Engineering, vol.7, no.4, 2018. p.269. [Електронний ресурс]. URL: https://sci-hub.se/10.1007/s41872-018-0060-x

25. Khan M. G. M., Ahmad N., Rafi L. S., Determining the optimal allocation of testing resource for modular software system using dynamic programming, Communications in Statistics - Theory and Methods, vol.45, no.3, 2016. p. 670-694. [Електронний ресурс]. URL: https://sci-hub.se/https://doi.org/10.1080/03610926.2013.834455.

26. Tahvili S. Multi-Criteria Optimization of System Integration Testing, 2018. [Електронний ресурс]. URL: https://www.diva-portal.org/smash/get/diva2:1283045/FULLTEXT01.pdf.

27. Arora R., Mittal R., Aggarwal A. Optimal Testing Resource Allocation for Software System: An approach combining Interval Valued Intuitionistic Fuzzy Sets and AHP. International Journal of Operational Research, 2022. [Електронний ресурс]. URL: http://dx.doi.org/10.1504/IJOR.2022.10046600.

28. Чабанюк Я., Кукурба В., Гнатів Л., Будз І., Петрович Р Оптимізація моделі тестування програмного забезпечення з показником величини проекту, SCSIT, Volume 694, 2011. p. 226 – 232. [Електронний ресурс]. URL: https://science.lpnu.ua/ scsit/all-volumes-and-issues/volume-694-2011/ optimizaciya-modeli-testuvannya-programnogo.

29. Shahzad B., Amin F., Abro A. Resource Optimization-Based Software Risk Reduction Model for Large-Scale Application Development. Sustainability, 2021. [Електронний ресурс]. URL: https://www.researchgate.net/publication/349716477_Resource_Optimization-Based_Software_Risk_Reduction_Model_for_Large-Scale_Application_Development.

30. Zhang G., Li L., Su Z., Yue F., Chen Y., Li M., Yao X.. On Estimating the Feasible Solution Space of Multi-objective Testing Resource Allocation. ACM Trans. Softw. Eng. Methodol. 33, 6, Article 145, 41p., 2024. [Електронний ресурс]. URL: https://doi.org/10.1145/3654444.

31. Su Z., Zhang G., Yue F., Zhan D., Li M., Li B., Yao X. Enhanced Constraint Handling for Reliability-Constrained Multi-objective Testing Resource Allocation. IEEE Transactions on Evolutionary Computation, 2021. [Електронний ресурс]. URL: https://sci-hub.se/http://dx.doi.org/10.1109/TEVC.2021.3055538.

32. Rani P., Mahapatra, G.S. Entropy based enhanced particle swarm optimization on multi-objective software reliability modelling for optimal testing resources allocation. Software Testing, Verification and Reliability, 2021. [Електронний ресурс]. URL: https://sci-hub.se/http://dx.doi.org/10.1002/stvr.1765.

33. Inoue, S., Minamino, Y., Yamada, S. Multi-criteria Decision Making in Optimal Software Testing-Allocation Problem. In: Aggarwal, A.G., Tandon, A., Pham, H. (eds) Optimization Models in Software Reliability. Springer Series in Reliability Engineering. Springer, Cham, 2022. [Електронний ресурс]. URL: https://doi.org/10.1007/978-3-030-78919-0_4.

34. Pan A., Shen B., Wang L. Ensemble of resource allocation strategies in decision and objective spaces for multiobjective optimization, Information Sciences, Volume 605, 2022, p. 393-412. [Електронний ресурс]. URL: https://doi.org/10.1016/j.ins.2022.05.005.

35. Jothi K., Lanjewar C., Jain L., Ramesh J., Singh P. Detecting Local Software Issues Using NSGA Multi-optimization, 2024. [Електронний ресурс]. URL: http://dx.doi.org/10.1007/978-3-031-73494-6_8.

36. Zhang G., Su Z., Li M., Yue F., Jiang J., Yao X. Constraint Handling in NSGA-II for Solving Optimal Testing Resource Allocation Problems. IEEE Transactions on Reliability, 2017. p.1-20. [Електронний ресурс]. URL: http://agent.hfut.edu.cn/_upload/article/files/68/b8/8bf3d471418a911c6e6a9b3e95ba/458a823d-450d-4c6a-ba8a-ab3aaaa364e9.pdf.

37. Su Z. Enhanced Constraint Handling for Reliability-Constrained Multiobjective Testing Resource Allocation. in IEEE Transactions on Evolutionary Computation, vol. 25, no. 3, 2021. p. 537-551. [Електронний ресурс]. URL: https://sci-hub.se/https://doi.org/10.1109/TEVC.2021.3055538.

38. Raheem A., Akthar S. Software Reliability Growth Models with Exponentiated-gompertz Testing Effort and Release Time Determination. International Journal of Computer Network and Information Security. 2023. [Електронний ресурс]. URL: https://www.mecs-press.org/ijcnis/ijcnis-v15-n1/IJCNIS-V15-N1-7.pdf.

39. Zhang G. New Reliability-Driven Bounds for Architecture-Based Multi-Objective Testing Resource Allocation. in IEEE Transactions on Software Engineering, vol. 49, no. 4, 2023. p. 2513-2529. [Електронний ресурс]. URL: https://doi.org/10.1109/TSE.2022.3223875.

40. Гамільтон Т. 14 найкращих інструментів керування тестами (2025). Guru99. [Електронний ресурс]. URL: https://www.guru99.com/uk/top-20-test-management-tools.html.

41. Vashchenko S. Best Test Management Tools In 2024. Svitla. [Електронний ресурс]. URL: https://svitla.com/blog/test-management-tools/.

42. Популярні інструменти для автоматизованого тестування. Varosh, 2025. [Електронний ресурс]. URL: https://varosh.com.ua/adv/populyarni-instrumenty-dlya-avtomatyzovanogo-testuvannya/.

43. Mohapatro S.K. 20 Best Automation Testing Tools to Know.Headspin, 2025. [Електронний ресурс]. URL: https://www.headspin.io/blog/the-ultimate-list-of-automated-testing-tools.

44. Araujo S. Test Environment Management Tools: Best Solution. ApwideGOLIVE, 2025. [Електронний ресурс]. URL: https://www.apwide.com /test-environment-management-tools-the-best-solution/.

45. Top 10 Code Coverage Tools Every Developer Should Know. Hypertest, 2024. [Електронний ресурс]. URL: https://www.hypertest.co/software-testing/top-10-code-coverage-tools.

46. 5 Best Test Coverage Tools for 2024. Appsurify, 2024. [Електронний ресурс]. URL: https://appsurify.com/resources/5-best-test-coverage-tools-for-2024/.

47. 20+ найкращих інструментів, програмного забезпечення та рішень для CI/CD на 2025 рік. Visure, 2025. [Електронний ресурс]. URL: https://visuresolutions.com/uk/alm-guide/%D0%BD%D0%B0%D0%B9%D0%BA

%D1%80%D0%B0%D1%89%D1%96-%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B8-ci-cd-%D0%BD%D0%B0-2023-%D1%80%D1%96%D0%BA/.

48. Dinu F., Ninawe S. 20+ Best CI/CD Tools for DevOps in 2025. Spacelift. 2025. [Електронний ресурс]. URL: https://spacelift.io/blog/ci-cd-tools.

49. Топ 20+ найкращих інструментів управління вимогами (повний список). [Електронний ресурс]. URL: https://uk.myservername.com/top-20-best-requirements-management-tools.

50. 9 НАЙКРАЩИХ інструментів керування вимогами (2025). [Електронний ресурс]. URL: https://www.guru99.com/uk/requirement-management-tools.html.

51. Золотухіна О.А., Негоденко О.В., Резник С.Ю., Разіна С.Я.. «Якість та тестування інформаційних систем». Навчальний посібник підготовлено до друку для самостійної роботи студентів вищих навчальних закладів. Київ: ННІІТ ДУТ, 2020. –128 с.

52. Шевченко Г.В. Дискретна математика. Навчально-методичний посібник. – К.: ДУТ, 2015. – 158 с.

53. Кузьменко І.М. Теорія графів навч. посіб. для здобувачів ступеня бакалавра за освітньою програмою «Комп’ютерний моніторинг та геометричне моделювання процесів і систем» спеціальності 122 «Комп’ютерні науки»/ І.М. Кузьменко; КПІ ім. Ігоря Сікорського. — Електронні текстові дані (1 файл: 1,7 Мбайт). — Київ: КПІ ім. Ігоря Сікорського, 2020. — 71 с [Електронний ресурс]. URL: https://ela.kpi.ua/server/api/core/bitstreams/fb0a4251-74d9-470b-88da-71abb4e85f93/content.

54. Щербина А.С., Золотухіна О. А. Гібридний підхід до багатоцілевої оптимізації розподілу тестових ресурсів ІТ-проекту за допомогою графової моделі оптимізації. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.605-607.

55. Звенігородський О.С., Зінченко О.В., Чичкарьов Є.А., Кисіль Т.М. Штучний інтелект. Вступний курс: Навчальний посібник. – К.: ДУТ, 2022. – 193 с. [Електронний ресурс]. URL: https://duikt.edu.ua/uploads/l_492_92652604.pdf.

56. Microsoft Learn. Common web application architectures. [Електронний ресурс]. URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.

57. Microsoft Learn. Windows Presentation Foundation documentation [Електронний ресурс]. URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/.

58. QuikGraph documentation [Електронний ресурс]. URL: https://kernelith.github.io/QuikGraph/

59. Чичкарьов Є.А., Зінченко О.В., Єльченко С.В. Прикладне програмування на Python. Частина 1. Основи програмування на Python.- Навчальний посібник.- Київ: ДУТ, 2022. - 160 с. [Електронний ресурс]. URL: https://www.duikt.edu.ua/uploads/l_546_48073069.pdf.

60. NumPy Documentation [Електронний ресурс]. URL: https://numpy.org/doc/

61. pymoo: Multi-objective Optimization in Python [Електронний ресурс]. URL: https://pymoo.org/.

62. NSGA-II: Non-dominated Sorting Genetic Algorithm [Електронний ресурс]. URL: https://pymoo.org/algorithms/moo/nsga2.html.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Оптимізація розподілу тестових ресурсів для процесу тестування програмного забезпечення»

Виконав: студент групи ПДМ-63 Андрій ЩЕРБИНА

Керівник: канд. хім. наук, доцент кафедри ІПЗ Людмила СОЛЯНИК

Київ - 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація розподілу тестових ресурсів для тестування програмного забезпечення за рахунок використання гібридного підходу на основі графової моделі та генетичного алгоритму.

Об'єкт дослідження: розподіл тестових ресурсів для процесу тестування програмного забезпечення.

Предмет дослідження: моделі та методи оптимального розподілу тестових ресурсів для процесу тестування програмного забезпечення.

2

АКТУАЛЬНІСТЬ РОБОТИ

Група методів	Приклади методів	Переваги	Недоліки
Точні методи (одноцільові моделі)	Метод Лагранжа Динамічне програмування Нелінійне програмування SRGM експоненційна моделі	Гарантують оптимальні рішення для простих задач	Не розглядають багатоцільність Важко масштабуються (NP-повні)
Метаевристичні методи	Генетичні алгоритми (NSGA-II) Еволюційні алгоритми (MODE) Ройові алгоритми (PSO)	Легко масштабуються Ефективні компромісні рішення	Не гарантує оптимальність рішення Не гарантує дотримання критичних обмежень (дефіцит кваліфікованих ресурсів)
Методи багатокритеріального вибору	Методи аналітичного ієрархічного процесу (AHP, Fuzzy AHP) Ризик-орієнтовані евристики	Легко масштабуються Ефективні компромісні рішення Оцінка та пріоритезація тестів і ресурсів	Не генерують оптимальний розклад (лише ранжування) Не враховують складні залежності
Методи ШІ Моделі глибокого навчання	DQN PPO (на основі векторних станів)	Адаптивній розподіл ресурсів у реальному часі Реагування на динамічні зміни ресурсів і пріоритетів.	Не враховує залежності між тестами Не враховує порядок виконання завдань
Графові моделі	Граф залежностей тестів RCPSP на графах Графові нейронні мережі (GNN).	Формалізація та структуризація задачі Забезпечують структурний аналіз GNN здатні обробляти змінну топологію графа	Не є вирішувачем Не має механізмів оптимізації та планування

3

ЗАДАЧІ ДОСЛІДЖЕННЯ

1. Провести огляд існуючих підходів до задачі розподілу тестових ресурсів.
2. Розробити математичну модель оптимізації розподілу тестових ресурсів, визначити та формалізувати обмеження на кваліфікацію ресурсів та та топологічні залежності тестових випадків
3. Розробити підхід та алгоритмічне забезпечення пошуку оптимальних рішень розподілу ресурсів.
4. Спроектувати архітектуру програмного забезпечення для розподілу тестових ресурсів, обрати технології та засоби реалізації компонентів системи.
5. Розробити програмне забезпечення для апробації розробленого підходу.
6. Провести моделювання роботи розробленого підходу та оцінити ефективность отриманних розподілів тестових ресурсів.

4

ГРАФОВА МОДЕЛЬ

Орієнтовний граф $G = (V, E)$, де:

V - множина вершин: T (тестові випадки t_i) та R (тестові ресурси r_j) $T \cup R = V, T \cap R = \emptyset$

E - множина орієнтованих ребер: E_{dep}(залежності між тестами), та E_{res} (зв'язки «тест-ресурс») $E_{dep} \cup E_{res} = E$, $E_{dep} \cap E_{res} = \emptyset$

Атрибути тестових випадків $\forall t_i \in T$:

- e_{t_i} – тип тестового випадку t_i
- $d_{t_i}^{base}$– базовий час виконання тестового випадку t_i
- p_{t_i} – пріоритет тестового випадку t_i
- c_{t_i} – множина вимог, яка покривається t_i

Атрибути вимог $\forall c \in C$:

- s_{c_k} – серйозність вимоги c_k

Атрибути ресурсів випадків $\forall\ r_j \in R$:

- e_{r_j} – тип ресурсу r_j
- q_{r_j} – кількість доступних одиниць ресурсу r_j
- f_{r_j} – коефіцієнт ефективності (швидкості) ресурсу r_j
- $K_{r_j}^{poss}$– множина усіх типів тестових завдань, які може виконувати ресурс r_j
- $K_{r_j}^{pref}$– множина профільних типів завдань, які може виконувати ресурс r_j ($K_{r_j}^{pref} \subseteq K_{r_j}^{poss}$)
- W_{r_j} – вартість неефективності (штраф) за призначення r_j на непрофільне завдання

Змінні для рішення

- $x_{t_i} \in \{0,1\} - 1$, якщо тест t_i обрано для виконання
- $z_{t_i,r_j} \in \{0,1\} - 1$, якщо тест t_i призначено ресурсу r_j
- $y_c \in \{0,1\} - 1$, якщо вимога покрита

Цільові функції

Покриття вимог (якість тестування):

$$Cover = max \sum_{c \in C} s_c y_c$$

Фактичний час (швидкість):

$$Time = \min \sum_{t_i \in T} \sum_{r_j \in R} (p_{t_i} d_{t_i}^{base} f_{r_i}) \cdot z_{t_i,r_j}$$

Ефективність використання ресурсів (оптимальність):

$$Inefficiency = min \sum_{t_i \in T} \sum_{r_j \in R} (I_{t_i,r_j}) \cdot z_{t_i,r_j}$$

де I_{t_i,r_j} – вартість невідповідності:

$I_{t_i,r_j} = 0$, якщо тип тесту $e_{t_i} \in K_{r_j}^{pref}$ (бажане призначення)

$I_{t_i,r_j} = W_{r_j}$, якщо $e_{t_i} \in (K_{r_j}^{poss} - K_{r_j}^{pref})$ (можливе, але небажане)

Обмеження на використання ресурсів

- порядок виконання тестів: $x_{t_k} \le x_{t_j}, \forall (t_k, t_j) \in E_{dep}$
- сумісність типу ресурсу: $z_{t_i,r_j} = 0$, якщо $e_{t_i} \notin K_{r_j}^{poss}$
- призначення ресурсу:

$$\sum_{r_j \in R} z_{t_i,r_j} = x_{t_i}, \forall t_i \in T$$

- обмеження на кількість ресурсів:

$$\sum_{t_i \in T} z_{t_i,r_j} \le q_{r_j}\ , \forall r_j \in R$$

- покриття вимог:

$$\sum_{t_i \in T, c \in c_{t_i}} x_{t_i} \ge y_c$$

5

ГІБРИДНИЙ ПІДХІД

Формування графу $G = (T, R, E)$: .NET, бібліотека QuikGraph

Аналіз та очищення, кластеризація: Scikit-learn

Евристичний пошук початкового рішення: жадібна евристична стратегія, точне рішення від Google OR-Tools

Евристична багатоцільова оптимізація (алгоритм NSGA-II, Pymoo)

Тип кодування: комбіноване (перестановкове (S) + цілочисельне (A))

Довжина хромосоми: $2 \mid T \mid$

Логіка гена:

S – вектор послідовності вибору на виконання тестових задач, s_i – ID тестового випадку

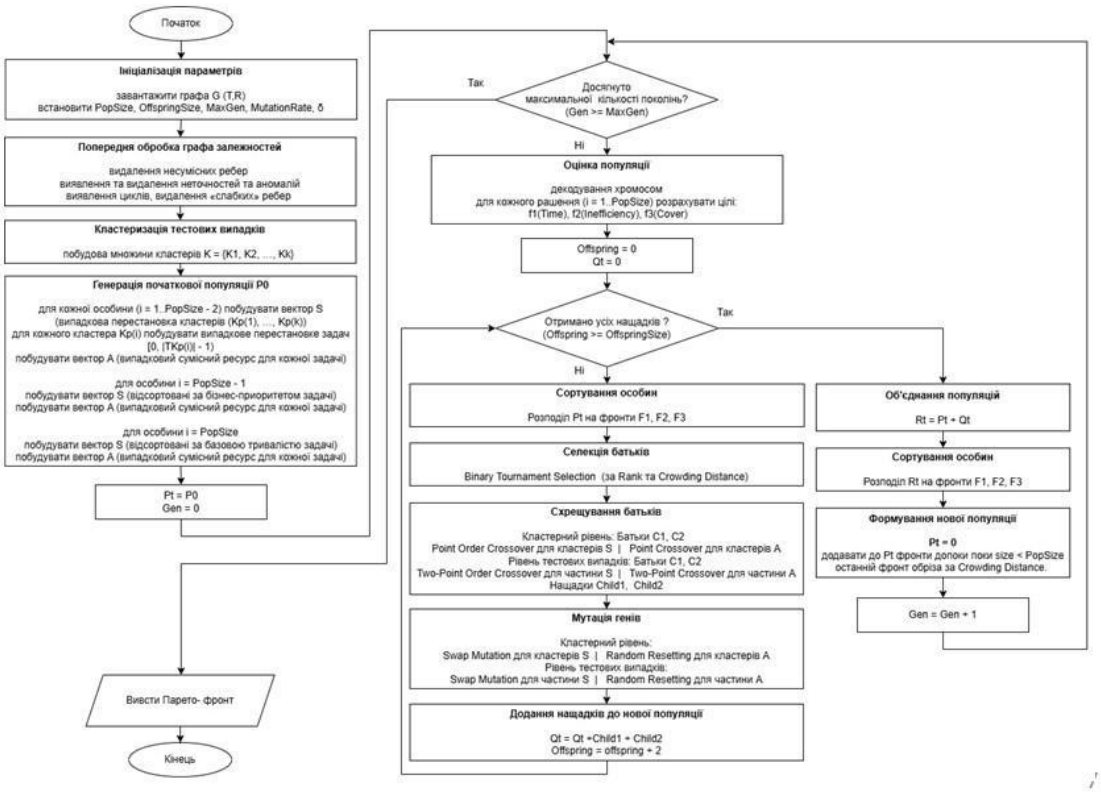
A – вектор призначення ресурсів a_i – ID ресурсу для виконання тестової задачі s_i

Фітнес-функція: *F(X) = [f1(Time), f2(Inefficiency), f3(Cover)]*

Остаточний розподіл ресурсів (логіка C#, зберігання варіантів рішення BD SQL Server)

6

УЗАГАЛЬНЕНА БЛОК-СХЕМА АЛГОРИТМУ ПОШУКУ ОПТИМАЛЬНОГО РІШЕННЯ



ПРАКТИЧНІ РЕЗУЛЬТАТИ

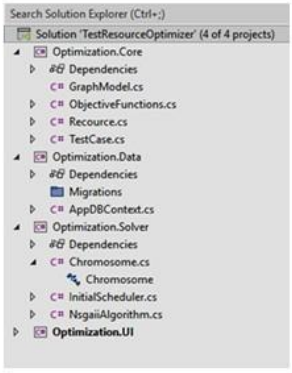


Рис. 1 Структура програмного рішення

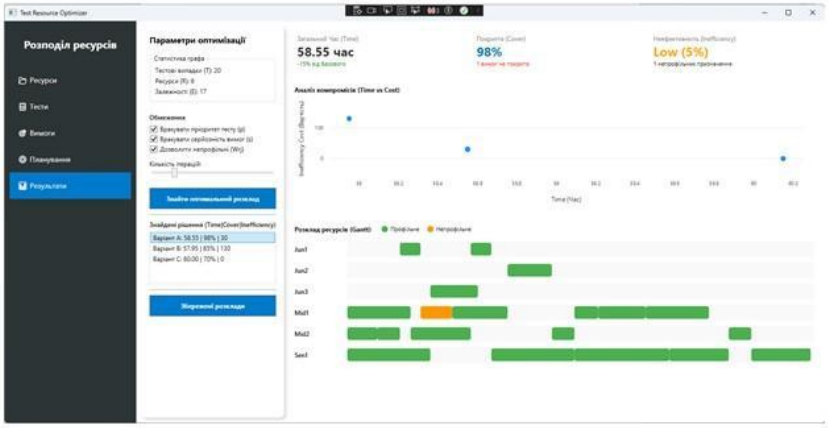


Рис. 2 Головне вікно Dashboard

МЕТРИКИ ОЦІНЮВАННЯ

Зважене покриття вимог (ступінь перевірки)

$$C_w = \frac{\sum_{i \in C_{covered}} s_i}{\sum_{j \in C_{all}} s_j}$$

де: $C_{covered}$ – множина покритих вимог;
C_{all} – множина усіх вимог
s_i, s_j – вага важливості (критичності) відповідної вимоги.

Завантаження ресурсів (ступінь використання)

$$U_w = \frac{1}{|R|} \sum_{r \in R} \frac{T_r^{load}}{T_r^{avail}}$$

де: R – множина доступних ресурсів (тестувальників);
r – ресурс (тестувальник);
$|R|$ – загальна кількість ресурсів;
T_r^{load} – фактичний час, коли ресурс r був зайнятий;
T_r^{avail} – увесь доступний робочий час ресурсу r.

Загальна тривалість тестування (час завершення всього тестового циклу)

$$T = \max_{r \in R} finish\_time(r)$$

де: R – множина доступних ресурсів (тестувальників);
r – ресурс (тестувальник);
$finish\_time(r)$ – час, коли ресурс завершив виконання всіх призначених тестів

Частка непрофільних призначень (оцінка відповідності тестових випадків спеціалізації ресурсів)

$$A = \frac{N_{non\_prof}}{N_{total}}$$

де: N_{non_prof} – кількість призначень для тестувальників не за профілем;
N_{total} – загальна кількість призначень.

Інтегральна оцінка побудованого рішення

$$F(S) = w_C C^{norm} + w_T T^{norm} + w_U U^{norm} + w_A A^{norm}$$

де: $C^{norm}, T^{norm}, U^{norm}, A^{norm}$ - нормовані значення критеріїв;
w_C, w_T, w_U, w_A - вагові коефіцієнти (важливість критерію у оцінці): $w_C + w_T + w_U + w_A = 1$

РЕЗУЛЬТАТИ МОДЕЛЮВАННЯ

Результати першого експерименту (кількість поколінь = 50)

Метрики	Базовий план	OR-Tools	Гібридний метод	Відносне покращення	
				Базовий/ Гібридний	OR/ Гібридний
Time (F1)	81.8	60.0	57.5	-29.6%	-4.1%
Inefficiency (F2)	510.0	140.0	180.0	-64.7%	+28.6%
Cover (F3)	382	382	382	0.0%	0.0%
C_w	0.430	0.430	0.430	0.0%	0.0%
U_w	21.4%	32.7%	33.5%	+56.4%	+2.5%
A, помилки	0.450 (11)	0.800 (4)	0.800 (4)	-63.6%	0.0%
F(S)	0.3145	0.4372	0.4462	+41.9%	+2.1%
T_w (сек)	0.0054	0.0817	11.1978	-	-

Результати першого експерименту (кількість поколінь = 100)

Метрики	Базовий план	OR-Tools	Гібридний метод	Відносне покращення	
				Базовий/ Гібридний	OR/ Гібридний
Time (F1)	81.8	60.0	57.4	-29.8%	-4.4%
Inefficiency (F2)	510.0	140.0	200.0	-60.8%	+42.9%
Cover (F3)	382	382	382	+0.0%	+0.0%
C_w	0.430	0.430	0.430	0.0%	0.0%
U_w	21.4%	32.7%	34.3%	+60.2%	+5.0%
A, помилки	0.450 (11)	0.800 (4)	0.800 (4)	-63.6%	+0.0%
F(S)	0.3145	0.4372	0.4484	+42.6%	+2.6%
T_w (сек)	0.0054	0.0817	21.4516	-	-

Загальні параметри моделі

Параметр моделі	Значення та опис
Кількість тестових випадків (T)	20,120
Кількість вимог (C)	40,240
Кількість тестувальників (R)	6 (3 Juniors, 2 Mids, 1 Senior)
Типи тестових випадків	Routine (40%), Standard (40%), Advanced (20%)
Глибина залежностей тестових випадків	довжина ланцюжків від 5 до 15 тестових випадків

Вагові коефіцієнти інтегральної оцінки

$w_C = 0.4, w_T = 0.3, w_U = 0.2, w_A = 0.1$

Результати другого експерименту (кількість поколінь = 50)

Метрики	Базовий план	OR-Tools	Гібридний метод	Відносне покращення	
				Базовий/ Гібридний	OR/ Гібридний
Time (F1)	346.8	231.0	271.1	-21.8%	+17.4%
Inefficiency (F2)	2100.0	620.0	640.0	-69.5%	+3.2%
Cover (F3)	2402	2458	2458	+2.3%	+0.0%
C_w	0.5120	0.524	0.524	+2.3%	+0.0%
U_w	26.4%	45.8%	39.4%	+49.2%	-14.0%
A, помилки	0.589 (46)	0.833 (20)	0.867 (16)	-65.2%	-20.0%
F(S)	0.3193	0.4865	0.4427	+38.6%	-9.0%
T_w (сек)	0.0495	11.7797	166.0041		

Результати другого експерименту (кількість поколінь = 100)

Метрики	Базовий план	OR-Tools	Гібридний метод	Відносне покращення	
				Базовий/ Гібридний	OR/ Гібридний
Time (F1)	346.8	231.0	244.0	-29.6%	+5.6%
Inefficiency (F2)	2100.0	620.0	500.0	-76.2%	-19.4%
Cover (F3)	2402	2458	2458	+2.3%	+0.0%
C_w	0.5120	0.524	0.524	+2.3%	+0.0%
U_w	26.4%	45.8%	43.9%	+66.5%	-4.1%
A, помилки	0.589 (46)	0.833 (20)	0.883 (14)	-69.6%	-30.0%
F(S)	0.3193	0.4865	0.4766	+49.3%	-2.0%
T_w (сек)	0.0495	11.7797	332.8195		

Характеристика ресурсів

Ресурс	Коефіцієнт ефективності (f_{r_j})	Вартість неефективності (W_{r_j})	Навички ($K_{r_j}^{poss}$)	Пріоритет ($K_{r_j}^{pref}$)
Jun1	1.3 (повільний)	10	Routine	Routine
Jun2	1.4 (повільний)	10	Routine	Routine
Jun3	1.5 (повільний)	10	Routine	Routine
Mid1	1.0 (стандарт)	30	Routine, Standard	Standard
Mid2	0.095 (стандарт)	30	Routine, Standard	Standard
Sen1	0.75 (швидкий)	50	Routine, Standard, Advanced	Advanced

ВИСНОВКИ

1. Проведено огляд існуючих підходів до розподілу тестових ресурсів та методів, що використовуються для вирішення задач оптимізації їх розподілу в умовах багатоцільових обмежень.

2. Розроблено математичну модель оптимізації розподілу тестових ресурсів, яка базується на графовій структурі та формалізує обмеження задачі (кваліфікація фахівців та топологічні залежності тестових випадків).

3. Розроблено гібридний підхід, який поєднує графовий аналіз для попереднього очищення обмежень з методами багатоцільової оптимізації на основі генетичного алгоритму та дозволяє знайти практичні рішення у просторі варіантів призначення ресурсів.

4. Спроектовано архітектуру програмного забезпечення для розподілу тестових ресурсів, обрано технології та засоби реалізації компонентів системи: середовище розробки MS Visual Studio 2022, .Net бібліотека работи з графами QuikGraph для побудови моделі, Python бібліотека Pymoo для реалізації гібридного алгоритму та технологія WPF для інтерфесу користувача.

5. Розроблено програмне забезпечення гібридного підхіду розподілу тестових ресурсів для процесу тестування програмного забезпечення.

6. Проведене моделювання роботи розробленого програмного забезпечення показало, що запропонований гібридний підхід формує кращі плани тестування порівняно з розкладом наданим точним рішенням OR-Tools та евристичним базовим розкладом, дозволяє скоротити час тестування приблизно на 30% порівняно з базовим планом та зменшити кількість непрофільних призначень на 60-80%. Збільшення популяції у двічі збільшує завантаженість ресурсів у середньому на 6% та додатково покращує інтегральну оцінку рішення у середньому на 3%, однак майже вдвічі збільшує тривалість пошуку рішення.

11

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Щербина А.С., Золотухіна О.А. Гібридний підхід до багатоцільової оптимізації розподілу тестових ресурсів ІТ-проекту за допомогою графової моделі оптимізації. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.605-607

2. Щербина А.С., Соляник Л.О., Оптимізаційна графова модель розподілу тестових ресурсів ІТ-проєкту II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії», 26 листопада 2025 р, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.

12

ДОДАТОК Б
ІНТЕРФЕЙС СИСТЕМИ

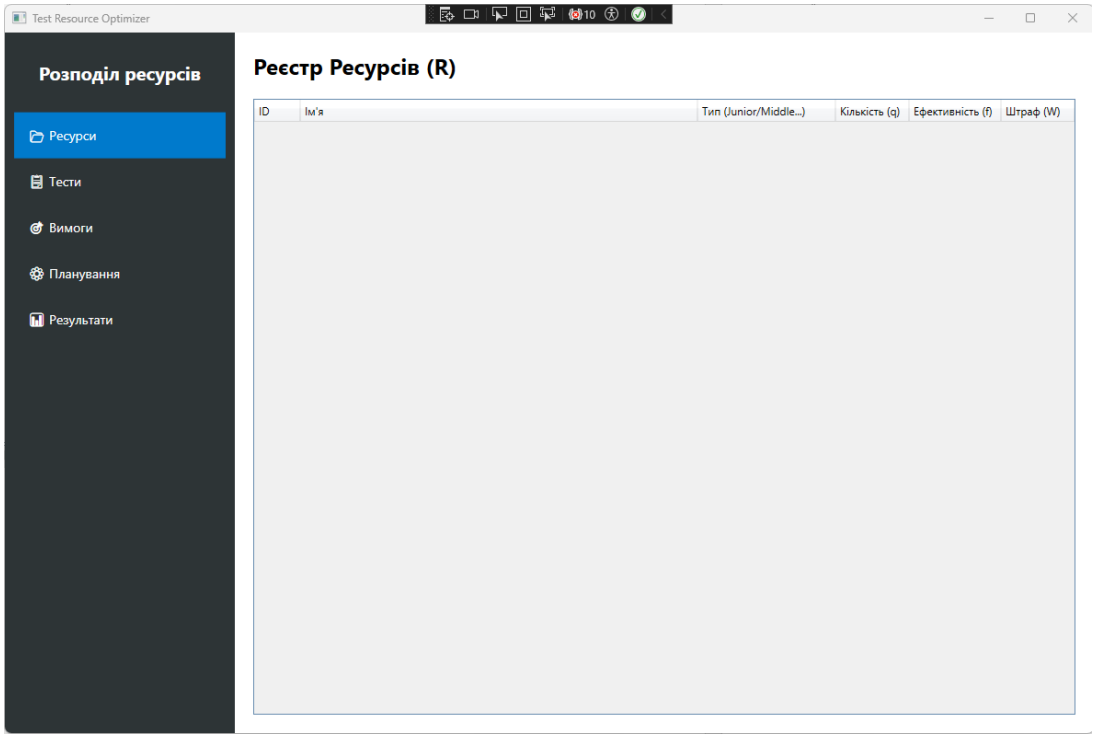


Рис В.1 Вікно модулю «Реєстр ресурсів»

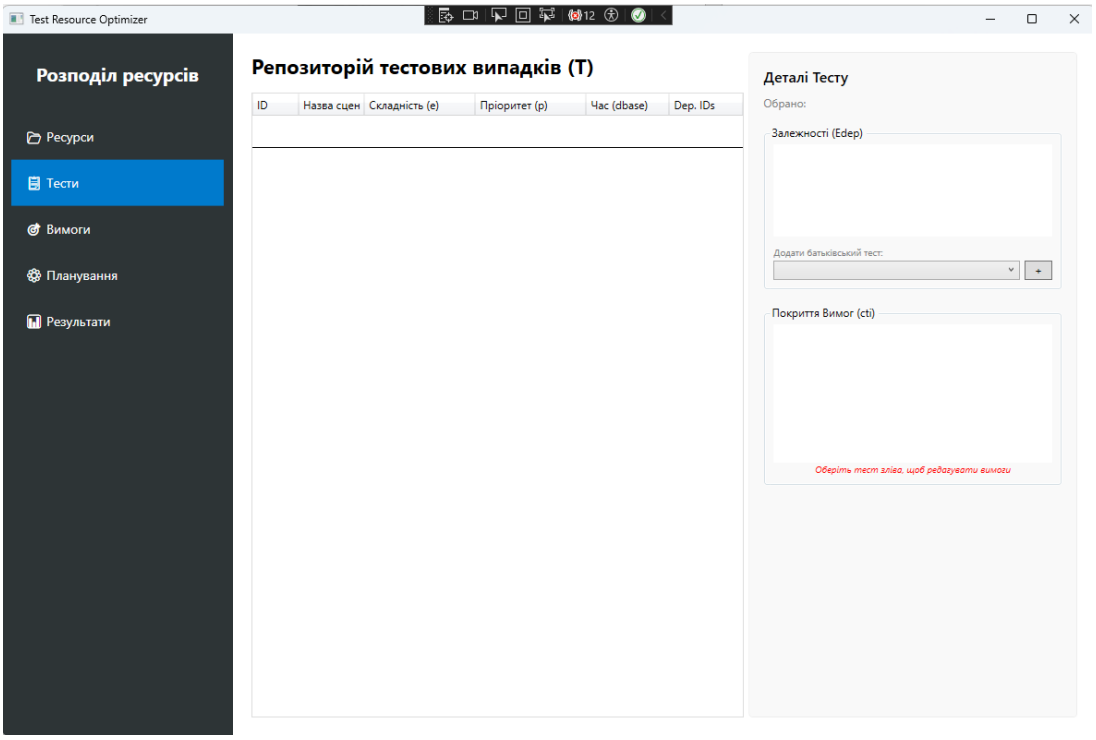


Рис В.2 Вікно модулю «Репозиторій тестових випадків»

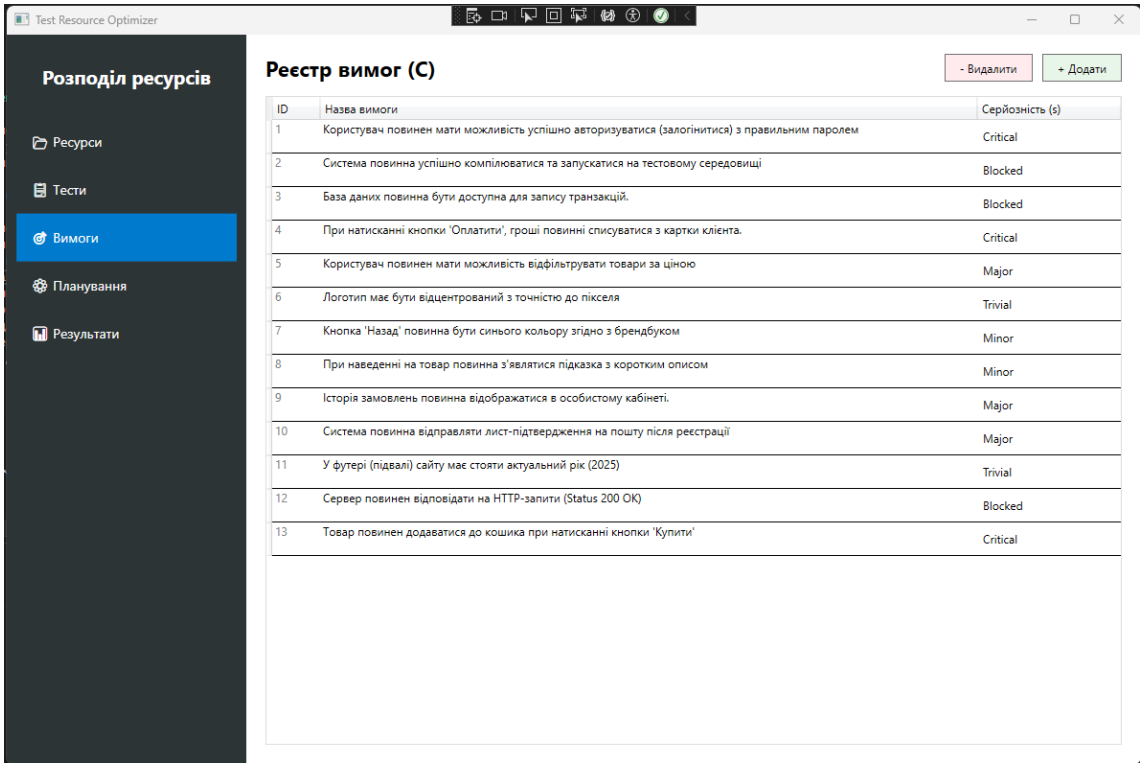


Рис В.3 Вікно модулю «Реєстр вимог»

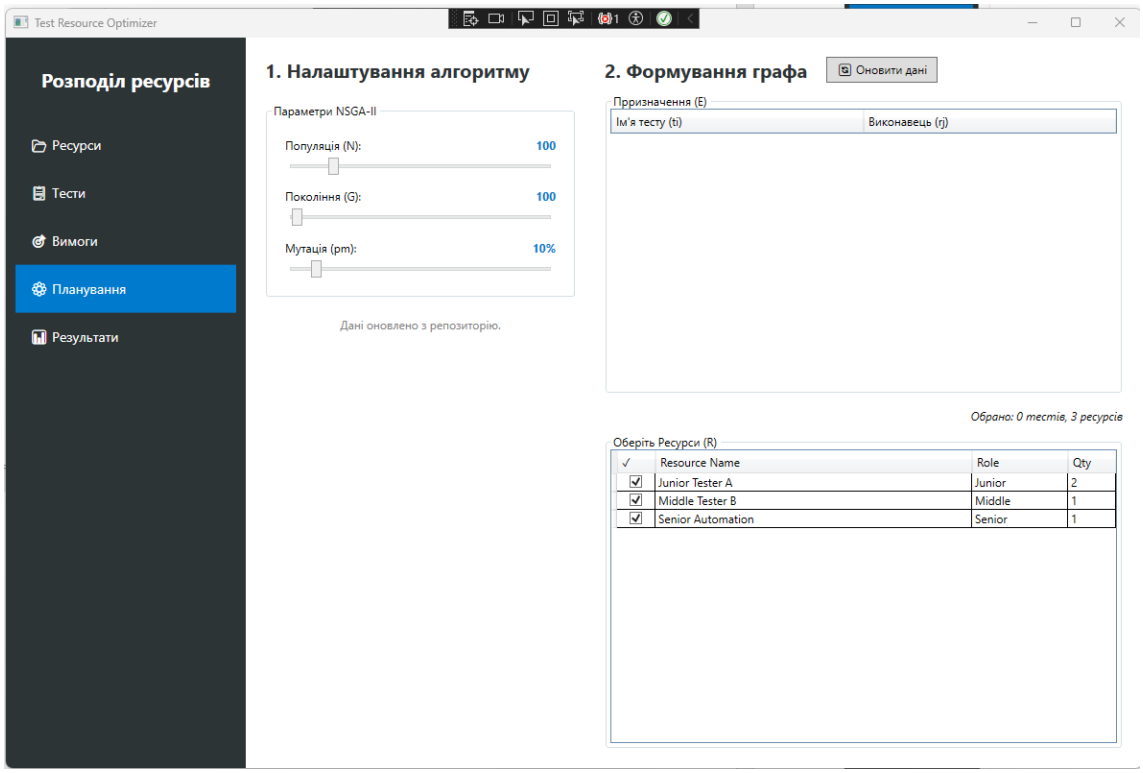


Рис В.4 Вікно модулю «Налаштування»

ДОДАТОК В. ЛИСТИНГИ ОСНОВНИХ МОДУЛІВ

В.1 Евристичний пошук початкового рішення

```
def run_heuristic_baseline():
    print("\n" + "="*60)
    print(" ЕТАП 4: ЕВРИСТИЧНИЙ БАЗОВИЙ АЛГОРИТМ")
    print("="*60)
        # --- перевіряємо дані ---
    sample_res = list(RESOURCES.values())[0]
    if not sample_res.get('specific_skills'):
        print("[ПОМИЛКА] У ресурсів відсутні
'specific_skills'!")
        return 0, 0, 0, {'K1_norm':0, 'K1_raw':0,
'K2_norm':0, 'K2_raw':0, 'K3_norm':0, 'K3_raw':0,
'K4_norm':0, 'K4_raw':0}, 0
    # --- жадібне формування черги ---
    heuristic_seq = []

    in_degree = {t: 0 for t in TASKS}
    graph = {t: [] for t in TASKS}
    for child, parents in DEPENDENCIES.items():
        in_degree[child] = len(parents)
        for p in parents:
            graph[p].append(child)

    ready_queue = [t for t in TASKS if in_degree[t] == 0]
    covered_reqs_so_far = set()

    while ready_queue:
        best_task = -1
        best_score = -1

        for t in ready_queue:
            reqs = TASKS[t]['reqs']
            new_cover_count = sum(1 for r in reqs if r not
in covered_reqs_so_far)
            prio_val = PRIORITY_W[TASKS[t]['prio']]
           # спочатку нові вимоги, потім пріоритет
            score = new_cover_count * 1000 + prio_val

            if score > best_score:
                best_score = score
                best_task = t

        if best_task == -1: # щоб не зациклело
(ready_queue не пуста, але задач немає)
             best_task = ready_queue[0]

        heuristic_seq.append(best_task)
        ready_queue.remove(best_task)

        for r in TASKS[best_task]['reqs']:
            covered_reqs_so_far.add(r)

        for neighbor in graph[best_task]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                ready_queue.append(neighbor)

    if len(heuristic_seq) < N_TASKS:
        missing = [t for t in TASKS if t not in
heuristic_seq]
        heuristic_seq.extend(missing)

    # --- жадібне призначення ресурсів ---
    heuristic_assign = [0] * N_TASKS

    for t_id in TASKS:
        # шукаємо серед тих, хто має specific_skills
        candidates = [r for r, val in RESOURCES.items() if
t_id in val['specific_skills']]

        if not candidates:
            # якщо нема кандидатів візьмем будь-кого
(помилка генерації)
            candidates = list(RESOURCES.keys())

        # # сортуємо кандидатів: профільність (0=Так) ->
ефективність (меньше неефективнjcns) -> вартістьь
        # candidates.sort(key=lambda r: (
        #     0 if t_id in RESOURCES[r]['specific_prefs']
else 1,
        #     RESOURCES[r]['eff'],
        #     RESOURCES[r]['cost']
        # ))

        # сортуємо кандидатів: швидкість, -> профільність
        candidates.sort(key=lambda r: (
         # 1. min коефіцієнт (найшвидший)
            RESOURCES[r]['eff'],
         # 2. якщо швидкість однакова -> візьмемо
профільного
            0 if t_id in RESOURCES[r]['specific_prefs']
else 1,
         # 3. min вартість за базовою ставкою
            RESOURCES[r]['cost']
        ))

        heuristic_assign[t_id] = candidates[0]

    # --- оцінюємо  ---
    local_problem = TestModel()

    h_sched, h_cov, h_ineff =
local_problem.decode(heuristic_seq, heuristic_assign)

    if not h_sched:
        print("Евристичний розклад порожній! (жорсткий
дедлайн або помилка в навичках?")

    h_integral, h_k = calculate_integral_score(h_sched,
h_cov)

    h_makespan = max(i['end'] for i in h_sched.values())
if h_sched else 0
    h_cover_score = sum(SEVERITY_W[REQ_INFO[r]] for r in
h_cov)

    # print(f" Результати Евристики:")
    # print(f"   Integral: {h_integral:.4f}")
    # print(f"   F1 Time:  {h_makespan:.1f}")
    # print(f"   F2 Cost:  {h_ineff:.1f}")
    # print(f"   F3 Cover: {h_cover_score:.0f}")
  return h_makespan, h_ineff, h_cover_score, h_k, h_integral
```

В.2 Пошук рішення інструмент OR-Tools

```
def run_ortools_comparison():
    print("\n" + "="*60)
    print(" ЕТАП 3: ПЕРЕВІРКА ТОЧНИМ МЕТОДОМ (GOOGLE OR-
TOOLS)")
    print("="*60)

    model = cp_model.CpModel()
    horizon = int(sum(t['dur'] * 2 for t in
TASKS.values()))

    starts, ends, assigns = {}, {}, {}
    intervals = {r: [] for r in RESOURCES}

      for t_id, task in TASKS.items():
          s = model.NewIntVar(0, horizon, f's_{t_id}')
          e = model.NewIntVar(0, horizon, f'e_{t_id}')
          starts[t_id], ends[t_id] = s, e
          # кандидатов беремо зі specific_skills
          cands = [r for r, v in RESOURCES.items() if t_id
   in v['specific_skills']]
          if not cands: return None

          lits = []
          assigns[t_id] = []
```

```
        for r in cands:
            dur = int(np.ceil(task['dur'] *
RESOURCES[r]['eff']))
            b = model.NewBoolVar(f't{t_id}_r{r}')
            lits.append(b)
            assigns[t_id].append((r, b))

intervals[r].append(model.NewOptionalIntervalVar(s, dur,
e, b, f'i{t_id}{r}'))
        model.Add(sum(lits) == 1)

    for c, parents in DEPENDENCIES.items():
        for p in parents: model.Add(starts[c] >=
ends[p])

    for i_list in intervals.values():
model.AddNoOverlap(i_list)


    makespan = model.NewIntVar(0, horizon, 'makespan')
    model.AddMaxEquality(makespan, list(ends.values()))
    model.Add(makespan <= int(PROJECT_DEADLINE))

    # оптимізація по часу
    model.Minimize(makespan)

    #оптимізація по вартості
    # total_cost_var = model.NewIntVar(0, 1000000,
'total_cost')
        # cost_terms = []
    # for t_id in TASKS:
    #     for r_id, bool_var in assigns[t_id]:
    ## вартість призначення ресурсу r_id на задачу t_id
    ## якщо задача профільна -> 0, якщо ні ->
RESOURCES[r_id]['cost']
    #         cost_val = 0 if t_id in
RESOURCES[r_id]['specific_prefs'] else
RESOURCES[r_id]['cost']
    ## якщо cost_val > 0, додати: bool_var * cost_val
    #         if cost_val > 0:
    #             term = model.NewIntVar(0, cost_val,
f'cost_{t_id}_{r_id}')
    #             model.Add(term ==
cost_val).OnlyEnforceIf(bool_var)
    #             model.Add(term ==
0).OnlyEnforceIf(bool_var.Not())
    #             cost_terms.append(term)
    # model.Add(total_cost_var == sum(cost_terms))
    # scale = 10000
    # model.Minimize(total_cost_var * scale + makespan)

    solver = cp_model.CpSolver()
    solver.parameters.max_time_in_seconds = 30.0
    # solver.parameters.num_search_workers = 8
    solver.parameters.num_search_workers = 1

    # print("Запуск солвера...")
    if solver.Solve(model) in [cp_model.OPTIMAL,
cp_model.FEASIBLE]:
        makespan = solver.ObjectiveValue()
        sched, cov = {}, set()
        ineff = 0.0

        for t_id in TASKS:
            res = -1
            for r, b in assigns[t_id]:
                if solver.Value(b): res = r; break
            sched[t_id] = {'start':
float(solver.Value(starts[t_id])),
                           'end':
float(solver.Value(ends[t_id])), 'res': res}
            # штрафуэмо
            if t_id not in
RESOURCES[res]['specific_prefs']:
                ineff += RESOURCES[res]['cost']
            for r in TASKS[t_id]['reqs']: cov.add(r)
        cover_score = sum(SEVERITY_W[REQ_INFO[r]] for r
in cov)
        integral, details =
calculate_integral_score(sched, cov)
        # print(f" OR-Tools: Time={makespan},
Integral={integral:.4f}")

        # # розклад від OR-Tools
        # print("\n   РОЗКЛАД ВІД OR-TOOLS:")
        # print(f"   {'Task':<25} | {'Res':<5} |
{'Start':<6} | {'End':<6}")
        # print("   " + "-"*50)
        # for t_id, d in sorted(sched.items(),
key=lambda x: x[1]['start']):
        # print(f"   {TASKS[t_id]['name']:<25} |
{RESOURCES[d['res']]['name']:<5} | {d['start']:<6.1f} |
{d['end']:<6.1f}")
        return makespan, ineff, cover_score, details
    return None
```

B.3 Генетичні оператори та модель

```
class HybridCrossover(Crossover):
    def __init__(self): super().__init__(2, 2)
    def _do(self, problem, X, **kwargs):
        n_parents, n_matings, n_var = X.shape
        Y = np.full_like(X, -1)
        for k in range(n_matings):
            p1, p2 = X[0, k], X[1, k]
            for i, (parent_a, parent_b) in
enumerate([(p1, p2), (p2, p1)]):
                a, b = np.sort(np.random.choice(N_TASKS,
2, replace=False))
                child_seq = np.full(N_TASKS, -1)
                child_seq[a:b+1] =
parent_a[a:b+1][:N_TASKS]
                curr, p2_pos = (b + 1) % N_TASKS, (b +
1) % N_TASKS
                while -1 in child_seq:
                    cand = parent_b[p2_pos]
                    if cand not in child_seq:
                        child_seq[curr] = cand
                        curr = (curr + 1) % N_TASKS
                    p2_pos = (p2_pos + 1) % N_TASKS
                Y[i, k, :N_TASKS] = child_seq

            cut1, cut2 =
np.sort(np.random.choice(range(N_TASKS, 2*N_TASKS), 2,
replace=False))
            Y[0, k, N_TASKS:] = p1[N_TASKS:]; Y[0, k,
cut1:cut2] = p2[cut1:cut2]
            Y[1, k, N_TASKS:] = p2[N_TASKS:]; Y[1, k,
cut1:cut2] = p1[cut1:cut2]
        return Y

class HybridMutation(Mutation):
    def _do(self, problem, X, **kwargs):
        for i in range(len(X)):
            if np.random.random() < 0.2:
                idx1, idx2 = np.random.choice(N_TASKS,
2, replace=False)
                X[i, idx1], X[i, idx2] = X[i, idx2],
X[i, idx1]
            for j in range(N_TASKS, 2 * N_TASKS):
                if np.random.random() < 0.05:
                    t_id = j - N_TASKS
                    # мутацію робимо лише в межах
specific_skills
                    feasible = [r for r, val in
RESOURCES.items() if t_id in val['specific_skills']]
                    if feasible: X[i, j] =
np.random.choice(feasible)
        return X

class HybridSampling(Sampling):
    def _do(self, problem, n_samples, **kwargs):
        X = np.full((n_samples, problem.n_var), 0,
dtype=int)
        for i in range(n_samples):
            X[i, :N_TASKS] =
np.random.permutation(N_TASKS)
            for t_idx in range(N_TASKS):
                # обираємо лише з specific_skills
                feasible = [r for r, val in
RESOURCES.items() if t_idx in val['specific_skills']]
                if not feasible: feasible =
list(RESOURCES.keys())
                X[i, N_TASKS + t_idx] =
np.random.choice(feasible)
        return X


class TestModel(ElementwiseProblem):
```

```
    def __init__(self): super().__init__(n_var=2 *
N_TASKS, n_obj=3, xl=0, xu=1)

    def _evaluate(self, x, out, *args, **kwargs):
        sch, cov, ineff = self.decode(x[:N_TASKS],
x[N_TASKS:])
        makespan = max(i['end'] for i in sch.values())
if sch else 0
        out["F"] = [makespan, ineff, -
sum(SEVERITY_W[REQ_INFO[r]] for r in cov)]

    def decode(self, seq, assigns):
        free_time = {r: 0.0 for r in RESOURCES}
        end_time = {}
        schedule = {}
        covered = set()
        total_ineff = 0.0
        pending = list(seq)
        completed = set()

        while pending:
            scheduled = False
            for i, t_id in enumerate(pending):
                preds = DEPENDENCIES.get(t_id, [])
                if not all(p in completed for p in
preds): continue

                task = TASKS[t_id]
                pref_res = assigns[t_id]

                # чи може ресурс виконати цю задачу
                if t_id not in
RESOURCES[pref_res]['specific_skills']:
                    pref_res = None

                candidates = [r for r, v in
RESOURCES.items() if t_id in v['specific_skills']]
                if not candidates: # нема канддидату((
                    pending.pop(i); scheduled = True;
break

                ready = max(end_time[p] for p in preds)
if preds else 0

                best_res, best_finish = None,
float('inf')
                for r in candidates:
                    finish = max(free_time[r], ready) +
task['dur'] * RESOURCES[r]['eff']
                    if finish < best_finish:
best_finish, best_res = finish, r

                chosen = best_res
                if pref_res is not None:
                    p_finish = max(free_time[pref_res],
ready) + task['dur'] * RESOURCES[pref_res]['eff']
                    if p_finish <= (1 + DELTA_TOLERANCE)
* best_finish: chosen = pref_res

                start = max(free_time[chosen], ready)
                dur = task['dur'] *
RESOURCES[chosen]['eff']
                end = start + dur

                if end <= PROJECT_DEADLINE:
                    schedule[t_id] = {'start': start,
'end': end, 'res': chosen}
                    free_time[chosen] = end
                    end_time[t_id] = end
                    completed.add(t_id)
                    # штрафуємо, якщо задача не в
specific_prefs
                    if t_id not in
RESOURCES[chosen]['specific_prefs']:
                        total_ineff +=
RESOURCES[chosen]['cost']
                    for r in task['reqs']:
covered.add(r)

                pending.pop(i); scheduled = True; break
            if not scheduled: break
        return schedule, covered, total_ineff
```

B.4 Конфігурація моделі

```
random.seed(42)  # Фіксація зерна, щоб дані були
однакові при кожному запуску
np.random.seed(42)

PRIORITY_W = {'Highest': 100, 'High': 50, 'Medium': 20,
'Low': 1}
SEVERITY_W = {'Blocked': 100, 'Critical': 50, 'Major':
20, 'Minor': 5, 'Trivial': 1}
DELTA_TOLERANCE = 0.2
PROJECT_DEADLINE = 100.0

#Ресурси
RESOURCES = {
    0: {'name': 'Jun1', 'eff': 1.3, 'skills':
['Routine'], 'pref': ['Routine'], 'cost': 10},
    1: {'name': 'Jun2', 'eff': 1.4, 'skills':
['Routine'], 'pref': ['Routine'], 'cost': 10},
    2: {'name': 'Jun3', 'eff': 1.5, 'skills':
['Routine'], 'pref': ['Routine'], 'cost': 10},
    3: {'name': 'Mid1', 'eff': 1.0, 'skills':
['Routine', 'Standard'], 'pref': ['Standard'], 'cost':
30},
    4: {'name': 'Mid2', 'eff': 0.95, 'skills':
['Routine', 'Standard'], 'pref': ['Standard'], 'cost':
30},
    5: {'name': 'Sen1', 'eff': 0.75, 'skills':
['Routine', 'Standard', 'Advanced'], 'pref':
['Advanced'], 'cost': 50},
}

N_RES = len(RESOURCES)
N_TASKS_GEN = 20
N_REQS_GEN = 40

#Вимоги
REQ_INFO = {}
severity_levels = list(SEVERITY_W.keys())
sev_weights = [0.05, 0.15, 0.3, 0.3, 0.2]
for r in range(1, N_REQS_GEN + 1):
    req_name = f"Req{r}"
    severity = random.choices(severity_levels,
weights=sev_weights, k=1)[0]
    REQ_INFO[req_name] = severity

all_req_keys = list(REQ_INFO.keys())

#Задачі
TASKS = {}
task_types = ['Routine', 'Standard', 'Advanced']
priorities = list(PRIORITY_W.keys())
modules = ['Auth', 'Payment', 'Catalog', 'Cart',
'Profile', 'API', 'DB', 'UI', 'Export', 'Search']
actions = ['Test', 'Verify', 'Check', 'Validate',
'Audit', 'Debug']

for t in range(N_TASKS_GEN):
    t_type = random.choices(task_types, weights=[0.4,
0.4, 0.2], k=1)[0]

    if t_type == 'Routine': dur = random.randint(1, 4)
    elif t_type == 'Standard': dur = random.randint(3,
8)
    else: dur = random.randint(6, 16)

    name = f"{random.choice(actions)}
{random.choice(modules)}_{t}"
    prio = random.choice(priorities)

    num_reqs = random.randint(1, 3)
    start_req_idx = random.randint(0, N_REQS_GEN - 5)
    local_req_pool = all_req_keys[start_req_idx :
start_req_idx + 10]
    t_reqs = random.sample(local_req_pool,
k=min(num_reqs, len(local_req_pool)))

    TASKS[t] = {'name': name, 'type': t_type, 'dur':
dur, 'prio': prio, 'reqs': t_reqs}

N_TASKS = len(TASKS)
```

```
#Залежності
DEPENDENCIES = {}
available_ids = list(range(N_TASKS))
random.shuffle(available_ids)

while available_ids:
    chain_len = random.randint(5, 15)
    if len(available_ids) < chain_len: chain_len =
len(available_ids)
    chain = [available_ids.pop() for _ in
range(chain_len)]
    for i in range(1, len(chain)):
        parent, child = chain[i-1], chain[i]
        if child not in DEPENDENCIES:
DEPENDENCIES[child] = []
        DEPENDENCIES[child].append(parent)

#Задачі по ресурсах (хто яку задачу може виконувати
skills, які по prefs
for r in RESOURCES:
    RESOURCES[r]['specific_skills'] = set() # може
виконати
    RESOURCES[r]['specific_prefs'] = set()  # виконує
без штрафу
```

```
# гарантуємо, що кожна задача має хоча б одного
виконавця
for t_id, task in TASKS.items():
    potential_candidates = [r_id for r_id, val in
RESOURCES.items() if task['type'] in val['skills']]
    if not potential_candidates:
        raise ValueError(f"CRITICAL: Задача {t_id} не
має кандидатів за типом!")
    guaranteed_res = random.choice(potential_candidates)

RESOURCES[guaranteed_res]['specific_skills'].add(t_id)

# розподіляємо решту можливостей (~70% задач свого
рівня)
SKILL_COVERAGE_RATE = 0.7
for r_id, r_data in RESOURCES.items():
    for t_id, task in TASKS.items():
        if task['type'] in r_data['skills']:
            if random.random() < SKILL_COVERAGE_RATE:
                r_data['specific_skills'].add(t_id)
        # якщо вміє це робити, чи це його профіль
        if t_id in r_data['specific_skills']:
            if task['type'] in r_data['pref']:
                r_data['specific_prefs'].add(t_id)
```

B.5 Формування та обробка графа

```
# орієнтований граф
G = nx.DiGraph()

# додаємо вершини-ресурси ('R_' -> відрізняти від задач)
for r_id, r_data in RESOURCES.items():
    node_id = f"R_{r_id}"
    # Можемо зберегти атрибути ресурсу прямо у вершині
    G.add_node(node_id, node_type='resource',
name=r_data['name'], cost=r_data['cost'])
# додаємо вершини-задачи
for t_id, t_data in TASKS.items():
    node_id = f"T_{t_id}"
    G.add_node(node_id, node_type='task',
name=t_data['name'], duration=t_data['dur'])
# додаємо ребра залежність задач(Task -> Task)
for child_id, parents in DEPENDENCIES.items():
    for parent_id in parents:
        u = f"T_{parent_id}"

        v = f"T_{child_id}"
        G.add_edge(u, v, edge_type='dependency',
color='black')
# додаємо ребра виконання задач (Resource -> Task) -
існує, якщо ресурс може виконуваати задачу (skills)
for r_id, r_data in RESOURCES.items():
    for t_id in r_data['specific_skills']:
        u = f"R_{r_id}"
        v = f"T_{t_id}"
        # якщо профільна задача
        is_pref = t_id in r_data['specific_prefs']
        edge_color = 'green' if is_pref else 'orange' #
зелений - добре, помаранчевий - штраф
        G.add_edge(u, v, edge_type='capability',
color=edge_color, is_preferred=is_pref)

# print(f"Граф побудовано:")
# print(f"- Вершин: {G.number_of_nodes()} # (Ресурсів:
{N_RES}, Задач: {N_TASKS})")
# print(f"- Ребер:  {G.number_of_edges()}")



# моделювання помилок
# параметри "шуму"
N_NOISE_DEPS = 50  # випадкові залежності (зациkл)
N_NOISE_ASSIGNS = 10 # випадкові призначенння (ресурс не
має skill)

print(f"1. Вносимо викривлення в граф:")
print(f"   - Додаємо {N_NOISE_DEPS} випадкових
залежностей між задачами...")
print(f"   - Додаємо {N_NOISE_ASSIGNS} випадкових
зв'язків 'Ресурс-Задача'...")

# випадкові залежності (Task -> Task)
all_task_ids = list(TASKS.keys())
added_deps = []
```

```
for _ in range(N_NOISE_DEPS):
    t1, t2 = random.sample(all_task_ids, 2)
    u, v = f"T_{t1}", f"T_{t2}"
    if not G.has_edge(u, v):
        G.add_edge(u, v, edge_type='dependency',
color='red', is_noise=True)
        added_deps.append(f"{u}->{v}")

#випадкові призначення (Resource -> Task)
all_res_ids = list(RESOURCES.keys())
added_assigns = []
for _ in range(N_NOISE_ASSIGNS):
    r_id = random.choice(all_res_ids)
    t_id = random.choice(all_task_ids)
    u, v = f"R_{r_id}", f"T_{t_id}"
    if not G.has_edge(u, v):
        G.add_edge(u, v, edge_type='capability',
color='red', is_noise=True)
        added_assigns.append(f"{u}->{v}")
# print("   [DONE] Граф модифіковано (внесено шум).")
# print(f"\n2. Діагностика циклічних залежностей:")
# підграф тільки з задач
task_nodes = [n for n, attr in G.nodes(data=True) if
attr.get('node_type') == 'task']
task_subgraph = G.subgraph(task_nodes)

try:
    # simple_cycles знаходить елементарні цикли
    cycles = list(nx.simple_cycles(task_subgraph))

    if cycles:
        print(f"   [CRITICAL] Знайдено {len(cycles)}
циклів! Це блокує виконання проекту.")
        print("   Приклади циклів:")
        for i, cycle in enumerate(cycles[:5]):
            print(f"     Cycle #{i+1}: {' ->
'.join(cycle)}")
        # виправити автоматично, видалємо "червоні"
ребра
    else:
        print("   [ОК] Циклів не виявлено (пощастило,
рандом не створив кілець).")

except Exception as e:
    print(f"   [ERROR] Помилка пошуку циклів: {e}")

# --- шукаємо помилкові призначення  ---
# print(f"\n3. Перевірка валідності зв'язків 'Ресурс-
Задача':")
invalid_assignments = []
# перебираємо ребра
for u, v, attrs in G.edges(data=True):
    # ребра типу 'capability' (Resource -> Task)
    if u.startswith("R_") and v.startswith("T_"):
        r_id = int(u.split("_")[1])
        t_id = int(v.split("_")[1])
        # чи є задача в skill
```

```
        if t_id not in
RESOURCES[r_id]['specific_skills']:
            invalid_assignments.append(f"{u}
({RESOURCES[r_id]['name']}) -> {v}")

if invalid_assignments:
    print(f"   [WARNING] Знайдено
{len(invalid_assignments)} невалідних призначень
(Resource lack skills):")
    for item in invalid_assignments:
        print(f"     X: {item}")
else:
    print("   [ОК] Всі зв'язки в графі відповідають
кваліфікації ресурсів.")

print(f"\n[КЛАСТЕРИЗАЦІЯ] Знайдено {len(components)}
незалежних компонент (груп).")
```

```
# zкщо 1 компонент - все пов'язано, ынакше э групи
задач+ресурсів, що  перетинаються.
for i, comp in enumerate(components):
    tasks_in_comp = [n for n in comp if
n.startswith('T_')]
    res_in_comp = [n for n in comp if
n.startswith('R_')]
    print(f"  - Група {i+1}: Задач {len(tasks_in_comp)},
Ресурсів {len(res_in_comp)}")
# ваги
W_INTEGRAL = {'coverage': 0.4, 'time': 0.3, 'util': 0.2,
'profile': 0.1}
MAX_COVER_SCORE = sum(SEVERITY_W[val] for val in
REQ_INFO.values())
```

B.6 Моделювання та аналіз

```
# розрахунок інтегральної
def calculate_integral_score(schedule, covered_reqs):
    if not schedule: return 0.0, {}

    curr_cov = sum(SEVERITY_W[REQ_INFO[r]] for r in
covered_reqs)
    k1 = curr_cov / MAX_COVER_SCORE if MAX_COVER_SCORE >
0 else 0

    makespan = max(i['end'] for i in schedule.values())
    k2 = max(0, 1 - (makespan / PROJECT_DEADLINE))

    work = sum((TASKS[t]['dur'] *
RESOURCES[d['res']]['eff']) for t, d in
schedule.items())
    cap = N_RES * makespan
    k3 = work / cap if cap > 0 else 0

    # помилки  - не в prefs)
    errs = sum(1 for t, d in schedule.items() if t not
in RESOURCES[d['res']]['specific_prefs'])
    k4 = 1 - (errs / len(schedule))

    integral = (W_INTEGRAL['coverage']*k1 +
W_INTEGRAL['time']*k2 +
                W_INTEGRAL['util']*k3 +
W_INTEGRAL['profile']*k4)

    return integral, {
        'K1_norm': k1, 'K1_raw': curr_cov,
        'K2_norm': k2, 'K2_raw': makespan,
        'K3_norm': k3, 'K3_raw': k3*100,
        'K4_norm': k4, 'K4_raw': errs
    }# моделювання ГА

problem = TestModel()
algorithm = NSGA2(pop_size=50,
                  n_offsprings=50,
                   sampling=HybridSampling(),
                   crossover=HybridCrossover(),
mutation=HybridMutation(), eliminate_duplicates=True)

st1 = time.time()
res = minimize(problem, algorithm, ('n_gen', 25),
seed=1, verbose=True)
et1 = time.time()
sols = []
for i, x in enumerate(res.X):
    seq, ass = x[:N_TASKS], x[N_TASKS:]
    sch, cov, _ = problem.decode(seq, ass)
    integ, det = calculate_integral_score(sch, cov)
    sols.append({'id': i, 'x': x, 'seq': seq, 'assign':
ass, 'obj': res.F[i],
                 'integral': integ, 'details': det,
'schedule': sch})

sols.sort(key=lambda s: s['integral'], reverse=True)

# евристичне рішення
st2 = time.time()
heur_res = run_heuristic_baseline()
h_time, h_ineff, h_cov, h_k, h_int = heur_res
et2 = time.time()

# точне від OR-Tools
st3 = time.time()
or_res = run_ortools_comparison()
et3 = time.time()

if or_res:
    # розпаковка результатів
    or_time, or_ineff, or_cov, or_k = or_res[:4]
    best_ga_idx = np.argmin(res.F[:, 0])
    # перерахуємо для кращого ГА по часу
    x_best = res.X[best_ga_idx]
    sch_ga, cov_ga, _ = problem.decode(x_best[:N_TASKS],
x_best[N_TASKS:])
    # оцінка та деталі ГА
    ga_int, ga_k = calculate_integral_score(sch_ga,
cov_ga)
    ga_time, ga_ineff, ga_cov = res.F[best_ga_idx][0],
res.F[best_ga_idx][1], -res.F[best_ga_idx][2]
    # інтегральна для OR-Tools ---
    or_int = (W_INTEGRAL['coverage'] * or_k['K1_norm'] +
              W_INTEGRAL['time']     * or_k['K2_norm'] +
              W_INTEGRAL['util']     * or_k['K3_norm'] +
              W_INTEGRAL['profile']  * or_k['K4_norm'])
```