БАКАЛАВРСЬКА РОБОТА

БР. ІП - 17.00.00.000 ПЗ

Група ІП-21-1

Решітник Богдан

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Решітник Богдан Романович

(прізвище, ім’я, по батькові)

УДК 004

(індекс)

БАКАЛАВРСЬКА РОБОТА

Програмне рішення для побудови мап залежностей процесу керування програмним проектом

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього рівня Решітник Б.Р.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Юрчишин Володимир Миколайович, д.т.н., професор

(підпис, прізвище, ім’я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**

доц. В.В. Бандура

“____” ________________ 2025 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Решітнику Богдану Романовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “ Програмне рішення для побудови мап залежностей процесу керування програмним проектом”

керівник проекту (роботи) Юрчишин Володимир Миколайович, професор

затверджені наказом закладу вищої освіти від “ 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз предметної області процесів керування програмним проектом
2. Алгоритмізація представлення мап залежностей процесу керування програмним проектом
3. Використання теорії графів для задачі планування виконання завдань
4. Програмна імплементація методів побудови мап залежностей процесу керування проектом
5. Експериментальна оцінка ефективності алгоритмів планування

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Дошка завдань в Jira (рис. 1.1)
2. Інтерфейс системи Scoro (рис. 1.2)
3. Вигляд системи Basecamp (рис. 1.3)
4. Roadmap в Jira (рис. 1.4)
5. Приклад мережевої діаграми PERT (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **28 квітня 2025 р.**

Керівник ______________________________
(підпис)

Завдання прийняв до виконання___ ______ ___________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Аналіз предметної області процесів керування програмним проектом	03.05.2025	виконано
2	Алгоритмізація представлення мап залежностей процесу керування програмним проектом	15.05.2025	виконано
3	Використання теорії графів для задачі планування виконання завдань	24.05.2025	виконано
4	Програмна імплементація методів побудови мап залежностей процесу керування проектом	01.06.2025	виконано
5	Експериментальна оцінка ефективності алгоритмів планування	07.06.2025	виконано
6	Оформлення пояснювальної записки дипломної роботи завідувачем кафедри	10.06.2025	виконано

Студент – дипломник _____________________
(підпис)

Керівник роботи _____________________
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 78 сторінок, 33 рисунки, список використаних джерел із 23 найменуваннями.

Мета роботи: розробка та експериментальна оцінка програмного інструменту і алгоритмів для планування виконання завдань з залежностями на основі даних системи управління проєктами Jira

Об'єкт дослідження: процеси планування виконання завдань у програмних проєктах з урахуванням залежностей

Предмет дослідження: методи та алгоритми оптимізації планування на основі графових моделей залежностей та даних систем управління проєктами

Результати дослідження: реалізовано підхід до моделювання залежностей між завданнями за допомогою теорії графів на основі даних, отриманих із системи управління проєктами Jira

В першому розділі проведено аналіз предметної області, виявлено проблеми в управлінні залежностями та оцінено можливості графового підходу

В другому розділі описано алгоритмічні та теоретичні засади побудови мап залежностей на основі теорії графів і методології PERT.

В третьому розділі реалізовано програмну систему побудови та візуалізації мап залежностей і проведено її експериментальну оцінку.

Висновок: розроблено програмний комплекс, який здійснює отримання, обробку та трансформацію даних завдань у графове представлення, виявляє повні ланцюжки залежностей та реалізує алгоритми планування виконання завдань на обмеженій кількості виконавців.

КЛЮЧОВІ СЛОВА: УПРАВЛІННЯ ПРОГРАМНИМИ ПРОЄКТАМИ, ПЛАНУВАННЯ ЗАВДАНЬ, ЗАЛЕЖНОСТІ, ТЕОРІЯ ГРАФІВ, ГРАФОВЕ МОДЕЛЮВАННЯ, АЛГОРИТМИ ПЛАНУВАННЯ, ОПТИМІЗАЦІЯ, JIRA, ПЛАНУВАННЯ.

ANNOTATION

The Bachelor's thesis contains 78 pages, 33 figures, a list of 23 references.

Purpose of the work: Development and experimental evaluation of a software tool and algorithms for planning the execution of tasks with dependencies based on data from the Jira project management system.

Object of research: Processes of planning task execution in software projects, taking dependencies into account.

Subject of research: Methods and algorithms for optimizing planning based on graph dependency models and data from project management systems.

Research results: An approach to modeling dependencies between tasks using graph theory, based on data obtained from the Jira project management system, was implemented.

The first chapter conducted an analysis of the subject domain, identified problems in dependency management, and evaluated the capabilities of the graph approach.

The second chapter described the algorithmic and theoretical foundations for constructing dependency maps based on graph theory and the PERT methodology.

The third chapter implemented a software system for building and visualizing dependency maps and conducted its experimental evaluation.

Conclusion: A software complex has been developed that performs the retrieval, processing, and transformation of task data into a graph representation, identifies complete dependency chains, and implements algorithms for planning task execution with a limited number of performers.

KEYWORDS: SOFTWARE PROJECT MANAGEMENT, TASK PLANNING, DEPENDENCIES, GRAPH THEORY, GRAPH MODELING, SCHEDULING ALGORITHMS, OPTIMIZATION, JIRA, PLANNING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 8
ВСТУП 9

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПРОЦЕСІВ КЕРУВАННЯ ПРОГРАМНИМ ПРОЕКТОМ 12
1.1. Оптимізація планування проектів розробки програмного забезпечення на основі графових моделей залежностей 12
1.2. Аналіз та управління міжкомандними та внутрішньокомандними залежностями у проектах розробки програмного забезпечення 15
1.2.1. Природа та вплив залежностей у проектах розробки ПЗ 17
1.2.2. Інструменти управління проектами та обмеження у роботі із залежностями 18
1.2.2. Моделювання залежностей за допомогою теорії графів 19
1.3. Аналіз функціональності відомих систем управління проектами розробки програмного забезпечення 21
1.3.1. Інтегрована система управління бізнесом для сервісних компаній Scoro 21
1.3.2. Веб-орієнтований інструмент управління проектами та комунікаціями Basecamp 24
1.3.3. Система управління проектами Jira 26

РОЗДІЛ 2. АЛГОРИТМІЗАЦІЯ І ФОРМАЛЬНЕ ПРЕДСТАВЛЕННЯ МАП ЗАЛЕЖНОСТЕЙ ПРОЦЕСУ КЕРУВАННЯ ПРОГРАМНИМ ПРОЕКТОМ . 29
2.1. Імплементаційна архітектура та інтеграція даних 29

2.2. Теоретичні основи планування на основі методології оцінки та аналізу програм (PERT) 32
2.3. Використання теорії графів для задачі планування виконання завдань 36
2.4. Топологічне впорядкування як інструмент управління залежностями 38
2.5. Розв'язання задачі про планування інтервалів методом жадібного вибору за часом завершення 40

РОЗДІЛ 3. ПРОГРАМНА ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ПОБУДОВИ МАП ЗАЛЕЖНОСТЕЙ ПРОЦЕСУ КЕРУВАННЯ ПРОГРАМНИМ ПРОЕКТОМ . 43
3.1. Формування вихідних даних та інфраструктури проекту 43
3.2. Обробка даних 46
3.3. Трансформація даних та побудова графової моделі 47
3.3.1. Виявлення повних ланцюжків залежностей 49
3.4 Алгоритм відображення мап залежностей 51
3.5. Реалізація фронтенд частини та формату виведення 58
3.6. Експериментальна оцінка ефективності алгоритмів планування 59

ВИСНОВКИ 74
ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 76
БІБЛІОГРАФІЧНА ДОВІДКА

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

SDLC – Software Development Life Cycle (Життєвий цикл розробки програмного забезпечення)

API – Application Programming Interface (Програмний інтерфейс застосунків)

WBS – Work Breakdown Structure (Структура декомпозиції робіт)

DAG – Directed Acyclic Graph (Орієнтований ациклічний граф) - згадується концепція "орієнтованому ациклічному графі" у зв'язку з критичним шляхом.

CPM – Critical Path Method (Метод критичного шляху) - згадується "критичний шлях", що є частиною методології CPM.

ВСТУП

Сучасні проєкти розробки програмного забезпечення характеризуються високою складністю, динамічністю та наявністю численних залежностей між завданнями, компонентами системи та учасниками команд. Ефективне управління цими залежностями є критично важливим для своєчасного завершення проєктів, оптимізації використання ресурсів та забезпечення передбачуваності процесів розробки. Неправильне врахування або ігнорування залежностей може призводити до затримок, перевитрат бюджету, нераціонального розподілу навантаження та зниження загальної продуктивності команди.

Існуючі системи управління проєктами (наприклад, Jira) надають базовий функціонал для фіксації завдань, встановлення зв'язків залежностей та оцінок часу. Проте, аналітичні інструменти цих систем часто обмежуються простими візуалізаціями та не пропонують потужних засобів для автоматизованого розрахунку оптимальних або субоптимальних планів виконання завдань з урахуванням складних графових структур залежностей, паралельного виконання та обмеженої кількості виконавців. Особливо це стосується планування в умовах гнучких методологій (спринтів), де швидке та ефективне розподілення завдань є ключовим.

Це створює нагальну потребу у розробці спеціалізованих інструментів та алгоритмів, здатних аналізувати дані з реальних систем управління проєктами, будувати на їх основі формальні моделі залежностей та надавати менеджерам проєктів дієву підтримку у прийнятті рішень щодо планування спринтів, розподілу завдань між виконавцями та оцінки реалістичних термінів виконання.

Актуальність роботи

Актуальність роботи зумовлена необхідністю вдосконалення процесів планування у програмній інженерії шляхом застосування науково

обґрунтованих підходів, зокрема теорії графів та алгоритмів оптимізації, для ефективного управління залежностями та ресурсами на основі реальних даних з інструментів управління проєктами, підвищення прозорості планування та оптимізації використання командних ресурсів.

Ефективне планування є одним з найважливіших факторів успіху проєктів розробки програмного забезпечення. У сучасних умовах, що характеризуються зростанням складності продуктів, розподіленими командами та необхідністю швидкого реагування на зміни, питання оптимізації процесів планування набуває особливої актуальності. Ключовим викликом при плануванні є врахування численних залежностей між окремими завданнями, що визначають порядок їх виконання та впливають на загальну тривалість проєкту.

Аналіз предметної області управління програмними проєктами показує, що існуючі інструменти, хоча й дозволяють фіксувати залежності, часто не надають повноцінних можливостей для їх глибокого аналізу та автоматизованої побудови оптимальних планів виконання завдань з урахуванням обмежених ресурсів. Це вимагає застосування більш досконалих математичних моделей та алгоритмів.

Об'єктом дослідження є процеси планування виконання завдань у програмних проєктах з урахуванням залежностей.

Предметом дослідження є методи та алгоритми оптимізації планування на основі графових моделей залежностей та даних систем управління проєктами.

Метою роботи є розробка та експериментальна оцінка програмного інструменту і алгоритмів для планування виконання завдань з залежностями на основі даних системи управління проєктами Jira.

Завдання дослідження:

1. Проаналізувати існуючі підходи до керування залежностями у програмних проєктах.

2. Сформулювати графову модель залежностей між завданнями.

3. Розробити алгоритми топологічного впорядкування та планування інтервалів.

4. Імплементувати систему обробки даних і візуалізації мап залежностей.

5. Провести експериментальну перевірку ефективності розроблених рішень.

Методи дослідження

- Методи теорії графів (побудова та аналіз графів залежностей);
- Алгоритмічні методи (топологічне сортування, жадібні алгоритми);
- Методологія PERT для аналізу часу виконання завдань;
- Методи програмної інженерії (архітектурне проектування, реалізація front-end та back-end частин);
- Експериментальне моделювання ефективності рішень.

Наукова новизна

Запропоновано підхід до формалізованої побудови карт залежностей у програмних проєктах на основі поєднання графових моделей, алгоритмів топологічного впорядкування та методів планування, що підвищує точність і прозорість процесів управління.

Практичне застосування

Розроблена система може бути використана в командній розробці ПЗ для візуалізації складних залежностей, оптимізації планування, зниження ризиків затримки та поліпшення комунікації між командами.

Бакалаврська робота містить 78 сторінки, 33 рисунки, 3 розділи список використаних джерел із 23 найменуваннями.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПРОЦЕСІВ КЕРУВАННЯ ПРОГРАМНИМ ПРОЕКТОМ

1.1. Оптимізація планування проектів розробки програмного забезпечення на основі графових моделей залежностей

В контексті життєвого циклу розробки програмного забезпечення (SDLC), ефективне управління проектами є критично важливим фактором успіху, проте його значення часто недооцінюється учасниками команди, зокрема розробниками. Процес планування зазвичай передбачає декомпозицію загального обсягу робіт на окремі завдання, які ідентифікуються та оцінюються членами команди розробки. Ця інформація далі обробляється менеджерами проектів з метою структурування, встановлення реалістичних очікувань для зацікавлених сторін (клієнтів) та забезпечення дотримання часових рамок і бюджетних обмежень проекту. Ключовою проблемою, що суттєво впливає на точність планування та ефективність виконання, є наявність складних залежностей між завданнями. Ці залежності формують ланцюги, виконання яких визначає загальну тривалість проекту. Неефективне управління залежностями може призвести до суттєвого перевищення запланованих термінів виконання, а також до субтимальної утилізації ресурсів, зокрема кваліфікації розробників. Незважаючи на широке використання спеціалізованих інструментів управління проектами, їх функціональні можливості щодо візуалізації та ефективного аналізу складних мереж залежностей між завданнями часто обмежені.

Метою даної роботи дослідження стало розробка підходу до автоматизованого аналізу та планування проектів розробки програмного забезпечення на основі даних про залежності між завданнями. Це включало взаємодію з програмним інтерфейсом (API) інструменту управління

проектами для отримання вихідних даних, екстракцію та обробку інформації про залежності, а також генерацію множини можливих часових графіків (таймлайнів) виконання проекту.

Успішне завершення проекту розробки програмного забезпечення вимагає не лише технічної експертизи команди, але й високоякісного управління проектом. Планування, моніторинг та контроль за ходом виконання є фундаментальними аспектами, які впливають на кінцевий результат, відповідність вимогам, своєчасність завершення та дотримання бюджету. Одним із найскладніших елементів планування є правильне врахування взаємозв'язків між окремими завданнями, відомих як залежності. Ігнорування або некоректна оцінка цих залежностей може призвести до значних ризиків та проблем на етапі виконання.

Традиційний процес планування проекту розробки ПЗ часто базується на ітеративному зборі інформації від команди розробки щодо обсягу робіт та оцінок часу. Менеджери проектів агрегують ці дані, формуючи структуру робіт (Work Breakdown Structure - WBS) та визначаючи послідовність виконання завдань. Однак, визначення та явне відображення залежностей "попередник-наступник" між завданнями залишається значним викликом. Складні ланцюги залежностей можуть створювати "вузькі місця", спричиняти простої ресурсів, якщо завдання-попередники затримуються, або, навпаки, генерувати нереалістичні плани, якщо залежності ігноруються. Існуючі інструменти управління проектами, хоча й підтримують введення залежностей, часто не надають розвинених функцій для їх аналізу, візуалізації складних мереж або автоматизованої оптимізації розкладу на їх основі. Це обмежує можливості менеджера проекту ефективно прогнозувати тривалість проекту, ідентифікувати критичний шлях та оптимізувати розподіл ресурсів.

Основною метою даного дослідження є розробка та аналіз алгоритмічних підходів для автоматизованого планування проектів розробки

програмного забезпечення з явним врахуванням складних залежностей між завданнями. Для досягнення цієї мети були поставлені наступні задачі:

- Встановити механізм взаємодії з типовими інструментами управління проектами для екстракції структурованих даних про завдання та їх залежності через програмні інтерфейси (API).
- Розробити формальну модель представлення проекту та його залежностей.
- Розробити або адаптувати алгоритми для побудови множини можливих та оптимальних за певними критеріями часових графіків виконання проекту на основі отриманих даних про залежності.
- Провести експериментальний аналіз ефективності розроблених алгоритмів на наборах даних з різною структурою та складністю залежностей.
- Розробити прототип програмного забезпечення, що імплементує запропоновані підходи.

Для формалізації проблеми планування проекту з урахуванням залежностей було запропоновано модель, засновану на теорії графів. Проект моделюється як орієнтований граф $G=(V,E)$, де множина вершин V представляє окремі завдання проекту, а множина орієнтованих ребер E відображає відношення залежності "попередник-наступник" між завданнями. Наявність ребра $(u,v)\in E$ означає, що завдання v не може розпочатися до завершення завдання u. Таке представлення дозволяє використовувати потужний апарат теорії графів для аналізу та вирішення поставленої задачі.

Було встановлено зв'язок між задачею планування проекту з урахуванням залежностей та класичними задачами теорії графів, такими як пошук найдовшого шляху в орієнтованому ациклічному графі (що відповідає критичному шляху в проекті) або задачами календарного планування. Це відношення послугувало основою для розробки та адаптації алгоритмічних підходів. У роботі розглядаються та аналізуються два альтернативні

алгоритми для побудови можливих часових графіків, які враховують визначені залежності. Ці алгоритми були протестовані в різних сценаріях із різноманітними структурами залежностей (лінійні ланцюги, розгалужені структури, злиття гілок). Експериментальний аналіз результатів дозволив оцінити обчислювальну складність та практичну ефективність алгоритмів, а також їхню здатність обробляти різні типи та складність ланцюгів залежностей.

Очікується, що розроблене програмне забезпечення забезпечить інструментарій для більш точного та ефективного планування проектів розробки програмного забезпечення шляхом автоматизованого аналізу залежностей та генерації оптимізованих розкладів. Впровадження такого інструменту потенційно може призвести до суттєвої економії ресурсів, включаючи час виконання проекту та фінансові витрати, за рахунок оптимізації розподілу завдань, більш точного прогнозування термінів та уникнення простоїв, спричинених неефективним управлінням залежностями. Результати дослідження також можуть слугувати основою для подальшої розробки більш складних алгоритмів, що враховують обмеження на ресурси, пріоритети завдань та інші фактори, що впливають на планування проектів.

1.2. Аналіз та управління міжкомандними та внутрішньокомандними залежностями у проектах розробки програмного забезпечення

Ефективне управління проектами є критично важливим компонентом успішної розробки програмного забезпечення, особливо в умовах сучасних організацій, що характеризуються складною структурою та розподіленими командами. Одним із найсуттєвіших викликів у цьому процесі є ідентифікація, аналіз та управління взаємозалежностями між окремими завданнями. Ці залежності можуть існувати як на макрорівні (між командами), так і на мікрорівні (всередині команд), формуючи складні

мережі та ланцюги, які безпосередньо впливають на послідовність виконання робіт, тривалість проекту та ефективність використання ресурсів. Незважаючи на наявність численних комерційних інструментів управління проектами, їхні можливості щодо візуалізації та аналізу складних залежностей, особливо в динамічному середовищі методологій гнучкої розробки, часто є обмеженими. Це призводить до необхідності ручного відстеження залежностей, що є неефективним та схильним до помилок. Дана робота досліджує проблему ефективного управління залежностями шляхом формалізації її за допомогою графових моделей. Пропонується моделювання проекту як орієнтованого графа, де вершини представляють завдання, а ребра – залежності. Тривалість виконання завдань асоціюється з вагами вершин. Обговорюються переваги такого підходу для візуалізації, аналізу та підтримки прийняття рішень щодо планування. Розглядається потенційне застосування графових алгоритмів для автоматизованої побудови часових графіків з урахуванням залежностей, а також сценарії використання такого інструментарію менеджерами проектів для оперативного аналізу та адаптивного управління.

Успіх проектів розробки програмного забезпечення значною мірою залежить від якості процесів управління, включаючи планування, координацію та контроль за виконанням робіт. Сучасна індустрія характеризується зростаючою складністю програмних продуктів та масштабів проектів, що часто реалізуються великими організаціями з багатокомпонентною структурою, поділеною на спеціалізовані підрозділи та команди. Ефективна взаємодія та синхронізація діяльності цих команд є критично важливою. Ключовим аспектом координації є управління залежностями – логічними обмеженнями, які визначають послідовність виконання окремих завдань або груп завдань. Незважаючи на очевидну значущість цього елемента, його ефективне врахування та візуалізація в практиці управління проектами часто становить значну складність.

1.2.1. Природа та вплив залежностей у проектах розробки ПЗ

Структура організації, що займається розробкою програмного забезпечення, зазвичай передбачає поділ на функціональні відділи або доменні команди, кожна з яких відповідає за певний компонент або аспект системи (наприклад, розробка API, інтерфейсу користувача, бази даних). При розробці комплексного продукту діяльність цих команд тісно пов'язана. Часто виконання завдань однією командою (наступником) залежить від результатів роботи іншої команди (попередника). Типовим прикладом є залежність розробки функціональності клієнтської частини (фронтенду) від готовності відповідного інтерфейсу програмного забезпечення (API) на стороні сервера (бекенду). Така міжкомандна залежність є прикладом залежності типу "старт-фініш", коли початок завдання-наступника можливий лише після завершення завдання-попередника.

Окрім міжкомандних, існують також внутрішньокомандні залежності – взаємозв'язки між окремими, часто меншими за обсягом, завданнями в рамках діяльності однієї команди. Ці залежності можуть виникати, наприклад, між розробкою окремих модулів, інтеграцією компонентів або послідовністю виправлення дефектів. Незважаючи на їх "мікроскопічний" рівень, кумулятивний ефект великої кількості внутрішньокомандних залежностей може суттєво впливати на загальну продуктивність команди та здатність доставляти функціональність у межах визначеної ітерації (наприклад, спринту в Scrum фреймворку).

Одне завдання може мати залежності від кількох попередників, формуючи ланцюги залежностей. Довгі та складні ланцюги залежностей є ключовим фактором ризику, оскільки затримка у виконанні будь-якого завдання в ланцюзі безпосередньо впливає на терміни початку та завершення всіх наступних завдань, потенційно спричиняючи значні затримки всього проекту. На менеджера проекту покладається відповідальність за

ідентифікацію, документування, аналіз та активне управління цими залежностями на всіх рівнях.

1.2.2. Інструменти управління проектами та обмеження у роботі із залежностями

На сучасному ринку представлено широкий спектр інструментів управління проектами (наприклад, Basecamp, Asana, Scoro, Jira), які підтримують основні функції планування, відстеження прогресу та комунікації. Серед команд розробки програмного забезпечення особливої популярності набули інструменти, що інтегровані з методологіями гнучкої розробки (Agile), зокрема Jira від Atlassian, який широко використовується у проектах, що застосовують Scrum.

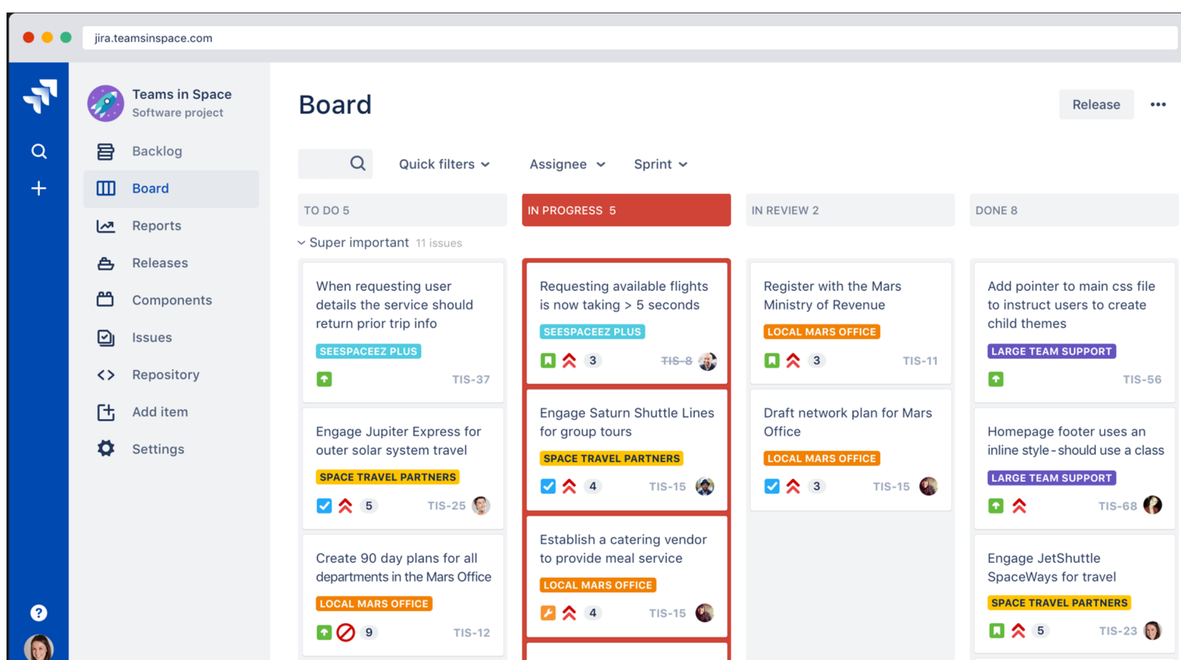

Рисунок 1.1 – Дошка завдань в Jira

Jira підтримує роботу з елементами беклогу, такими як Завдання (Tasks) та Користувацькі історії (User Stories), а також надає функціональність для зв'язування цих елементів. Механізми зв'язування часто використовуються для позначення залежностей між завданнями або

історіями. Наприклад, можна вказати, що Завдання B "блокується" Завданням A, що імпліцитно означає залежність. Однак, незважаючи на можливість перегляду безпосередніх залежностей для окремого елемента, стандартний функціонал Jira та інших подібних інструментів часто є недостатнім для комплексного аналізу мережі залежностей у масштабі спринту або проекту. Зокрема, відсутні вбудовані інструменти для:

- Візуалізації повної мережі залежностей для заданого набору завдань.
- Автоматичного визначення загальної довжини ланцюгів залежностей.
- Ідентифікації всіх критичних шляхів (найдовших ланцюгів) у спринті.
- Моделювання впливу змін (наприклад, затримки завдання, зміни ресурсів) на загальний розклад з урахуванням залежностей.

Ці обмеження призводять до того, що менеджери проектів часто змушені виконувати візуалізацію та аналіз залежностей вручну (наприклад, за допомогою електронних таблиць або діаграм), що є трудомістким, схильним до помилок та не масштабується на великі проекти або динамічні Agile-середовища.

1.2.2. Моделювання залежностей за допомогою теорії графів

Графове представлення дозволяє застосовувати відомі алгоритми теорії графів для аналізу залежностей:

1. Пошук найдовшого шляху в DAG відповідає визначенню критичного шляху проекту – послідовності завдань, які визначають мінімальну можливу тривалість проекту.

2. Топологічне сортування вершин графа надає коректну послідовність виконання завдань з урахуванням усіх залежностей.

3. Алгоритми обходу графа можуть використовуватися для візуалізації мережі залежностей, ідентифікації вузьких місць та аналізу впливу змін.

Розробка програмного забезпечення, що імплементує описану графову модель та відповідні алгоритми, має значний потенціал для покращення

процесів планування та управління проектами розробки ПЗ. Такий інструмент може автоматизувати аналіз залежностей, який зараз часто виконується вручну, знижуючи ризик помилок та економлячи час менеджера проекту.

Прикладом практичного сценарію використання може бути щоденна або ітераційна оцінка статусу проекту. Менеджер проекту може використовувати інтегроване або зовнішнє програмне забезпечення, яке через API отримує актуальні дані про завдання, їхній статус, оцінки часу та залежності з інструменту управління проектами (наприклад, Jira). На основі цих даних програмне забезпечення може:

- Візуалізувати поточну мережу залежностей.

- Автоматично розрахувати актуальний критичний шлях та прогнозовану дату завершення спринту/проекту.

- Ідентифікувати завдання з найбільшою кількістю залежностей або ті, що знаходяться на критичному шляху.

- Моделювати сценарії впливу змін (наприклад, додавання/видалення завдань, зміна оцінок, тимчасова відсутність ресурсів) на розклад та навантаження команди. Наприклад, у випадку хвороби члена команди, менеджер може ввести оновлені дані про доступність ресурсів, і програмне забезпечення швидко перерахує можливі часові графіки та оцінить, чи є поточний обсяг робіт реалістичним.

Такий інструмент надасть менеджерам проектів більш точне, оперативне та обґрунтоване уявлення про стан проекту та ризики, пов'язані із залежностями, дозволяючи приймати своєчасні та ефективні рішення. Це, в свою чергу, може призвести до оптимізації використання ресурсів, скорочення термінів розробки, підвищення якості планування та точності бюджетних оцінок, що є прямою економічною вигодою для компанії.

Управління залежностями є центральною проблемою у плануванні та виконанні проектів розробки програмного забезпечення. Існуючі інструменти

управління проектами часто не надають адекватних можливостей для комплексного аналізу цих залежностей. Моделювання проекту як орієнтованого ациклічного графа з вагами вершин є ефективним підходом для формалізації цієї проблеми. Застосування алгоритмів теорії графів дозволяє автоматизувати процес аналізу залежностей, ідентифікації критичного шляху та генерації можливих часових графіків. Розробка програмного забезпечення на основі цих принципів може суттєво підвищити ефективність управління проектами, забезпечуючи більш точне планування, краще використання ресурсів та зменшення ризиків, пов'язаних із затримками виконання завдань через залежності.

1.3. Аналіз функціональності відомих систем управління проектами розробки програмного забезпечення

1.3.1. Інтегрована система управління бізнесом для сервісних компаній Scoro

Scoro – це комплексна хмарна платформа управління бізнесом, розроблена спеціально для сервісних компаній, таких як маркетингові та креативні агенції, консалтингові фірми, IT-сервіси та інші професійні послуги. Її ключова відмінність полягає в підході "все-в-одному", який має на меті об'єднати різні бізнес-функції в єдиній інтегрованій системі.

Основна філософія Scoro полягає в наданні повної прозорості та контролю над усіма аспектами бізнесу, від першого контакту з клієнтом до виставлення рахунку та аналізу прибутковості.

Це дозволяє компаніям уникнути використання множини розрізнених інструментів (для CRM, управління проектами, фінансів, звітності тощо), що часто призводить до фрагментації даних, неефективності та адміністративних накладних витрат.

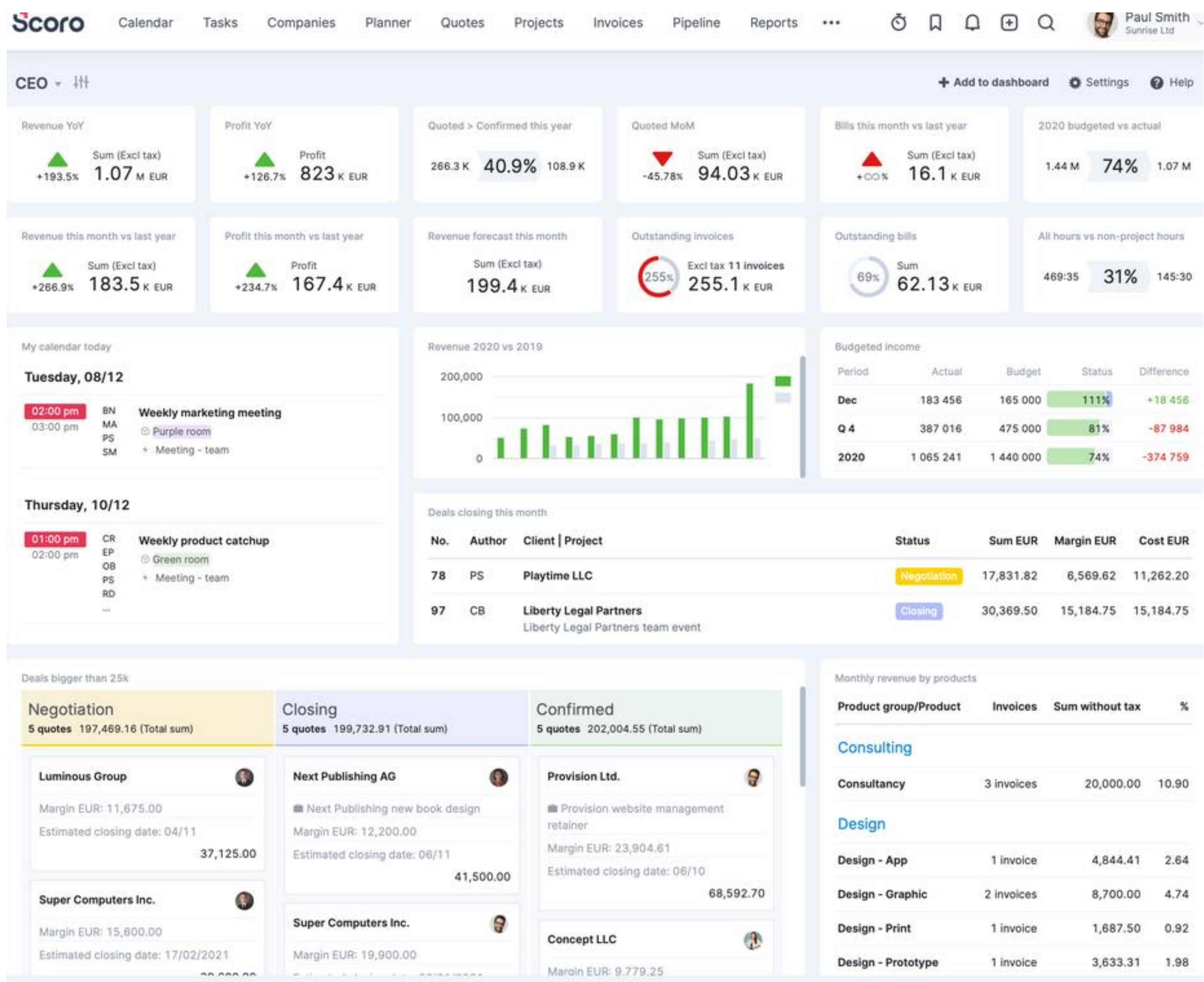

Рисунок 1.2 – Інтерфейс системи Scoro

Розглянемо основні функціональні модулі Scoro.

1. Управління проектами (Project Management). Це один із центральних модулів системи. Він охоплює повний життєвий цикл проекту, включаючи:

2. Планування. Створення проектів, декомпозиція на завдання та підзавдання. Використання різних візуалізацій, таких як діаграми Ганта, дошки Kanban, календарні сітки.

3. Відстеження завдань. Призначення завдань відповідальним особам, встановлення дедлайнів, відстеження прогресу.

4. Облік часу. Можливості точного відстеження часу, витраченого на завдання та проекти (тайм-трекінг), що є критично важливим для білінгу в сервісних компаніях.

5. Бюджетування та контроль витрат. Планування бюджетів проектів, відстеження фактичних витрат та доходів, порівняння з планом.

6. Управління ресурсами. Розподіл членів команди на проекти та завдання, візуалізація завантаженості команди.

7. Звітність за проектами. Детальні звіти про прогрес проекту, прибутковість, використання ресурсів, виконану роботу.

8. CRM та управління продажами (CRM & Sales). Інтеграція з клієнтською базою та процесами продажів. Включає управління контактами, компаніями, угодами (deals), відстеження комунікацій, управління пайплайном продажів, створення та надсилання комерційних пропозицій.

9. Фінанси та виставлення рахунків (Billing & Finance): Можливості для створення рахунків-фактур, управління платежами, обліку витрат. Часто інтегрується з популярними бухгалтерськими системами для синхронізації даних. Білінг може базуватися на відстеженому часі або фіксованих цінах за проектами.

10. Звітність та аналітика (Reporting & Business Intelligence): Надання комплексних звітів та дашбордів, що дозволяють аналізувати ключові бізнес-метрики: прибутковість проектів, завантаженість персоналу, ефективність продажів, фінансові показники тощо. Це дозволяє менеджерам приймати обґрунтовані рішення.

Ключові Переваги Scoro:

- Вся інформація про клієнтів, проекти, фінанси та команду зберігається в одному місці.

- Надання повного огляду бізнесу в реальному часі для керівництва та менеджерів.

- Оптимізація робочих процесів, автоматизація рутинних завдань (наприклад, білінгу на основі часу).

- Прямий зв'язок між відстеженим часом/витратами та виставленням рахунків.

- Візуалізація завантаженості команди та ефективний розподіл ресурсів.

Scoro позиціонується як універсальне рішення для сервісних компаній, що прагнуть оптимізувати свою діяльність, покращити фінансові показники та отримати глибоке розуміння ефективності бізнесу через інтегровану платформу.

1.3.2. Веб-орієнтований інструмент управління проектами та комунікаціями Basecamp

Basecamp – це популярний веб-орієнтований інструмент для управління проектами та командної співпраці, розроблений компанією з однойменною назвою (раніше відомою як 37signals). На відміну від багатьох інших систем управління проектами, які пропонують широкий спектр функцій та можливостей для глибокого налаштування, Basecamp відомий своїм свідомо спрощеним та чітко визначеним підходом.

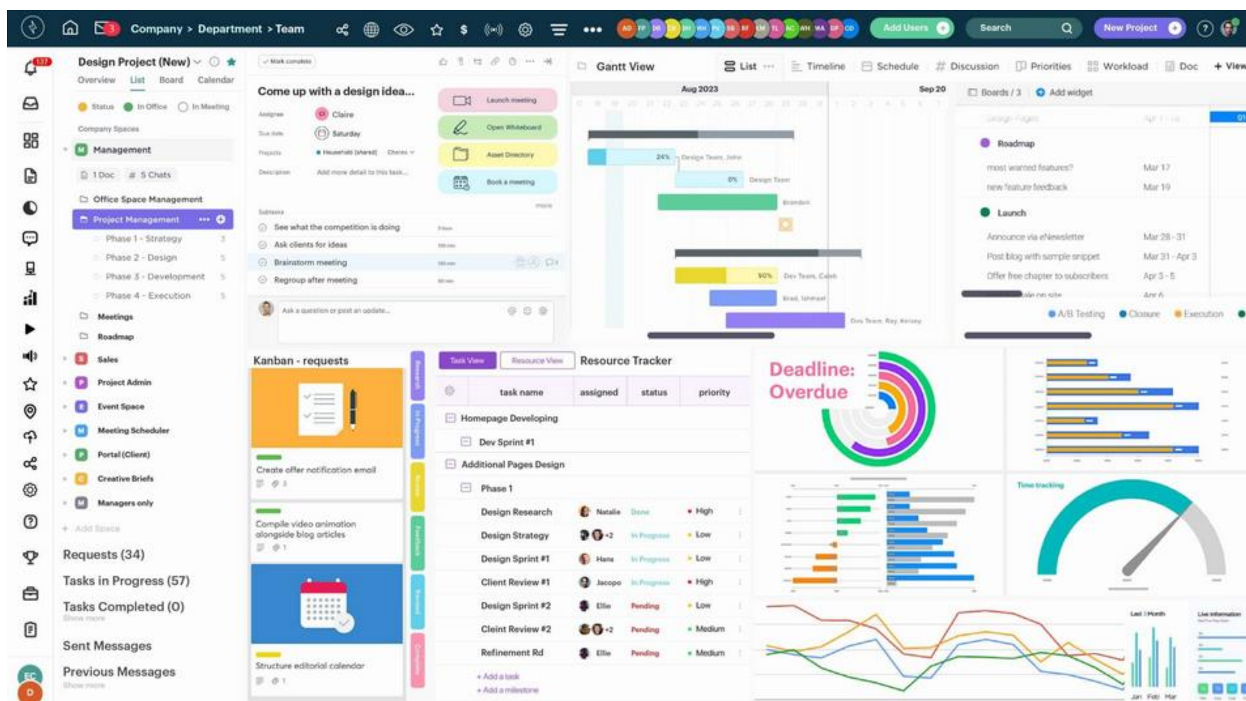

Рисунок 1.3 – Вигляд системи Basecamp

Основна філософія Basecamp полягає у створенні централізованого простору для кожної команди або проекту, де зібрано все необхідне для роботи, що дозволяє зменшити залежність від електронної пошти, розрізнених чатів та файлових сховищ. Система спрямована на покращення комунікації та організації роботи, роблячи її прозорою та легкою для відстеження.

Basecamp пропонує фіксований набір інструментів для кожного "Basecamp" (який може представляти проект, команду, відділ або навіть всю компанію). Цей набір включає:

- дошка повідомлень (Message Board). Центральне місце для оголошень, оновлень та обговорень за конкретними темами. Дозволяє вести організовані, структуровані дискусії, на відміну від безперервного потоку чату.

- списки справ (To-dos). Простий та інтуїтивно зрозумілий інструмент для створення завдань, призначення відповідальних осіб та встановлення термінів виконання. Завдання можна групувати за категоріями або етапами.

- розклад (Schedule). Спільний календар для відстеження дедлайнів, важливих дат, подій та етапів проекту. Може синхронізуватися із зовнішніми календарями.

- документи та файли (Docs & Files). Централізоване сховище для всіх документів, файлів, зображень та нотаток, пов'язаних із проектом. Дозволяє легко обмінюватися файлами та організовувати інформацію.

- чат (Campfire). Вбудований інструмент для швидкого обміну повідомленнями в реальному часі між членами команди або групи.

- автоматичні перевірки (Automatic Check-ins). Функція для налаштування регулярних запитань до команди (наприклад, "Над чим ви працювали сьогодні?", "Які є перешкоди?"), відповіді на які збираються в одному місці, замінюючи необхідність частих статусних зустрічей.

Переваги:

- Basecamp відомий своїм чистим інтерфейсом та легкістю освоєння, що робить його доступним для користувачів з будь-яким рівнем технічної підготовки.

- Зменшує необхідність використання електронної пошти для внутрішніх обговорень та обміну файлами.

- Надає всім учасникам проекту чітке уявлення про статус завдань, обговорення та важливі дати.

- Наявність стандартного набору інструментів спрощує процес прийняття рішень та впровадження.

Однією з відмінних рис Basecamp є його спрощена модель ціноутворення, яка часто базується на фіксованій щомісячній платі незалежно від кількості користувачів (для бізнес-планів), що робить його привабливим для команд, розмір яких може змінюватися.

Basecamp є ефективним інструментом для команд та компаній, які цінують простоту, прозору комунікацію та централізоване управління проектами без надмірної складності. Він чудово підходить для невеликих та середніх проектів, а також для команд, які хочуть впорядкувати свою внутрішню співпрацю та відійти від хаосу електронної пошти та розрізнених чатів.

1.3.3. Система управління проектами Jira

В контексті систем управління проектами, що застосовуються у сфері розробки програмного забезпечення, Jira від компанії Atlassian посідає одне з провідних місць і є фактично стандартом де-факто для багатьох команд по всьому світу. Вона є високоспеціалізованою платформою, основне призначення якої – відстеження запитів (issue tracking) та управління робочими процесами в рамках проектів розробки.

Архітектура Jira спроектована з урахуванням потреб сучасних методологій розробки, демонструючи високу адаптивність до принципів

гнучкої розробки (Agile), зокрема фреймворків Scrum та Kanban, хоча може застосовуватися і в більш традиційних моделях управління проектами. Фундаментальним елементом, навколо якого будується вся робота в системі, є "запит" (issue). Цей термін є узагальнюючим і може репрезентувати широкий спектр одиниць роботи або відстежуваних елементів: від технічного завдання (Task) або дефекту (Bug) до більш високорівневих сутностей, таких як користувацька історія (User Story), епік (Epic) або навіть запит на зміну.

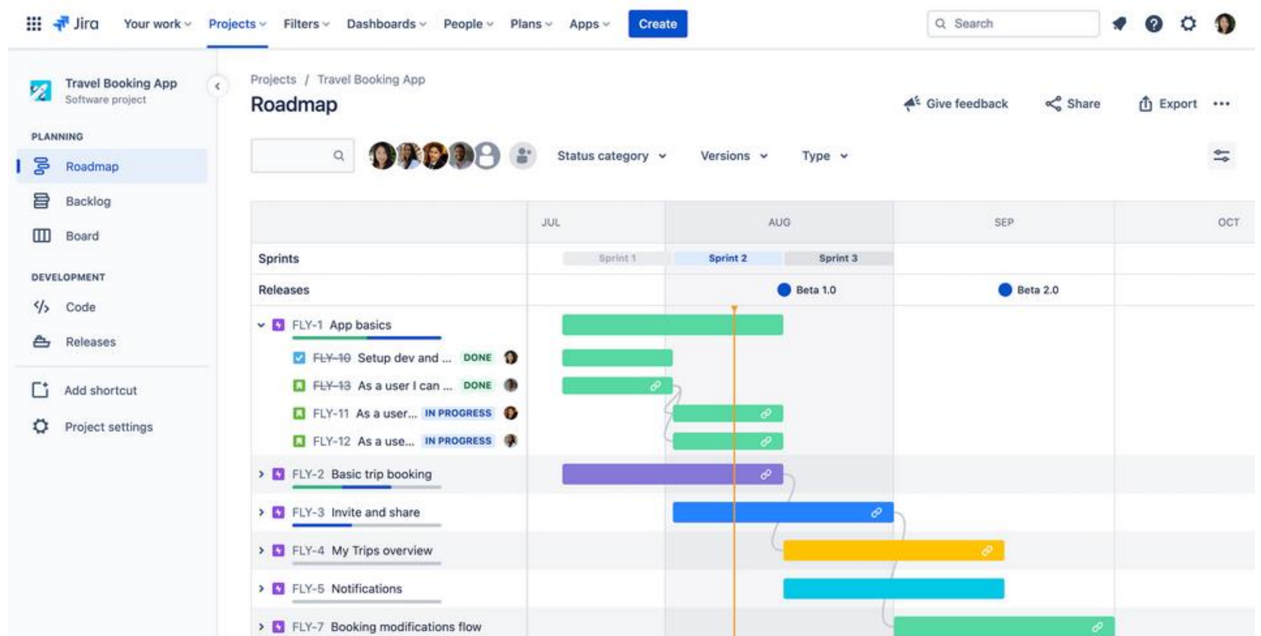

Рисунок 1.4 – Roadmap в Jira

Система надає гнучкі можливості конфігурації для визначення типів запитів, налаштування кастомних полів для збору специфічної інформації та, що особливо важливо, для моделювання робочих процесів (workflows). Кожен тип запиту може мати свій унікальний робочий процес, що відображає послідовність станів, через які проходить елемент роботи (наприклад, "До виконання", "В роботі", "На рев'ю", "Виконано"). Це дозволяє командам точно відобразити свій унікальний флоу розробки.

Для візуалізації та управління потоком робіт Jira пропонує різні типи дошок (Boards). Scrum-дошки оптимізовані для роботи з беклогом продукту,

планування та виконання спринтів (коротких ітерацій фіксованої тривалості) та відстеження прогресу в межах цих ітерацій за допомогою таких інструментів, як burndown-діаграми. Kanban-дошки, у свою чергу, зосереджені на візуалізації безперервного потоку робіт, управлінні незавершеною роботою (Work In Progress - WIP) та оптимізації часу виконання запитів.

Jira також надає функціональність для встановлення зв'язків між запитами (issue linking). Цей механізм дозволяє відображати різноманітні типи відношень, включаючи залежності ("блокує", "блокується"), асоціації ("пов'язаний з"), ієрархії ("є частиною", "має складові") тощо. Це є важливим інструментом для розуміння взаємозв'язків між окремими елементами роботи.

Завдяки своїй гнучкості, масштабованості та орієнтації на потреби розробників, Jira стала центральною платформою для управління розробкою у багатьох компаніях, забезпечуючи структуроване середовище для планування, виконання, моніторингу та звітності за проектами у відповідності до обраної методології. Незважаючи на її потужність та широкий функціонал, управління специфічними аспектами, такими як складні мережі залежностей між завданнями на рівні спринту чи проекту, як зазначалося в попередньому обговоренні, може вимагати додаткових інструментів або специфічних підходів через обмеження стандартних засобів візуалізації таких структур.

РОЗДІЛ 2. АЛГОРИТМІЗАЦІЯ І ФОРМАЛЬНЕ ПРЕДСТАВЛЕННЯ МАП ЗАЛЕЖНОСТЕЙ ПРОЦЕСУ КЕРУВАННЯ ПРОГРАМНИМ ПРОЕКТОМ

2.1. Імплементаційна архітектура та інтеграція даних

Реалізація програмного забезпечення, призначеного для аналізу та візуалізації мап залежностей у проектах розробки програмного забезпечення, була виконана з використанням технологічного стеку, основу якого складають Node.js та веб-фреймворк Express.

Node.js являє собою кросплатформове середовище виконання JavaScript, що функціонує на базі високопродуктивного рушія V8 від Google. Ключовою архітектурною особливістю Node.js є його подійно-орієнтована неблокуюча модель вводу/виводу (event-driven, non-blocking I/O). Така парадигма робить Node.js особливо ефективним для розробки асинхронних мережевих додатків, які вимагають обробки великої кількості одночасних з'єднань без створення окремих потоків для кожного запиту. Це надзвичайно важливо для сценаріїв, де додаток взаємодіє із зовнішніми сервісами, такими як API інших систем, оскільки дозволяє ефективно управляти очікуванням відповідей, не блокуючи виконання інших операцій. Екосистема Node.js також включає вбудований менеджер пакетів npm (Node Package Manager). Npm надає доступ до одного з найбільших у світі репозиторіїв бібліотек та модулів з відкритим вихідним кодом, що значно спрощує процес управління залежностями проекту, прискорення розробки шляхом використання готових компонентів та їх інтеграції у власний додаток.

Express є мінімалістичним та гнучким веб-фреймворком для Node.js, який встановлюється як пакет через npm. Він надає надійний набір функціоналу для розробки веб-додатків та API. До його ключових можливостей входять: обробка HTTP-запитів та відповідей, система

маршрутизації (routing), що дозволяє визначати ендпоінти (endpoints) додатка та логіку їх обробки, а також підтримка проміжних програмних шарів (middleware) для виконання різноманітних завдань (наприклад, парсинг тіла запиту, автентифікація, логування) на шляху обробки запиту. Express формує основу серверної частини розробленого програмного забезпечення, забезпечуючи необхідну інфраструктуру для прийому запитів та взаємодії із зовнішніми сервісами.

Вибір Node.js та Express як технологічної основи обумовлений їхньою ефективністю у побудові додатків, орієнтованих на мережеву взаємодію, що є основним завданням даного проекту – отримання даних із системи управління проектами Jira.

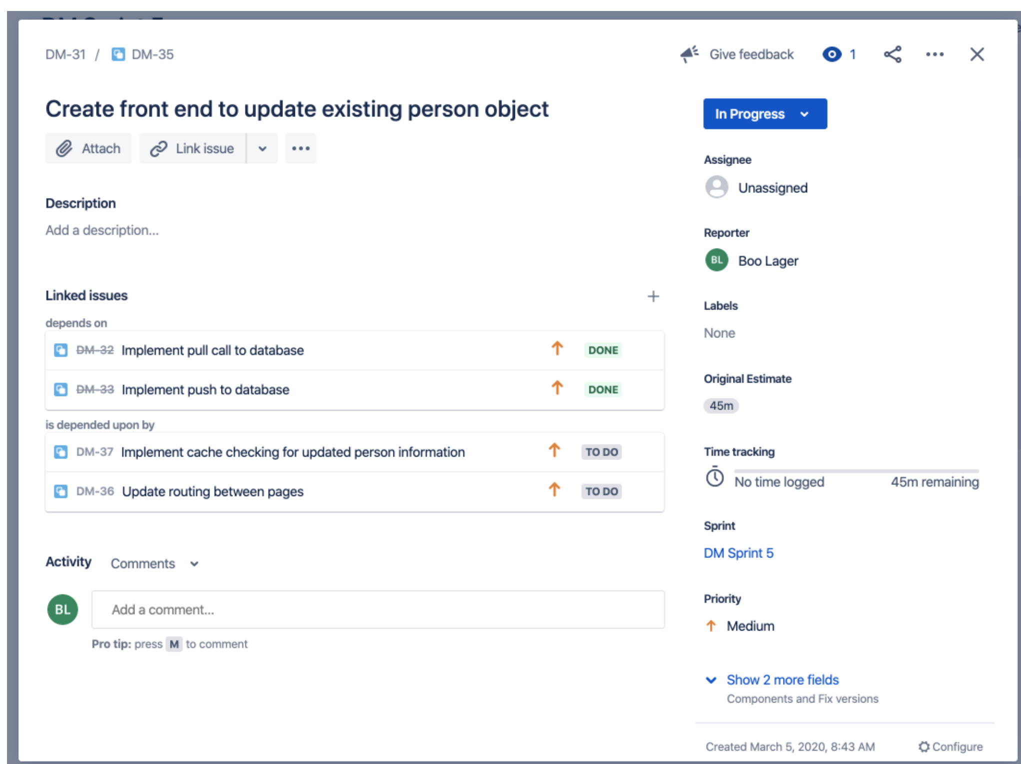

Рисунок 2.1 - Приклад відображення зв'язків між завданнями в інтерфейсі Jira (концептуальна ілюстрація)

Як вже зазначалося у розділі 1, Jira є широко adopted інструментом у середовищі розробки програмного забезпечення, що надає користувацький

інтерфейс для управління завданнями, спринтами, беклогом та зв'язками між елементами роботи.

Однією з функціональних можливостей Jira є встановлення та відображення зв'язків між завданнями, що часто використовується для позначення залежностей. При перегляді деталей окремого завдання інтерфейс користувача відображає ідентифікатори пов'язаних завдань (попередників та наступників), дозволяючи перейти до їх детального перегляду (рисунок 2.1).

Однак, незважаючи на можливість перегляду безпосередніх зв'язків, стандартний інтерфейс Jira не надає вбудованого функціоналу для автоматичної візуалізації або аналізу повних ланцюгів залежностей або всієї мережі залежностей у масштабі спринту чи проекту без необхідності вручну переходити за кожним посиланням та будувати граф поза системою.

Для подолання цього обмеження та отримання структурованих даних про проект, включаючи вичерпну інформацію про завдання, їхні атрибути (як-от оцінка часу) та всі встановлені зв'язки, розроблене програмне забезпечення використовує розширені REST API, які надає Jira. Ці API є ключовим інтерфейсом для програмної взаємодії із системою, дозволяючи здійснювати операції читання (завантаження даних) та запису (надсилання або оновлення даних) до облікового запису Jira. Доступ до ресурсів API вимагає належної авторизації, яка зазвичай реалізується за допомогою персональних токенів доступу (Personal Access Tokens) або протоколів авторизації, таких як OAuth, що забезпечує безпеку та контроль доступу до даних проекту.

Використання REST API дозволяє отримати дані про залежності у форматах, придатних для автоматизованої обробки та подальшого алгоритмічного аналізу, що є критично важливим для побудови графової моделі проекту та генерації можливих часових графіків.

2.2. Теоретичні основи планування на основі методології оцінки та аналізу програм (PERT)

Серед класичних методологій управління проектами, які мають значний концептуальний зв'язок з проблематикою планування робіт з урахуванням залежностей, варто виділити Методологію Оцінки та Аналізу Програм (Program Evaluation and Review Technique – PERT). Розроблена у 1957 році в рамках масштабного та високотехнологічного проекту "Поларіс" Військово-морських сил США, PERT була спеціально призначена для управління великими, складними науково-дослідними та інженерними проектами, що характеризувалися високим ступенем невизначеності та значною кількістю взаємозалежних завдань [3]. Основною метою PERT є надання структурованого підходу до планування, оцінки тривалості та прогнозування ймовірності своєчасного завершення проекту шляхом аналізу мережі робіт.

Реалізація методології PERT передбачає декілька послідовних етапів. По-перше, здійснюється детальна декомпозиція проекту на окремі завдання (activities) або роботи та встановлюється логічна послідовність їх виконання, що визначає залежності "попередник-наступник". По-друге, для кожного завдання проводиться триточкова оцінка тривалості. Цей етап є ключовим для врахування невизначеності, притаманної складним проектам. Оцінюються три значення часу:

- Оптимістичний час (T_оптимістичний) – мінімально можлива тривалість виконання завдання за ідеальних умов, без несподіванок та проблем.

- Песимістичний час (T_песимістичний) – максимально можлива тривалість виконання завдання за найгірших умов, що можуть виникнути (за винятком повного провалу завдання).

- Найбільш ймовірний час (T_найімовірніший) – оцінка тривалості, яка є найбільш реалістичною за звичайних умов виконання завдання.

На основі цих трьох оцінок обчислюється очікуваний час виконання завдання (T_очікуваний) за формулою, що базується на припущенні про бета-розподіл ймовірності тривалості завдання:

$$T_{expected} = \frac{1}{6}(T_{optimistic} + 4T_{most\ likely} + T_{pessimistic})$$

Ця формула надає зважене середнє, де найбільш ймовірний час має найбільший вплив (вага 4) на очікувану тривалість порівняно з оптимістичним та песимістичним часом.

Наступним кроком є побудова мережевої діаграми PERT. Ця діаграма візуально представляє структуру проекту, де вершини (nodes) зазвичай представляють завдання або події (залежно від стилю діаграми – Activity-on-Node або Activity-on-Arrow), а орієнтовані ребра (arcs) позначають логічні залежності між ними, вказуючи на послідовність виконання. Прикладом такої діаграми може слугувати рисунок 2.2 (концептуально).

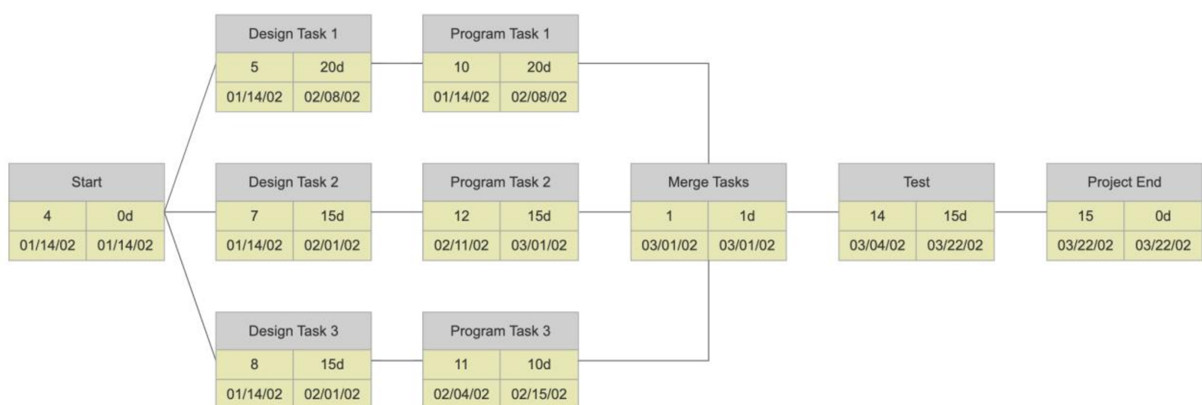

Рисунок 2.2 - Приклад мережевої діаграми PERT (концептуальна ілюстрація)

Діаграма зазвичай включає фіктивні вершини початку (Start) та кінця (End) проекту з нульовою тривалістю для зручності розрахунків. Для кожної вершини-завдання на діаграмі можуть відображатися її атрибути (назва, очікувана тривалість) та розрахункові часові параметри.

Після побудови мережевої діаграми та визначення очікуваних тривалостей завдань здійснюється розрахунок ключових часових параметрів для кожного завдання шляхом виконання прямого та зворотного проходу по мережі. Основними параметрами є:

- Час раннього початку (Early Start Time - EST): Найраніший можливий час, коли може розпочатися завдання, виходячи з завершення всіх його безпосередніх попередників.

- Час раннього завершення (Early Finish Time - EFT): Найраніший можливий час, коли може завершитися завдання (EFT=EST+T_очікуваний).

- Час пізнього завершення (Late Finish Time - LFT): Найпізніший можливий час, коли має завершитися завдання, щоб не спричинити затримку будь-якого наступного завдання або проекту в цілому.

- Час пізнього початку (Late Start Time - LST): Найпізніший можливий час, коли має розпочатися завдання, щоб не спричинити затримку будь-якого наступного завдання або проекту в цілому (LST=LFT−T_очікуваний).

- Запас часу (Slack / Float - S): Різниця між пізнім та раннім часом початку (або завершення) завдання (S=LST−EST=LFT−EFT). Це час, на який може бути відкладено виконання завдання без впливу на загальну тривалість проекту.

Розрахунки ранніх часів виконуються під час прямого проходу (від початку до кінця проекту), де EST завдання є максимальним EFT усіх його безпосередніх попередників. Розрахунки пізніх часів виконуються під час зворотного проходу (від кінця до початку проекту), де LFT завдання є мінімальним LST усіх його безпосередніх наступників.

Після обчислення цих часових параметрів для всіх завдань здійснюється ідентифікація критичного шляху (Critical Path). Критичний шлях визначається як найдовший за сукупною очікуваною тривалістю завдань шлях від початкової до кінцевої вершини в мережевій діаграмі. Завдання, що лежать на критичному шляху, мають нульовий запас часу (S=0), що означає, що будь-яка затримка у їх виконанні призведе до відповідної затримки завершення проекту в цілому. Таким чином, критичний шлях представляє мінімально можливу тривалість проекту та вимагає особливої уваги та контролю з боку менеджера проекту.

Хоча методологія PERT була розроблена для інших типів проектів (переважно з фізичними артефактами та чітко визначеними послідовностями), її базові принципи – декомпозиція робіт, ідентифікація залежностей, мережеве планування та аналіз критичного шляху – мають пряме застосування до проблематики управління проектами розробки програмного забезпечення, особливо у контексті аналізу залежностей та прогнозування термінів. Моделювання проекту як мережевої діаграми PERT (або її варіації у вигляді орієнтованого графа) дозволяє візуалізувати складні взаємозв'язки між завданнями та системно розрахувати часові параметри на основі оцінок тривалості. Однак, слід враховувати певні обмеження та відмінності при застосуванні PERT у чистому вигляді до сучасної розробки ПЗ, зокрема в Agile-середовищах. Триточкова оцінка може бути складною в умовах високої невизначеності та частих змін вимог. Крім того, PERT традиційно фокусується на часі та залежностях, менше уваги приділяючи управлінню ресурсами, що є критичним для проектів з обмеженими командами розробників. Тим не менш, фундаментальний підхід до представлення проекту як мережі залежних завдань залишається актуальним і є теоретичною основою для розробки більш сучасних алгоритмів планування, що враховують специфіку розробки ПЗ та можливості автоматизованого збору даних через API, як це реалізовано у даній роботі [3].

2.3. Використання теорії графів для задачі планування виконання завдань

Розробці програмного забезпечення, що моделює описану проблему, передує аналіз відповідної математичної моделі. Залежності між елементами системи можуть бути ефективно представлені за допомогою теорії графів. Зокрема, розглянуті залежності можуть бути змодельовані у вигляді незв'язаних, зважених, орієнтованих графів. Слід зазначити, що вага може бути асоційована як з вершинами графа, так і з його ребрами (наприклад, шляхом введення уявної кореневої вершини для моделювання ваги ребер).

У даній роботі терміни "вершина" та "вузол" використовуються як синоніми, а орієнтовані ребра графа представляють залежності між вершинами. Розглянемо орієнтований граф зі зваженими вершинами, представлений на рисунку 2.3.

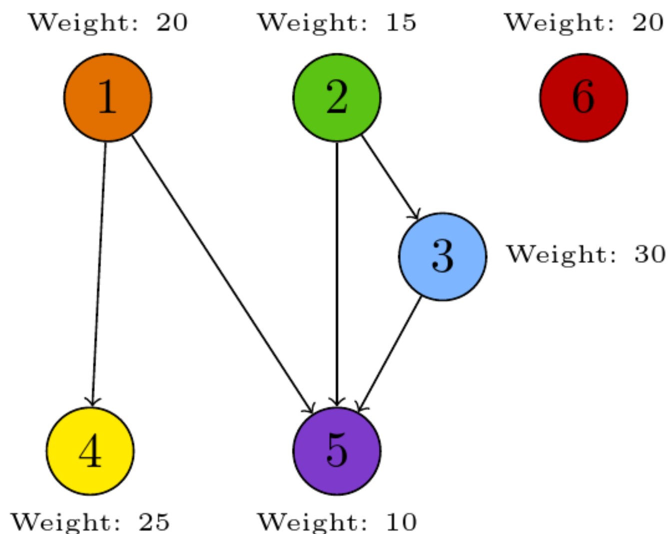

Рисунок 2.3 - Простий орієнтований граф зі зваженими вершинами

Нехай кожна вершина (вузол) представляє певне завдання, а кожне орієнтоване ребро (u,v) вказує на залежність, де виконання завдання v

залежить від завершення виконання завдання u. Вага кожної вершини відповідає оцінці часу, необхідного для виконання відповідного завдання.

Задача, що розглядається, полягає в оптимальному розподіленні цих завдань на паралельних часових лініях (ресурсах) таким чином, щоб дотримувалася умова залежності: виконання завдання, представленого вершиною v, може розпочатися лише після завершення виконання всіх завдань, представлених вершинами u, для яких існує ребро (u,v).

З метою ілюстрації постановки задачі представимо бажаний результат такого розподілу для випадку двох паралельних часових ліній (ресурсів) для графа, зображеного на рисунку 2.3. Цей результат показано на рисунку 2.4.

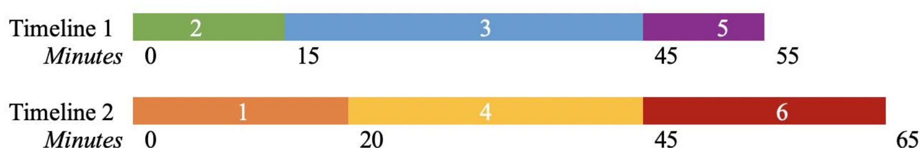

Рисунок 2.4 - Приклад розподілу завдань (вершин) графа з рисунка 2.2 на двох паралельних часових лініях

На рисунку 2.4 видно, що кожному вузлу (завданню), представленому на рисунку 2.3, відповідає певний часовий інтервал на одній із часових ліній. Так, вузол 1 з вагою 20 (рисунок 2.3) займає часовий інтервал тривалістю 20 одиниць часу (рисунок 2.4), вузол 5 – інтервал тривалістю 10 одиниць часу, тощо.

Слід зазначити, що розподіл завдань повинен задовольняти умову залежності: часовий інтервал виконання завдання v не може передувати або перетинатися з часовим інтервалом виконання будь-якого завдання u, від якого залежить завдання v. Це означає, що якщо у графі існує орієнтований шлях від вершини v_i до вершини v_j, то завершення виконання завдання,

представленого вершиною v_i, повинно відбутися суворо до початку виконання завдання, представленого вершиною v_j.

Хоча для графів з невеликою кількістю вершин розв'язання цієї задачі є відносно простим, її обчислювальна складність швидко зростає зі збільшенням кількості вершин, що вказує на NP-складність проблеми в загальному випадку.

2.4. Топологічне впорядкування як інструмент управління залежностями

Для врахування залежностей між завданнями, що моделюються орієнтованими ребрами графа, використовується топологічне сортування. Дано орієнтований ациклічний граф (ОАГ, або DAG) G = (V,E), де V – множина вершин, а E – множина орієнтованих ребер. Топологічне сортування (або топологічне впорядкування) множини його вершин V є лінійним впорядкуванням вершин v_1,v_2,dots,v_n таким, що для кожного орієнтованого ребра (v_i,v_j)inE, вершина v_i передує вершині v_j у цьому впорядкуванні ($i \< j$). Можливість побудови топологічного сортування є необхідною і достатньою умовою ациклічності орієнтованого графа.

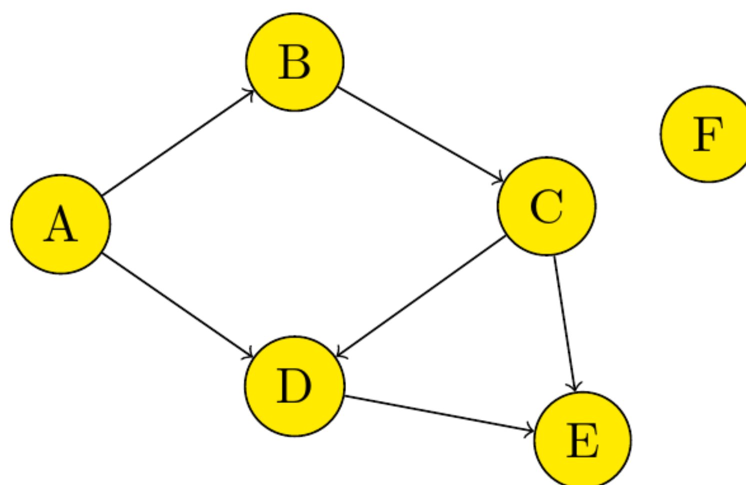

Рисунок 2.5 - Приклад орієнтованого ациклічного графа

Наприклад, для графа, зображеного на рисунку 2.5, можна побудувати топологічне впорядкування. Послідовність вершин (A,B,C,D,E,F) є прикладом дійсного топологічного сортування цього графа. Однак послідовність (A,B,F,C,D,E) також є дійсним топологічним впорядкуванням, що ілюструє відсутність унікальності топологічного сортування для деяких графів. Слід також зазначити, що кількість можливих топологічних впорядкувань обернено пропорційна кількості та структурі залежностей (ребер) у графі. Додавання орієнтованого ребра можна інтерпретувати як накладання додаткового обмеження на відносне положення двох вершин у лінійному впорядкуванні, тим самим зменшуючи множину можливих топологічних сортувань.

Продемонструємо це на прикладі. Нехай задано граф G=(V,E) з множиною вершин V=W,X,Y,Z та порожньою множиною ребер E=emptyset. Оскільки відсутні ребра, відсутні й обмеження на порядок слідування вершин. Будь-яке лінійне впорядкування вершин є дійсним топологічним сортуванням. Кількість таких впорядкувань дорівнює 4=24. Тепер додамо до графа ребра, визначивши множину E=(X,Y),(Y,Z). Введені ребра накладають обмеження: вершина X повинна передувати вершині Y, а вершина Y – вершині Z. Множина всіх дійсних топологічних сортувань для цього графа складається з наступних послідовностей: (W,X,Y,Z), (X,W,Y,Z), (X,Y,W,Z), (X,Y,Z,W). Таким чином, виникає задача розробки алгоритму для знаходження хоча б одного топологічного сортування заданого орієнтованого ациклічного графа.

Алгоритм Кана є одним із методів побудови топологічного сортування. Він ґрунтується на властивості орієнтованих ациклічних графів (DAG), яка полягає в тому, що кожен непорожній DAG має принаймні одну вершину з вхідним степенем (індегрі) рівним 0. Відсутність циклів є необхідною умовою існування топологічного сортування, що відповідає припущенням розглядуваної задачі розподілу завдань.

Кроки алгоритму Кана:

1. Обчислити вхідний степінь (індегрі) для кожної вершини графа.

2. Ініціалізувати порожню чергу Q. Додати всі вершини з вхідним степенем, рівним 0, до черги Q.

3. Ініціалізувати порожній список L, який буде містити результат топологічного сортування.

4. Поки черга Q не порожня:

4.1. Видалити вершину u з початку черги Q.

4.2. Додати вершину u до кінця списку L.

4.3. Для кожної вершини v, суміжної з u (для кожного ребра):

4.3.1. Зменшити вхідний степінь вершини v на 1.

4.3.1. Якщо після зменшення вхідний степінь вершини v стає рівним 0, додати v до кінця черги Q.

5. Після завершення циклу:

5.1. Якщо кількість вершин у списку L дорівнює загальній кількості вершин у графі, то список L містить дійсне топологічне сортування.

5.2. В іншому випадку граф містить цикл, і топологічне сортування неможливе.

Цей алгоритм дозволяє ефективно побудувати одне з можливих топологічних сортувань, що є важливим кроком для визначення коректного порядку виконання завдань з урахуванням існуючих залежностей у розглядуваній задачі.

2.5. Розв'язання задачі про планування інтервалів методом жадібного вибору за часом завершення

Задача про планування інтервалів (або задача вибору активностей) є класичною задачею оптимізації. Вона може бути формально описана наступним чином:

Нехай задано множину з n завдань (або активностей) $A=\{a1,a2,\ldots,an\}$. Кожне завдання ai характеризується часовим інтервалом $[si,fi)$, де si – час початку виконання, а fi – час завершення, причому $si<fi$. Інтервали зазвичай вважаються напіввідкритими $[si,fi)$, що означає, що завдання починається в момент si і закінчується до моменту fi.

Два завдання ai та aj є сумісними, якщо їхні відповідні часові інтервали не перетинаються, тобто $[si,fi)\cap[sj,fj)=\varnothing$. Це еквівалентно умові, що або $fi\leq sj$, або $fj\leq si$. Несумісні завдання не можуть бути виконані одночасно на одному ресурсі.

Основна постановка задачі про планування інтервалів полягає у знаходженні такої підмножини $S\subseteq A$ сумісних (попарно несумісних) завдань, яка має максимально можливий розмір, тобто $|S|$ є максимальним. Ця задача відповідає ситуації, коли необхідно виконати якомога більше завдань на одному ресурсі, маючи фіксовані інтервали їх виконання.

На відміну від більш складних задач планування, задача про планування інтервалів з критерієм максимізації кількості завдань може бути ефективно розв'язана за допомогою жадібного алгоритму. Доведено, що такий алгоритм знаходить оптимальне розв'язання.

Жадібний алгоритм для цієї задачі базується на наступній ідеї: завжди вибирати завдання, яке завершується найраніше, оскільки це залишає максимум вільного часу для виконання наступних завдань.

Кроки алгоритму:

1. Сортування: Відсортувати всі n завдань у порядку неспадання (за зростанням) часу їх завершення fi. Якщо два завдання мають однаковий час завершення, їх відносний порядок не має значення для коректності алгоритму. Нехай відсортовані завдання позначено як $a1,a2,\ldots,an$ так, що $f1 \leq f2\leq\cdots\leq fn$.

2. Ініціалізація: Створити порожню множину S, яка буде містити вибрані (заплановані) завдання. Додати перше завдання a1 (з найранішим

часом завершення) до множини S. Зберегти час завершення цього завдання як час завершення останнього вибраного завдання: last_finish_time=f1.

3. Ітеративний вибір: Для кожного наступного завдання ai у відсортованому списку, починаючи з i=2 до n:

- Перевірити, чи є завдання ai сумісним з останнім доданим до S завданням. Це виконується, якщо час початку завдання ai не раніше часу завершення останнього вибраного завдання, тобто si≥last_finish_time.

- Якщо умова сумісності виконується, додати завдання ai до множини S і оновити last_finish_time=fi.

4. Результат: Множина S містить максимально можливу кількість сумісних завдань.

Цей простий жадібний підхід гарантує знаходження оптимального розв'язання для задачі про планування інтервалів з критерієм максимізації кількості. Важливо розрізняти цю задачу від більш складних варіантів, наприклад, задачі зваженого планування інтервалів (де кожне завдання має вагу, і мета – максимізувати суму ваг, а не кількість), або задач планування з залежностями, розглянутих раніше, де застосування лише жадібного алгоритму на основі інтервалів може бути недостатнім.

РОЗДІЛ 3. ПРОГРАМНА ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ПОБУДОВИ МАП ЗАЛЕЖНОСТЕЙ ПРОЦЕСУ КЕРУВАННЯ ПРОГРАМНИМ ПРОЕКТОМ

3.1. Формування вихідних даних та інфраструктури проекту

Центральним аспектом даного дослідження стало опрацювання даних системи управління проектами Jira. Початковий етап роботи включав створення тестового облікового запису в Jira та моделювання структури даних, необхідної для проведення експериментального аналізу. Як зазначено у вступі, Jira широко використовується як інструмент для планування та відстеження виконання завдань у проєктах, зокрема в розробці програмного забезпечення.

Функціонал системи дозволяє користувачам створювати та редагувати завдання (issues), історії користувачів (user stories) та інші типи записів, встановлювати зв'язки між ними (наприклад, залежності) та вказувати оцінки часу на їх виконання (estimated time).

Для формування репрезентативного набору даних зі складними та різноманітними структурами залежностей було змодельовано низку завдань у межах тестового проєкту (спринту). Створена структура залежностей охоплювала типові випадки, такі як:

- незалежні завдання;

- завдання з однією або кількома вхідними залежностями (від нього залежить декілька інших завдань);

- завдання з однією або кількома вихідними залежностями (завдання залежить від кількох інших);

- комбіновані та розгалужені структури, включаючи ланцюжки, де безпосередній послідовник завдання мав спільні залежності з попередніми завданнями у ланцюгу.

Візуалізація змодельованих залежностей між завданнями представлена на рисунку 3.1.

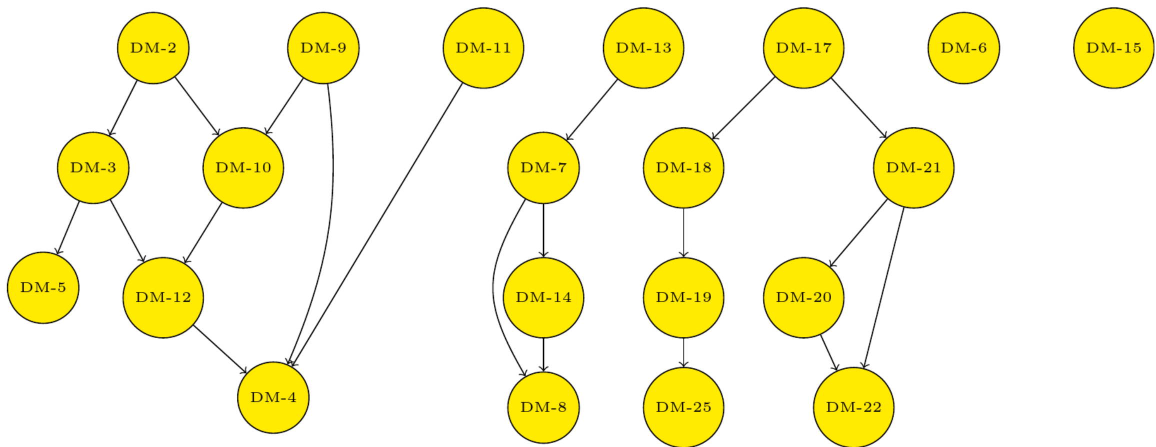

Рисунок 3.1 - Змодельовані залежності між завданнями в тестовому проєкті Jira

Аналіз рисунка 3.1 показує наявність складних та перетинаючихся ланцюжків залежностей, які утворюють орієнтований граф. Оцінки часу виконання для кожного із змодельованих завдань наведено у таблиці 3.1.

Таблиця 3.1 - Початкові оцінки часу виконання змодельованих завдань у Jira

Завдання	**Оцінка часу**	**Завдання**	**Оцінка часу**	**Завдання**	**Оцінка часу**
DM-2	30 хв	DM-9	1 год	DM-17	5 год
DM-3	3 год	DM-10	3 год 30 хв	DM-18	2 год
DM-4	30 хв	DM-11	2 год 30 хв	DM-19	4 год
DM-5	2 год 30 хв	DM-12	6 год	DM-20	3 год
DM-6	2 год 15 хв	DM-13	4 год	DM-21	1 год
DM-7	3 год	DM-14	1 год	DM-22	1 год 15 хв
DM-8	6 год	DM-15	3 год 15 хв	DM-23	45 хв

Слід відзначити, що через різницю в оцінках часу виконання окремих завдань, сумарна тривалість ланцюжка залежностей з меншою кількістю завдань може перевищувати тривалість ланцюжка з більшою кількістю завдань. Це підкреслює важливість врахування ваги вершин (тривалості завдань) при плануванні.

Після внесення тестових даних у систему Jira було проведено дослідження доступних кінцевих точок (endpoints) її програмного інтерфейсу (API). API Jira надає можливість стороннім додаткам програмно взаємодіяти з даними користувачів, зокрема отримувати інформацію про завдання, їх залежності та оцінки часу.

Для автентифікації та авторизації запитів до API Jira було згенеровано персональний токен API, який використовувався для здійснення необхідних HTTP-запитів.

Наявність підготовлених даних та визначення методу програмного доступу до них стали основою для початку розробки програмного забезпечення, призначеного для обробки цих даних та реалізації алгоритмів планування.

Початковий етап налаштування середовища розробки включав встановлення платформи Node.js, яка використовується для виконання JavaScript коду на стороні сервера, та веб-фреймворку Express.js для спрощення розробки серверної частини додатку. Встановлення здійснювалося за допомогою пакетного менеджера Homebrew для операційних систем macOS та Linux.

Далі було структуровано директорію нашого проєкту, створивши піддиректорії для розміщення необхідних програмних модулів, файлів вихідного коду JavaScript (наприклад, для реалізації алгоритмів), основного файлу програми (app.js) та файлів, що відповідають за обробку запитів (маршрутизація).

3.2. Обробка даних

За допомогою пакета запитів npm ми надсилали HTTP-запити до API Jira. Ці запити включали наш URL, токен авторизації та параметри запиту. Відповідні кінцеві точки API включали запити на всі дошки проекту, на всі спринти дошки та на всі проблеми для спринту. Кінцеві точки повертали об'єкти JSON, які можна було скоротити до включення лише інформації, яка буде використана для побудови ланцюжків залежностей. Запит на всі проблеми в спринті призвів би до понад 7000 рядків даних, які не можна було б ефективно сортувати. Зверніть увагу, що проблема — це термін Jira для будь-якого завдання, історії або помилки програмного забезпечення в спринті.

У файлі api.js ми визначили наступні виклики API за допомогою пакета запитів npm: getDashboard, getAllBoards, getAllSprints, getIssuesForSprint та getIssuesForBoard.

Ці запити повертають обіцянки JavaScript, які розв'яжуться в об'єкт JSON, що містить дані. В файлі parser.js ми використовували ці функції та продовжували знімати необхідну інформацію з об'єкта JSON. Потім повернута оброблена інформація поверталася.

Обробка getAllBoards поверне обіцянку, яка розв'язується в об'єкт JSON, що містить id, ім'я, ключ проекту та ім'я проекту дошки. Обробка getSprints поверне обіцянку, яка розв'язується в об'єкт JSON, який містить id, стан (активний, майбутній або завершений), ім'я, дату початку, дату завершення та boardId спринту. Мета обробки дошок та спринтів — знайти дошки, які містять активні спринти. Після того, як знайдені ідентифікатори активних спринтів, ми можемо обробляти getIssuesForSprint, щоб знайти інформацію, необхідну для побудови ланцюжків залежностей. Структура та інформація в повернутому об'єкті JSON показані нижче.

Лістинг 3.1. Повернутий JSON об’єкт

```
// об'єкт JSON проблеми
{
    id : 10019,
    key : DM-7,
    sprintId : 3,
    inwardLinks : [{id : 10020, key : DM-8}, {id : 10026, key : DM-14}],
    outwardLinks : [{id : 10025, key : DM-13}],
    issueType : Підзавдання,
    subtasksId : [],
    worklogs : [{timeSpent : 18}{timeSpent : 36}],
    originalTimeEstimate : 16200,
    timeRemaining : 108,
    status : до виконання
}
```

Важливо зазначити, що inwardLinks містить id та key проблем, від яких це завдання залежить, а outwardLinks містить id та key проблем, від яких це завдання залежить. Наприклад, розглянемо дані на рисунку 3.1. Масив inwardLinks для DM-7 включав би DM-14 та DM-8, тоді як масив outwardLinks містив би DM-13.

Були деякі проблеми при початковій обробці цих даних, оскільки необхідні частини інформації часто були в дуже вкладених об'єктах, і якщо в об'єкті не було доступної інформації, були помилки невизначеності, з якими потрібно було впоратися. Тип проблеми іноді був полем, до якого не можна було отримати доступу. Приклади типів проблем — це завдання, підзавдання, помилка та історія.

3.3. Трансформація даних та побудова графової моделі

Перед побудовою та аналізом ланцюжків залежностей, отримані через API Jira дані потребували трансформації у структуру, придатну для ефективної графової обробки та подальшого застосування алгоритмів планування.

Оскільки кожне завдання (issue) в системі Jira має унікальний ідентифікатор (ключ), було реалізовано механізм зберігання об'єктів, що представляють завдання, у структурі даних типу "карта" (асоціативний масив, словник). Ключем у цій карті виступав унікальний ключ завдання, що забезпечувало швидкий доступ до даних будь-якого завдання за його ідентифікатором. Ця операція виконувалася в рамках функції buildMap.

Процес отримання даних з Jira включав послідовні виклики до її API:

1. Запит на отримання списку доступних дошок (через відповідну кінцеву точку getBoards).

2. Вилучення ідентифікатора цільової дошки з об'єкта відповіді.

3. Використання ідентифікатора дошки для запиту списку спринтів у межах цієї дошки (getSprints).

4. Отримання ідентифікаторів потрібних спринтів з відповіді getSprints.

5. Використання ідентифікаторів спринтів для запиту інформації про всі завдання, включені до цих спринтів (getIssuesForSprint).

На останньому етапі отримані об'єкти завдань ітерувалися, і для кожного завдання створювався відповідний запис у карті. Значенням для кожного ключа (ідентифікатора завдання) був об'єкт "вузол" (node), що представляв завдання у графовій моделі.

Структура об'єкта "вузол" визначалася конструктором Node, представленим нижче (лістинг 3.2). Цей об'єкт містив як основні атрибути завдання з Jira, так і додаткові поля, необхідні для алгоритмів планування.

Конструктор ініціалізував вузол з ключовими атрибутами завдання та масивами, що представляють зв'язки залежностей. Поля completed, completedTime, minStartTime, endTime були додані спеціально для потреб алгоритмів планування та відстеження стану розподілу завдань на часових лініях.

Таким чином, карта, що містила об'єкти вузлів, забезпечувала централізоване, структуроване та легкодоступне сховище інформації про всі

завдання та їхні безпосередні залежності, формуючи графове представлення даних.

Лістинг 3.2. Структура об'єкта "вузол"

```
// Конструктор об'єкта вузла графа
function Node(issueKey, inwardLinkArray, outwardLinkArray, originalTime,
    remainingTime, status) {
    this.key = issueKey; // Унікальний ключ завдання
    this.originalTimeEstimate = originalTime; // Початкова оцінка часу
    this.timeRemaining = remainingTime; // Час, що залишився
    this.status = status; // Поточний статус завдання
    this.inwardlinks = []; // Масив ключів завдань, що залежать від поточного (вихідні залежн
    this.outwardlinks = []; // Масив ключів завдань, від яких залежить поточне (вхідні залежн
    this.completed = false; // Прапорець завершення
    this.completedTime = -1; // Час завершення
    this.minStartTime = 0; // Мінімальний час початку виконання з урахуванням залежностей
    this.endTime = -1; // Час фактичного завершення після планування

    // Заповнення масивів залежностей
    inwardLinkArray.forEach(link => {
        this.inwardlinks.push(link.key);
    });
    outwardLinkArray.forEach(link => {
        this.outwardlinks.push(link.key);
    });
}
```

3.3.1. Виявлення повних ланцюжків залежностей

Наступним важливим кроком після побудови графової моделі стала ідентифікація та структуроване зберігання повних ланцюжків залежностей у графі завдань. Для потреб подальшого аналізу та планування було визначено поняття "повного ланцюжка залежностей" як орієнтованого шляху у графі завдань, який починається з вершини, що не має вхідних залежностей (вершина-джерело, in-degree = 0), і закінчується вершиною, що не має вихідних залежностей (вершина-стік, out-degree = 0).

Наприклад, у графі, представленому на рисунку 3.1, послідовність завдань DM-13 → DM-7 → DM-14 → DM-8 є повним ланцюжком залежностей згідно з цим визначенням, оскільки DM-13 не залежить від інших завдань, а від DM-8 ніщо не залежить. Натомість, послідовність DM-7

→ DM-14 не є повним ланцюжком, оскільки DM-7 має вхідну залежність (від DM-13), а від DM-14 є вихідна залежність (до DM-8).

Для виявлення всіх повних ланцюжків залежностей та їхнього зберігання у структурованому вигляді було застосовано рекурсивний підхід. Спочатку було ідентифіковано всі вершини-джерела графа, тобто завдання, чий масив outwardlinks (вхідні залежності в термінології коду) має довжину 0. Примітка: У даному контексті outwardlinks в коді відповідає вхідним залежностям завдання в термінології графа (тобто, від яких завдання залежить), а inwardlinks – вихідним залежностям (тобто, які залежать від завдання). Таким чином, вершинами-джерелами є вузли з порожнім масивом outwardlinks.

Для кожної такої вершини-джерела запускалася рекурсивна функція buildArrayOfDependencies. Ця функція приймала карту всіх вузлів, ключ поточного вузла, поточний масив-шлях (що будується) та поточну сумарну оцінку часу для цього шляху. Алгоритм рекурсивного формування ланцюжка працював наступним чином:

1. Додати ключ поточного вузла до поточного масиву-шляху.

2. Додати оцінку часу виконання поточного вузла до сумарної оцінки часу для шляху.

3. Перевірити, чи має поточний вузол вихідні залежності (тобто, чи є завдання, які залежать від нього). Це визначалося за довжиною масиву inwardlinks.

4. Якщо вихідні залежності відсутні (довжина inwardlinks дорівнює 0), це означає, що досягнуто вершини-стоку, і поточний шлях є повним ланцюжком залежностей. Сумарна оцінка часу додається як останній елемент до масиву-шляху, і цей повний ланцюжок зберігається (наприклад, додається до глобального списку).

5. Якщо вихідні залежності існують (довжина inwardlinks > 0), для кожного завдання v, що залежить від поточного завдання u (тобто, для

кожного ключа v у масиві inwardlinks вузла u), рекурсивно викликати функцію buildArrayOfDependencies, передаючи ключ v, оновлений масив-шлях (з доданим u) та оновлену сумарну оцінку часу.

Результатом виконання цієї рекурсивної процедури для всіх вершин-джерел є формування глобального масиву dependencyChains. Кожен елемент цього масиву є окремим масивом, що представляє собою повний ланцюжок залежностей (послідовність ключів завдань) від вершини-джерела до вершини-стоку, з сумарною оцінкою часу виконання всіх завдань у цьому ланцюжку, що зберігається як останній елемент внутрішнього масиву. Ця структура даних dependencyChains надалі використовувалася для реалізації алгоритмів планування з урахуванням залежностей.

3.4 Алгоритм відображення мап залежностей

Основна увага цього проекту була зосереджена на алгоритмі відображення залежностей. Ми хотіли дозволити менеджерам проектів швидко визначати навантаження, яке команда може виконати, та створювати життєздатні часові лінії для посилання. Простий перший випадок для цього — це: за наявності необмеженої кількості розробників, як швидко можна виконати набір завдань. Ми реалізували цю функціональність у функції minTimeUnlimitedDevelopers. Аналізуючи цю проблему, ми бачимо, що ми можемо розподілити кожне незалежне завдання між різними розробниками, а їхні часи завершення збігатимуться з їхніми оціненими часами. Отже, якщо цій функції буде надано набір абсолютно незалежних завдань, результат повинен бути час оцінки найбільшого завдання. Однак, якщо існують залежності всередині завдань, потрібно врахувати більше інформації. Питання полягає в тому, як впоратися з цими залежностями та забезпечити їх ефективне виконання.

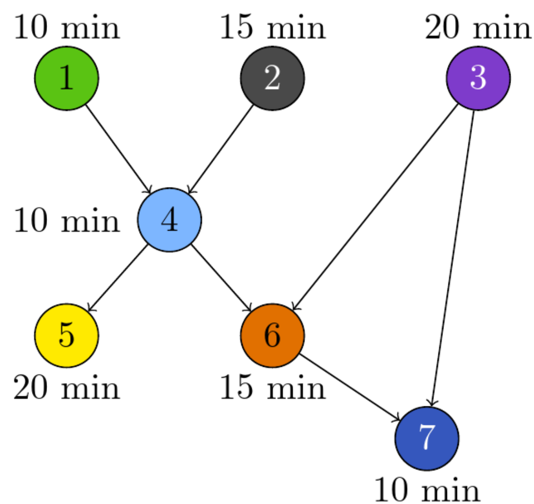

Рисунок 3.2 - Тестові дані для необмеженої кількості розробників

Розглянемо завдання на рисунку 3.2. Очевидно, що для завершення набору завдань потрібно більше часу, ніж найбільше окреме завдання. Це правда, оскільки 4 потрібно завершити перед тим, як можна почати 5, 6 та 7. Зверніть увагу, що обмеження на порядок можна розглянути в термінах ланцюжків залежностей. Тобто, якщо ми побудуємо кожен можливий ланцюжок залежностей та надамо кожному розробнику різний ланцюжок залежностей, ми можемо гарантувати, що кожне завдання буде виконано якомога швидше. Цей розподіл завдань міститиме повторювані завдання, тому нам потрібно буде видалити всі, крім останнього появлення завдання.

Щоб зрозуміти цей процес, візьмімо всі ланцюжки залежностей на рисунку 3.2: [1,4,5], [1,4,6,7], [2,4,5], [2,4,6,7], [3,6,7], [3,7]. Ми надаємо кожному розробнику ланцюжок залежностей та стежимо за часом початку кожного завдання в ланцюжку. Ми визначаємо час початку завдання як суму оцінок часу всіх завдань, які з'являються перед ним у ланцюжку. Потім ми видаляємо кожне повторюване завдання, залишаючи копію з найпізнішим часом початку. Якщо ми не перерахуємо часи початку завдань, ми залишимося з часовою лінією для кожного розробника, де всі залежності для завдання будуть завершені перед тим, як ми дійдемо до залежного завдання.

Рисунок 3.3 показує відповідну часову лінію для кожного ланцюжка залежностей. Розробник 6 повинен почати завдання 7 о 20 хвилин, але зверніть увагу, що всі завдання, від яких залежить 7, не будуть завершені до 40 хвилин. Саме тому ми повинні видалити всі копії завдання, крім останнього, що з'являється.

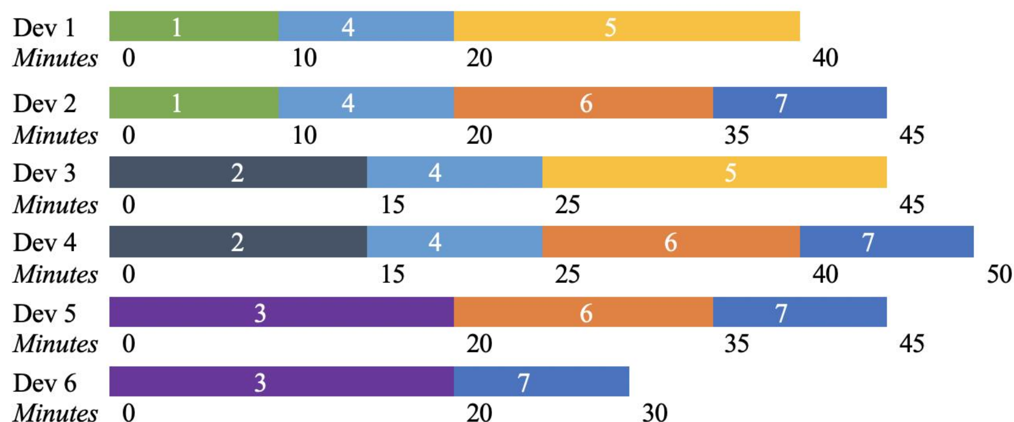

Рисунок 3.3 - Часові лінії для необмеженої кількості розробників до видалення

Рисунок 3.4 показує кожну часову лінію після видалення дублікатів. Ми можемо побачити, що до кожного завдання можна дістатися лише після завершення його залежностей. Також зверніть увагу, що час, необхідний для завершення всіх завдань, буде довжиною найдовшого ланцюжка залежностей.

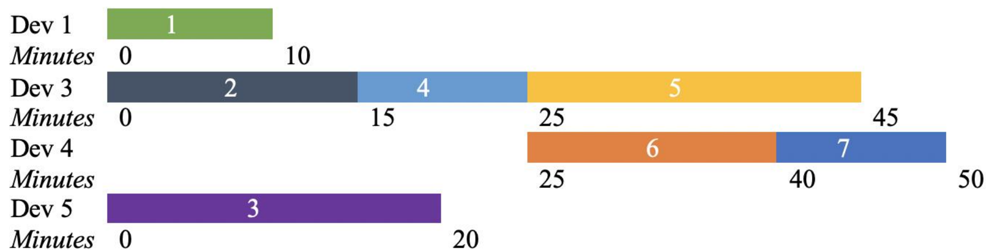

Рисунок 3.4 - Часові лінії для необмеженої кількості розробників після видалення

Тепер ми можемо перейти до реального випадку обмеженої кількості розробників, які можуть виконати завдання. Припустимо, у нас є лише певна кількість розробників для виконання завдань на рисунку 3.2. Це виявляється значно складнішим, і ми використаємо версію жадібного алгоритму для вирішення цієї проблеми. Спочатку ми створюємо масив для представлення всіх розробників. Довжина масиву розробників буде такою ж, як і кількість розробників. Звідси ми створюємо два мапи вузлів, порожній мап, який представляє всі виконані завдання, та мап, який містить всі невиконані завдання. Спочатку мапа невиконаних завдань міститиме всі завдання в оригінальному мапі вузлів. Згадаймо проблему планування інтервалів, описану раніше. Алгоритм, використаний для вирішення цієї проблеми, вимагає від нас знати часи завершення кожного завдання. Тому ми можемо перевірити найраніший час завершення для кожного завдання, але це вимагає додаткових роздумів у випадку обмеженої кількості розробників.

Відомо, що всі залежності завдання повинні бути завершені перед тим, як можна почати завдання. Але також згадайте, що обмеження кількості розробників спричинить те, що часи початку деяких завдань будуть відсунуті. Отже, ми повинні обчислити найпізніший час завершення будь-якої залежності завдання та додати оцінений час для завдання, щоб визначити найраніший можливий час завершення. Ми позначаємо довжину завдання G як ｜G｜, час завершення як TG, а найраніший можливий час завершення як Gend. Найраніший можливий час завершення обчислюється виключно на основі довжини завдань та їхніх залежностей, тоді як час завершення обчислюється по мерезі виконання завдань. Зверніть увагу, що коли ми обмежуємо кількість розробників, завдання може мати пізніший час завершення, ніж його найраніший можливий час завершення. Наприклад, нехай A — це завдання, яке залежить від завдань B та C. Обчислено, що B буде завершено в час TB=30хв, однак C не буде завершено до часу TC=40хв. Оскільки TB<TC, A_{end} = ｜A｜ + TC.

Тоді ми визначаємо найраніший час завершення всіх невиконаних завдань та вибираємо завдання з найранішим часом завершення, ми позначаємо це завдання A. Ми позначаємо кількість часу, яка нараховується розробнику d як ∣d∣. Потім ми визначаємо, якому розробнику потрібно надати завдання. Ми надаємо завдання розробнику di таким чином, що ∣di∣ ≥Aend−∣A∣ і якщо існує dkdk такий, що ∣dk∣<∣di∣, то ∣dk∣ ≱Aend−∣A∣. Щоб сформулювати це неформально, ми вибираємо розробника з найменшою кількістю часу, проведеного на роботі, таким чином, що час, проведений на роботі, більший або дорівнює найранішому часу початку завдання. Потім ми надаємо завдання розробнику, оновлюємо розробника з новим часом, проведеним на роботі, видаляємо завдання з мапи невиконаних та поміщаємо завдання в мапу виконаних. Цей процес повторюється, поки мапа невиконаних не стане порожньою.

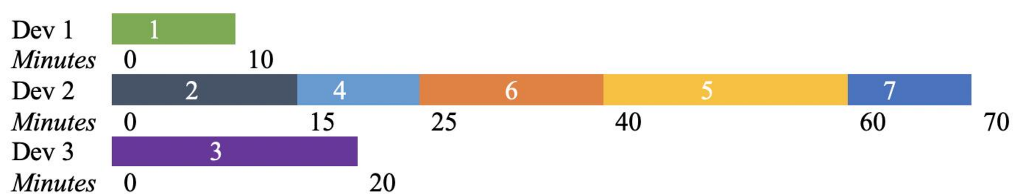

Рисунок 3.5 - Неперервні часові лінії для обмеженої кількості розробників

Наприклад, нехай A — це завдання, а d_1, d_2, d_3 - розробники.

$$|A| = 13 \qquad A_{end} = 45 \qquad A_{end} - |A| = 32$$

$$|d_1| = 38 \qquad |d_2| = 29 \qquad |d_3| = 60$$

Ми вибираємо d_1, він задовольняє умову ∣d1∣ ≥ A_{end} −∣A∣. Хоча ∣d3∣також задовольняє цю умову, ∣d1∣<∣d3∣, тому ми надаємо завдання d_1.

Візьмімо набір завдань на рисунку 3.2. Ми обмежимо кількість розробників трьома. Тепер у нас є масив довжини 3 для розробників та часи найранішого початку, прикріплені до кожного завдання. Потім ми вибираємо завдання з найранішим часом завершення та надаємо його розробнику 1. В цьому випадку це завдання 1. Ми збільшуємо час, проведений на роботі для розробника 1, поміщаємо завдання 1 у мапу виконаних та видаляємо його з мапи невиконаних. Завдання 2 тепер має найраніший час завершення, ми надаємо його розробнику 2 та вносимо необхідні зміни в змінні, і процес повторюється. Цікава частина цього випадку настає, коли ми дійдемо до завдання 5. Якщо ми розглянемо рисунок 3.5, ми можемо побачити, що стан процесу буде наступним, коли ми виберемо завдання 5. Розробники 1, 2 та 3 матимуть 10, 40 та 20 хвилин відповідно. Завдання 5 має найраніший час початку 25 хвилин. Згідно з нашим алгоритмом, ми вибираємо розробника з найменшою кількістю часу, проведеного на роботі, таким чином, що час, проведений на роботі, більший або дорівнює найранішому часу початку завдання. В цьому випадку розробник 2 відповідає цьому опису, і ми продовжуємо, як очікувалося. Ми можемо побачити на рисунку 3.5, що найдовша часова лінія прикріплена до розробника 2, а загальний час, необхідний для завершення всіх завдань з 3 розробниками, становить 70 хвилин. Ми назвемо цю стратегію неперервною стратегією, оскільки вона може створювати часові лінії без прогалин. Очевидно, що якщо завдання 5 було б надано розробнику 3, залишивши прогалину в часовій лінії розробника 3, ми могли б створити менший загальний час.

Є досить простий трюк до критерію вибору розробника, який допоможе впоратися з випадками такого роду. Знову нехай A — це завдання, а d_i — розробник. Ми надаємо A розробнику d_i таким чином, що $|d_i| = A_{end} - |A|$ і якщо такого d_i не існує, то ми вибираємо d_i таким чином, що не існує d_k таким чином, що $||d_k| - (A_{end} - |A|)| < ||d_i| - (A_{end} - |A|)|$.

Наприклад, ми знову розглядаємо завдання A та розробників d_1, d_2, d_3.

$$|A| = 13 \qquad A_{end} = 45 \qquad A_{end} - |A| = 32$$

$$|d_1| = 38 \qquad |d_2| = 29 \qquad |d_3| = 60$$

В цьому випадку ми вибираємо d_2, оскільки

$$||d_2| - (A_{end} - |A|)| < ||d_3| - (A_{end} - |A|)|$$

$$|29 - 32| < |38 - 32|$$

$$3 < 6$$

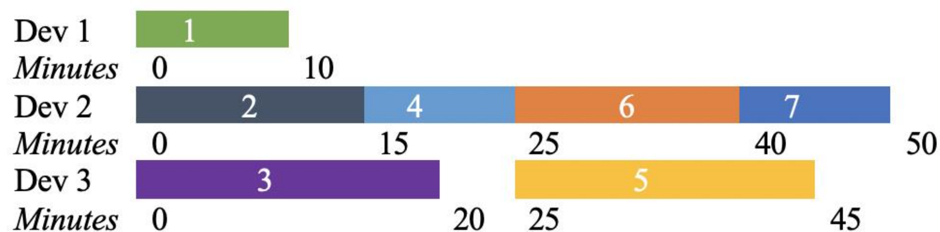

Рисунок 3.6 - Неперервні часові лінії для обмеженої кількості розробників

Коли ми враховуємо цю зміну стратегії, ми отримуємо кращий результат від нашого алгоритму, як показано на рисунку 3.6. Звернимо увагу, що завдання 5 було переміщено до розробника 3, що дозволяє завершити завдання 7 на 20 хвилин раніше. Ми назвемо цю стратегію неперервною стратегією. Хоча ця зміна критерію вибору покращила результати в цьому наборі завдань, ми побачимо, що існують випадки, коли неперервний метод дає кращий результат, ніж неперервна стратегія, і навпаки. Ці випадки будуть розглянуті пізніше.

3.5. Реалізація фронтенд частини та формату виведення

На початковому етапі розробки програмного забезпечення передбачалося створення веб-орієнтованого застосунку з графічним інтерфейсом користувача (GUI), призначеного для взаємодії з менеджерами проєктів. Функціонал серверної частини було реалізовано на базі фреймворку Express.js з тимчасовим розгортанням у локальному середовищі Node.js. Планувався подальший перенос застосунку на спеціалізований веб-сервер для забезпечення постійної доступності.

Під час підготовки до розгортання на цільовому веб-сервері та інтеграції серверної частини з запланованим фронтендом, що мав працювати в браузері, були виявлені технічні обмеження цільової веб-платформи стосовно прямої підтримки певних можливостей середовища Node.js, інтенсивно використаних при розробці серверної логіки. Ключовою проблемою стала несумісність стандартного для Node.js механізму імпорту модулів require (специфікація CommonJS) з середовищем виконання у клієнтських браузерах або певних конфігураціях веб-серверів, що вимагало б додаткової та складної конфігурації процесів транспіляції та бандлінгу коду.

Виходячи з виявлених обмежень та з метою забезпечення працездатності ключового функціоналу, було прийнято рішення змінити стратегію реалізації інтерфейсу користувача. Розроблено простий інтерфейс командного рядка (CLI), що надає можливість менеджеру проєктів інтерактивно вказувати ідентифікатори спринтів для аналізу та обирати тип необхідних обчислень (наприклад, візуалізація залежностей, планування на ресурсах).

Відображення виявлених ланцюжків залежностей здійснюється шляхом виведення в термінал текстового представлення. Для кожного повного ланцюжка (від вершини-джерела до вершини-стоку) спочатку відображається сумарна оцінка його тривалості (сума ваг вершин у ланцюжку), а потім –

послідовність ідентифікаторів завдань у цьому ланцюжку, розділених символом '→', що ілюструє напрямок залежностей.

Формат виведення результатів планування завдань на паралельних часових лініях (ресурсах), таких як розробники, також є текстовим, але структурований для наочності розподілу. Для кожного ресурсу виводиться окремий блок інформації. Перший рядок блоку містить ідентифікатор ресурсу та сумарну тривалість виконання завдань, призначених цьому ресурсу. Ця сумарна тривалість визначається часом завершення останнього завдання на даній часовій лінії. Наступний рядок ілюструє послідовність виконання завдань на цій часовій лінії. Кожне завдання в межах часової лінії представлене у форматі [час_початку] ІДЕНТИФІКАТОР_ЗАВДАННЯ [час_завершення], розташовані послідовно у часі. Хоча візуальне відображення періодів простою (gap/idle time) між завданнями не є явним графічним елементом, їх можна ідентифікувати шляхом аналізу часових міток: період простою між завданням A (завершується о fA) та завданням B (починається о sB) на одній часовій лінії відповідає інтервалу [fA,sB).

Потенційним напрямком для подальшого розвитку програмного забезпечення є реалізація повноцінного графічного інтерфейсу користувача. Оптимальним варіантом візуалізації результатів було б генерація статичних або інтерактивних графічних зображень (наприклад, у форматі SVG або PNG/JPEG), що наочно представляють структури ланцюжків залежностей (можливо, у вигляді графа) та розподіл завдань на часових лініях ресурсів (наприклад, у вигляді діаграми Ганта).

3.6. Експериментальна оцінка ефективності алгоритмів планування

Для оцінки ефективності, порівняння характеристик та виявлення сильних і слабких сторін розроблених алгоритмів планування було

проведено їх тестування на різних наборах вхідних даних із варіюванням параметрів, зокрема кількості доступних паралельних ресурсів. Тестові дані були змодельовані в системі управління проєктами Jira у межах проєкту з умовним позначенням "Dependency Mapper", де кожне завдання має унікальний ідентифікатор з префіксом "DM-". Для спрощення представлення результатів у подальшому тексті цей префікс в ідентифікаторах завдань буде опущено.

Аналіз результатів розпочнемо з найпростішого сценарію: набору завдань, що не мають залежностей між собою.

На рисунку 3.7 представлено структуру тестового набору, що складається з п'яти завдань без будь-яких залежностей між ними. Ваги (оцінки часу виконання) для цих завдань різняться, як це типово для реальних проєктів.

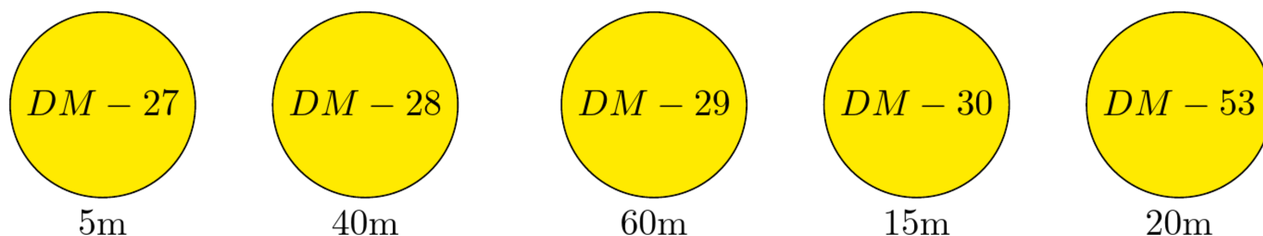

Рисунок 3.7 - Набір завдань без залежностей

Тестування алгоритмів планування на цьому наборі даних проводилося з використанням фіксованої кількості паралельних ресурсів – 3 розробники. Були виконані розрахунки для обох розроблених алгоритмів (надалі умовно називатимемо їх алгоритм А та алгоритм Б) з 3 розробниками, та проаналізовано отримані плани виконання.

На рисунках 3.8 та 3.9 показано результати планування, отримані обома алгоритмами.

```
Total time: 45
 | 0--- Task: DM-27 ---5 | 5--- Task: DM-28 ---45
Total time: 75
 | 0--- Task: DM-30 ---15 | 15--- Task: DM-29 ---75
Total time: 20
 | 0--- Task: DM-53 ---20
```

Рисунок 3.8 - Результат роботи алгоритму А для набору без залежностей, 3 розробники

```
Total time: 45
 | 0--- Task: DM-27 ---5 | 5--- Task: DM-28 ---45
Total time: 75
 | 0--- Task: DM-30 ---15 | 15--- Task: DM-29 ---75
Total time: 20
 | 0--- Task: DM-53 ---20
```

Рисунок 3.9 - Результат роботи алгоритму Б для набору без залежностей, 3 розробники

Аналіз результатів, представлених на рисунках 3.8 та 3.9, показує, що обидва алгоритми продемонстрували ідентичні часові лінії для даного набору завдань. Цей результат очікуваний, оскільки відсутність залежностей унеможливлює потребу у відкладанні виконання завдань через очікування завершення інших, що могло б спричинити відмінності у плануванні між алгоритмами.

Проте, очевидно, що отримане планування не є оптимальним з точки зору загальної тривалості виконання (makespan). Оптимальне розв'язання для задачі планування на паралельних машинах без залежностей, що мінімізує makespan, досягається шляхом ефективного пакування завдань, наприклад, розміщенням завдань з найбільшою тривалістю першими на різних ресурсах або розподілом коротших завдань для максимально щільного заповнення ресурсів. Згідно з інтерпретацією даних з рисунка 3.8 як тривалості завдань:

DM-27(5), DM-28(40), DM-30(15), DM-29(60), DM-53(20)), оптимальне планування на 3 розробниках з мінімальною тривалістю (makespan T_опт) складає 60 одиниць часу, наприклад:

Розробник 1: DM-29 (60);

Розробник 2: DM-28 (40) + DM-27 (5) = 45;

Розробник 3: DM-53 (20) + DM-30 (15) = 35.

Максимальна тривалість серед ресурсів становить 60.

Тривалість планування, отримана обома алгоритмами (T_алг), складає 75 одиниць часу (максимальна тривалість серед ресурсів на рисунку 3.8). Відхилення результатів алгоритмів від оптимального значення може бути кількісно оцінене за формулою відносної похибки:

$$\frac{Continuous\ Time - Optimal\ Time}{Optimal\ Time} = \frac{75 - 60}{60} = 0.25$$

Таким чином, для цього набору завдань отримана відносна похибка становить 25%. Цей результат свідчить, що алгоритми не є оптимізованими для найпростішого випадку планування без залежностей. Однак важливо підкреслити, що їхнє основне призначення полягає у розв'язанні складніших задач планування, що включають обробку великої кількості завдань зі складними перетинаючимися ланцюжками залежностей, де прості жадібні евристики можуть бути неефективними.

Далі розглянемо аналіз результатів тестування на наборі завдань, структура залежностей яких формує дерево. Дані, що відповідають структурі дерева залежностей (представлені на рисунку 3.10), були внесені до системи Jira. Були виконані розрахунки для обох алгоритмів планування з 3 розробниками.

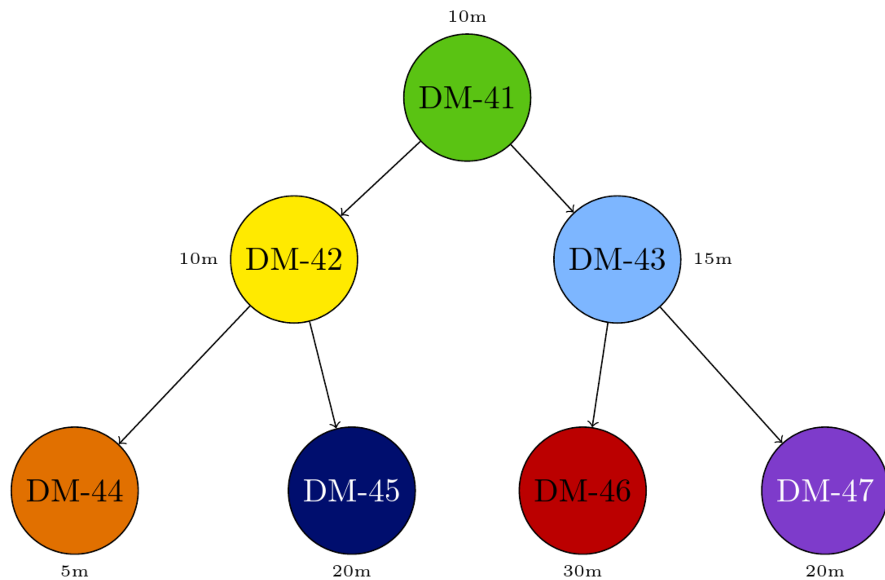

Рисунок 3.10 - Набір завдань зі структурою залежностей типу "дерево"

Результати роботи одного з алгоритмів (наприклад, Алгоритму А, або "неперервного" згідно з оригінальною термінологією) з 3 розробниками представлені на рисунку 3.11.

```
Total time: 110
 | 0--- Task: DM-41 ---10 | 10--- Task: DM-42 ---20 | 20--- Task: DM-43 ---35 | 35--- Task: DM-44 ---40 | 40--- Task: DM-45 ---60 | 60--- Task: DM-47 ---80 | 80--- Task: DM-46 ---110
Total time: 0

Total time: 0
```

Рисунок 3.11 - Результат роботи Алгоритму А для набору зі структурою "дерево", 3 розробники

```
Total time: 80
 | 0--- Task: DM-41 ---10 | 10--- Task: DM-42 ---20 | 20--- Task: DM-43 ---35 | 35--- Task: DM-44 ---40 | 40--- Task: DM-45 ---60 | 60--- Task: DM-47 ---80|
Total time: 65
 | 35--- Task: DM-46 ---65|
Total time: 0
|
```

Рисунок 3.12 - Результат роботи Алгоритму Б для набору зі структурою "дерево", 3 розробники

З рисунка 3.12 видно, що тривалості виконання завдань для ресурсів становлять 80, 65 та 0 хвилин. Попри те, що цей результат є кращим за

результат, отриманий неперервним алгоритмом, він все ще свідчить про субоптимальність планування, побудованого переривчастим алгоритмом. Оптимальне планування для цих даних, отримане ручним аналізом, є наступним:

$$\frac{Discontinuous - Optimal}{Optimal} = \frac{80 - 55}{55} \approx 0.4545$$

Отримана приблизна похибка складає 45%. Слід також відзначити, що переривчастий алгоритм недостатньо ефективно обробляє ситуації, які призводять до значних проміжків (періодів простою) між виконанням завдань на ресурсах.

Перейдемо до розгляду випадку зі складнішою, більш взаємопов'язаною структурою залежностей, що нагадує конфігурації, типові для реальних промислових проєктів.

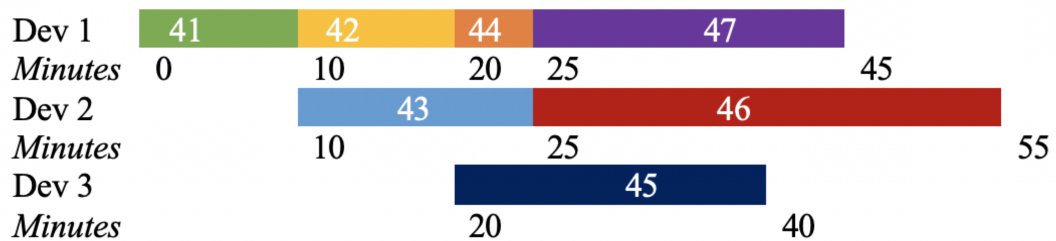

Рисунок 3.13 - Часові лінії для дерева залежностей

Аналогічно, завдання, представлені на рисунку 3.14, були внесені до системи Jira, і обидва алгоритми були запущені для обробки цих даних. На першому етапі тестування алгоритми було запущено для сценарію з 2 паралельними ресурсами.

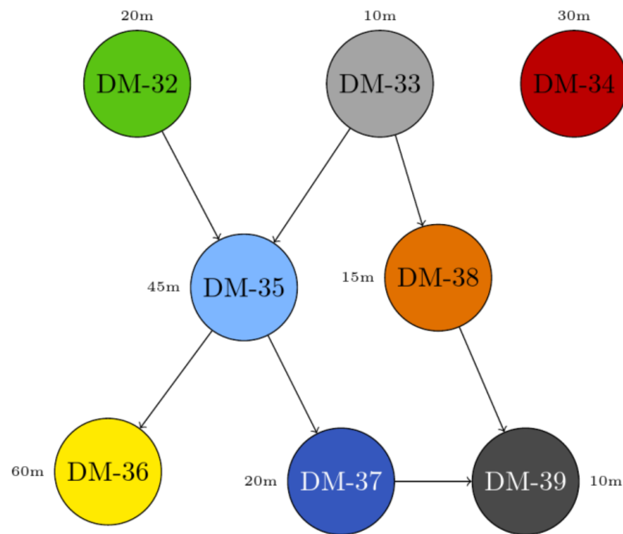

Рисунок 3.14 - Перетинаючі залежності

```
Total time: 160
 | 0--- Task: DM-33 ---10 | 10--- Task: DM-38 ---25 | 25--- Task: DM-35 ---70 | 70--- Task: DM-37 ---90 | 90--- Task: DM-39 ---100 | 100--- Task: DM-36 ---160
Total time: 50
 | 0--- Task: DM-32 ---20 | 20--- Task: DM-34 ---50
```

Рисунок 3.15 - Неперервний алгоритм, 2 розробники

З рисунка 3.15 видно, що тривалості планування для 2 розробників складають 160 та 50 хвилин. Знову ж таки, переривчастий алгоритм продемонстрував кращі результати порівняно з неперервним алгоритмом, забезпечивши тривалості 100 та 130 хвилин (рисунок 3.16). Слід відзначити, що хоча загальна сума тривалостей на всіх ресурсах (сума makespan) зросла на 20 хвилин (230 проти 210), максимальна тривалість виконання завдань на одному ресурсі (makespan) зменшилася на 30 хвилин (з 160 до 130).

```
Total time: 100
 | 0--- Task: DM-33 ---10 | 10--- Task: DM-38 ---25 | 25--- Task: DM-35 ---70 | 70--- Task: DM-37 ---90 | 90--- Task: DM-39 ---100|
Total time: 130
 | 0--- Task: DM-32 ---20 | 20--- Task: DM-34 ---50 | 70--- Task: DM-36 ---130|
```

Рисунок 3.16 - Переривчастий алгоритм, 2 розробники

```
Total time: 25
 | 0--- Task: DM-33 ---10 | 10--- Task: DM-38 ---25
Total time: 155
 | 0--- Task: DM-32 ---20 | 20--- Task: DM-35 ---65 | 65--- Task: DM-37 ---85 | 85--- Task: DM-39 ---95 | 95--- Task: DM-36 ---155
Total time: 30
 | 0--- Task: DM-34 ---30
```

Рисунок 3.17 - Неперервний алгоритм, 3 розробники

```
Total time: 25
 | 0--- Task: DM-33 ---10 | 10--- Task: DM-38 ---25|
Total time: 155
 | 0--- Task: DM-32 ---20 | 20--- Task: DM-35 ---65 | 65--- Task: DM-37 ---85 | 85--- Task: DM-39 ---95 | 95--- Task: DM-36 ---155|
Total time: 30
 | 0--- Task: DM-34 ---30|
```

Рисунок 3.18 - Переривчастий алгоритм, 3 розробники

Тестування було повторено зі збільшенням кількості паралельних розробників до 3. Результати роботи неперервного алгоритму з 3 розробниками представлені на рисунку 3.17, де тривалості планування становлять 25, 155 та 30 хвилин. Для переривчастого алгоритму отримані ідентичні тривалості (рисунок 3.18).

Цей випадок є особливо цікавим, оскільки збільшення кількості ресурсів призвело до погіршення результатів переривчастого алгоритму порівняно зі сценарієм з 2 розробниками (максимальна тривалість зросла зі 130 до 155 хвилин). Це узгоджується зі спостереженнями, отриманими під час тестування набору даних зі структурою "дерево". Виявлено, що переривчастий алгоритм не містить ефективного механізму для визначення та оптимізації використання ресурсів, що мають значні періоди простою або недозавантажені. Одним із можливих шляхів подолання цього недоліку може бути модифікація алгоритму, яка передбачатиме відкладення планування незалежних завдань та їх подальше призначення ресурсам з найменшим поточним або прогнозованим завантаженням.

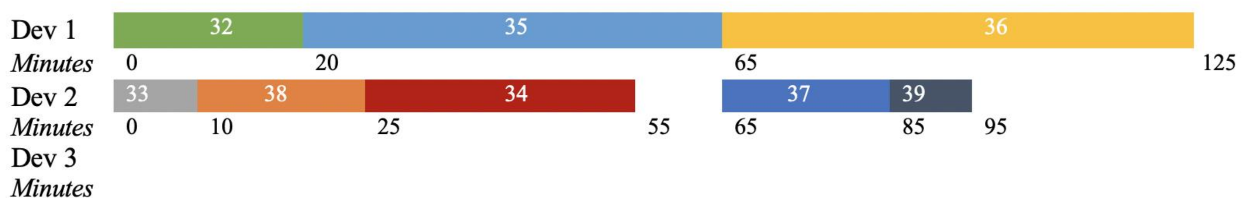

Рисунок 3.19 - Оптимальне планування для набору завдань з перетинаючими залежностями

Оптимальне планування для цього набору даних, отримане шляхом ручного аналізу, представлено на рисунку 3.19. Відповідні тривалості для ресурсів складають 125, 95 та 0 хвилин (максимальна тривалість становить 125 хвилин).

Слід відзначити, що завдання DM-34 є незалежним і, теоретично, могло б бути призначене розробнику 3 (з нульовим завантаженням).

Однак, з практичної точки зору (у реальному сценарії), розробник з нульовим завантаженням може бути більш ефективно використаний на іншому проєкті або звільнений. Тому в представленому оптимальному плануванні завдання DM-34 розміщено на часовій лінії між завданнями DM-38 та DM-37.

Для розрахунку похибки алгоритму буде використано результати планування, отримані переривчастим алгоритмом зі сценарієм з 2 розробниками (максимальна тривалість 130 хвилин), порівняно з цим оптимальним результатом (125 хвилин). Відносна похибка становить:

$$\frac{Discontinuous - Optimal}{Optimal} = \frac{130 - 125}{125} = 0.04$$

Таким чином, похибка для даного набору даних становить 4%. Це свідчить про те, що алгоритм демонструє значно кращу продуктивність у випадку наявності множинних перетинаючихся ланцюжків залежностей порівняно з простішими структурами.

Для подальшого аналізу ефективності на структурах з великою кількістю перетинаючихся залежностей розглянемо дані, представлені на рисунку 3.1 (Початкові залежності завдань), що є ще одним прикладом складної структури залежностей, типової для реального проєкту. Результати тестування для цього набору даних з 3 розробниками представлені, наприклад, для неперервного алгоритму на рисунку 3.20.

```
Total time: 1035
 | 0--- Task: DM-2 ---30 | 30--- Task: DM-11 ---180 | 180--- Task: DM-13 ---420 | 420--- Task: DM-5 ---570 | 570--- Task: DM-12 ---930 | 930--- Task: DM-4 ---960 | 960--- Task: DM-22 ---1035
Total time: 1185
 | 0--- Task: DM-9 ---60 | 60--- Task: DM-15 ---255 | 255--- Task: DM-10 ---465 | 465--- Task: DM-7 ---645 | 645--- Task: DM-18 ---765 | 765--- Task: DM-20 ---945 | 945--- Task: DM-19 ---1185
Total time: 1095
 | 0--- Task: DM-6 ---135 | 135--- Task: DM-3 ---315 | 315--- Task: DM-17 ---615 | 615--- Task: DM-21 ---675 | 675--- Task: DM-14 ---735 | 735--- Task: DM-8 ---1095
```

Рисунок 3.20 - Неперервний алгоритм, 3 розробники

Розглянемо далі результати тестування алгоритмів планування на початковому наборі даних із системи Jira, представленому на рисунку 3.1, оцінки часу виконання завдань для якого наведено у таблиці 3.1.

Розглянемо спочатку сценарій з 3 розробниками. З рисунка 3.20 видно, що тривалості планування для трьох розробників у випадку використання неперервного алгоритму складають 1035, 1185 та 1095 хвилин відповідно (максимальна тривалість - 1185 хвилин).

На рисунку 3.21 представлені тривалості виконання завдань для кожного розробника у випадку використання переривчастого алгоритму, які становлять 1320, 1020 та 1005 хвилин відповідно (максимальна тривалість - 1320 хвилин).

```
Total time: 1320
 | 0--- Task: DM-2 ---30 | 30--- Task: DM-11 ---180 | 180--- Task: DM-13 ---420 | 420--- Task: DM-5 ---570 | 570--- Task: DM-12 ---930 | 930--- Task: DM-4 ---960 | 960--- Task: DM-8 ---1320
Total time: 1020
 | 0--- Task: DM-9 ---60 | 60--- Task: DM-15 ---255 | 255--- Task: DM-10 ---465 | 465--- Task: DM-7 ---645 | 645--- Task: DM-18 ---765 | 765--- Task: DM-20 ---945 | 945--- Task: DM-22 ---1020
Total time: 1005
 | 0--- Task: DM-6 ---135 | 135--- Task: DM-3 ---315 | 315--- Task: DM-17 ---615 | 615--- Task: DM-21 ---675 | 675--- Task: DM-14 ---735 | 765--- Task: DM-19 ---1005
```

Рисунок 3.21 - Переривчастий алгоритм, 3 розробники.

Отже, для сценарію з 3 розробниками неперервний алгоритм забезпечує результат, ближчий до оптимального розв'язання, маючи меншу максимальну тривалість планування (makespan). Порівнявши makespan для обох алгоритмів у цьому випадку, можна кількісно оцінити покращення, яке забезпечує неперервний алгоритм порівняно з переривчастим:

$$\frac{Discontinuous - Continuous}{Continuous} = \frac{1320 - 1185}{1185} \approx 0.1139$$

Таким чином, у даному випадку неперервний алгоритм демонструє покращення на близько 11% порівняно з переривчастим алгоритмом за критерієм мінімізації makespan.

Аналіз планування, отриманого неперервним алгоритмом з 3 розробниками, виявляє наявність проміжку (періоду простою) на часовій лінії розробника 3 між завданнями DM-14 (завершується о 735 хв) та DM-19 (починається о 765 хв). Цей проміжок впливає на загальний розподіл завдань: завдання DM-8 призначається розробнику 1, а завдання DM-22 – розробнику 3. Такий розподіл подовжує загальну тривалість виконання, навіть якщо на часовій лінії з максимальною тривалістю відсутні періоди простою.

Далі розглянемо, як зміняться ці результати при збільшенні кількості розробників до 4. Тривалості планування для неперервного алгоритму з 4 розробниками становлять 1005, 390, 930 та 990 хвилин відповідно (рисунок 3.22).

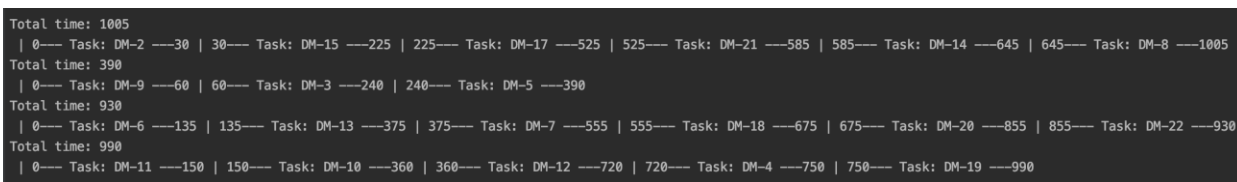
```
Total time: 1005
 | 0--- Task: DM-2 ---30 | 30--- Task: DM-15 ---225 | 225--- Task: DM-17 ---525 | 525--- Task: DM-21 ---585 | 585--- Task: DM-14 ---645 | 645--- Task: DM-8 ---1005
Total time: 390
 | 0--- Task: DM-9 ---60 | 60--- Task: DM-3 ---240 | 240--- Task: DM-5 ---390
Total time: 930
 | 0--- Task: DM-6 ---135 | 135--- Task: DM-13 ---375 | 375--- Task: DM-7 ---555 | 555--- Task: DM-18 ---675 | 675--- Task: DM-20 ---855 | 855--- Task: DM-22 ---930
Total time: 990
 | 0--- Task: DM-11 ---150 | 150--- Task: DM-10 ---360 | 360--- Task: DM-12 ---720 | 720--- Task: DM-4 ---750 | 750--- Task: DM-19 ---990
```

Рисунок 3.22 - Неперервний алгоритм, 4 розробники

Слід відзначити, що це зменшення максимальної тривалості (з 1185 до 1005 хвилин) не є значним порівняно з результатом для 3 розробників. Це пояснюється впливом довжини найдовшого ланцюжка залежностей у графі, який встановлює теоретичну нижню межу для makespan. Теоретично, при необмеженій кількості розробників мінімальна тривалість виконання буде визначатися саме довжиною найдовшого критичного шляху (ланцюжка залежностей) у графі.

Знову проведемо порівняння з результатами роботи переривчастого алгоритму (рисунок 3.23).

```
Total time: 915
 | 0--- Task: DM-2 ---30 | 30--- Task: DM-15 ---225 | 225--- Task: DM-17 ---525 | 525--- Task: DM-21 ---585 | 585--- Task: DM-14 ---645 | 675--- Task: DM-19 ---915
Total time: 390
 | 0--- Task: DM-9 ---60 | 60--- Task: DM-3 ---240 | 240--- Task: DM-5 ---390
Total time: 930
 | 0--- Task: DM-6 ---135 | 135--- Task: DM-13 ---375 | 375--- Task: DM-7 ---555 | 555--- Task: DM-18 ---675 | 675--- Task: DM-20 ---855 | 855--- Task: DM-22 ---930
Total time: 1110
 | 0--- Task: DM-11 ---150 | 150--- Task: DM-10 ---360 | 360--- Task: DM-12 ---720 | 720--- Task: DM-4 ---750 | 750--- Task: DM-8 ---1110
```

Рисунок 3.23 - Переривчастий алгоритм, 4 розробники

Отримані тривалості для переривчастого алгоритму з 4 розробниками становлять 915, 390, 930 та 1110 хвилин відповідно (максимальна тривалість - 1110 хвилин). Як і у випадку з 3 розробниками, неперервний алгоритм забезпечує меншу максимальну тривалість (1005 хвилин) порівняно з переривчастим алгоритмом (1110 хвилин), тобто знову демонструє кращий результат за критерієм makespan.

$$\frac{Discontinuous - Continuous}{Continuous} = \frac{1110 - 1005}{1005} \approx 0.1044$$

Слід відзначити, що результат неперервного алгоритму з 3 розробниками (makespan 1185 хвилин) лише на 75 хвилин гірший за результат переривчастого алгоритму з 4 розробниками (makespan 1110 хвилин).

Відзначається покращення результатів неперервного алгоритму порівняно з переривчастим і для 4 розробників, хоча вплив додавання розробників є обмеженим через довжину критичного шляху, що домінує у визначенні загальної тривалості.

Проте, шляхом ручного аналізу встановлено, що жоден з отриманих результатів для 4 розробників не є оптимальним. Оптимальне планування, отримане ручним розрахунком, представлене нижче. Для зручності читання префікс "DM-" в ідентифікаторах завдань опущено.

$$\text{Developer 1: } [9, 15, 10, 12, 4] \rightarrow 855 \text{ minutes}$$

$$\text{Developer 2: } [2, 13, 7, 14, 20, 22] \rightarrow 765 \text{ minutes}$$

$$\text{Developer 3: } [6, 3, 5, 8] \rightarrow 870 \text{ minutes}$$

$$\text{Developer 4: } [11, 17, 21, 18, 19] \rightarrow 870 \text{ minutes}$$

Слід відзначити, що у представленому оптимальному плануванні на часовій лінії розробника 3 є проміжок з 465-ї по 510-у хвилину, зумовлений необхідністю дотримання залежності завдання DM-8 від завдання DM-14.

Похибка (результату неперервного алгоритму) порівняно з оптимальним плануванням розраховується наступним чином:

$$\frac{Continuous - Optimal}{Optimal} = \frac{1005 - 870}{870} \approx 0.1551$$

Таким чином, для цього набору даних спостерігається приблизно 16% похибка (результату неперервного алгоритму) порівняно з оптимальним плануванням.

Цікавим є той факт, що, хоча неперервний алгоритм забезпечує кращий результат (менший makespan) порівняно з переривчастим алгоритмом на цьому наборі даних, сама структура оптимального планування (рисунок 3.19) містить періоди простою, що типово для планування з можливістю переривання або гнучким розподілом завдань тобто, структура відповідає можливостям переривчастого планування.

Отже, в рамках даної роботи було розроблено програмне забезпечення, призначене для підтримки прийняття рішень менеджерами проєктів шляхом розрахунку можливих планів виконання завдань у межах спринту на основі даних системи управління проєктами Jira. Програмний комплекс реалізовано мовою JavaScript на базі серверного середовища Node.js з використанням фреймворку Express.js.

Архітектура програмного забезпечення має багатошарову структуру, що забезпечує модульність та розподілення відповідальності:

- Шар отримання та парсингу даних: Відповідає за взаємодію з Jira API, отримання "сирих" даних про завдання та їх трансформацію у структурований формат.

- Шар побудови графової моделі та ланцюжків залежностей: На основі оброблених даних формує графове представлення завдань та виявляє повні ланцюжки залежностей.

- Шар планування (необмежені ресурси): Реалізує алгоритм планування за умови гіпотетичної необмеженої кількості виконавців (розробників).

- Шар планування (обмежені ресурси): Містить алгоритми планування для реалістичної, обмеженої кількості виконавців.

Аналіз окремих компонентів програмного комплексу виявив наступне:

1. Отримання та парсинг даних. Процес взаємодії з Jira API може бути неефективним через значний обсяг інформації, що передається при кожному запиті. Оптимізація цього етапу може бути досягнута шляхом імплементації механізму кешування даних та валідації кешу (наприклад, перевірки дати останньої модифікації спринту) перед виконанням повного запиту до API, що дозволить зменшити латентність та навантаження на сервіс.

2. Побудова ланцюжків залежностей. Після успішного парсингу даних та трансформації їх у графове представлення (вузли), етап побудови ланцюжків залежностей є відносно невитратним за часом, враховуючи типові розміри графів завдань у межах одного спринту.

3. Планування (необмежені ресурси). Побудова планування для необмеженої кількості виконавців після визначення ланцюжків залежностей є обчислювально простою (тривіальною) задачею, що дозволяє швидко визначити теоретичний мінімальний час виконання спринту, обмежений лише найдовшим критичним шляхом.

4. Планування (обмежені ресурси). Шар планування для обмеженої кількості виконавців виявився найбільш критичним і таким, що потребує суттєвого вдосконалення.

Результати проведеного тестування двох реалізованих алгоритмів (неперервного та переривчастого) на модельованих даних показали, що їхня ефективність варіюється залежно від структури вхідних даних. Кожен з алгоритмів може демонструвати кращі результати (з точки зору мінімізації makespan) у певних сценаріях залежностей. Проте, жоден із розроблених алгоритмів у загальному випадку не гарантує знаходження глобально оптимального розв'язання. Спостерігається похибка отриманих рішень порівняно з оптимальним плануванням, яка, за результатами тестування на різних наборах даних, варіюється в діапазоні від 4% до 45%.

Незважаючи на наявність значної похибки в окремих сценаріях, програмне забезпечення може надавати менеджерам проєктів корисну початкову евристику для побудови плану виконання. Виведення алгоритму може слугувати основою для ручного корегування планування, зокрема шляхом оптимізації розміщення незалежних завдань у випадках великої похибки. Окрім надання початкового плану, програмне забезпечення дозволяє проводити аналіз "що-якщо" (what-if analysis), багаторазово запускаючи розрахунки для одного набору завдань із різною кількістю виконавців. Це дає змогу швидко оцінити вплив чисельності команди на тривалість спринту та оптимізувати розподіл завдань для покращення завантаження виконавців.

ВИСНОВКИ

В дипломній роботі було досліджено проблематику планування та управління процесами розробки програмних проєктів в умовах наявності міжзавданню та внутрішньокомандних залежностей. Проведений аналіз предметної області підтвердив значний вплив залежностей на терміни та ефективність виконання проєктів, а також виявив певні обмеження існуючих інструментів управління проєктами щодо їхнього комплексного аналізу та оптимізації планування.

Визначено, що теорія графів є ефективним математичним апаратом для моделювання складних структур залежностей між завданнями. В роботі формалізовано представлення мап залежностей у вигляді орієнтованих зважених графів. Розглянуто теоретичні основи планування, включаючи використання топологічного впорядкування як інструменту врахування залежностей, а також проаналізовано класичну задачу про планування інтервалів як базову модель для планування на ресурсах.

В рамках практичної частини роботи було розроблено та імплементовано програмний комплекс для підтримки процесу планування, який включає етапи отримання та обробки даних із системи Jira, їх трансформації у графову модель та виявлення повних ланцюжків залежностей. Реалізовано два алгоритми планування виконання завдань на обмеженій кількості паралельних виконавців (розробників), що базуються на різних евристичних підходах (надалі умовно "неперервний" та "переривчастий").

Проведена експериментальна оцінка ефективності розроблених алгоритмів на різних наборах тестових даних, що імітують типові структури залежностей, показала, що:

- ефективність алгоритмів варіюється залежно від структури вхідних даних.

- жоден із реалізованих алгоритмів не гарантує знаходження глобально оптимального розв'язання у загальному випадку.

- спостерігається похибка отриманих планів порівняно з оптимальними, яка, за результатами тестування, може сягати значних значень.

- виявлено, що "переривчастий" алгоритм має недолік в ефективному використанні ресурсів при певних конфігураціях залежностей та їх ваг.

Незважаючи на субоптимальність, розроблений програмний комплекс може слугувати корисним інструментом для менеджерів проєктів, надаючи початкову евристику та можливість проведення аналізу "що-якщо" для оцінки впливу кількості виконавців на загальну тривалість виконання.

Подальший розвиток проєкту передбачає вдосконалення алгоритмів планування, можливо, шляхом розробки спеціалізованих підходів для різних класів структур залежностей, а також розробку веб-орієнтованого графічного інтерфейсу для наочної візуалізації та зручності використання.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Jira Review | Features, Pricing, Pros & Cons - https://project-management.com/jira-software-review/
2. Dashboard setup – Scoro Help Center - https://support.scoro.com/hc/en-us/articles/12379977388173-Dashboard-setup
3. What is Basecamp? How It Works, Features, and Pros & Cons - Ahsuite Blog - https://blog.ahsuite.com/what-is-basecamp/
4. Cataldo, M., Mockus, A., Roberts, J. A., & Herbsleb, J. D. (Дата публікації, якщо відома). Software Dependencies, Work Dependencies, and Their Impact on Failures. ResearchGate.
5. Javan Jafari Bojnordi. (S2024). Dependency Management Practices for the npm Software Ecosystem. Spectrum: Concordia University Research Repository. https://spectrum.library.concordia.ca/993232/1/Javan%20Jafari%20Bojnordi_PhD_S2024.pdf
6. Managing and utilizing dependencies between components in component-based systems. Lund University. Отримано з https://fileadmin.cs.lth.se/cs/Personal/Lars_Bendix/Teaching/Lund/Theses/AL17/AL17.pdf
7. Using Graph Theory for Project Management. Rememo Blog. Отримано з https://rememo.io/blog/graph-theory-in-project-management
8. F. Pferschy, R. Stanislawska. 2009. (PDF) Graph models for scheduling systems with machine saturation property. ResearchGate. Отримано з https://www.researchgate.net/publication/227313408_Graph_models_for_scheduling_systems_with_machine_saturation_property
9. Project Scheduling Problem for Software Development Library - PSPSWDLIB. CORE. Отримано з https://core.ac.uk/download/pdf/61419872.pdf

10. Implementing the Program Evaluation and Review Technique (PERT) in Software Projects. DZone. Отримано з https://dev.to/teamcamp/implementing-the-program-evaluation-and-review-technique-pert-in-software-projects-536k
11. G. Malcolm J. H. Roseboom C. E. Clark W. Fazar. \Application of a Technique for Research and Development Program Evaluation". In: Operations Research (1959). doi: https://doi.org/10.1287/opre.7.5.646.
12. How to Use PERT to Find the Critical Path in Projects. ClickUp Blog. Отримано з https://clickup.com/blog/pert-critical-path/
13. Project Evaluation and Review Technique (PERT). 2024 - GeeksforGeeks. Отримано з https://www.geeksforgeeks.org/project-evaluation-and-review-technique-pert/
14. PERT and CPM in Project Management with Practical Examples. Scientific Research Publishing (SCIRP). 2022 - https://www.scirp.org/journal/paperinformation?paperid=110980
15. Dependency Visualization Tool for Decision Support Systems with Preferential Dependencies. ResearchGate. https://www.researchgate.net/publication/365840173_Dependency_Visualization_Tool_for_Decision_Support_Systems_with_Preferential_Dependencies
16. Leiseron Cormen and Rivest. Introduction to Algorithms. Cambridge, Massachusetts London, England: The MIT Press, 2009.
17. Nora Harts eld and Gerhard Ringel. Pearls in Graph Theory. Mineola, New York: Dover Publications Inc., 2003. isbn: 0486432327.
18. Project Dependencies [Types & Strategies]. Atlassian Agile Coach. https://www.atlassian.com/ru/agile/project-management/project-management-dependencies
19. How dependency mapping can lead to better project outcomes. Nulab Blog. Отримано з https://nulab.com/learn/project-management/how-dependency-mapping-can-lead-to-better-project-outcomes/

20. Project Management and Visualization Techniques A Details Study. International Research Journal of Engineering and Science (IRJES). Отримано з https://www.irjes.com/Papers/vol13-issue5/13052844.pdf
21. Building Data Pipelines With Jira API. DZone. Отримано з https://dzone.com/articles/building-data-pipelines-with-jira-api
22. Is there a possibility to analyze the Jira issue using REST API? Atlassian Community. https://community.atlassian.com/forums/Jira-Service-Management/Is-there-a-possibility-to-analyze-the-Jira-issue-using-REST-API/qaq-p/2779105
23. Atlassian Developer. (2024). Jira REST API examples. Atlassian Developer Documentation. https://developer.atlassian.com/server/jira/platform/jira-rest-api-examples/
24. Atlassian Developer. (2023). The Jira Cloud platform REST API. Atlassian Developer Documentation. https://developer.atlassian.com/cloud/jira/platform/rest/v2/
25. Insights into Dependency Maintenance Trends in the Maven Ecosystem. arXiv preprint arXiv:2503.22902. https://arxiv.org/html/2503.22902v1
26. Generation of Construction Scheduling through Machine Learning and BIM: A Blueprint. MDPI. https://www.mdpi.com/2075-5309/14/4/934

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: “Програмне рішення для побудови мап залежностей процесу керування програмним проектом”

Обсяг пояснювальної записки: __78__ аркушів.

Дата закінчення роботи: _10 червня_ 2025 р.

Підпис студента _______________