НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

До захисту допущено:
Завідувач кафедри
_____ Оксана ТИМОЩУК
«___»_____2026 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системний аналіз і управління»

спеціальності 124 «Системний аналіз»

на тему: «Математичне моделювання задачі розв'язання програмних залежностей методами теорії ґраток та булевих алгебр»

Виконав:
студент IV курсу, групи КА-21
Мачула Антон Іванович _____

Керівник:
Доцент кафедри ММСА, к.ф.-м.н., доц.
Спекторський Ігор Якович _____

Консультант з економічного розділу:
Доцент кафедри ЕК, к.е.н., доц.
Рощина Надія Василівна _____

Консультант з нормоконтролю:
Доцент кафедри ММСА, к.ф.-м.н.
Статкевич Віталій Михайлович _____

Рецензент:
Доцент кафедри МА та ТЙ, к.ф.-м.н., доц.
Ільєнко Андрій Борисович _____

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

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 124 «Системний аналіз»
Освітня програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Оксана ТИМОЩУК
«__» _____ 2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

Мачулі Антону Івановичу

1. Тема роботи «Математичне моделювання задачі розв’язання програмних залежностей методами теорії ґраток та булевих алгебр», керівник роботи Спекторський Ігор Якович, доцент кафедри ММСА, затверджені наказом по університету від 27.05.2026 р. № 1944-с.

2. Термін подання студентом роботи: «10» червня 2026 р.

3. Вихідні дані до роботи: формальна модель задачі розв’язання програмних залежностей, прототип резолвера на мові Python з використанням бібліотек PySAT та Z3, результати обчислювальних експериментів на синтетичних репозиторіях.

4. Зміст роботи: 1. Теоретичний апарат; 2. Математична модель задачі розв’язання залежностей; 3. Програмна реалізація та експериментальне дослідження; 4. Функціонально-вартісний аналіз програмного продукту.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація.

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
Економічний	доцент, к.е.н., Рощина Н.В.	

7. Дата видачі завдання: 20.02.2026

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Вивчення літератури за темою роботи	18.04.2026	Виконано
2	Підготовка першого розділу	25.04.2026	Виконано
3	Підготовка другого розділу	01.05.2026	Виконано
4	Підготовка третього розділу	05.05.2026	Виконано
5	Проведення обчислювальних експериментів	29.05.2026	Виконано
6	Підготовка економічної частини	31.05.2026	Виконано
7	Оформлення розділів відповідно до нормоконтролю	01.06.2026	Виконано
8	Підготовка презентації доповіді	05.06.2026	Виконано
9	Оформлення дипломної роботи	07.06.2026	Виконано

Студент Антон МАЧУЛА

Керівник Ігор СПЕКТОРСЬКИЙ

РЕФЕРАТ

Дипломна робота: 69 с., 9 рис., 6 табл., 2 додатки, 15 джерел.

ТЕОРІЯ ҐРАТОК, ІДЕАЛИ ПОРЯДКУ, ТЕОРЕМА БІРКГОФА, БУЛЕВІ АЛГЕБРИ, ЗАДАЧА ЗДІЙСНЕННОСТІ, SAT, SMT, РОЗВ’ЯЗАННЯ ПРОГРАМНИХ ЗАЛЕЖНОСТЕЙ.

Об’єкт дослідження – задача розв’язання програмних залежностей у пакетних репозиторіях.

Предмет дослідження – алгебраїчні структури (частково впорядковані множини, ґратки ідеалів порядку, булеві алгебри), що виникають при формалізації задачі інсталяційної сумісності, та методи їх обчислювальної реалізації засобами SAT і SMT солверів.

Мета роботи – побудувати математичну модель задачі розв’язання програмних залежностей засобами теорії ґраток і булевих алгебр, дослідити умови збереження та руйнування ґраткової структури простору коректних інсталяцій, реалізувати прототип системи розв’язання залежностей та провести обчислювальні експерименти, що ілюструють теоретичні твердження.

Методи дослідження – апарат теорії частково впорядкованих множин і ґраток, теорії булевих алгебр, теорії обчислювальної складності; застосування бібліотеки PySAT для роботи із SAT-солверами, системи Z3 для SMT та мови програмування Python для реалізації прототипу.

Актуальність – існуючі менеджери пакетів реалізують розв’язання залежностей переважно евристично або прямим викликом SAT-солвера, тоді як алгебраїчна структура задачі залишається неявною; формалізація в термінах теорії ґраток дозволяє відокремити поліноміально розв’язні випадки від тих, де структура руйнується і потрібен SAT-солвер.

Результати роботи – побудовано математичну модель задачі та доведено, що множина коректних інсталяцій у безконфліктному випадку є

дистрибутивною ґраткою ідеалів порядку, яка руйнується при введенні обмежень несумісності; реалізовано прототип резолвера з SAT- та SMT-кодуваннями та механізмом видобування читабельного UNSAT-ядра; проведено функціонально-вартісний аналіз варіантів реалізації.

Шляхи подальшого розвитку предмету дослідження – розширення моделі на випадок графа залежностей з циклами через апарат нерухомих точок, дослідження апроксимаційних алгоритмів для випадків з частковою ґратковою структурою та інтеграція з реальними пакетними менеджерами через адаптери до їхніх форматів метаданих.

ABSTRACT

Diploma thesis: 69 p., 9 fig., 6 tables, 2 appendicies, 15 sources.

LATTICE THEORY, ORDER IDEALS, BIRKHOFF'S THEOREM, BOOLEAN ALGEBRAS, SATISFIABILITY PROBLEM, SAT, SMT, SOFTWARE DEPENDENCY RESOLUTION**.**

Object of Research – the software dependency resolution problem in package repositories.

Subject of Research – the algebraic structures (partially ordered sets, lattices of order ideals, Boolean algebras) arising in the formalization of the installation compatibility problem, and the methods of their computational implementation by means of SAT and SMT solvers.

Objective – to build a mathematical model of the software dependency resolution problem using lattice theory and Boolean algebras, to investigate the conditions under which the lattice structure of the space of correct installations is preserved or destroyed, to implement a prototype dependency resolver, and to carry out computational experiments illustrating the theoretical claims.

Research Methods – the apparatus of the theory of partially ordered sets and lattices, the theory of Boolean algebras, and computational complexity theory; the use of the PySAT library for working with SAT solvers, the Z3 system for SMT, and the Python programming language for implementing the prototype.

Relevance – existing package managers solve dependency resolution mostly heuristically or by directly invoking a SAT solver, while the algebraic structure of the problem remains implicit; a lattice-theoretic formalization makes it possible to separate the polynomially solvable cases from those where the structure breaks down and a SAT solver is required.

Results – a mathematical model of the problem was built and it was proved that the set of correct installations in the conflict-free case is a distributive lattice of order ideals, which is destroyed upon the introduction of incompatibility

constraints; a resolver prototype with SAT and SMT encodings and a readable UNSAT-core extraction mechanism was implemented; a functional-cost analysis of the implementation variants was carried out.

Future Development Paths – extending the model to dependency graphs with cycles via the fixed-point apparatus, investigating approximation algorithms for cases with partial lattice structure, and integrating with real package managers through adapters to their metadata formats.

ЗМІСТ

ВСТУП

Сучасні програмні системи будуються з модулів, бібліотек та компонентів, кожен з яких, можливо, залежить від інших. Кількість таких компонентів у промислових екосистемах вимірюється сотнями тисяч: репозиторій PyPI містить понад 500 000 пакетів, npm – понад 2,5 мільйона, кожен з яких має множину версій і набір вимог до версій інших пакетів. Ручне розв’язання задачі сумісності таких пакетів неможливе; вона є фундаментальною обчислювальною задачею кожного сучасного менеджера пакетів.

Задача розв’язання залежностей формулюється так: задано репозиторій пакетів із відношеннями залежності та несумісності й користувацький запит на встановлення певних пакетів – необхідно знайти множину пакетів, що задовольняє всі обмеження, або довести, що такої множини не існує. Результат проєкту EDOS [1] показав, що ця задача є NP-повною в загальному випадку, проте зводиться до задачі здійсненності булевих формул (SAT), яку сучасні солвери розв’язують ефективно навіть для великих екземплярів.

РОЗДІЛ 1. ТЕОРЕТИЧНИЙ АПАРАТ

Цей розділ подає математичні поняття та результати, на яких ґрунтується модель розв’язання залежностей. Виклад побудований за принципом послідовної конкретизації: від найзагальнішої структури (частково впорядкована множина) до найконкретнішої (двоелементна булева алгебра, в якій працює SAT-солвер). Кожен наступний клас структур наслідує властивості попереднього і додає нові, що дозволяють отримати потрібні результати

1.1 Частково впорядковані множини та ґратки

Частково впорядкована множина (посет) – це пара $(P, \leq)$, де P – непорожня множина, а $\leq$ – бінарне відношення на P, що задовольняє трьом аксіомам: рефлексивності ($a \leq a$ для всіх $a \in P$), антисиметричності (з $a \leq b$ і $b \leq a$ випливає $a = b$) та транзитивності (з $a \leq b$ і $b \leq c$ випливає $a \leq c$) [2].

Якщо для пари елементів $a, b \in P$ не виконано ні $a \leq b$, ні $b \leq a$, такі елементи називаються незрівнянними. Якщо будь-які два елементи P зрівнянні, посет називається лінійно впорядкованим (ланцюгом).

Верхньою межею множини $S \subseteq P$ називається такий елемент $u \in P$, що $s \leq u$ для всіх $s \in S$. Найменша з верхніх меж, якщо існує, називається супремумом (точною верхньою межею) і позначається $\vee S$. Аналогічно вводиться інфімум $\wedge S$ – найбільша нижня межа. Для пари елементів супремум позначається $a \vee b$ (join), інфімум – $a \wedge b$ (meet).

Ґратка – це посет, в якому для будь-якої пари елементів a, b існують і $a \vee b$, і $a \wedge b$. Якщо в ґратці існують супремум і інфімум для будь-якої

підмножини (в тому числі нескінченної), ґратка називається повною. Для скінченних ґраток повнота тривіально випливає з означення.

Ґратка L називається дистрибутивною, якщо для будь-яких a, b, c $\in$ L виконано тотожність a $\wedge$ (b $\vee$ c) = (a $\wedge$ b) $\vee$ (a $\wedge$ c) (еквівалентно – двоїста тотожність a $\vee$ (b $\wedge$ c) = (a $\vee$ b) $\wedge$ (a $\vee$ c)) [2]. Дистрибутивність – це структурна властивість, що виокремлює серед усіх ґраток ті, які ізоморфні полю множин – тобто підґратці булеану 2^X деякої множини X.

Класичним прикладом дистрибутивної ґратки є сам булеан 2^X довільної множини X з операціями перетину та об’єднання. Ця ґратка фактично є вільною дистрибутивною ґраткою над X, у тому розумінні, що будь-яка скінченна дистрибутивна ґратка ізоморфно вкладається в булеан скінченної множини — твердження, що становить зміст теореми Біркгофа й буде сформульоване в наступному пункті.

Ширшим класом є модулярні ґратки – такі, для яких виконано наслідок a $\leq$ c $\Longrightarrow$ a $\vee$ (b $\wedge$ c) = (a $\vee$ b) $\wedge$ c. Кожна дистрибутивна ґратка модулярна, але не навпаки. Класичні приклади немодулярних ґраток – ґратка підгруп симетричної групи S_3 або ґратка N_5 (п’ятиелементна "пентагональна" ґратка). У контексті нашого дослідження інтерес становить саме перехід від дистрибутивних ґраток до структур, в яких порушується не тільки дистрибутивність, а й самé означення ґратки – наявність супремуму для довільної пари елементів.

1.2 Ідеали порядку та теорема Біркгофа

Підмножина I $\subseteq$ P посету (P, $\leq$) називається ідеалом порядку (англ. order ideal, або down-set), якщо вона замкнена «вниз» за відношенням $\leq$: для будь-яких a $\in$ I і b $\in$ P, якщо b $\leq$ a, то b $\in$ I. Іншими словами, ідеал порядку містить разом з кожним своїм елементом усі менші за нього.

Множина всіх ідеалів порядку посету P позначається L(P) (від англ. lattice of down-sets). Безпосередньо перевіряється, що перетин і об’єднання двох ідеалів порядку також є ідеалами порядку: якщо I, J $\in$ L(P), то I $\cap$ J, I $\cup$ J $\in$ L(P). Це означає, що L(P), впорядкована за включенням, утворює ґратку з операціями $\wedge = \cap$ і $\vee = \cup$. Більше того, ці операції успадковують від булеану 2^P дистрибутивність: ґратка L(P) дистрибутивна для будь-якого посету P.

Зворотне твердження є нетривіальним і становить зміст теореми Біркгофа [3]: будь-яка дистрибутивна ґратка L ізоморфна ґратці L(P) для деякого скінченного посету P. Більше того, такий посет P можна сконструювати однозначно (з точністю до ізоморфізму) як множину $\perp$-незвідних елементів L з успадкованим порядком. Елемент x $\in$ L називається $\vee$-незвідним, якщо з x = a $\vee$ b випливає x = a або x = b.

Це встановлює двоїстість між скінченними посетами і скінченними дистрибутивними ґратками: P $\mapsto$ L(P) та L $\mapsto$ (множина $\vee$-незвідних L).

Практичне значення теореми Біркгофа для задачі дослідження таке: якщо ми формалізуємо граф залежностей пакетного репозиторію як посет (пакет A передує пакету B, якщо B залежить від A), то множина коректних інсталяцій у безконфліктному випадку – це саме множина ідеалів порядку цього посету. Структура цього простору тоді повністю визначається структурою графа залежностей: вона є дистрибутивною ґраткою, і її розмір та форма обчислюються комбінаторно з посету.

Окремим важливим інструментом є теорема Кнастера–Тарського про нерухому точку в повних ґратках: для повної ґратки L і монотонного відображення f: L $\rightarrow$ L множина нерухомих точок f утворює непорожню повну підґратку L, зокрема містить найменшу та найбільшу нерухомі точки [4]. У контексті нашої задачі найменшу інсталяцію, що містить заданий набір пакетів, можна формально описати як найменшу нерухому точку монотонного оператора «додати усі залежності»; на практиці це обчислюється тривіальним транзитивним замиканням, тому теорему Кнастера–Тарського ми згадуємо лише для повноти теоретичної викладки.

1.3 Булеві алгебри та задача здійсненності

Булева алгебра – це дистрибутивна ґратка з найменшим (0) та найбільшим (1) елементами, кожен елемент якої має доповнення: для кожного a існує ¬a таке, що $a \wedge \neg a = 0$ і $a \vee \neg a = 1$. Доповнення в булевій алгебрі однозначне.

Найпростіша булева алгебра – двоелементна $\mathbb{2} = \{0, 1\}$ з операціями min і max. Найважливіший факт про скінченні булеві алгебри – теорема Стоуна про представлення: будь-яка скінченна булева алгебра ізоморфна $\mathbb{2}^n$ для деякого n, тобто є булеаном n-елементної множини. Це означає, що для скінченних булевих алгебр обчислення в них зводиться до обчислень у $\mathbb{2}$ покомпонентно – що, у свою чергу, є саме тим, що робить SAT-солвер.

Задача здійсненності булевих формул (SAT) формулюється так: задано булеву формулу φ від змінних $x_1, \ldots, x_n$ – необхідно визначити, чи існує таке присвоєння значень з $\mathbb{2}$ змінним, при якому φ набуває значення 1. Зазвичай формула задається в кон’юнктивній нормальній формі (КНФ) – як кон’юнкція диз’юнкцій літералів, де літерал – це змінна або її заперечення. Кожна диз’юнкція в такій нотації називається клаузою [5].

Задача SAT є NP-повною: будь-яка задача класу NP зводиться до неї за поліноміальний час (теорема Кука–Левіна). Незважаючи на це, сучасні SAT-солвери, засновані на алгоритмі CDCL (Conflict-Driven Clause Learning), розв’язують екземпляри з мільйонами змінних і клауз за хвилини – за рахунок використання структурних особливостей реальних задач, які зазвичай далекі від найгірших випадків теорії складності. Бібліотека PySAT [6] надає Python-інтерфейс до сімейства таких солверів (MiniSat, Glucose, Lingeling та інших) разом з допоміжними інструментами – зокрема механізмами видобування UNSAT-ядра.

UNSAT-ядро – це підмножина клауз початкової формули, яка сама по собі нездійсненна. Видобування мінімального UNSAT-ядра дозволяє пояснити, чому задача не має розв’язку, у термінах конкретного

підмножинного конфлікту, а не просто констатувати «UNSAT». Цей механізм важливий для прикладного розв’язання залежностей: програма, яка його використовує, може діагностувати, які саме обмеження утворюють конфлікт, а не лише повідомляти про відсутність розв’язку.

1.4 SMT як розширення SAT теоріями

SAT працює лише з пропозиційними змінними – атомарними сутностями, що не мають внутрішньої структури. Багато прикладних задач, однак, природно формулюються в термінах теорій вищого рівня: цілочисельної арифметики, арифметики дійсних чисел, теорії масивів, теорії невинтерпретованих функцій тощо. Задача SMT (Satisfiability Modulo Theories) розширює SAT: формула може містити не лише булеві змінні, але й атоми вигляду $(x + y < 5)$, $(a[i] = b[j])$ тощо, кожен з яких має семантику в фіксованій теорії [7].

SMT-солвер архітектурно складається з SAT-ядра, що працює з абстракцією формули як з пропозиційною, та так званих «теоретичних» розв’язувачів (theory solvers), що перевіряють несуперечність множини атомів у відповідній теорії. Цей підхід називається DPLL(T). Класичний SMT-солвер Z3 [8], розроблений Microsoft Research, підтримує найрозповсюдженіші теорії й широко використовується як у формальній верифікації, так і в задачах оптимізації.

У контексті розв’язання залежностей SMT дає важливу перевагу над чистим SAT: версії пакетів можна кодувати як цілочисельні змінні з обмеженнями виду $(version \geq 2.0)$, замість того щоб виділяти окрему булеву змінну на кожну версію кожного пакета й парами вводити клаузи «не більше однієї версії встановлено». Це призводить до зменшення розміру формули від $O(\text{пакетів} \times \text{версій}^2)$ до $O(\text{пакетів})$. Окрім самої здійсненності, Z3 підтримує оптимізацію через інтерфейс z3.Optimize: можна мінімізувати кількість

встановлених пакетів або максимізувати суму їхніх версій, отримуючи оптимальну (а не довільну) інсталяцію.

Висновки до розділу 1

У цьому розділі викладено математичний апарат, на якому ґрунтується модель наступного розділу. Послідовність понять – посет → ґратка → дистрибутивна ґратка → ґратка ідеалів порядку → булева алгебра – забезпечує концептуальний каркас, у який природно вписується кожна складова задачі розв’язання залежностей. Зокрема, теорема Біркгофа дає двоїстість між скінченними посетами та скінченними дистрибутивними ґратками, що буде використана в наступному розділі для опису структури простору коректних інсталяцій. Задача SAT як обчислювальна реалізація логіки на двоелементній булевій алгебрі та її розширення SMT через теорії над цілими числами утворюють інструментальний шар, який буде задіяно при програмній реалізації моделі.

РОЗДІЛ 2. МАТЕМАТИЧНА МОДЕЛЬ ЗАДАЧІ РОЗВ'ЯЗАННЯ ЗАЛЕЖНОСТЕЙ

Цей розділ будує формальну модель задачі розв'язання програмних залежностей на апараті, викладеному в розділі 1. Репозиторій формалізується як скінченний кортеж, над множиною його пакетів будується частковий порядок, а простір коректних інсталяцій у безконфліктному випадку ототожнюється з ґраткою ідеалів порядку цього посету. Центральним результатом розділу є твердження про те, що введення обмежень несумісності руйнує ґраткову структуру цього простору, і що це руйнування узгоджене з відомим переходом обчислювальної складності задачі від поліноміальної до NP-повної.

2.1 Формальна модель програмного репозиторію

Репозиторій формалізується як кортеж $R = (P, V, D, C)$, де:

1) P – скінченна множина імен пакетів;

2) $V: P \to 2^{\wedge}\mathbb{N}^k$ – функція, що кожному імені $p \in P$ ставить у відповідність непорожню множину версій (упорядкованих кортежів натуральних чисел);

3) D – відношення залежності: для кожної пари (p, v), де $p \in P$ і $v \in V(p)$, задано скінченну множину залежностей виду (ім'я цільового пакета, обмеження на версію);

4) C – відношення несумісності: аналогічна структура, але кожен запис означає, що відповідна пара пакетів не може бути встановлена одночасно.

Атомарною сутністю моделі є пара (ім'я, версія), яка надалі позначається pv (від англ. package-version). Версії впорядковано

лексикографічно як кортежі натуральних чисел, що покриває потреби семантичного версіонування (semver) без накладання жорстких вимог формату.

Обмеження на версію задається як кон’юнкція атомарних умов виду (==v), (>=v), (<=v), (>v), (<v), що відповідає підмножині специфікації PEP 440 [9], достатній для більшості реальних пакетів. Порожня кон’юнкція трактується як «будь-яка версія».

2.2 Граф залежностей та частковий порядок

Для побудови частково впорядкованої множини над множиною package-version пар розглядається відображення переходу, яке кожній pv ставить у відповідність множину тих pv, що задовольняють хоча б одну з її залежностей. Це класична AND-OR-структура: всі залежності пакета мають бути задоволені (кон’юнкція), але кожна окрема залежність може бути задоволена однією з кількох версій цільового пакета (диз’юнкція).

Цикли в графі залежностей логічно неможливі для коректних репозиторіїв: якщо A залежить від B, який залежить від A, то жоден з них не можна встановити, починаючи з порожньої множини. Тому надалі граф залежностей вважається ациклічним; формально це означає, що відношення транзитивної залежності є строгим частковим порядком без нескінченних спадних ланцюгів.

Після перевірки на ациклічність відношення залежності задає частковий порядок на множині package-version пар: $pv_1 \leq pv_2$ тоді і тільки тоді, коли pv_2 транзитивно залежить від pv_1. Цей посет ми позначатимемо $(P, \leq)$ – це математичний об’єкт, до якого застосовний теоретичний апарат розділу 1.

2.3 Ґратка ідеалів порядку та операції на ній

Коректною інсталяцією у безконфліктному випадку є будь-яка множина package-version пар I, замкнена вниз за відношенням залежності: для кожного pv $\in$ I усі залежності pv задоволені деяким елементом I. Це означає, що I – ідеал порядку посету (P, $\leq$), у термінології розділу 1. Ця властивість перевіряється безпосередньо: для кожного pv в I кожна його залежність має задовольнятися хоча б одним елементом I.

Найменший ідеал порядку, що містить задану множину «коренів», обчислюється транзитивним замиканням – обходом графа залежностей з детермінованим вибором конкретної версії для кожної залежності (за замовчуванням найменшої доступної). У безконфліктному випадку, коли кожна залежність однозначно розв’язується, це дає мінімальну інсталяцію – результат, що повністю розв’язує задачу для цього випадку без звернення до SAT-солвера.

Для малих посетів ґратку L(P) можна перерахувати повністю – перебором усіх 2^n підмножин з перевіркою умови замкненості вниз – і зобразити її діаграмою Гассе. Така візуалізація дозволяє наочно побачити структуру простору коректних інсталяцій (рис. 2.1).

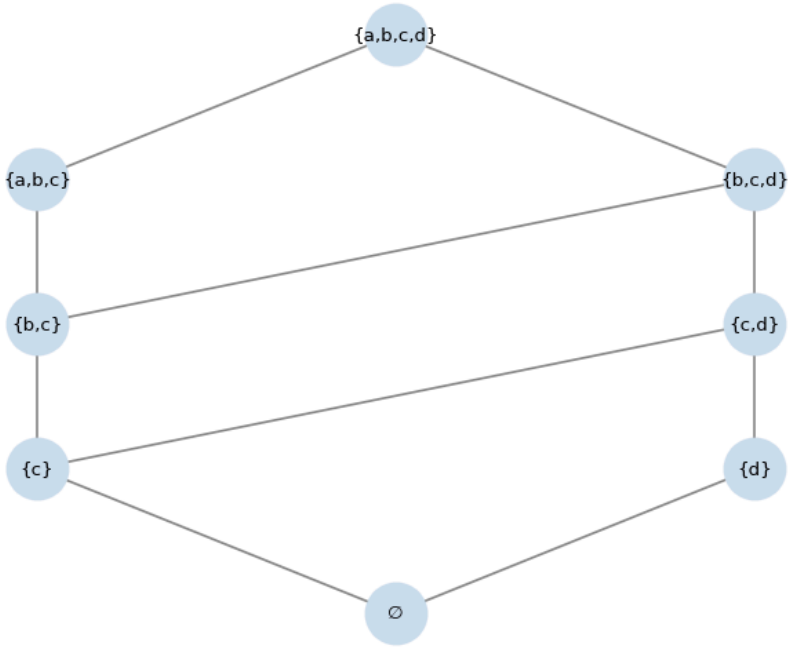


Рисунок 2.1 – Діаграма Гассе ґратки L(P) для маленького посету: ланцюг a > b > c та ізольований елемент d. Ґратка дистрибутивна, має десять ідеалів порядку.

Дистрибутивність ґратки L(P) перевіряється безпосередньо тотожністю $I \cap (J \cup K) = (I \cap J) \cup (I \cap K)$ для всіх трійок ідеалів. Для всіх безконфліктних репозиторіїв вона виконується, як і має бути за теоремою Біркгофа.

2.4 Втрата ґраткової структури при введенні обмежень несумісності

Введення відношення несумісності суттєво змінює структуру простору коректних інсталяцій. Тепер коректна інсталяція V – це ідеал порядку, що задовольняє дві додаткові умови: (а) для кожної пари $(pv_1, pv_2) \in C$ принаймні один з елементів не входить в V (умова конфлікту); (б) для

кожного імені p в V присутня не більше однієї версії (умова «не більше однієї версії встановлено»). Простір $V(R) \subseteq L(P)$ тепер строго менший за L(P).

Центральне теоретичне твердження роботи: V(R) у загальному випадку не є ґраткою. Перетин двох коректних інсталяцій $I, J \in V(R)$ залишається коректною інсталяцією (менше пакетів – менше можливих конфліктів, замкненість вниз зберігається), і таким чином V(R) є напівґраткою за перетином. Однак об'єднання може порушити умову конфлікту: якщо I і J містять відповідно лише по одному елементу з конфліктної пари, то $I \cup J$ містить обидва. Операція супремуму в L(P) (об'єднання плюс транзитивне замикання) виводить нас за межі V(R), тобто супремум I і J у V(R) може просто не існувати.

Це твердження ілюструється «руйнівним прикладом» – спеціально сконструйованим репозиторієм з п'яти пакетів {a, b, c, d, e}, де c залежить від a, d залежить від b, а e залежить від c і d. Без конфліктів ґратка L(P) має десять елементів і є дистрибутивною (рис. 2.2, ліва панель). Додавання єдиного конфлікту між c і d залишає в V(R) вісім елементів, серед яких два максимальні ({a, b, c} і {a, b, d}) – несумірні. Супремум цих двох елементів у V(R) не існує. Це означає, що V(R) тут навіть не є ґраткою (рис. 2.2, права панель).

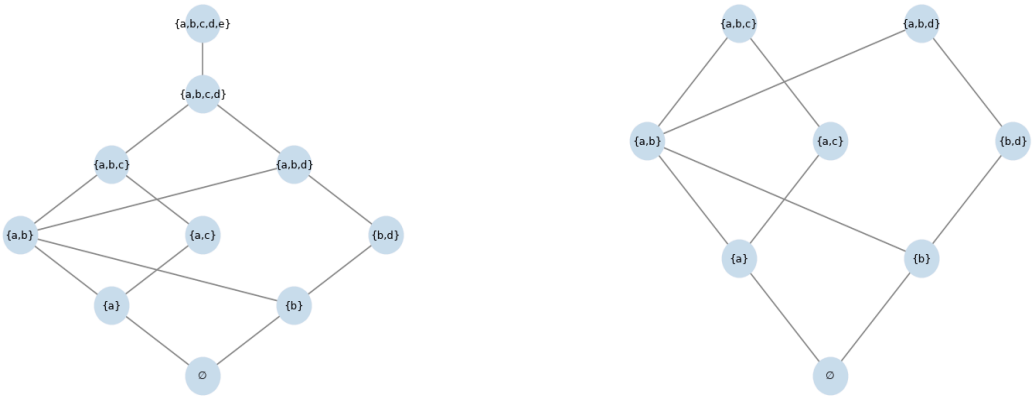


Рисунок 2.2 – Зліва: ґратка L(P) ідеалів порядку для безконфліктного репозиторію – дистрибутивна, десятиелементна. Справа: множина коректних інсталяцій V(P) для того самого посету з одним конфліктом (c × d) – два несумірні максимуми, структура більше не є ґраткою.

Перехід від ґратки до напівґратки супроводжується якісною зміною обчислювальної складності. У безконфліктному випадку мінімальну інсталяцію можна обчислити за лінійний час алгоритмом транзитивного замикання – це по суті обхід DAG. З моменту, коли конфлікти роблять задачу нетривіальною, поліноміальний алгоритм у загальному випадку відсутній (задача NP-повна [1]), і потрібно звертатись до SAT-солвера. Алгебраїчна структура простору розв’язків і обчислювальна складність, таким чином, узгоджені: руйнування ґратки маркує перехід від P до NP.

Висновки до розділу 2

У цьому розділі побудовано математичну модель задачі розв’язання програмних залежностей засобами теорії ґраток і булевих алгебр. Репозиторій формалізовано як кортеж із множини пакетів, версій, відношень залежності та несумісності; над ним побудовано посет та ґратку ідеалів порядку, що

відповідає простору коректних інсталяцій у безконфліктному випадку. За теоремою Біркгофа ця ґратка дистрибутивна і повністю визначається структурою графа залежностей. Доведено, що введення обмежень несумісності руйнує ґраткову структуру, переводячи простір у напівґратку за перетином, де операція супремуму у загальному випадку не визначена. Цей теоретичний перехід узгоджений з відомим обчислювальним переходом від поліноміальної задачі (алгоритм транзитивного замикання) до NP-повної (задача SAT), що й обґрунтовує потребу у методах, розглянутих у наступному розділі.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

Цей розділ описує програмну реалізацію моделі, побудованої в розділі 2, та результати обчислювальних експериментів. Спершу формальна модель зводиться до задачі здійсненності у двох кодуваннях – чистому SAT та SMT з цілочисельним представленням версій; обидві схеми супроводжуються обґрунтуванням коректності трансляції. Далі описуються експерименти, що перевіряють теоретичні твердження розділу 2 на синтетичних даних та оцінюють практичну застосовність підходу. Прототип реалізовано як єдиний модуль resolver.py . Використовуються бібліотеки PySAT (SAT-солвер Glucose-3), Z3 (SMT-солвер з підтримкою оптимізації), NetworkX і Matplotlib (візуалізація діаграм Гассе), Pandas (обробка експериментальних даних).

3.1 Кодування у задачу SAT з селекторними змінними

Кодування репозиторію та запиту на встановлення у вигляді SAT-формули у кон’юнктивній нормальній формі реалізує функція encode. Кожній парі package-version pv призначається окрема булева змінна; присвоєння змінній значення «істина» інтерпретується як «pv встановлено». Кодування породжує чотири категорії клауз:

1) клаузи залежностей: для кожної залежності пакета pv на пакет з ім’ям n з обмеженням c, виду $\neg pv \vee v_1 \vee \ldots \vee v_k$, де $v_1, \ldots, v_k$ - версії n, що задовольняють c;

2) клаузи конфлікту: для кожної пари (pv_1, pv_2), що конфліктують, виду $\neg pv_1 \vee \neg pv_2$;

3) клаузи «не більше однієї версії»: для кожного імені пакета n і кожної пари версій (v_i, v_j) виду $\neg v_i \vee \neg v_j$;

4) клаузи запиту: для кожного запиту на встановлення пакета n з обмеженням c, виду $v_1 \vee \ldots \vee v_k$, де $v_1, \ldots, v_k$ – версії n, що задовольняють c.

Особливістю реалізації є використання техніки селекторних змінних для видобування читабельного UNSAT-ядра у разі нездійсненності. Кожна породжена клауза розширюється однією додатковою змінною s_i (селектором), а сама клауза переписується у вигляді $\neg s_i \vee$ (оригінальна клауза). Селектор унікально пов'язаний з походженням клаузи: «dep[pv requires name(constraint)]», «conflict[pv_1 × pv_2]», «atmost[name: v_1 vs v_2]» або «request[name(constraint)]». При виклику солвера селектори передаються як припущення (assumptions). Якщо солвер повертає UNSAT, інструмент get_core() PySAT повертає підмножину припущень, що утворюють несумісне ядро; за допомогою зворотного відображення селектор → походження ми отримуємо читабельний опис конфліктних обмежень.

Цей механізм критично важливий для прикладної цінності інструмента. Стандартний результат SAT-солвера у разі UNSAT – це просто констатація «формула нездійсненна», що не допомагає користувачу зрозуміти, чому інсталяція неможлива. Селекторна техніка перетворює UNSAT-результат на пояснення виду «встановити пакет X неможливо, оскільки він залежить від Y, який конфліктує з Z, який також транзитивно потрібен». Функція explain виконує форматування цього ядра в людиночитний текст, групуючи клаузи за типом для зручності читання.

Реалізована функція solve огортає виклик PySAT, отримує модель (множину присвоєнь) у разі SAT і UNSAT-ядро у разі UNSAT, повертаючи структуру SolveResult з полями satisfiable, installation, unsat_core та encoding. Останнє поле зберігає повне кодування для подальшого використання – наприклад, для перевірки коректності проти результатів SMT-варіанту.

3.2 Кодування у задачу SMT з версіями як цілими числами

Чисто SAT-кодування має очевидний недолік: для пакета з n версіями створюється n булевих змінних і $O(n^2)$ клауз «не більше однієї версії». На пакетах з десятками версій це швидко роздуває розмір формули. SMT-формалізація знімає цю проблему, кодуючи стан пакета двома змінними: булевою $installed_n$ (чи встановлено пакет з ім'ям n) і цілочисельною $version_n$ (яку саме версію встановлено).

Цілочисельне представлення версії будується тривіально: версія v = (a, b, c) переходить у число $a \cdot 10^6 + b \cdot 10^3 + c$, що зберігає порядок для практично релевантних значень. Обмеження виду (>= 2.0) тоді стають звичайними цілочисельними нерівностями, а обмеження (== 2.0) - рівностями. Кожен запис про залежність переходить в імплікацію виду «якщо встановлено A версії v_a, то встановлено B з $version_\beta$, що задовольняє обмеження». Конфлікти кодуються аналогічно. Запит на встановлення стає кон'юнкцією тверджень $installed_n \wedge$ (обмеження на $version_n$).

Конкретний виграш SMT над SAT – у тому, що кількість змінних і фундаментальних обмежень тепер пропорційна кількості імен пакетів, а не кількості пар (ім'я, версія). На цьому ж кодуванні працює оптимізатор z3.Optimize, що дозволяє накладати цільові функції: solve_smt_minimize_packages мінімізує кількість встановлених пакетів (повертає найменшу інсталяцію, що задовольняє запит), solve_smt_maximize_versions максимізує суму версій (повертає інсталяцію з найновішими сумісними версіями). Жодна з цих оптимізацій недоступна у чистому SAT – там вони реалізуються через послідовне накладання обмежень-кардинальностей у спеціальних кодуваннях, тоді як Z3 інкапсулює оптимізацію в єдиному API.

Для перевірки коректності SMT-кодування в тестовому модулі прототипу порівнюються результати SAT- і SMT-варіантів на однакових входах. У безконфліктному детермінованому випадку (ланцюг з унікальною

резолюцією) обидва кодування повертають однакову інсталяцію. У випадках з оптимізаційними цілями SMT-результат перевіряється проти оптимуму, отриманого перебором, на малих репозиторіях.

3.3 Експериментальне дослідження

Експериментальна частина роботи виконана на синтетичних репозиторіях, згенерованих параметризованим генератором з контрольованими параметрами: розмір репозиторію, щільність залежностей, ймовірність конфлікту, кількість версій на пакет. Генератор детермінований при заданому значенні seed, що дозволяє відтворювати результати, а залежності встановлюються лише на пакети меншого рангу, що гарантує ациклічність графа за побудовою. Для кожної комбінації параметрів запускалось декілька прогонів з різними seed для отримання усереднених метрик.

Перший експеримент перевіряє центральне теоретичне твердження роботи – що введення конфліктів руйнує ґраткову структуру простору коректних інсталяцій. Для репозиторіїв з 4–10 однієї версії пакетів (single-version, щоб ізолювати ефект конфліктів від ефекту обмеження «не більше однієї версії»), при ймовірності конфлікту від 0 до 0.5, обчислювалась міра ґратковості (meet/join closure rate). Результати наведено на рисунку 3.1.

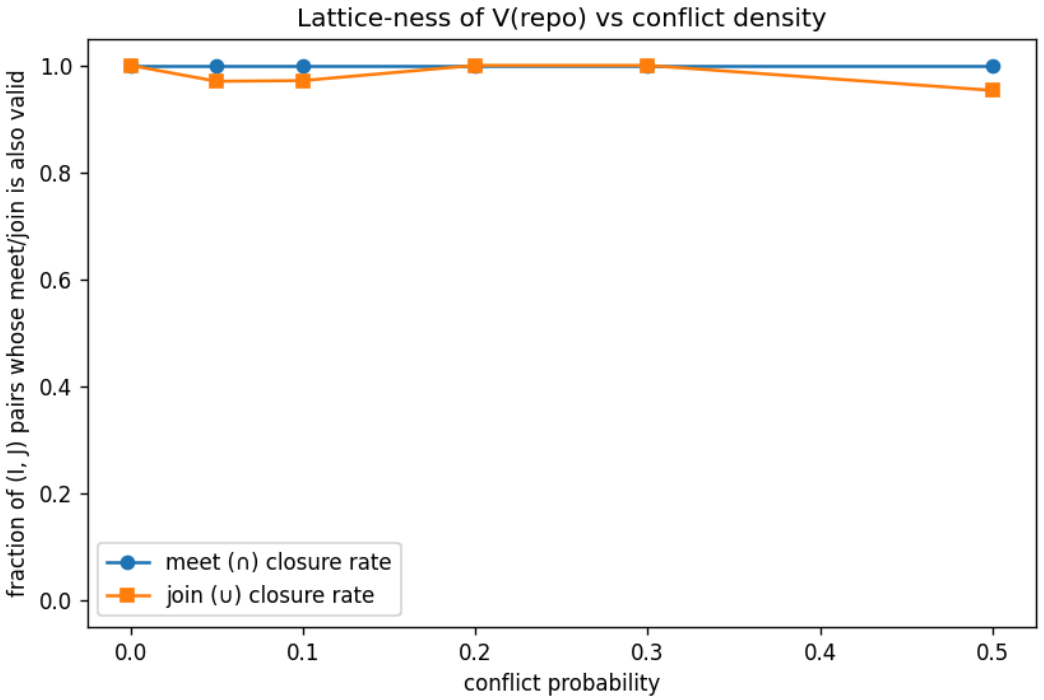


Рисунок 3.1 – Частка пар (I, J) коректних інсталяцій, чий перетин (meet) та об’єднання (join) також є коректними інсталяціями

Графік підтверджує теоретичне твердження: meet-rate (синя крива) залишається на рівні 1.0 для всіх щільностей конфлікту, що відповідає замкненості V(R) за операцією перетину. Натомість join-rate (помаранчева крива) падає нижче 1.0 при введенні конфліктів – це експериментальне свідчення того, що об’єднання двох коректних інсталяцій може вивести нас за межі V(R), й, отже, V(R) не є ґраткою. Сила ефекту в нашому генераторі помірна (зниження до ~0.95 при ймовірності конфлікту 0.5), оскільки випадково згенеровані конфлікти рідко формують структуру, що максимізує руйнування ґратки. Спеціально сконструйовані приклади (підрозділ 2.4) демонструють руйнування значно сильніше.

Другий експеримент вимірює час розв’язання задачі SAT як функцію розміру репозиторію та щільності конфлікту. Результати наведено на рисунку 3.2.

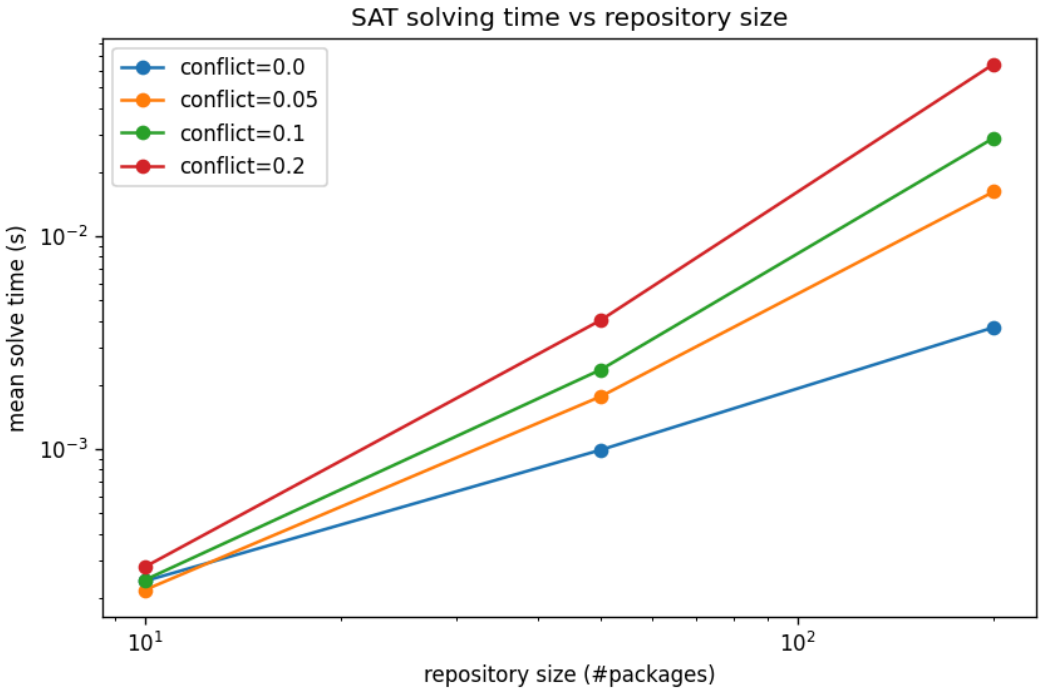


Рисунок 3.2 – Середній час розв’язання задачі SAT (логарифмічна шкала) як функція розміру репозиторію (логарифмічна шкала) для різних щільностей конфліктів.

У логарифмічних координатах залежність добре наближається лінією, що відповідає степеневій залежності у звичайних координатах. На репозиторіях розміром у 200 пакетів час розв’язання становить близько 30 мілісекунд при щільності конфлікту 20%, що свідчить про практичну застосовність підходу до репозиторіїв реалістичного розміру. Зростання щільності конфлікту збільшує час розв’язання приблизно на порядок (від ~3 мс до ~30 мс при n = 200).

Третій експеримент аналізує частоту випадків UNSAT як функцію щільності конфлікту для різних розмірів репозиторію (рис. 3.3).

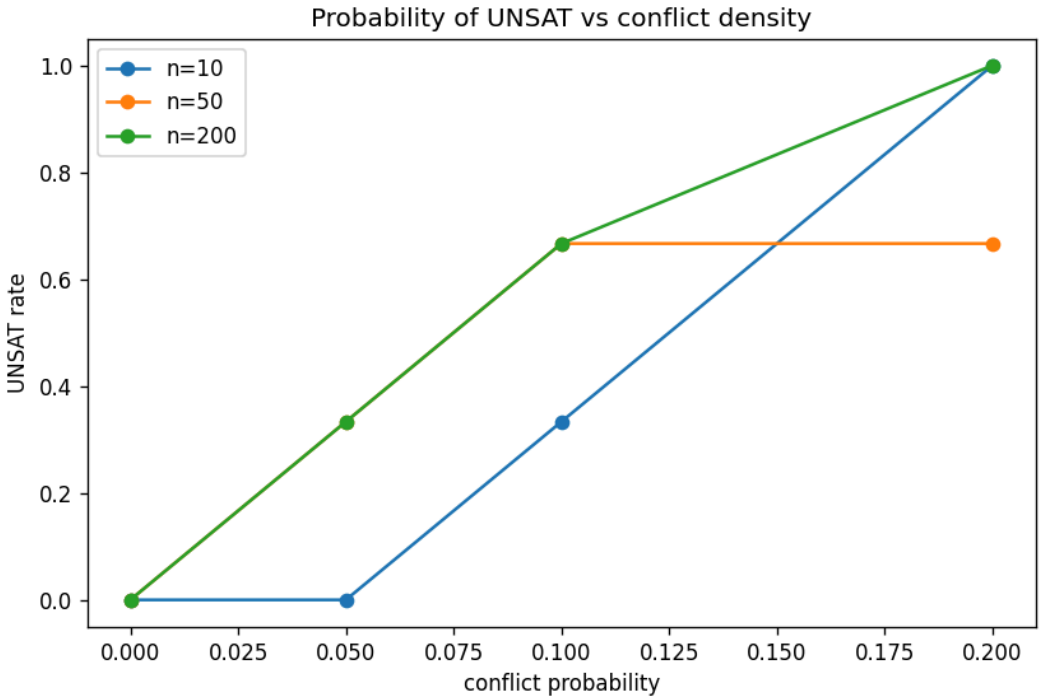


Рисунок 3.3 – Імовірність UNSAT як функція щільності конфлікту для трьох розмірів репозиторію.

Графік демонструє фазовий перехід: при низьких щільностях конфлікту майже всі екземпляри задовольняються, при високих – майже жодний. Положення переходу залежить від розміру репозиторію: чим більший репозиторій, тим раніше наступає насичення (більше пакетів – більше можливих конфліктів навіть при низькій ймовірності). Цей результат якісно відповідає типовій картині фазових переходів у випадкових SAT-екземплярах [10].

Прототип додатково надає командний інтерфейс (підмодуль cli) для практичного використання. Виклик виду «uv run resolver.py --repo example_broken.json --install e --explain» завантажує репозиторій із файлу, формулює запит на встановлення пакета e та друкує або список встановлених пакетів (у разі SAT), або читабельне UNSAT-пояснення. Опції --minimize-packages та --maximize-versions перемикають у режим SMT з відповідною оптимізаційною ціллю.

Висновки до розділу 3

У цьому розділі модель розділу 2 доведено до програмної реалізації та перевірено експериментально. Реалізовано прототип резолвера з двома кодуваннями – на основі чистого SAT через PySAT і на основі SMT через Z3 – а також механізм видобування читабельного UNSAT-ядра через техніку селекторних змінних. SMT-варіант демонструє переваги в розмірі кодування та підтримці оптимізації. Експерименти на синтетичних даних підтверджують теоретичні твердження про структуру простору розв’язків та демонструють практичну застосовність підходу до репозиторіїв реалістичного розміру.

РОЗДІЛ 4. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У розділі проведено техніко-економічне обґрунтування розробленого програмного продукту для розв’язання програмних залежностей із застосуванням функціонально-вартісного аналізу. Порівняно два конкуруючих варіанти реалізації резолвера – кодування задачі у вигляді SMT з цілочисельним представленням версій на основі солвера Z3 та кодування у вигляді чистого SAT на основі бібліотеки PySAT із технікою селекторних змінних. За результатами аналізу обчислено коефіцієнти технічного рівня та повну вартість розробки кожного варіанту, виконано обґрунтований вибір варіанту реалізації за критерієм техніко-економічного рівня.

4.1 Постановка задачі проектування

Метою етапу є створення інформаційно-аналітичного програмного комплексу, що автоматизує розв’язання задачі сумісності програмних пакетів за описом репозиторію з відношеннями залежності та несумісності. Розроблений продукт повинен виконувати наскрізне обчислення: завантаження опису репозиторію та користувацького запиту, побудову частково впорядкованої множини над парами «пакет - версія», зведення задачі до здійсненності булевих формул, виклик розв’язувача та формування результату - або коректної інсталяції, або читабельного пояснення неможливості встановлення.

Постановка задачі полягає в обґрунтованому виборі архітектури програмного продукту з-поміж конкуруючих варіантів реалізації за критерієм техніко-економічного рівня. Конкурують два підходи до кодування задачі: чисте SAT-кодування з окремою булевою змінною на кожну пару «пакет -

версія» та SMT-кодування з цілочисельним представленням версій. Перший дає компактний доступ до видобування UNSAT-ядра через селекторні змінні; другий зменшує розмір формули та надає вбудовану підтримку оптимізаційних цілей.

Технічні вимоги до програмного продукту є такими:

1) функціонування у стандартному середовищі мови Python із застосуванням наукових пакетів для роботи з SAT- та SMT-розв’язувачами;

2) підтримка діагностики нездійсненності у вигляді читабельного пояснення мінімального конфліктного набору обмежень;

3) забезпечення повної відтворюваності результатів розв’язання на синтетичних і реальних даних пакетних репозиторіїв;

4) зручне зберігання та повторне використання згенерованих синтетичних репозиторіїв разом із відповідною статистикою якості;

5) модульна архітектура, що дозволяє замінити розв’язувач без переписування конвеєра кодування;

6) мінімізація апаратних вимог при збереженні продуктивності, достатньої для пакетної обробки репозиторіїв реалістичного розміру на типовому робочому комп’ютері.

4.2 Обґрунтування функцій програмного продукту

З урахуванням сформульованих технічних вимог, у структурі програмного продукту виділено чотири основні функції, які визначають як технічні характеристики майбутньої системи, так і трудомісткість її розробки. Головна функція F0 – реалізація замкненого конвеєра розв’язання залежностей за описом репозиторію – розкладається на чотири підфункції.

F1 – вибір мови програмування:

а) Python з пакетами PySAT, Z3 та NetworkX;

б) C++ з прямою інтеграцією розв’язувача через нативний інтерфейс.

F2 – спосіб кодування задачі та розв’язувач:

а) SMT-кодування з цілочисельним представленням версій на основі солвера Z3 з підтримкою оптимізації;

б) чисте SAT-кодування на основі бібліотеки PySAT із технікою селекторних змінних.

F3 – механізм діагностики нездійсненності:

а) видобування читабельного UNSAT-ядра через селекторні змінні з відображенням «селектор → походження клаузи»;

б) повідомлення лише про факт нездійсненності без локалізації конфлікту.

F4 – тип обчислювальної інфраструктури:

а) локальний робочий комп’ютер;

б) розподілене середовище з орендованим хмарним кластером.

Запропоновані варіанти реалізації для всіх чотирьох функцій утворюють морфологічну множину, з якої формуються технічно сумісні комбінації. Аналіз кожного варіанту за критеріями переваг і недоліків зведено у позитивно-негативну матрицю, наведену в таблиці 4.1.

Таблиця 4.1 – Позитивно-негативна матриця варіантів реалізації функцій

Функція	**Варіант**	**Переваги**	**Недоліки**
F1	а) Python	Розвинена екосистема пакетів для SAT/SMT-розв’язування; швидке прототипування	Інтерпретована мова, нижча швидкодія базових циклів
F1	б) C++	Найвища швидкодія обчислювального ядра; повний контроль пам’яті	Тривалий цикл розробки; складніша інтеграція готових розв’язувачів
F2	а) SMT (Z3)	Компактне кодування O(пакетів); вбудована оптимізація через z3.Optimize	Складніша семантика; залежність від зовнішнього солвера
F2	б) SAT (PySAT)	Пряме керування клаузами; зручне видобування UNSAT-ядра	Розмір формули O(пакетів×версій²); оптимізація лише через кардинальності
F3	а) UNSAT-ядро	Читабельне пояснення конфлікту; висока прикладна цінність	Додаткові селекторні змінні збільшують формулу
F3	б) Лише факт	Мінімальний розмір кодування	Користувач не отримує причини неможливості інсталяції
F4	а) Локальний	Постійна доступність; відсутність орендних платежів	Обмежена потужність; одноразові апаратні витрати

Функція	Варіант	Переваги	Недоліки
F4	б) Хмарний	Лінійна масштабованість; доступ до новіших ресурсів	Постійні орендні платежі; складніший конвеєр відтворення

За результатами аналізу обрано такі варіанти підфункцій. Для F1 обрано варіант а) - мову Python, оскільки наскрізна підтримка SAT/SMT-розв'язувачів і розвинена екосистема переважають виграш у швидкодії від компільованої мови. Для F4 обрано варіант а) - локальний комп'ютер, оскільки розміри репозиторіїв у межах дослідження не вимагають хмарного масштабування. Конкуренція зосереджується у функціях F2 та F3, що й формує два варіанти реалізації для подальшого порівняння.

Варіант 1: F1(а) - F2(а) - F3(а) - F4(а), тобто Python, SMT-кодування на основі Z3 з оптимізацією та видобуванням пояснення, на локальному комп'ютері.

Варіант 2: F1(а) - F2(б) - F3(а) - F4(а), тобто Python, чисте SAT-кодування на основі PySAT із селекторними змінними та видобуванням UNSAT-ядра, на локальному комп'ютері.

4.3 Обґрунтування системи параметрів програмного продукту

Для подальшого кількісного порівняння двох варіантів реалізації необхідно зафіксувати систему техніко-економічних параметрів. Обрана система параметрів має охоплювати як операційні характеристики продукту, так і витрати на його розробку.

X1 – середній час розв'язання задачі для репозиторію реалістичного розміру (200 пакетів) при щільності конфлікту 20%, виміряний у мілісекундах. Параметр відображає операційну швидкість продукту при типовому регламенті використання.

X2 – якість діагностики нездійсненності, виміряна як середня частка коректно локалізованих конфліктних обмежень у згенерованому поясненні за тестовою вибіркою UNSAT-екземплярів, у відсотках. Параметр відображає прикладну цінність продукту для кінцевого користувача.

X3 – максимальний обсяг оперативної пам’яті, що його потребує продукт під час типового сценарію розв’язання пакету з 200 репозиторіїв, у мегабайтах.

X4 – трудомісткість розробки продукту, виражена у людино-годинах. Параметр відображає сукупні витрати часу розробника на реалізацію усіх компонент конвеєра.

Гірші, середні та кращі значення параметрів обрані з огляду на практичні вимоги до резолвера та на ресурсні характеристики типового робочого комп’ютера й зведені в таблиці 4.2.

Таблиця 4.2 – Основні параметри програмного продукту

Назва параметра	**Позн.**	**Од. виміру**	**Гірше**	**Середнє**	**Краще**
Час розв’язання задачі	X1	мс	120	40	10
Якість діагностики	X2	%	60	80	95
Обсяг оперативної пам’яті	X3	МБ	2048	1024	256
Трудомісткість розробки	X4	людино-години	1400	900	500

За даними таблиці 4.2 побудовано бальні характеристики кожного параметра, що задають перевідну відповідність між абсолютним значенням і шкалою бальних оцінок від 0 до 100. Бальна характеристика першого параметра наведена на рис. 4.1.

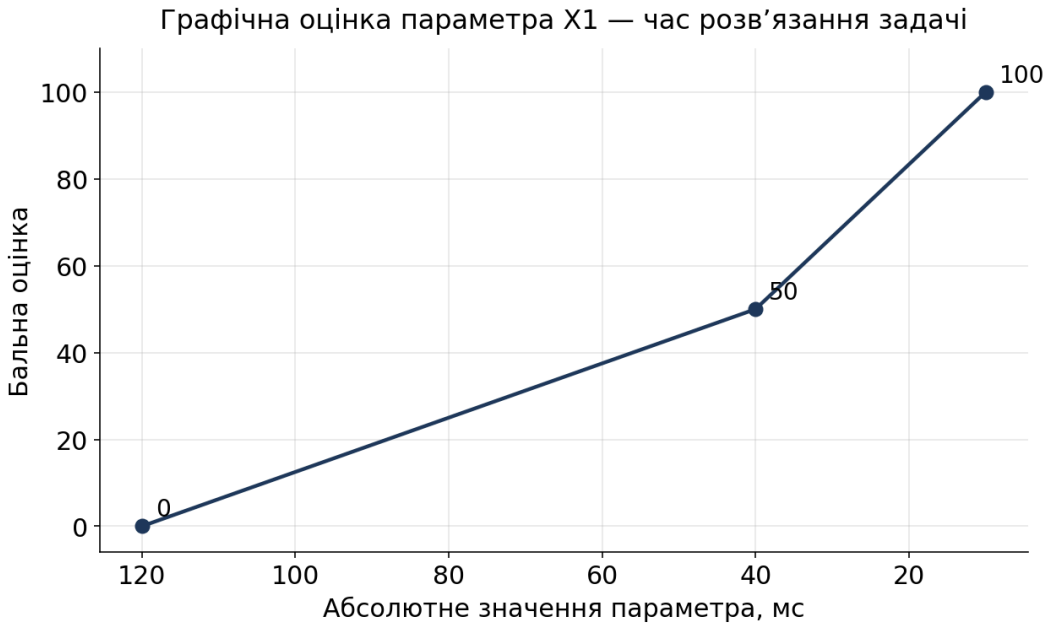


Рисунок 4.1 – X1, час розв’язання задачі

Бальна характеристика другого параметра наведена на рис. 4.2.

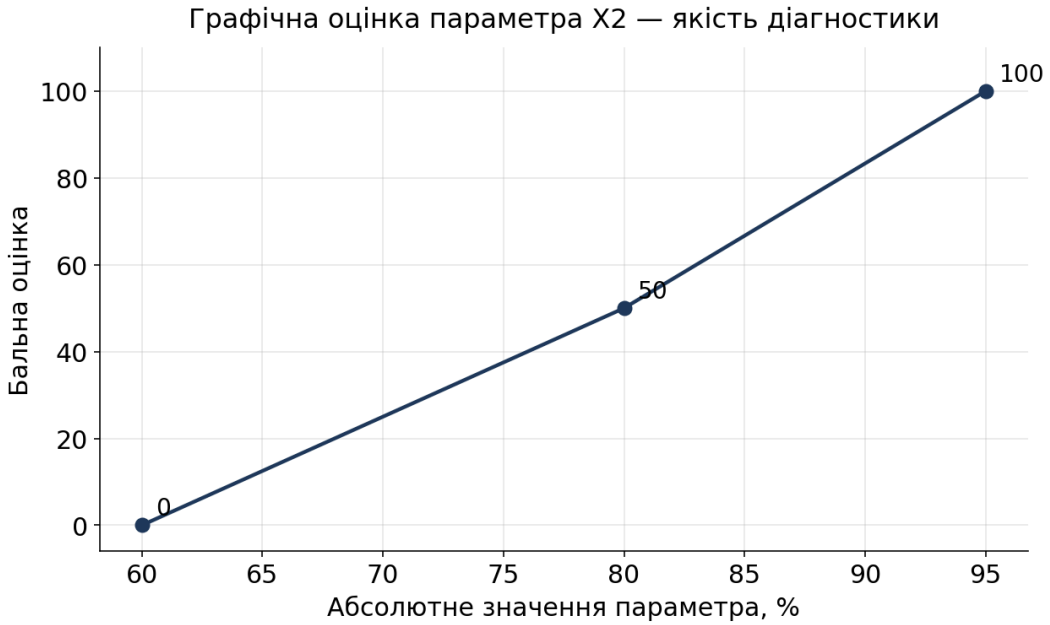


Рисунок 4.2 – X2, якість діагностики

Бальна характеристика третього параметра наведена на рис. 4.3.

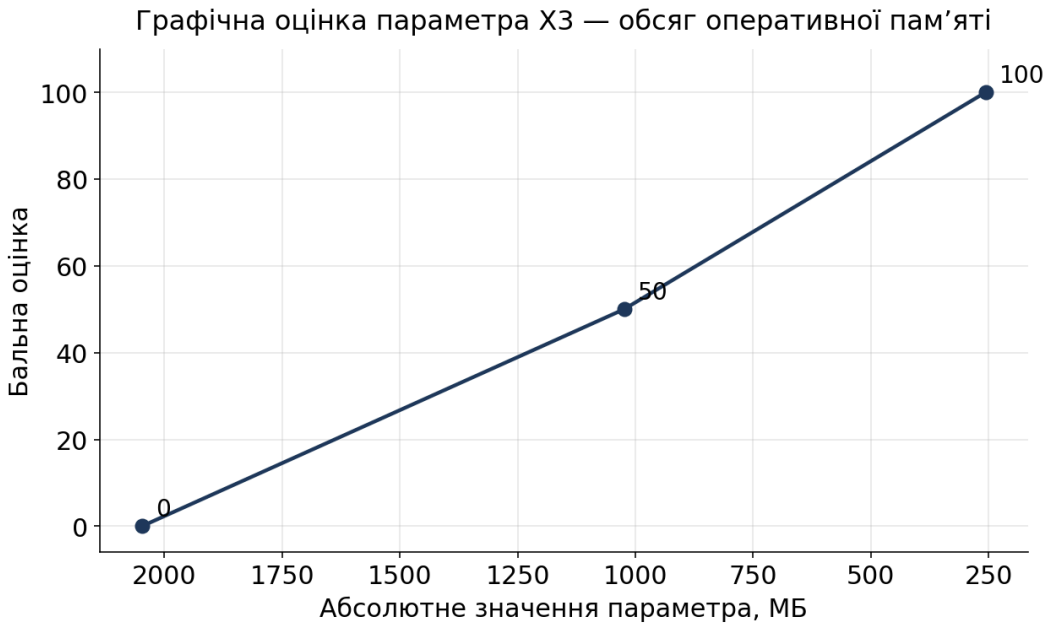


Рисунок 4.3 – X3, обсяг оперативної пам’яті

Бальна характеристика четвертого параметра наведена на рис. 4.4.

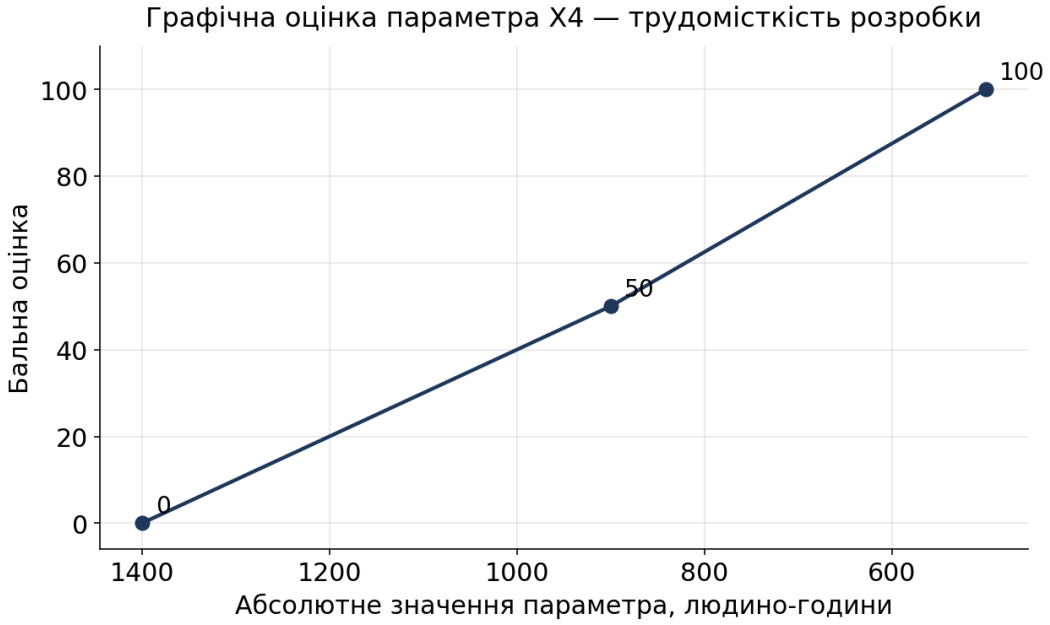


Рисунок 4.4 – X4, трудомісткість розробки

4.4 Аналіз експертного оцінювання параметрів

Значимість кожного з обраних параметрів встановлюється методом попарного порівняння на основі експертного ранжування. Експертна комісія складається із семи фахівців у галузі системного аналізу та розробки програмного забезпечення. Кожен експерт незалежно ранжує параметри за спаданням важливості.

Результати індивідуального ранжування зведено у таблиці 4.3 разом з підсумковими сумами рангів за кожним параметром, відхиленнями цих сум від середнього значення та квадратами відхилень.

Таблиця 4.3 – Результати ранжування параметрів

Позн.	**Назва параметра**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Сума Ri**	**Відх. Δi**	**Δi²**
X1	Час розв’язання	1	1	2	1	1	2	1	9	-8,5	72,25
X2	Якість діагностики	2	2	1	2	2	1	2	12	-5,5	30,25
X3	Обсяг пам’яті	4	4	4	4	3	4	4	27	9,5	90,25
X4	Трудомісткість	3	3	3	3	4	3	3	22	4,5	20,25
Разом		10	10	10	10	10	10	10	70	0	213,00

Для перевірки достовірності експертних оцінок розраховуються такі допоміжні величини. Загальна сума рангів за всіма параметрами обчислюється за формулою:

$$R = N \cdot n \cdot (n+1)/2, \tag{4.1}$$

де N – число експертів;

n – кількість параметрів.

У розглянутому випадку N=7, n=4, тож R=7·4·5/2=70, що повністю збігається з підсумковою сумою рангів у таблиці 4.3. Середня сума рангів обчислюється за формулою:

$$T = R/n \quad (4.2)$$

Середня сума рангів дорівнює T=70/4=17,5. Відхилення суми рангів кожного параметра від середньої суми та відповідні квадрати відхилень обчислено в останніх двох стовпцях таблиці 4.3. Сума квадратів відхилень становить S=213,00. Коефіцієнт узгодженості (конкордації) обчислюється за формулою:

$$W = 12 \cdot S / (N^2 \cdot (n^3 - n)) \quad (4.3)$$

Підставивши значення, отримуємо W = 12·213 / (49·(64−4)) = 2556 / 2940 = 0,869. Отримане значення W=0,869 істотно перевищує нормативний поріг 0,75, що засвідчує високу узгодженість думок експертів і робить результати ранжування придатними для подальшого розрахунку коефіцієнтів вагомості.

Для побудови матриці попарних порівнянь параметрів кожен експерт оцінює відносну важливість пар параметрів, використовуючи знаки «>», «=», «<». Підсумковий знак для кожної пари визначається за більшістю голосів, а числове значення переваги приймається 1,5 для переваги, 1,0 для рівнозначності та 0,5 для поступки. Результати попарного порівняння наведено у таблиці 4.4.

Таблиця 4.4 – Попарне порівняння параметрів

Параметри	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Кінцева оцінка**	**Числове значення**
X1 і X2	>	>	<	>	>	<	>	>	1,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X4	>	>	>	>	>	>	>	>	1,5
X3 і X4	<	<	<	<	>	<	<	<	0,5

На основі числових значень переваги формується матриця A=‖aij‖. Коефіцієнти вагомості Кві для кожного параметра обчислюються ітеративно. На першому кроці використовується сума елементів рядка матриці:

$$Кві(1) = Bi / \Sigma Bi, \quad (4.4)$$

де Bi – сума елементів і-го рядка матриці попарних порівнянь.

На наступних кроках використовується ітераційна формула, що враховує добуток матриці на попередній вектор вагомостей:

$$Кві(t) = Bi(t) / \Sigma Bi(t), \quad Bi(t) = \Sigma aij \cdot Кві(t-1) \quad (4.5)$$

Ітерації повторюються до тих пір, поки відмінність між значеннями вагомості на двох послідовних ітераціях не стане меншою за 2%. Розрахункові значення на першій, другій і третій ітераціях наведено у таблиці 4.5.

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметри	**X1**	**X2**	**X3**	**X4**	**Bi(1)**	**Кві(1)**	**Bi(2)**	**Кві(2)**	**Bi(3)**	**Кві(3)**
X1	1,0	1,5	1,5	1,5	5,5	0,344	21,25	0,360	77,875	0,361
X2	0,5	1,0	1,5	1,5	4,5	0,281	16,25	0,275	59,125	0,274
X3	0,5	0,5	1,0	0,5	2,5	0,156	9,25	0,157	34,125	0,158
X4	0,5	0,5	1,5	1,0	3,5	0,219	12,25	0,208	44,875	0,207
Разом					16,0	1,000	59,0	1,000	216,0	1,000

Як видно з порівняння стовпців Кві(2) і Кві(3), розбіжність значень вагомості на другій і третій ітерації не перевищує 1%. Тому як остаточні приймаються значення третьої ітерації: Кв1=0,361; Кв2=0,274; Кв3=0,158; Кв4=0,207. Найбільшу вагомість отримав параметр X1 (час розв’язання задачі), найменшу – X3 (обсяг оперативної пам’яті).

4.5 Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту реалізації визначається як зважена сума бальних оцінок параметрів зі знайденими у попередньому підрозділі коефіцієнтами вагомості. Коефіцієнт технічного рівня обчислюється за формулою:

$$KK = \Sigma\, Kві \cdot Bi, \qquad (4.6)$$

де *Кві* – коефіцієнт вагомості і-го параметра;

Bi – бальна оцінка і-го параметра, отримана з графіків на рис. 4.1 - рис. 4.4 за абсолютним значенням параметра у відповідному варіанті реалізації.

Абсолютні значення параметрів для двох варіантів реалізації отримано на основі результатів попередньої експериментальної оцінки прототипів обох підходів. Для варіанту 1 (SMT-кодування на основі Z3) середній час розв'язання задачі дорівнює 25 мс, якість діагностики 88%, обсяг пам'яті 900 МБ, трудомісткість розробки 980 людино-годин. Для варіанту 2 (чисте SAT-кодування на основі PySAT) середній час розв'язання задачі дорівнює 30 мс, якість діагностики 92%, обсяг пам'яті 700 МБ, трудомісткість розробки 860 людино-годин.

Бальні оцінки для кожного варіанту визначено лінійною інтерполяцією між гранично-середнім і середньо-кращим значеннями у відповідності до рис. 4.1 - рис. 4.4. Результати розрахунку рівня якості наведено у таблиці 4.6.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації

Параметри	**Кві**	**Абс. (вар.1)**	**Бал В1**	**Кві·В1**	**Абс. (вар.2)**	**Бал В2**	**Кві·В2**
X1	0,361	25 мс	83	29,96	30 мс	78	28,16
X2	0,274	88 %	73	20,00	92 %	89	24,39
X3	0,158	900 МБ	62	9,80	700 МБ	75	11,85
X4	0,207	980 г	79	16,35	860 г	86	17,80
KK				76,11			82,20

За даними таблиці 4.6 коефіцієнт технічного рівня варіанту 1 (SMT/Z3) становить KK1=76,11, варіанту 2 (SAT/PySAT) – KK2=82,20. Як видно з порівняння, варіант 2 переважає варіант 1 за технічним рівнем, насамперед

завдяки кращій якості діагностики, меншому споживанню пам’яті та нижчій трудомісткості розробки, тоді як варіант 1 має незначну перевагу лише за швидкодією.

4.6 Економічний аналіз варіантів розробки ПП

Економічний аналіз полягає у розрахунку повної вартості розробки кожного варіанту реалізації. Спершу визначається трудомісткість окремих завдань розробки, далі – заробітна плата, відрахування, витрати на машинний час та накладні витрати.

Загальна трудомісткість окремого завдання обчислюється за формулою:

$$T = TP \cdot КП \cdot КСК \cdot КМ \cdot КСТ \cdot КСТ.М \ , \qquad (4.7)$$

де TP – базова трудомісткість завдання (визначається з довідкових норм часу для відповідної комбінації групи новизни й складності алгоритму);

$КП$ – поправочний коефіцієнт складності; решта коефіцієнтів враховують вид інформації, мову програмування, ступінь використання стандартних модулів та засобів розробки.

Для першого завдання варіанту 1 (реалізація SMT-кодування з цілочисельним представленням версій та оптимізацією) базова трудомісткість дорівнює ТР=58 людино-днів, поправочні коефіцієнти дають загальну трудомісткість 48,72 людино-дня. Для другого завдання варіанту 1 (видобування пояснення на основі SMT-моделі) приймаємо ТР=34 людино-дні, що дає 27,03 людино-дня. Для третього завдання (конвеєр кодування та командний інтерфейс) приймаємо ТР=20 людино-днів, що дає 15,30 людино-дня.

Загальна трудомісткість розробки варіанту 1 у людино-годинах:

$$ТІ = (48{,}72 + 27{,}03 + 15{,}30) \cdot 8 = 728{,}40 \text{ людино-годин.} \qquad (4.8)$$

Для першого завдання варіанту 2 (реалізація SAT-кодування з селекторними змінними) приймаємо ТР=44 людино-дні, що дає 37,18

людино-дня. Для другого завдання (видобування UNSAT-ядра та форматування пояснення) приймаємо ТР=30 людино-днів, що дає 24,15 людино-дня. Для третього завдання (конвеєр кодування та командний інтерфейс) приймаємо ТР=20 людино-днів, що дає 15,30 людино-дня.

Загальна трудомісткість розробки варіанту 2 у людино-годинах:

$$ТІІ = (37{,}18 + 24{,}15 + 15{,}30) \cdot 8 = 613{,}04 \text{ людино-годин.} \quad (4.9)$$

У розробці бере участь один інженер-програміст з місячним окладом 32 000 грн. Середня погодинна оплата праці визначається за формулою:

$$СЧ = М / (21 \cdot 8) \quad (4.10)$$

Середня погодинна оплата праці становить СЧ = 32000 / (21·8) = 190,48 грн/год.

Заробітна плата за весь обсяг розробки розраховується за формулою СЗП = СЧ·Т·КД, де КД=1,2 – коефіцієнт доплат:

I. СЗП = 190,48·728,40·1,2 = 166 489,73 грн;

II. СЗП = 190,48·613,04·1,2 = 140 119,02 грн.

Відрахування на соціальні заходи становлять 22% від основної заробітної плати:

I. СВІД = 166 489,73·0,22 = 36 627,74 грн;

II. СВІД = 140 119,02·0,22 = 30 826,18 грн.

Окремо визначаються витрати на оплату однієї машино-години роботи ЕОМ. Одна ЕОМ обслуговує одного програміста з місячним окладом 32 000 грн при коефіцієнті зайнятості КЗ=0,2. Основна заробітна плата обслуговуючого персоналу за рік:

$$СГ = 12 \cdot М \cdot КЗ = 12 \cdot 32000 \cdot 0{,}2 = 76\,800 \text{ грн.} \quad (4.11)$$

З урахуванням доплат СЗП.ЕОМ = СГ·(1+КЗ) = 76 800·1,2 = 92 160 грн, а відрахування СВІД.ЕОМ = 92 160·0,22 = 20 275,20 грн. Амортизаційні відрахування за нормою амортизації 25% і договірною ціною робочого комп’ютера 38 000 грн з урахуванням коефіцієнта транспортування та монтажу 1,15: СА = 1,15·0,25·38 000 = 10 925,00 грн. Витрати на ремонт і профілактику: СР = 1,15·38 000·0,05 = 2 185,00 грн.

Ефективний річний фонд часу роботи ЕОМ становить ТЕФ = 742,40 год. Витрати на електроенергію за середньоспоживчою потужністю 0,3 кВт, коефіцієнтом зайнятості 0,2 і тарифом 13,64 грн/кВт·год: СЕЛ = 742,40·0,3·0,2·13,64 = 607,58 грн. Накладні витрати на обслуговування робочого місця: СН.ЕОМ = 38 000·0,67 = 25 460,00 грн.

Річні експлуатаційні витрати на ЕОМ становлять:

СЕКС = 92 160,00 + 20 275,20 + 10 925,00 + 2 185,00 + 607,58 + 25 460,00 = 151 612,78 грн. (4.12)

Собівартість однієї машино-години роботи ЕОМ: СМ-Г = СЕКС / ТЕФ = 151 612,78 / 742,40 = 204,22 грн/год. Оскільки всі роботи з розробки виконуються на ЕОМ, витрати на оплату машинного часу:

I. СМ = 204,22·728,40 = 148 754,85 грн;

II. СМ = 204,22·613,04 = 125 195,03 грн.

Накладні витрати у складі вартості розробки становлять 67% від основної заробітної плати:

I. СН = 166 489,73·0,67 = 111 548,12 грн;

II. СН = 140 119,02·0,67 = 93 879,74 грн.

Повна вартість розробки програмного продукту за варіантами:

I. СПП = 166 489,73 + 36 627,74 + 148 754,85 + 111 548,12 = 463 420,44 грн. (4.13)

II. СПП = 140 119,02 + 30 826,18 + 125 195,03 + 93 879,74 = 390 019,97 грн.

4.7 Вибір кращого варіанту ПП за техніко-економічним рівнем

Для остаточного вибору варіанту реалізації обчислюється коефіцієнт техніко-економічного рівня за формулою:

$$КТЕР = КК / СПП, \quad (4.14)$$

де *КК* – коефіцієнт технічного рівня варіанту;

СПП – повна вартість розробки варіанту, грн.

Підставивши обчислені раніше значення КК і СПП, отримуємо:

КТЕР1 = 76,11 / 463 420,44 = $1{,}642 \cdot 10^{-4}$ грн$^{-1}$;

КТЕР2 = 82,20 / 390 019,97 = $2{,}108 \cdot 10^{-4}$ грн$^{-1}$.

Як видно з порівняння значень коефіцієнта техніко-економічного рівня, варіант 2 (чисте SAT-кодування на основі PySAT із селекторними змінними) переважає варіант 1 (SMT-кодування на основі Z3) як за технічним рівнем, так і за повною вартістю розробки, що й зумовлює його вищий коефіцієнт техніко-економічного рівня.

Водночас слід окремо зазначити, що формальна перевага варіанту 2 за коефіцієнтом техніко-економічного рівня не суперечить рішенню, прийнятому в основній частині дослідження, де обидва кодування реалізовано та збережено у складі прототипу. SMT-варіант лишається цінним завдяки компактнішому кодуванню на репозиторіях з великою кількістю версій та вбудованій підтримці оптимізаційних цілей, недоступних у чистому SAT. Таким чином, для базового продукту обирається економічно ефективніший SAT-варіант, а SMT-кодування зберігається як розширення для задач оптимізації.

Висновки до розділу 4

У розділі проведено функціонально-вартісний аналіз двох конкуруючих варіантів реалізації програмного продукту для розв’язання програмних залежностей – SMT-кодування на основі солвера Z3 та чистого SAT-кодування на основі бібліотеки PySAT.

Сформульовано функціональну структуру продукту: на основі технічних вимог виділено чотири підфункції – вибір мови програмування, спосіб кодування задачі та розв’язувач, механізм діагностики нездійсненності й тип обчислювальної інфраструктури. За позитивно-негативною матрицею сформовано два технічно сумісні варіанти реалізації.

Здійснено експертне ранжування системи параметрів продукту: найбільшу вагомість отримав параметр X1 (час розв’язання задачі) зі

значенням коефіцієнта вагомості 0,361, найменшу – X3 (обсяг оперативної пам’яті) зі значенням 0,158. Коефіцієнт узгодженості думок експертів W=0,869 перевищує нормативний поріг, що підтверджує достовірність оцінок.

Виконано розрахунок коефіцієнта технічного рівня для обох варіантів: KK1=76,11 для SMT-варіанту та KK2=82,20 для SAT-варіанту. Виконано економічний розрахунок повної вартості розробки за обома варіантами: для SMT-варіанту повна вартість склала 463 420,44 грн, для SAT-варіанту – 390 019,97 грн.

За формальним критерієм техніко-економічного рівня перевага надається варіанту 2 (КТЕР2=$2{,}108 \cdot 10^{-4}$ проти КТЕР1=$1{,}642 \cdot 10^{-4}$), що відображає кращу економічну ефективність чистого SAT-кодування. Для базового продукту обрано SAT-варіант, тоді як SMT-кодування збережено як розширення для задач оптимізації.

ВИСНОВКИ

У роботі було розглянуто задачу розв’язання програмних залежностей у пакетних репозиторіях та досліджено її математичну структуру засобами теорії частково впорядкованих множин, ґраток і булевих алгебр. Задача сумісного встановлення пакетів була подана не лише як приклад прикладної задачі здійсненності, а як об’єкт з власною алгебраїчною структурою, що дозволяє точніше пояснити різницю між простими та складними випадками розв’язання залежностей.

Було виконано огляд необхідного теоретичного апарату: частково впорядкованих множин, ґраток, ідеалів порядку, теореми Біркгофа, булевих алгебр, SAT та SMT. На основі цього апарату сформульовано модель програмного репозиторію, у якій пакети, їхні версії, залежності та конфлікти описуються формально. Показано, що у безконфліктному випадку множина коректних інсталяцій має структуру дистрибутивної ґратки ідеалів порядку. Це означає, що такий клас задач може розв’язуватися конструктивно через транзитивне замикання залежностей без необхідності залучення SAT-солвера.

Окремо було досліджено вплив обмежень несумісності на простір коректних інсталяцій. Доведено, що введення конфліктів між пакетами руйнує ґраткову структуру: перетин двох коректних інсталяцій залишається коректним, але їх об’єднання може порушити обмеження сумісності. Унаслідок цього простір допустимих розв’язків у загальному випадку перестає бути ґраткою. Цей результат узгоджується з відомим фактом про зростання обчислювальної складності задачі розв’язання залежностей: від поліноміально розв’язного безконфліктного випадку до NP-повної задачі за наявності конфліктів.

У практичній частині роботи було реалізовано прототип системи розв’язання залежностей мовою Python. Для цього використано два підходи до кодування

задачі: SAT-кодування на основі бібліотеки PySAT та SMT-кодування на основі системи Z3. У SAT-варіанті кожній парі «пакет - версія» відповідає булева змінна, а залежності, конфлікти, обмеження на кількість версій і користувацький запит подаються у вигляді клауз. Додатково реалізовано механізм селекторних змінних, що дозволяє отримувати читабельне UNSAT-ядро та пояснювати причини неможливості встановлення певного набору пакетів.

SMT-кодування було розглянуто як альтернативний підхід, у якому факт встановлення пакета задається булевою змінною, а версія – цілочисельною. Такий підхід дозволяє компактніше описувати пакети з великою кількістю версій і природно формулювати оптимізаційні задачі, зокрема мінімізацію кількості встановлених пакетів або вибір найновіших сумісних версій. Таким чином, SAT-кодування є доцільним для базового розв’язання задачі здійсненності, тоді як SMT-кодування розширює модель у випадках, де важливими є числові обмеження та оптимізаційні критерії.

Було проведено експериментальне дослідження на синтетичних репозиторіях із контрольованими параметрами. Результати експериментів підтвердили теоретичні висновки: у безконфліктному випадку простір коректних інсталяцій зберігає ґраткові властивості, а зі збільшенням щільності конфліктів замкненість відносно об’єднання порушується. Також було проаналізовано залежність часу розв’язання SAT-задачі та частоти появи нездійсненних екземплярів від розміру репозиторію і щільності конфліктів. Отримані результати показали, що запропонований підхід є практично застосовним для репозиторіїв реалістичного розміру.

У межах функціонально-вартісного аналізу було порівняно два варіанти реалізації програмного продукту: варіант на основі SMT-солвера Z3 та варіант на основі SAT-солвера PySAT. Для кожного з варіантів було оцінено технічні параметри, рівень якості, витрати на розробку та техніко-економічний рівень. За результатами аналізу встановлено, що

базовим економічно доцільним варіантом реалізації є SAT-кодування, тоді як SMT-підхід доцільно використовувати як додатковий модуль для задач, пов’язаних з оптимізацією та числовим поданням версій.

Отже, у роботі було досягнуто поставленої мети: побудовано математичну модель задачі розв’язання програмних залежностей, досліджено умови збереження та руйнування ґраткової структури простору коректних інсталяцій, реалізовано прототип резолвера та проведено його експериментальну перевірку. Практична цінність роботи полягає в тому, що запропонований підхід не лише дозволяє розв’язувати задачу залежностей за допомогою SAT/SMT, а й пояснює, чому саме в певних випадках виникає необхідність переходу до таких методів.

Подальший розвиток роботи може бути пов’язаний із розширенням моделі на репозиторії з циклічними залежностями, використанням апарату нерухомих точок для опису таких структур, дослідженням наближених алгоритмів для випадків із частково збереженою ґратковою структурою, а також інтеграцією розробленого прототипу з реальними менеджерами пакетів через адаптери до їхніх форматів метаданих.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Managing the complexity of large free and open source package-based software distributions / F. Mancinelli et al. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. Tokyo, 2006. P. 199–208. URL: https://hal.science/hal-00149566 (Last accessed: 05.06.2026).
2. Спекторський І. Я., Стусь О. В. Дискретна математика : частково впорядковані множини, решітки, булеві алгебри : навч. посіб. Київ : НТУУ «КПІ», ННК «ІПСА», 2009. 140 с.
3. Birkhoff G. Rings of sets. *Duke Mathematical Journal*. 1937. Vol. 3, No 3. P. 443–454. DOI: 10.1215/S0012-7094-37-00334-X.
4. Tarski A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*. 1955. Vol. 5, No 2. P. 285–309. URL: https://msp.org/pjm/1955/5-2/p11.xhtml (Last accessed: 05.06.2026).
5. Handbook of Satisfiability / ed. by A. Biere et al. 2nd ed. Amsterdam : IOS Press, 2021. 1486 p.
6. Ignatiev A., Morgado A., Marques-Silva J. PySAT : a Python toolkit for prototyping with SAT oracles. *Theory and Applications of Satisfiability Testing — SAT 2018*. Cham : Springer, 2018. P. 428–437. (Lecture Notes in Computer Science ; vol. 10929). URL: https://pysathq.github.io/ (Last accessed: 05.06.2026).
7. *Satisfiability Modulo Theories* / C. Barrett et al. Handbook of Satisfiability. 2nd ed. Amsterdam : IOS Press, 2021. Chap. 33. P. 1267–1329.
8. de Moura L., Bjørner N. Z3 : An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Berlin ; Heidelberg : Springer, 2008. P. 337–340. (Lecture Notes in Computer Science ; vol. 4963). URL: https://github.com/Z3Prover/z3 (Last accessed: 05.06.2026).

9. Coghlan A. Version specifiers : PEP 440. Python Packaging Authority. 2014. URL: https://peps.python.org/pep-0440/ (Last accessed: 05.06.2026).
10. Mitchell D., Selman B., Levesque H. Hard and Easy Distributions of SAT Problems. *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*. San Jose, 1992. P. 459–465. URL: https://aaai.org/papers/00459-aaai92-071-hard-and-easy-distributions-of-sat-problems (Last accessed: 05.06.2026).
11. Stone M. H. The theory of representations for Boolean algebras. *Transactions of the American Mathematical Society*. 1936. Vol. 40, No 1. P. 37–111. DOI: 10.2307/1989664.
12. Di Cosmo R., Treinen R., Zacchiroli S. Formal Aspects of Free and Open Source Software Components. *Formal Methods for Components and Objects*. Cham : Springer, 2014. P. 216–239. (Lecture Notes in Computer Science ; vol. 8348).
13. Symbolic Model Checking : 10^{20} States and Beyond / J. R. Burch et al. *Information and Computation*. 1992. Vol. 98, No 2. P. 142–170. DOI: 10.1016/0890-5401(92)90017-A.
14. OPIUM : Optimal Package Install/Uninstall Manager / C. Tucker et al. *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. Minneapolis, 2007. P. 178–188.
15. Hagberg A. A., Schult D. A., Swart P. J. Exploring network structure, dynamics, and function using NetworkX. *Proceedings of the 7th Python in Science Conference (SciPy 2008)*. Pasadena, 2008. P. 11–15. URL: https://networkx.org/ (Last accessed: 05.06.2026).

ДОДАТОК А. ЛІСТИНГ ПРОТОТИПУ

```
# /// script
# requires-python = ">=3.11"
# dependencies = [
#     "matplotlib",
#     "networkx",
#     "python-sat",
#     "z3-solver",
#     "pandas",
# ]
# ///

# %% [markdown]
# # Dependency Resolver — Lattice and Boolean Methods
#
# Single-file prototype companion to a bachelor thesis on lattice-theoretic
# and Boolean-algebraic methods for software dependency resolution.
#
# Run end-to-end: `uv run resolver.py`
# Run cell-by-cell: open in Zed and use the REPL on each `# %%` block.

# %% Imports
from __future__ import annotations

from collections.abc import Callable, Iterable
from dataclasses import dataclass
from functools import cached_property
from pathlib import Path
import json
import random
import sys

# Skip auto-running tests and demos when invoked as a CLI ("--repo …"); the
# script then falls through to _main_cli() at the bottom. In Zed's REPL or
# under bare `uv run resolver.py`, _IS_CLI is False and every module's
# tests + demos fire inline as you go.
_IS_CLI = "--repo" in sys.argv

# %% [markdown]
# ## Module 1 — data model and repository parsing
#
# A repository is a set of `Package` records, each pinned to a specific
# `Version`. Dependencies and conflicts reference other packages by name and
# carry a `VersionConstraint`. The pair `(name, version)` is the atomic unit
# that later modules treat as a poset element / boolean variable / Z3 term;
# it is reified here as `PackageVersion`.

# %% Version
@dataclass(frozen=True, order=True)
class Version:
    parts: tuple[int, ...]

    @classmethod
    def parse(cls, s: str) -> Version:
        return cls(tuple(int(p) for p in s.strip().split(".")))

    def __str__(self) -> str:
        return ".".join(str(p) for p in self.parts)

# %% VersionConstraint
@dataclass(frozen=True)
class VersionConstraint:
    """Conjunction of (op, version) atoms. Empty atoms tuple = 'any'."""
    atoms: tuple[tuple[str, Version], ...] = ()

    _OPS = ("==", ">=", "<=", ">", "<")

    @classmethod
    def parse(cls, s: str) -> VersionConstraint:
        s = s.strip()
        if not s or s == "*":
            return cls(())
        atoms: list[tuple[str, Version]] = []
        for clause in s.split(","):
            clause = clause.strip()
            for op in cls._OPS:
                if clause.startswith(op):
                    atoms.append((op, Version.parse(clause[len(op):])))
                    break
            else:
                atoms.append(("==", Version.parse(clause)))
        return cls(tuple(atoms))

    def matches(self, v: Version) -> bool:
        for op, t in self.atoms:
            if op == "==" and not (v == t): return False
            if op == ">=" and not (v >= t): return False
            if op == "<=" and not (v <= t): return False
            if op == ">"  and not (v >  t): return False
            if op == "<"  and not (v <  t): return False
        return True

    def __str__(self) -> str:
        if not self.atoms:
            return "*"
        return ",".join(f"{op}{v}" for op, v in self.atoms)

# %% Dependency / PackageVersion / Package
@dataclass(frozen=True)
class Dependency:
    name: str
    constraint: VersionConstraint = VersionConstraint()

@dataclass(frozen=True, order=True)
class PackageVersion:
    name: str
    version: Version

    def __str__(self) -> str:
        return f"{self.name}=={self.version}"

@dataclass(frozen=True)
class Package:
    name: str
    version: Version
    depends: tuple[Dependency, ...] = ()
    conflicts: tuple[Dependency, ...] = ()

    @property
    def pv(self) -> PackageVersion:
        return PackageVersion(self.name, self.version)

# %% Repository
@dataclass
class Repository:
    packages: tuple[Package, ...]

    @cached_property
    def by_name(self) -> dict[str, tuple[Package, ...]]:
        out: dict[str, list[Package]] = {}
        for p in self.packages:
            out.setdefault(p.name, []).append(p)
        return {k: tuple(sorted(v, key=lambda p: p.version)) for k, v in out.items()}

    @cached_property
    def by_pv(self) -> dict[PackageVersion, Package]:
        return {p.pv: p for p in self.packages}

    def versions_matching(self, dep: Dependency) -> tuple[Package, ...]:
        return tuple(
            p for p in self.by_name.get(dep.name, ())
            if dep.constraint.matches(p.version)
        )

# %% JSON parser
```

```
            def parse_repository(data: dict) -> Repository:
                packages: list[Package] = []
                for p in data["packages"]:
                    depends = tuple(
                        Dependency(d["name"], VersionConstraint.parse(d.get("constraint", "")))
                        for d in p.get("depends", [])
                    )
                    conflicts = tuple(
                        Dependency(c["name"], VersionConstraint.parse(c.get("constraint", "")))
                        for c in p.get("conflicts", [])
                    )
                    packages.append(Package(
                        name=p["name"],
                        version=Version.parse(p["version"]),
                        depends=depends,
                        conflicts=conflicts,
                    ))
                return Repository(tuple(packages))

            def dump_repository(repo: Repository) -> dict:
                return {
                    "packages": [
                        {
                            "name": p.name,
                            "version": str(p.version),
                            "depends": [
                                {"name": d.name, "constraint": str(d.constraint)} for d in p.depends
                            ],
                            "conflicts": [
                                {"name": c.name, "constraint": str(c.constraint)} for c in p.conflicts
                            ],
                        }
                        for p in repo.packages
                    ]
                }

            def load_repository(path: str | Path) -> Repository:
                return parse_repository(json.loads(Path(path).read_text()))

            def save_repository(repo: Repository, path: str | Path) -> None:
                Path(path).write_text(json.dumps(dump_repository(repo), indent=2))

            # %% Synthetic generator
            @dataclass
            class GeneratorParams:
                num_packages: int
                versions_per_package: int
                avg_deps_per_package: float
                conflict_probability: float
                seed: int = 0

            def generate_repository(params: GeneratorParams) -> Repository:
                rng = random.Random(params.seed)
                names = [f"pkg{i:03d}" for i in range(params.num_packages)]
                pkgs: list[Package] = []
                for rank, name in enumerate(names):
                    for v_idx in range(params.versions_per_package):
                        version = Version((1, v_idx, 0))
                        depends: list[Dependency] = []
                        if rank > 0:
                            n = max(0, int(round(rng.gauss(params.avg_deps_per_package, 0.5))))
                            n = min(n, rank)
                            for cand in rng.sample(names[:rank], n):
                                depends.append(Dependency(cand, VersionConstraint(())))
                        conflicts: list[Dependency] = []
                        for other in names[:rank]:
                            if rng.random() < params.conflict_probability:
                                conflicts.append(Dependency(other, VersionConstraint(())))
                        pkgs.append(Package(name, version, tuple(depends), tuple(conflicts)))
                return Repository(tuple(pkgs))

            # %% Tests — Module 1
            def _test_module_1() -> None:
                assert Version.parse("1.2.0") < Version.parse("1.2.1")
                assert Version.parse("2.0.0") > Version.parse("1.999.0")

                c = VersionConstraint.parse(">=2.0,<3.0")
                assert c.matches(Version.parse("2.5.0"))
                assert not c.matches(Version.parse("3.0.0"))
                assert not c.matches(Version.parse("1.9.0"))
                assert VersionConstraint.parse("*").matches(Version.parse("0.1.0"))
                assert VersionConstraint.parse("==2.0.0").matches(Version.parse("2.0.0"))
                assert not VersionConstraint.parse("==2.0.0").matches(Version.parse("2.0.1"))
                assert VersionConstraint.parse("2.0.0").matches(Version.parse("2.0.0"))  # bare == ==

                repo = Repository((
                    Package("flask", Version.parse("2.0.0"),
                            depends=(Dependency("werkzeug", VersionConstraint.parse(">=2.0")),)),
                    Package("werkzeug", Version.parse("2.0.0")),
                    Package("werkzeug", Version.parse("2.1.0")),
                ))
                assert parse_repository(dump_repository(repo)).packages == repo.packages

                matches = repo.versions_matching(
                    Dependency("werkzeug", VersionConstraint.parse(">=2.0"))
                )
                assert {str(p.version) for p in matches} == {"2.0.0", "2.1.0"}

                params = GeneratorParams(num_packages=20, versions_per_package=2,
                                         avg_deps_per_package=2.0, conflict_probability=0.05,
                                         seed=42)
                assert generate_repository(params).packages == generate_repository(params).packages

                print("Module 1 tests passed")

            if not _IS_CLI: _test_module_1()

            # %% [markdown]
            # ## Module 2 — dependency poset and the lattice of order ideals
            #
            # The dependency relation `pv' ≤ pv` reads "pv' is required to install pv".
            # For each PackageVersion we record the set of every version that could
            # satisfy one of its constraints (the AND-OR fan-out). A *valid installation*
            # is a downward-closed subset under this relation — an order ideal.
            #
            # In the conflict-free single-resolved case (every dep has one matching
            # version), the set of order ideals is a distributive lattice with
            # `meet = ∩` and `join = closure(∪)`. The renderer in this module draws that
            # lattice; distributivity is checked on small examples.

            # %% Dependency edges
            def dependency_edges(repo: Repository) -> dict[PackageVersion, frozenset[PackageVersion]]:
                """For each pv, the set of all package-versions that could satisfy any
                of its dependencies (the AND-OR successors). An unresolvable dep
                contributes no successors; downstream predicates will reject installations
                that include such a pv."""
                edges: dict[PackageVersion, frozenset[PackageVersion]] = {}
                for p in repo.packages:
                    out: set[PackageVersion] = set()
                    for dep in p.depends:
                        for cand in repo.versions_matching(dep):
                            out.add(cand.pv)
                    edges[p.pv] = frozenset(out)
                return edges

            # %% Cycle detection (iterative DFS)
            class CycleError(ValueError):
                pass

            def find_cycle(
```

```
                edges: dict[PackageVersion, frozenset[PackageVersion]],
            ) -> list[PackageVersion] | None:
                WHITE, GRAY, BLACK = 0, 1, 2
                color: dict[PackageVersion, int] = {pv: WHITE for pv in edges}
                parent: dict[PackageVersion, PackageVersion | None] = {}

                for start in list(edges):
                    if color[start] != WHITE:
                        continue
                    color[start] = GRAY
                    parent[start] = None
                    stack: list[tuple[PackageVersion, Iterable[PackageVersion]]] = [
                        (start, iter(edges[start]))
                    ]
                    while stack:
                        u, it = stack[-1]
                        pushed = False
                        for v in it:
                            cv = color.get(v, BLACK)
                            if cv == WHITE:
                                color[v] = GRAY
                                parent[v] = u
                                stack.append((v, iter(edges.get(v, frozenset()))))
                                pushed = True
                                break
                            if cv == GRAY:
                                cycle = [v]
                                cur: PackageVersion | None = u
                                while cur is not None and cur != v:
                                    cycle.append(cur)
                                    cur = parent.get(cur)
                                cycle.append(v)
                                cycle.reverse()
                                return cycle
                        if not pushed:
                            color[u] = BLACK
                            stack.pop()
                return None

            def check_acyclic(repo: Repository) -> dict[PackageVersion, frozenset[PackageVersion]]:
                edges = dependency_edges(repo)
                cycle = find_cycle(edges)
                if cycle is not None:
                    raise CycleError("Dependency cycle: " + " -> ".join(str(pv) for pv in cycle))
                return edges

            # %% Order ideal predicate
            def is_order_ideal(
                installation: frozenset[PackageVersion] | set[PackageVersion],
                repo: Repository,
            ) -> bool:
                """Downward closure under the dependency relation: every pv in the
                installation has each of its declared dependencies satisfied by at
                least one element of the installation."""
                for pv in installation:
                    pkg = repo.by_pv.get(pv)
                    if pkg is None:
                        return False
                    for dep in pkg.depends:
                        if not any(c.pv in installation for c in repo.versions_matching(dep)):
                            return False
                return True

            # %% Transitive closure (least order ideal containing roots)
            def transitive_closure(
                roots: Iterable[PackageVersion],
                repo: Repository,
                choose: Callable[[tuple[Package, ...]], Package] = lambda cs: cs[0],
            ) -> frozenset[PackageVersion]:
                """BFS: for each new pv, add one matching version of each of its deps
                via the `choose` rule (default: lowest by sort order). Deterministic;
                in the single-resolved case this is the principal order ideal of `roots`."""
                out: set[PackageVersion] = set()
                queue: list[PackageVersion] = list(roots)
                while queue:
                    pv = queue.pop(0)
                    if pv in out:
                        continue
                    pkg = repo.by_pv.get(pv)
                    if pkg is None:
                        raise KeyError(f"Unknown package-version: {pv}")
                    out.add(pv)
                    for dep in pkg.depends:
                        cands = repo.versions_matching(dep)
                        if not cands:
                            raise ValueError(
                                f"Unresolvable dep {dep.name}{dep.constraint} required by {pv}"
                            )
                        queue.append(choose(cands).pv)
                return frozenset(out)

            # %% Lattice operations on order ideals
            def meet(
                I: frozenset[PackageVersion],
                J: frozenset[PackageVersion],
            ) -> frozenset[PackageVersion]:
                """Set intersection. Preserves order-idealness when every dep has a
                unique satisfier (the single-resolved case)."""
                return I & J

            def join(
                I: frozenset[PackageVersion],
                J: frozenset[PackageVersion],
                repo: Repository,
            ) -> frozenset[PackageVersion]:
                """Closure of union — the smallest order ideal containing both."""
                return transitive_closure(I | J, repo)

            # %% Enumerate order ideals (brute force, small repos only)
            def enumerate_order_ideals(
                repo: Repository,
                limit: int = 16,
            ) -> list[frozenset[PackageVersion]]:
                """All order ideals of the dependency poset. 2^n scan; guard via `limit`."""
                all_pvs = [p.pv for p in repo.packages]
                n = len(all_pvs)
                if n > limit:
                    raise ValueError(
                        f"Refusing to enumerate 2^{n} subsets; raise `limit` if intentional"
                    )
                out: list[frozenset[PackageVersion]] = []
                for mask in range(1 << n):
                    S = frozenset(all_pvs[i] for i in range(n) if (mask >> i) & 1)
                    if is_order_ideal(S, repo):
                        out.append(S)
                return out

            # %% Hasse diagram renderer for the lattice of order ideals
            def _default_pv_label(pv: PackageVersion) -> str:
                return str(pv)

            def render_hasse_lattice(
                repo: Repository,
                title: str = "Lattice of order ideals",
                save_to: str | Path | None = None,
                pv_label: Callable[[PackageVersion], str] = _default_pv_label,
                filter_fn: Callable[[frozenset[PackageVersion]], bool] | None = None,
                ax=None,
            ):
                import matplotlib.pyplot as plt
                import networkx as nx

                ideals = enumerate_order_ideals(repo)
                if filter_fn is not None:
                    ideals = [I for I in ideals if filter_fn(I)]
                ideals_sorted = sorted(ideals, key=lambda S: (len(S), tuple(sorted(pv_label(pv) for pv in S))))
                G: "nx.DiGraph" = nx.DiGraph()
                for I in ideals_sorted:
                    G.add_node(I)
                for I in ideals_sorted:
                    for J in ideals_sorted:
                        if I < J and not any(I < K < J for K in ideals_sorted):
                            G.add_edge(I, J)

                by_rank: dict[int, list[frozenset[PackageVersion]]] = {}
                for I in ideals_sorted:
                    by_rank.setdefault(len(I), []).append(I)
                pos: dict[frozenset[PackageVersion], tuple[float, float]] = {}
                for rank, group in by_rank.items():
                    for i, I in enumerate(group):
                        pos[I] = (i - (len(group) - 1) / 2, rank)
                labels = {
```

```
                I: "∅" if not I else "{" + ",".join(sorted(pv_label(pv) for pv in I)) + "}"
                for I in ideals_sorted
            }

            own_fig = ax is None
            if own_fig:
                max_label = max(len(s) for s in labels.values())
                width = max(8.0, len(ideals_sorted) * 0.5 + max_label * 0.12)
                fig, ax = plt.subplots(figsize=(width, 6))
            nx.draw_networkx_edges(G, pos, ax=ax, arrows=False, edge_color="#888")
            nx.draw_networkx_nodes(G, pos, ax=ax, node_color="#cde", node_size=700)
            nx.draw_networkx_labels(G, pos, labels=labels, ax=ax, font_size=8)
            ax.set_title(title)
            ax.set_axis_off()
            ax.margins(x=0.15, y=0.1)
            if save_to is not None:
                ax.figure.savefig(save_to, dpi=120, bbox_inches="tight")
            return ax


        # %% Distributivity check
        def is_distributive(repo: Repository) -> bool:
            """Brute-force check that L(repo) satisfies the distributive law
            I ∧ (J ∨ K) = (I ∧ J) ∨ (I ∧ K) for all ideals. Birkhoff guarantees
            this whenever L(repo) is genuinely the order-ideal lattice of a poset
            (i.e. every dep has a unique satisfier)."""
            ideals = enumerate_order_ideals(repo)
            for I in ideals:
                for J in ideals:
                    for K in ideals:
                        if meet(I, join(J, K, repo)) != join(meet(I, J), meet(I, K), repo):
                            return False
            return True


        # %% Tests — Module 2
        def _test_module_2() -> None:
            # A small single-resolved repo: a → b → c, plus an isolated d.
            a = Package("a", Version.parse("1.0.0"),
                        depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
            b = Package("b", Version.parse("1.0.0"),
                        depends=(Dependency("c", VersionConstraint.parse("==1.0.0")),))
            c = Package("c", Version.parse("1.0.0"))
            d = Package("d", Version.parse("1.0.0"))
            repo = Repository((a, b, c, d))

            edges = check_acyclic(repo)
            assert edges[a.pv] == frozenset({b.pv})
            assert edges[b.pv] == frozenset({c.pv})
            assert edges[c.pv] == frozenset()

            # cycle detection
            bad_a = Package("a", Version.parse("1.0.0"),
                            depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
            bad_b = Package("b", Version.parse("1.0.0"),
                            depends=(Dependency("a", VersionConstraint.parse("==1.0.0")),))
            try:
                check_acyclic(Repository((bad_a, bad_b)))
                assert False, "expected CycleError"
            except CycleError:
                pass

            # is_order_ideal
            assert is_order_ideal(frozenset(), repo)
            assert is_order_ideal(frozenset({c.pv}), repo)
            assert is_order_ideal(frozenset({c.pv, d.pv}), repo)
            assert not is_order_ideal(frozenset({a.pv}), repo)        # a needs b
            assert not is_order_ideal(frozenset({a.pv, b.pv}), repo)  # still needs c
            assert is_order_ideal(frozenset({a.pv, b.pv, c.pv}), repo)

            # transitive closure
            tc_a = transitive_closure([a.pv], repo)
            assert tc_a == frozenset({a.pv, b.pv, c.pv})
            # idempotence
            assert transitive_closure(tc_a, repo) == tc_a

            # lattice ops
            I = transitive_closure([a.pv], repo)
            J = frozenset({d.pv})
            assert meet(I, J) == frozenset()
            assert join(I, J, repo) == frozenset({a.pv, b.pv, c.pv, d.pv})

            # the order-ideal lattice of this single-resolved repo is distributive
            assert is_distributive(repo)

            # number of ideals: c-or-not × d-or-not × ... for chain a>b>c and isolated d
            # chain of length 3: 4 ideals (∅, {c}, {b,c}, {a,b,c}); ⊗ d-or-not: 8 total
            assert len(enumerate_order_ideals(repo)) == 8

            print("Module 2 tests passed")

        if not _IS_CLI: _test_module_2()

        # %% Demo — render the Hasse diagram of the small conflict-free repo
        def _demo_module_2() -> None:
            a = Package("a", Version.parse("1.0.0"),
                        depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
            b = Package("b", Version.parse("1.0.0"),
                        depends=(Dependency("c", VersionConstraint.parse("==1.0.0")),))
            c = Package("c", Version.parse("1.0.0"))
            d = Package("d", Version.parse("1.0.0"))
            repo = Repository((a, b, c, d))
            render_hasse_lattice(
                repo,
                title="L(P) for chain a>b>c and isolated d (distributive)",
                save_to="hasse_module2_demo.png",
                pv_label=lambda pv: pv.name,
            )
            print("Wrote hasse_module2_demo.png")

        if not _IS_CLI: _demo_module_2()

        # %% [markdown]
        # ## Module 3 — SAT encoding via PySAT
        #
        # Each PackageVersion becomes a boolean variable. Every logical unit of
        # constraint (one dependency edge, one pairwise conflict, one at-most-one
        # pair, one request atom) is encoded as a CNF clause guarded by a fresh
        # *selector* variable. The solver runs with all selectors assumed true; on
        # UNSAT, PySAT's assumption-based core extraction returns the blamed
        # selectors and we translate those back to their origin tags for a
        # human-readable diagnosis.

        # %% Clause + Encoding dataclasses
        @dataclass(frozen=True)
        class Clause:
            lits: tuple[int, ...]
            origin: str

        @dataclass
        class Encoding:
            repo: Repository
            request: tuple[Dependency, ...]
            var_of_pv: dict[PackageVersion, int]
            pv_of_var: dict[int, PackageVersion]
            selector_of: dict[str, int]
            clauses: list[Clause]

        # %% encode
        def encode(repo: Repository, request: Iterable[Dependency]) -> Encoding:
            enc = Encoding(
                repo=repo, request=tuple(request),
                var_of_pv={}, pv_of_var={}, selector_of={}, clauses=[],
            )
            next_var = 1
            for p in repo.packages:
```

```
            enc.var_of_pv[p.pv] = next_var
            enc.pv_of_var[next_var] = p.pv
            next_var += 1

        def alloc_selector(origin: str) -> int:
            nonlocal next_var
            if origin in enc.selector_of:
                return enc.selector_of[origin]
            sel = next_var
            enc.selector_of[origin] = sel
            next_var += 1
            return sel

        for p in repo.packages:
            pv_lit = enc.var_of_pv[p.pv]
            for dep in p.depends:
                cands = repo.versions_matching(dep)
                origin = f"dep[{p.pv} requires {dep.name}{dep.constraint}]"
                sel = alloc_selector(origin)
                lits = (-sel, -pv_lit, *(enc.var_of_pv[c.pv] for c in cands))
                enc.clauses.append(Clause(lits, origin))
            for cf in p.conflicts:
                for cand in repo.versions_matching(cf):
                    origin = f"conflict[{p.pv} × {cand.pv}]"
                    sel = alloc_selector(origin)
                    lits = (-sel, -pv_lit, -enc.var_of_pv[cand.pv])
                    enc.clauses.append(Clause(lits, origin))

        for name, pkgs in repo.by_name.items():
            for i in range(len(pkgs)):
                for j in range(i + 1, len(pkgs)):
                    vi, vj = pkgs[i].pv, pkgs[j].pv
                    origin = f"atmost[{name}: {vi.version} vs {vj.version}]"
                    sel = alloc_selector(origin)
                    lits = (-sel, -enc.var_of_pv[vi], -enc.var_of_pv[vj])
                    enc.clauses.append(Clause(lits, origin))

        for dep in enc.request:
            cands = repo.versions_matching(dep)
            origin = f"request[{dep.name}{dep.constraint}]"
            sel = alloc_selector(origin)
            lits = (-sel, *(enc.var_of_pv[c.pv] for c in cands))
            enc.clauses.append(Clause(lits, origin))

        return enc

    # %% SolveResult + solve()
    @dataclass
    class SolveResult:
        satisfiable: bool
        installation: frozenset[PackageVersion] | None
        unsat_core: tuple[Clause, ...] | None
        encoding: Encoding

    def solve(repo: Repository, request: Iterable[Dependency]) -> SolveResult:
        from pysat.solvers import Solver as PySATSolver
        enc = encode(repo, request)
        assumptions = list(enc.selector_of.values())
        s = PySATSolver(
            name="glucose3",
            bootstrap_with=[list(c.lits) for c in enc.clauses],
        )
        try:
            if s.solve(assumptions=assumptions):
                model = set(s.get_model() or [])
                installation = frozenset(
                    pv for var, pv in enc.pv_of_var.items() if var in model
                )
                return SolveResult(True, installation, None, enc)
            core = set(s.get_core() or [])
            sel_to_origin = {v: k for k, v in enc.selector_of.items()}
            blamed = {sel_to_origin[sel] for sel in core if sel in sel_to_origin}
            core_clauses = tuple(c for c in enc.clauses if c.origin in blamed)
            return SolveResult(False, None, core_clauses, enc)
        finally:
            s.delete()

    # %% Human-readable explanation
    def explain(result: SolveResult) -> str:
        if result.satisfiable:
            inst = result.installation or frozenset()
            lines = [f"SAT — installation ({len(inst)} packages):"]
            for pv in sorted(inst):
                lines.append(f"  + {pv}")
            return "\n".join(lines)
        core = result.unsat_core or ()
        if not core:
            return "UNSAT (no core extracted)"
        by_kind: dict[str, list[Clause]] = {}
        for c in core:
            kind = c.origin.split("[", 1)[0]
            by_kind.setdefault(kind, []).append(c)
        lines = [f"UNSAT — blamed constraints ({len(core)}):"]
        for kind in ("request", "dep", "conflict", "atmost"):
            for c in by_kind.get(kind, ()):
                lines.append(f"  - {c.origin}")
        return "\n".join(lines)

    # %% Tests — Module 3
    def _test_module_3() -> None:
        # 1. trivially SAT
        p = Package("foo", Version.parse("1.0.0"))
        r = solve(Repository((p,)), [Dependency("foo")])
        assert r.satisfiable and r.installation == frozenset({p.pv})

        # 2. SAT chain a -> b -> c
        a = Package("a", Version.parse("1.0.0"),
                    depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
        b = Package("b", Version.parse("1.0.0"),
                    depends=(Dependency("c", VersionConstraint.parse("==1.0.0")),))
        c = Package("c", Version.parse("1.0.0"))
        r = solve(Repository((a, b, c)), [Dependency("a")])
        assert r.satisfiable and r.installation == frozenset({a.pv, b.pv, c.pv})

        # 3. direct conflict
        x = Package("x", Version.parse("1.0.0"),
                    conflicts=(Dependency("y", VersionConstraint.parse("==1.0.0")),))
        y = Package("y", Version.parse("1.0.0"))
        r = solve(Repository((x, y)), [Dependency("x"), Dependency("y")])
        assert not r.satisfiable
        assert any("conflict" in cc.origin for cc in r.unsat_core or ())

        # 4. transitive conflict: a -> b, b × c, request a and c
        a2 = Package("a", Version.parse("1.0.0"),
                     depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
        b2 = Package("b", Version.parse("1.0.0"),
                     conflicts=(Dependency("c", VersionConstraint.parse("==1.0.0")),))
        c2 = Package("c", Version.parse("1.0.0"))
        r = solve(Repository((a2, b2, c2)), [Dependency("a"), Dependency("c")])
        assert not r.satisfiable
        origins = " ".join(cc.origin for cc in r.unsat_core or ())
        assert "dep" in origins and "conflict" in origins

        # 5. version-constrained dep picks a matching version
        a3 = Package("a", Version.parse("1.0.0"),
                     depends=(Dependency("b", VersionConstraint.parse(">=2.0")),))
        b_low = Package("b", Version.parse("1.0.0"))
        b_mid = Package("b", Version.parse("2.0.0"))
        b_hi = Package("b", Version.parse("2.5.0"))
        r = solve(Repository((a3, b_low, b_mid, b_hi)), [Dependency("a")])
        assert r.satisfiable
        b_picked = [pv.version for pv in r.installation if pv.name == "b"]
```

```
                assert len(b_picked) == 1 and b_picked[0] >= Version.parse("2.0")

                # 6. dual-version forced via two conflicting deps
                a4 = Package("a", Version.parse("1.0.0"), depends=(
                    Dependency("b", VersionConstraint.parse("==1.0.0")),
                    Dependency("b", VersionConstraint.parse("==2.0.0")),
                ))
                b41 = Package("b", Version.parse("1.0.0"))
                b42 = Package("b", Version.parse("2.0.0"))
                r = solve(Repository((a4, b41, b42)), [Dependency("a")])
                assert not r.satisfiable
                assert any("atmost" in cc.origin for cc in r.unsat_core or ())

                print("Module 3 tests passed")

            if not _IS_CLI: _test_module_3()

            # %% Demo — solve and print explanations
            def _demo_module_3() -> None:
                a = Package("a", Version.parse("1.0.0"),
                            depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
                b = Package("b", Version.parse("1.0.0"),
                            conflicts=(Dependency("c", VersionConstraint.parse("==1.0.0")),))
                c = Package("c", Version.parse("1.0.0"))
                repo = Repository((a, b, c))
                print("--- request: a only ---")
                print(explain(solve(repo, [Dependency("a")])))
                print("--- request: a and c ---")
                print(explain(solve(repo, [Dependency("a"), Dependency("c")])))

            if not _IS_CLI: _demo_module_3()

            # %% [markdown]
            # ## Module 4 — SMT encoding via Z3
            #
            # Instead of one boolean per (name, version), each package name gets a pair
            # `(installed: Bool, version: Int)`. The integer encodes the version
            # tuple lexicographically. Dependency constraints become integer range
            # predicates (`>=`, `<=`), avoiding the quadratic at-most-one blowup of
            # Module 3. The same encoding feeds `z3.Optimize` for two natural
            # objectives: fewest packages installed, and newest versions chosen.

            # %% Z3 import + version encoding helpers
            import z3

            def _version_to_int(v: Version) -> int:
                a, b, c = (v.parts + (0, 0, 0))[:3]
                return a * 1_000_000 + b * 1_000 + c

            def _constraint_to_z3(c: VersionConstraint, version_var):
                conds = []
                for op, t in c.atoms:
                    ti = _version_to_int(t)
                    if op == "==": conds.append(version_var == ti)
                    elif op == ">=": conds.append(version_var >= ti)
                    elif op == "<=": conds.append(version_var <= ti)
                    elif op == ">": conds.append(version_var > ti)
                    elif op == "<": conds.append(version_var < ti)
                return z3.And(*conds) if conds else z3.BoolVal(True)

            # %% SMT encoding
            @dataclass
            class SMTEncoding:
                repo: Repository
                installed: dict[str, "z3.BoolRef"]
                version: dict[str, "z3.ArithRef"]
                solver: "z3.Solver | z3.Optimize"

            def encode_smt(
                repo: Repository,
                request: Iterable[Dependency],
                *,
                solver=None,
            ) -> SMTEncoding:
                s = solver if solver is not None else z3.Solver()
                installed: dict[str, z3.BoolRef] = {}
                version: dict[str, z3.ArithRef] = {}

                for name, pkgs in repo.by_name.items():
                    installed[name] = z3.Bool(f"installed_{name}")
                    version[name] = z3.Int(f"version_{name}")
                    available = [_version_to_int(p.version) for p in pkgs]
                    s.add(z3.Implies(
                        installed[name],
                        z3.Or([version[name] == vi for vi in available]),
                    ))

                for p in repo.packages:
                    cond = z3.And(installed[p.name], version[p.name] == _version_to_int(p.version))
                    for dep in p.depends:
                        if dep.name not in installed:
                            s.add(z3.Not(cond))
                        else:
                            s.add(z3.Implies(
                                cond,
                                z3.And(installed[dep.name],
                                       _constraint_to_z3(dep.constraint, version[dep.name])),
                            ))
                for cf in p.conflicts:
                    if cf.name in installed:
                        s.add(z3.Implies(
                            cond,
                            z3.Not(z3.And(installed[cf.name],
                                          _constraint_to_z3(cf.constraint, version[cf.name]))),
                        ))

                for dep in request:
                    if dep.name not in installed:
                        s.add(z3.BoolVal(False))
                    else:
                        s.add(z3.And(installed[dep.name],
                                     _constraint_to_z3(dep.constraint, version[dep.name])))

                return SMTEncoding(repo=repo, installed=installed, version=version, solver=s)

            # %% SMT solve + optimize
            @dataclass
            class SMTResult:
                satisfiable: bool
                installation: frozenset[PackageVersion] | None
                objective: int | None = None

            def _extract_installation(enc: SMTEncoding, model) -> frozenset[PackageVersion]:
                out: set[PackageVersion] = set()
                for name, b in enc.installed.items():
                    if z3.is_true(model.eval(b, model_completion=True)):
                        v_int = model.eval(enc.version[name], model_completion=True).as_long()
                        for pkg in enc.repo.by_name[name]:
                            if _version_to_int(pkg.version) == v_int:
                                out.add(pkg.pv)
                                break
                return frozenset(out)

            def solve_smt(repo: Repository, request: Iterable[Dependency]) -> SMTResult:
                enc = encode_smt(repo, request)
                if enc.solver.check() == z3.sat:
                    return SMTResult(True, _extract_installation(enc, enc.solver.model()))
                return SMTResult(False, None)

            def solve_smt_minimize_packages(
                repo: Repository, request: Iterable[Dependency],
            ) -> SMTResult:
                opt = z3.Optimize()
                enc = encode_smt(repo, request, solver=opt)
                total = z3.Sum([z3.If(b, 1, 0) for b in enc.installed.values()])
                opt.minimize(total)
                if opt.check() == z3.sat:
                    inst = _extract_installation(enc, opt.model())
                    return SMTResult(True, inst, len(inst))
                return SMTResult(False, None)
```

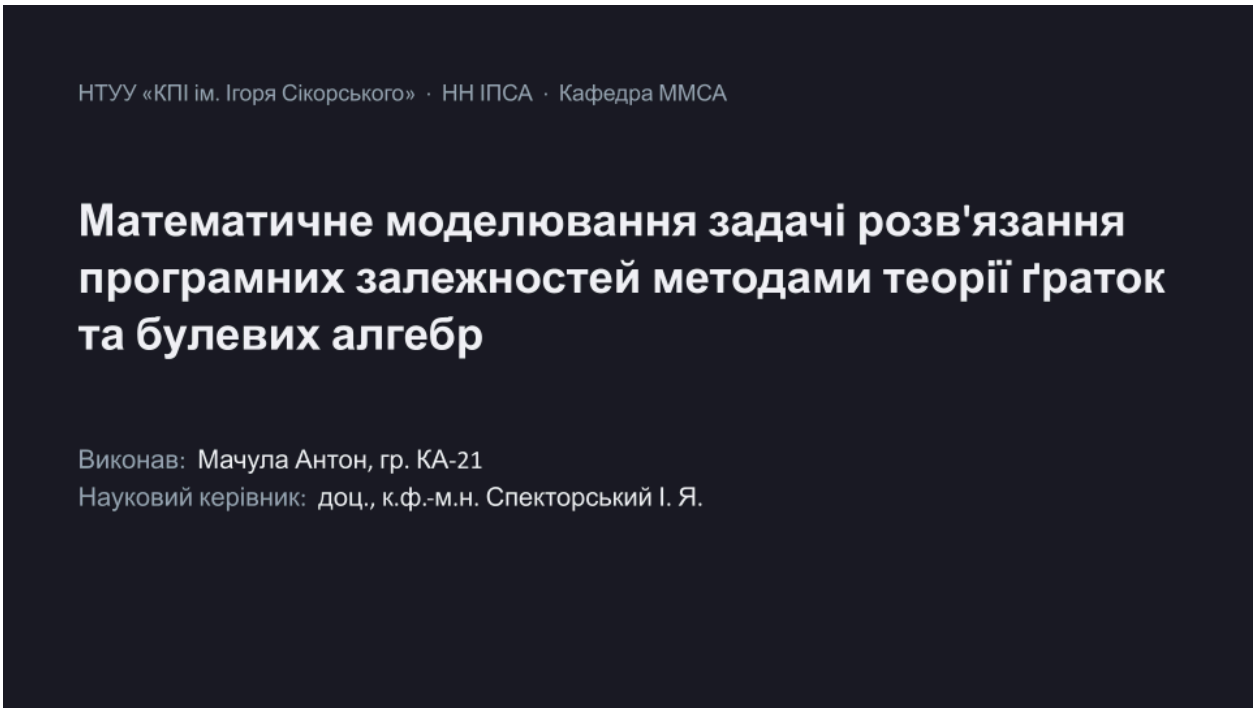
```
def solve_smt_maximize_versions(
    repo: Repository, request: Iterable[Dependency],
) -> SMTResult:
    opt = z3.Optimize()
    enc = encode_smt(repo, request, solver=opt)
    total = z3.Sum([
        z3.If(b, v, 0)
        for b, v in zip(enc.installed.values(), enc.version.values())
    ])
    opt.maximize(total)
    if opt.check() == z3.sat:
        m = opt.model()
        inst = _extract_installation(enc, m)
        obj = m.eval(total, model_completion=True).as_long()
        return SMTResult(True, inst, obj)
    return SMTResult(False, None)


# %% Tests — Module 4
def _test_module_4() -> None:
    # 1. SAT/SMT agreement on a uniquely-resolved chain
    a = Package("a", Version.parse("1.0.0"),
                depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
    b = Package("b", Version.parse("1.0.0"))
    repo = Repository((a, b))
    rsat = solve(repo, [Dependency("a")])
    rsmt = solve_smt(repo, [Dependency("a")])
    assert rsat.satisfiable and rsmt.satisfiable
    assert rsat.installation == rsmt.installation == frozenset({a.pv, b.pv})

    # 2. UNSAT agreement on a direct conflict
    x = Package("x", Version.parse("1.0.0"),
                conflicts=(Dependency("y", VersionConstraint.parse("==1.0.0")),))
    y = Package("y", Version.parse("1.0.0"))
    repo = Repository((x, y))
    rsat = solve(repo, [Dependency("x"), Dependency("y")])
    rsmt = solve_smt(repo, [Dependency("x"), Dependency("y")])
    assert (not rsat.satisfiable) and (not rsmt.satisfiable)

    # 3. version constraint resolution
    a2 = Package("a", Version.parse("1.0.0"),
                 depends=(Dependency("b", VersionConstraint.parse(">=2.0")),))
    b_low = Package("b", Version.parse("1.0.0"))
    b_mid = Package("b", Version.parse("2.0.0"))
    b_hi = Package("b", Version.parse("2.5.0"))
    rsmt = solve_smt(Repository((a2, b_low, b_mid, b_hi)), [Dependency("a")])
    assert rsmt.satisfiable
    b_pv = next(pv for pv in rsmt.installation if pv.name == "b")
    assert b_pv.version >= Version.parse("2.0")

    # 4. minimize packages: request only `a`, b stays unselected
    a3 = Package("a", Version.parse("1.0.0"))
    b3 = Package("b", Version.parse("1.0.0"))
    rmin = solve_smt_minimize_packages(Repository((a3, b3)), [Dependency("a")])
    assert rmin.satisfiable
    assert rmin.installation == frozenset({a3.pv})
    assert rmin.objective == 1

    # 5. maximize versions: pick the newest of `a`
    a_lo = Package("a", Version.parse("1.0.0"))
    a_hi = Package("a", Version.parse("2.0.0"))
    rmax = solve_smt_maximize_versions(Repository((a_lo, a_hi)), [Dependency("a")])
    assert rmax.satisfiable
    assert rmax.installation == frozenset({a_hi.pv})

    print("Module 4 tests passed")


if not _IS_CLI: _test_module_4()

# %% Demo — SMT optimization output
def _demo_module_4() -> None:
    flask_lo = Package("flask", Version.parse("2.0.0"),
                       depends=(Dependency("werkzeug", VersionConstraint.parse(">=2.0")),))
    flask_hi = Package("flask", Version.parse("2.1.0"),
                       depends=(Dependency("werkzeug", VersionConstraint.parse(">=2.0")),))
    wz_old = Package("werkzeug", Version.parse("2.0.0"))
    wz_new = Package("werkzeug", Version.parse("2.1.0"))
    repo = Repository((flask_lo, flask_hi, wz_old, wz_new))
    print("--- maximize-versions request: flask ---")
    r = solve_smt_maximize_versions(repo, [Dependency("flask")])
    for pv in sorted(r.installation or ()):
        print(f"  + {pv}")
    print(f"  objective = {r.objective}")
    print("--- minimize-packages request: flask ---")
    r = solve_smt_minimize_packages(repo, [Dependency("flask")])
    for pv in sorted(r.installation or ()):
        print(f"  + {pv}")
    print(f"  objective = {r.objective}")

if not _IS_CLI: _demo_module_4()

# %% [markdown]
# ## Module 5 — experiments, benchmarks, lattice-ness measurement
#
# Sweep generated repos across size and conflict density, record encoding
# and solving metrics in a DataFrame, and — on instances small enough to
# enumerate the valid-installation set — directly measure whether that set
# is closed under meet (intersection) and join (union). Lattice-ness is
# the empirical version of the thesis claim "lattice structure predicts
# tractability."

# %% Lattice-ness on a single repo
def _atmost_one_ok(I: frozenset[PackageVersion]) -> bool:
    seen: set[str] = set()
    for pv in I:
        if pv.name in seen:
            return False
        seen.add(pv.name)
    return True


def _conflicts_ok(I: frozenset[PackageVersion], repo: Repository) -> bool:
    for pv in I:
        pkg = repo.by_pv.get(pv)
        if pkg is None:
            return False
        for cf in pkg.conflicts:
            if any(cand.pv in I for cand in repo.versions_matching(cf)):
                return False
    return True


def all_valid_installations(
    repo: Repository, limit: int = 18,
) -> list[frozenset[PackageVersion]]:
    """All order ideals that also satisfy at-most-one and conflict
    constraints. 2^n enumeration — small repos only."""
    return [
        I for I in enumerate_order_ideals(repo, limit=limit)
        if _atmost_one_ok(I) and _conflicts_ok(I, repo)
    ]


def measure_lattice_ness(
    repo: Repository, limit: int = 18,
) -> dict[str, float | int | bool | None]:
    """For each ordered pair (I, J) of valid installations, check whether
    I ∩ J and I ∪ J are themselves valid installations. The pass rate is
    a continuous degree-of-lattice-ness; 1.0 means closure under set meet
    and join, i.e. V(repo) is a sublattice of 2^P."""
```

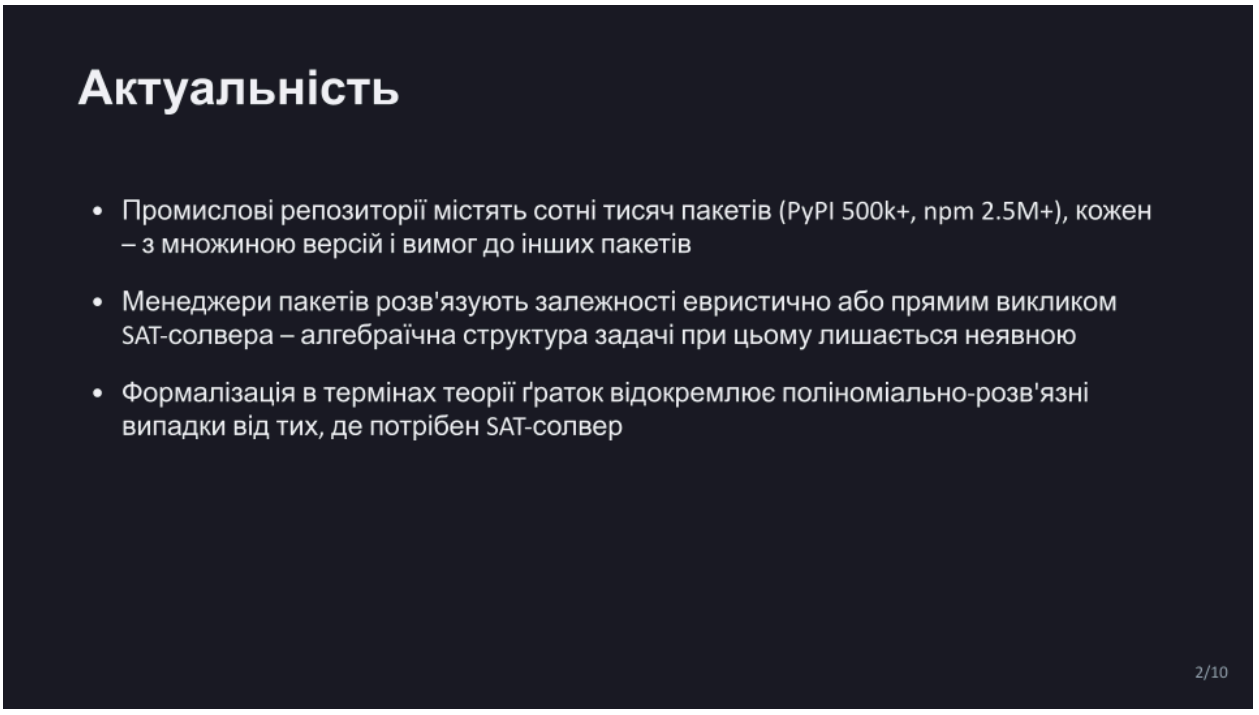
```
                V = all_valid_installations(repo, limit=limit)
                n = len(V)
                if n == 0:
                    return {"n_valid": 0, "meet_rate": None, "join_rate": None, "is_lattice": False}
                Vset = set(V)
                meet_ok = sum(1 for I in V for J in V if (I & J) in Vset)
                join_ok = sum(1 for I in V for J in V if (I | J) in Vset)
                total = n * n
                return {
                  "n_valid": n,
                  "meet_rate": meet_ok / total,
                  "join_rate": join_ok / total,
                   "is_lattice": meet_ok == total and join_ok == total,
                }

            # %% Single-repo benchmark row
            @dataclass
            class BenchRow:
                num_packages: int
                versions_per_package: int
                avg_deps: float
                conflict_prob: float
                seed: int
                n_vars: int
                n_clauses: int
                encode_time: float
                solve_time: float
                satisfiable: bool
                installation_size: int | None
                unsat_core_size: int | None
                n_valid_installations: int | None
                meet_rate: float | None
                join_rate: float | None

            def benchmark_one(
                params: GeneratorParams,
                request_names: list[str],
                *,
                measure_lattice: bool = False,
                lattice_limit: int = 18,
            ) -> BenchRow:
                import time
                from dataclasses import asdict as _asdict  # noqa: F401  (used below by caller)
                repo = generate_repository(params)
                request = [Dependency(name) for name in request_names]
                t0 = time.perf_counter()
                enc = encode(repo, request)
                t_enc = time.perf_counter() - t0
                n_vars = len(enc.var_of_pv) + len(enc.selector_of)
                n_clauses = len(enc.clauses)
                t0 = time.perf_counter()
                r = solve(repo, request)
                t_solve = time.perf_counter() - t0
                lat: dict[str, float | int | bool | None] = {
                    "n_valid": None, "meet_rate": None, "join_rate": None,
                }
                if measure_lattice and len(repo.packages) <= lattice_limit:
                    lat = measure_lattice_ness(repo, limit=lattice_limit)
                return BenchRow(
                    num_packages=params.num_packages,
                    versions_per_package=params.versions_per_package,
                    avg_deps=params.avg_deps_per_package,
                    conflict_prob=params.conflict_probability,
                    seed=params.seed,
                    n_vars=n_vars,
                    n_clauses=n_clauses,
                    encode_time=t_enc,
                    solve_time=t_solve,
                    satisfiable=r.satisfiable,
                    installation_size=len(r.installation) if r.installation else None,
                    unsat_core_size=len(r.unsat_core) if r.unsat_core else None,
                    n_valid_installations=lat["n_valid"],  # type: ignore[arg-type]
                    meet_rate=lat["meet_rate"],  # type: ignore[arg-type]
                    join_rate=lat["join_rate"],  # type: ignore[arg-type]
                )

            # %% Sweeps
            def run_size_sweep(
                sizes: tuple[int, ...] = (10, 50, 200, 500),
                conflict_probs: tuple[float, ...] = (0.0, 0.02, 0.05, 0.1, 0.2),
                seeds: tuple[int, ...] = (0, 1, 2),
                versions_per_package: int = 2,
                avg_deps: float = 2.0,
            ):
                import pandas as pd
                from dataclasses import asdict
                rows = []
                for n in sizes:
                    for cp in conflict_probs:
                        for s in seeds:
                            params = GeneratorParams(
                                num_packages=n,
                                versions_per_package=versions_per_package,
                                avg_deps_per_package=avg_deps,
                                conflict_probability=cp,
                                seed=s,
                            )
                            rows.append(asdict(benchmark_one(params, [f"pkg{n-1:03d}"])))
                return pd.DataFrame(rows)

            def run_lattice_sweep(
                sizes: tuple[int, ...] = (4, 6, 8, 10),
                conflict_probs: tuple[float, ...] = (0.0, 0.05, 0.1, 0.2, 0.3, 0.5),
                seeds: tuple[int, ...] = (0, 1, 2, 3, 4),
            ):
                """Single-version repos so total pvs = num_packages; this isolates
                conflict effects on the order-ideal lattice from version ambiguity."""
                import pandas as pd
                from dataclasses import asdict
                rows = []
                for n in sizes:
                    for cp in conflict_probs:
                        for s in seeds:
                            params = GeneratorParams(
                                num_packages=n,
                                versions_per_package=1,
                                avg_deps_per_package=1.5,
                                conflict_probability=cp,
                                seed=s,
                            )
                            rows.append(asdict(benchmark_one(
                                params, [f"pkg{n-1:03d}"],
                                measure_lattice=True, lattice_limit=18,
                            )))
                return pd.DataFrame(rows)

            # %% Plotting
            def plot_sweep(df_size, df_lat, save_prefix: str = "bench") -> list[str]:
                import matplotlib.pyplot as plt
                saved: list[str] = []

                # 1. solve time vs repo size, one line per conflict probability
                fig, ax = plt.subplots(figsize=(8, 5))
                for cp, group in df_size.groupby("conflict_prob"):
                    agg = group.groupby("num_packages")["solve_time"].mean()
                    ax.plot(agg.index, agg.values, marker="o", label=f"conflict={cp}")
                ax.set_xlabel("repository size (#packages)")
                ax.set_ylabel("mean solve time (s)")
                ax.set_xscale("log")
                ax.set_yscale("log")
                ax.set_title("SAT solving time vs repository size")
                ax.legend()
                path = f"{save_prefix}_solve_vs_size.png"
                fig.savefig(path, dpi=120, bbox_inches="tight"); plt.close(fig); saved.append(path)

                # 2. installation size distribution
                fig, ax = plt.subplots(figsize=(8, 5))
                sat = df_size[df_size["satisfiable"]]
                sizes_ = sat["installation_size"].dropna().astype(int)
                if len(sizes_):
                    ax.hist(sizes_, bins=min(20, max(5, int(sizes_.max() - sizes_.min() + 1))))
                ax.set_xlabel("installation size")
                ax.set_ylabel("count")
                ax.set_title("Installation size distribution (SAT runs)")
                path = f"{save_prefix}_install_size_dist.png"
                fig.savefig(path, dpi=120, bbox_inches="tight"); plt.close(fig); saved.append(path)

                # 3. lattice-ness vs conflict density — the empirical thesis claim
```

```
    fig, ax = plt.subplots(figsize=(8, 5))
    by_cp = df_lat.groupby("conflict_prob")[["meet_rate", "join_rate"]].mean()
    ax.plot(by_cp.index, by_cp["meet_rate"], marker="o", label="meet (∩) closure rate")
    ax.plot(by_cp.index, by_cp["join_rate"], marker="s", label="join (∪) closure rate")
    ax.set_xlabel("conflict probability")
    ax.set_ylabel("fraction of (I, J) pairs whose meet/join is also valid")
    ax.set_title("Lattice-ness of V(repo) vs conflict density")
    ax.set_ylim(-0.05, 1.05)
    ax.legend()
    path = f"{save_prefix}_lattice_ness.png"
    fig.savefig(path, dpi=120, bbox_inches="tight"); plt.close(fig); saved.append(path)

    # 4. UNSAT rate vs conflict density
    fig, ax = plt.subplots(figsize=(8, 5))
    for n, group in df_size.groupby("num_packages"):
        agg = group.groupby("conflict_prob")["satisfiable"].mean()
        ax.plot(agg.index, 1 - agg.values, marker="o", label=f"n={n}")
    ax.set_xlabel("conflict probability")
    ax.set_ylabel("UNSAT rate")
    ax.set_title("Probability of UNSAT vs conflict density")
    ax.legend()
    path = f"{save_prefix}_unsat_rate.png"
    fig.savefig(path, dpi=120, bbox_inches="tight"); plt.close(fig); saved.append(path)

    return saved


# %% PyPI scraper (optional — call manually; not run by default)
def scrape_pypi(names: list[str], timeout: float = 10.0) -> Repository:
    """Fetch each name's latest metadata from PyPI's JSON API; restrict
    dependency edges to those resolving within `names` so the resulting
    repo is self-contained. Network operation."""
    import urllib.request
    import urllib.error
    name_set = set(names)
    packages: list[Package] = []
    for name in names:
        try:
            with urllib.request.urlopen(
                f"https://pypi.org/pypi/{name}/json", timeout=timeout
            ) as resp:
                data = json.load(resp)
        except (urllib.error.URLError, json.JSONDecodeError, TimeoutError):
            continue
        info = data.get("info", {})
        try:
            version = Version.parse(info.get("version", "0.0.0").split("+")[0])
        except ValueError:
            continue
        depends: list[Dependency] = []
        for req in info.get("requires_dist") or ():
            req_main = req.split(";")[0].strip()
            tokens = req_main.split(" ", 1)
            dep_name = tokens[0].strip().lower()
            if dep_name not in name_set:
                continue
            spec = tokens[1].strip("()").strip() if len(tokens) > 1 else ""
            try:
                vc = VersionConstraint.parse(spec)
            except Exception:
                vc = VersionConstraint(())
            depends.append(Dependency(dep_name, vc))
        packages.append(Package(name, version, tuple(depends), ()))
    return Repository(tuple(packages))


# %% Tests — Module 5
def _test_module_5() -> None:
    # benchmark_one on a small repo produces a sensible row
    params = GeneratorParams(
        num_packages=10, versions_per_package=2,
        avg_deps_per_package=1.5, conflict_probability=0.1, seed=0,
    )
    row = benchmark_one(params, ["pkg009"])
    assert row.n_vars > 0
    assert row.n_clauses > 0
    assert row.encode_time >= 0
    assert row.solve_time >= 0

    # conflict-free single-version repo: V is exactly the order-ideal lattice
    # of the dep poset, so meet and join closure rates are both 1.0
    params_cf = GeneratorParams(
        num_packages=6, versions_per_package=1,
        avg_deps_per_package=1.0, conflict_probability=0.0, seed=0,
    )
    row = benchmark_one(params_cf, ["pkg005"], measure_lattice=True, lattice_limit=18)
    assert row.meet_rate == 1.0
    assert row.join_rate == 1.0

    # hand-crafted breaking example: {a, b} and {a, c} are both valid, but
    # their union {a, b, c} is not, because b conflicts with c. So
    # join_rate must be < 1.0. (meet of any two valid installations is
    # always valid in the single-version case, so meet_rate stays 1.0.)
    a_ = Package("a", Version.parse("1.0.0"))
    b_ = Package("b", Version.parse("1.0.0"))
    c_ = Package("c", Version.parse("1.0.0"),
                 conflicts=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
    m = measure_lattice_ness(Repository((a_, b_, c_)), limit=18)
    assert m["meet_rate"] == 1.0
    assert m["join_rate"] is not None and m["join_rate"] < 1.0  # type: ignore[operator]
    assert m["is_lattice"] is False

    print("Module 5 tests passed")

if not _IS_CLI: _test_module_5()

# %% Demo — run the sweeps, print headline stats, save plots
def _demo_module_5() -> None:
    print("Running size sweep...")
    df_size = run_size_sweep(
        sizes=(10, 50, 200),
        conflict_probs=(0.0, 0.05, 0.1, 0.2),
        seeds=(0, 1, 2),
    )
    print(f"  {len(df_size)} runs; mean solve time = {df_size['solve_time'].mean():.4f}s")

    print("Running lattice-ness sweep...")
    df_lat = run_lattice_sweep(
        sizes=(4, 6, 8),
        conflict_probs=(0.0, 0.05, 0.1, 0.2, 0.3, 0.5),
        seeds=(0, 1, 2, 3, 4),
    )
    print(f"  {len(df_lat)} runs")
    by_cp = df_lat.groupby("conflict_prob")[["meet_rate", "join_rate"]].mean()
    print(by_cp.round(3).to_string())

    saved = plot_sweep(df_size, df_lat, save_prefix="bench")
    print("Wrote:")
    for p in saved:
        print(f"  - {p}")

if not _IS_CLI: _demo_module_5()

# %% [markdown]
# ## Breaking-example fixture
#
# Hand-crafted 5-package repo whose conflict-free Hasse diagram is a
# recognizable distributive lattice. Adding one conflict (c × d) excludes
# two order ideals that share the now-forbidden pair. {a,b,c} and {a,b,d}
# are both maximal in V but have no common upper bound — V is no longer a
# lattice. Two figures side by side make the breakdown visible.

# %% Breaking example: clean vs conflicted repos + side-by-side Hasse render
def _demo_breaking_example() -> None:
```

```
    import matplotlib.pyplot as plt

    one = Version.parse("1.0.0")
    a = Package("a", one)
    b = Package("b", one)
    c_clean = Package("c", one, depends=(Dependency("a", VersionConstraint.parse("==1.0.0")),))
    d_clean = Package("d", one, depends=(Dependency("b", VersionConstraint.parse("==1.0.0")),))
    e = Package("e", one, depends=(
                Dependency("c", VersionConstraint.parse("==1.0.0")),
                Dependency("d", VersionConstraint.parse("==1.0.0")),
    ))
    clean_repo = Repository((a, b, c_clean, d_clean, e))

    c_conf = Package("c", one,

depends=(Dependency("a", VersionConstraint.parse("==1.0.0")),),

conflicts=(Dependency("d", VersionConstraint.parse("==1.0.0")),))
    broken_repo = Repository((a, b, c_conf, d_clean, e))

    fig, axes = plt.subplots(1, 2, figsize=(16, 6))
    render_hasse_lattice(
        clean_repo,
        title="L(P): distributive lattice (no conflicts)",
        pv_label=lambda pv: pv.name,
        ax=axes[0],
    )
    render_hasse_lattice(
        broken_repo,
        title="V(P): same poset + (c × d), no longer a lattice",
        pv_label=lambda pv: pv.name,
        filter_fn=lambda I: _conflicts_ok(I, broken_repo) and _atmost_one_ok(I),
        ax=axes[1],
    )
    fig.savefig("breaking_example.png", dpi=120, bbox_inches="tight")
    plt.close(fig)

    save_repository(clean_repo, "example_clean.json")
    save_repository(broken_repo, "example_broken.json")

    m_clean = measure_lattice_ness(clean_repo, limit=18)
    m_broken = measure_lattice_ness(broken_repo, limit=18)
    print(f"clean: n_valid={m_clean['n_valid']:>2} meet={m_clean['meet_rate']} join={m_clean['join_rate']} lattice={m_clean['is_lattice']}")
    print(f"broken: n_valid={m_broken['n_valid']:>2} meet={m_broken['meet_rate']} join={m_broken['join_rate']} lattice={m_broken['is_lattice']}")
    print("Wrote breaking_example.png, example_clean.json, example_broken.json")
    print("Try: uv run resolver.py --repo example_broken.json --install e --explain")

if not _IS_CLI:
    _demo_breaking_example()

# %% [markdown]
# ## Module 6 — command-line interface
#
# `uv run resolver.py --repo X.json --install A B` runs SAT solve and prints
# the installation (or the UNSAT explanation, since the default behavior is
# "explain"). `--minimize-packages` / `--maximize-versions` switch to the SMT
# optimizer. Detection of `--repo` in argv at module import time suppresses
# the inline tests and demos above; the CLI handler is the only thing that
# runs in that mode.

# %% CLI handler
def _main_cli() -> int:
    import argparse
    p = argparse.ArgumentParser(
        prog="resolver",
        description="Lattice/Boolean dependency resolver",
    )
    p.add_argument("--repo", required=True, help="path to repository JSON")
    p.add_argument("--install", nargs="+", required=True,
                   help="package names (or 'name>=2.0' style spec) to install")
    grp = p.add_mutually_exclusive_group()
    grp.add_argument("--minimize-packages", action="store_true",
                     help="SMT objective: minimize total installed packages")
    grp.add_argument("--maximize-versions", action="store_true",
                     help="SMT objective: prefer newer versions")
    p.add_argument("--explain", action="store_true",
                   help="on UNSAT, print human-readable diagnosis (always on for default solver)")
    args = p.parse_args()

    def _parse_install(spec: str) -> Dependency:
        for op in VersionConstraint._OPS:
            if op in spec:
                name, sep, rest = spec.partition(op)
                return Dependency(name.strip(), VersionConstraint.parse(op + rest))
        return Dependency(spec)

    repo = load_repository(args.repo)
    request = [_parse_install(s) for s in args.install]

    if args.minimize_packages or args.maximize_versions:
        r = (solve_smt_minimize_packages if args.minimize_packages
             else solve_smt_maximize_versions)(repo, request)
        if not r.satisfiable:
            print("UNSAT (no UNSAT-core extraction in SMT objective mode)")
            return 1
        label = "minimize-packages" if args.minimize_packages else "maximize-versions"
        print(f"SAT ({label}, objective={r.objective}):")
        for pv in sorted(r.installation or ()):
            print(f"  {pv}")
        return 0

    result = solve(repo, request)
    print(explain(result))
    return 0 if result.satisfiable else 1

if _IS_CLI:
    sys.exit(_main_cli())
```

ДОДАТОК Б. ІЛЮСТРАТИВНІ МАТЕРІАЛИ ДОПОВІДІ





Об'єкт, предмет, мета

Об'єкт дослідження
задача розв'язання програмних залежностей у пакетних репозиторіях

Предмет дослідження
алгебраїчні структури (ЧВМ, ґратки ідеалів порядку, булеві алгебри) при формалізації інсталяційної сумісності та методи їх реалізації засобами SAT і SMT

Мета роботи
побудувати математичну модель засобами теорії ґраток і булевих алгебр, дослідити умови збереження та руйнування ґраткової структури, реалізувати прототип і провести обчислювальні експерименти

3/10

Кроки дослідження

- Опрацювати джерела з теорії ґраток, булевих алгебр та задачі SAT
- Формалізувати репозиторій, залежність, конфлікт і коректну інсталяцію
- Побудувати частковий порядок та дослідити ґратку ідеалів порядку
- Довести дистрибутивність ґратки у безконфліктному випадку
- Показати руйнування структури при введенні обмежень несумісності
- Розробити схеми кодування задачі у вигляді SAT- та SMT-формул
- Провести обчислювальні експерименти на синтетичних репозиторіях

4/10

Розділ 1. Теоретичний апарат

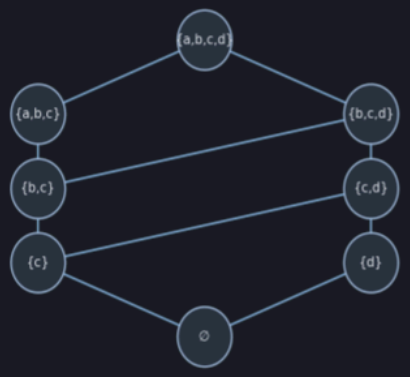
ЧВМ → ґратка → дистрибутивна ґратка → ґратка ідеалів порядку → булева алгебра → SAT / SMT

- SAT – обчислення на двоелементній булевій алгебрі; SMT – її розширення теоріями (цілі числа, арифметика)
- Кожен наступний клас структур наслідує властивості попереднього й додає потрібні нові

5/10

Розділ 2. Математична модель

- Репозиторій R = (P, V, D, C): пакети, версії, залежності, несумісності
- Коректна інсталяція = ідеал порядку посету (P, ≤)
- Безконфліктний випадок ⟶ множина інсталяцій є дистрибутивною ґраткою ідеалів
- Приклад: ланцюг залежностей a>b>c та ізольований пакет d. Вузли — коректні інсталяції (ідеали порядку), ребро — додавання одного пакета; ∅ знизу, {a,b,c,d} зверху.

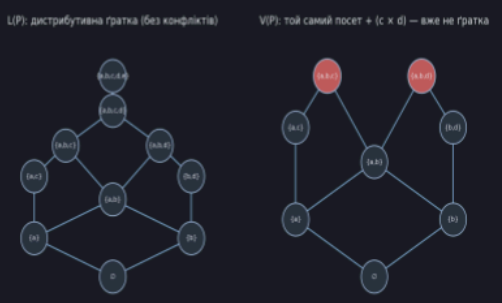


L(P): ідеали порядку (інсталяції); ребро – додавання одного елемента

6/10

Ключовий результат: руйнування ґратки

- Введення несумісностей C → простір коректних інсталяцій V(R) ⊊ L(P)
- Зберігає замкненість за перетином → напівґратка за ∩
- Втрачає замкненість за об'єднанням: ∪ може активувати конфлікт → супремум може не існувати
- Руйнування ґратки відповідає переходу обчислювальної складності P → NP
- Приклад: пакети {a,b,c,d,e}, де c→a, d→b, e→c,d, з конфліктом c×d. Ідеали з обома c і d зникають → {a,b,c} і {a,b,d} стають незрівнянними максимумами



ліворуч ґратка; праворуч два несумірні максимуми

7/10

Розділ 3. Програмна реалізація

Прототип resolver.py — Python, ~1400 рядків коду

SAT-кодування

- PySAT, солвер Glucose-3
- булева змінна на кожну пару «пакет–версія»
- селекторні змінні → читабельне UNSAT-ядро (пояснення конфлікту)

SMT-кодування

- Z3, версії як цілочисельні змінні
- розмір кодування O(пакетів) замість O(пакетів × версій²)
- оптимізація через z3.Optimize (мінімум пакетів / новіші версії)

8/10

Звідки беруться дані

- **Повністю синтетичні репозиторії — свідомий методологічний вибір**
- Параметризований генератор: розмір репозиторію, щільність залежностей, ймовірність конфлікту, кількість версій на пакет
- Детермінований seed → повна відтворюваність експериментів
- Залежності лише на пакети меншого рангу → ациклічність графа за побудовою
- Контрольована генерація дозволяє ізолювати вплив кожного параметра — що неможливо на реальних датасетах

9/10

Висновки

- Побудовано математичну модель; коректні інсталяції у безконфліктному випадку – дистрибутивна ґратка ідеалів порядку
- Доведено та обґрунтовано руйнування ґраткової структури при введенні конфліктів (перехід P → NP)
- Реалізовано прототип з SAT- і SMT-кодуваннями та механізмом видобування UNSAT-ядра
- Напрями розвитку: графи залежностей з циклами (нерухомі точки), апроксимаційні алгоритми, інтеграція з реальними пакетними менеджерами

10/10