**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ**

**КАФЕДРА КОМП'ЮТЕРНИХ НАУК**

# КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

**на тему:**

**«Інформаційна технологія адаптації евристичних алгоритмів оптимального пакування об'єктів в контейнери»**

**Завідувач**
**випускаючої кафедри** **Довбиш А.С.**

**Керівник роботи** **Шаповалов С.П.**

**Студент гр. ІН.м.-02** **Ковальов О.В.**

**Суми 2021**

*Сумський державний університет*

(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«122 - Комп'ютерні науки»*

Затверджую:

зав.кафедрою _____

_____

"_____" _____ 20___р.

# ЗАВДАННЯ
# НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

*Ковальову Олексію Віталійовичу*

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна технологія адаптації евристичних алгоритмів оптимального пакування об'єктів в контейнери*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проекту (роботи)_____

3. Вхідні данні до проекту (роботи) _____

_____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

*1) Інформаційний огляд проблеми та постановка задачі дослідження 2) Створення математичної моделі та проектування методу рішення; 3) Створення програмної реалізації алгоритму та аналіз результатів 4) Тестування додатку.*

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

_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання _____

Керівник _____

(підпис)

Завдання прийняв до виконання _____

(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № п/п | Назва етапів дипломного проекту (роботи) | Термін виконання проекту (роботи) | Примітка |
|---|---|---|---|
| 1. | Аналіз поставленої проблеми предметної області, постановка задачі | | |
| 2. | Аналітичний огляд існуючих алгоритмів | | |
| 3. | Розробка алгоритму та програмна реалізація | | |
| 4. | Аналіз та тестування отриманої системи | | |
| 5. | Оформлення пояснювальної записки до кваліфікаційної магістерської роботи | | |

Студент – дипломник _____

(підпис)

Керівник проекту _____

(підпис)

# РЕФЕРАТ

**Записка:** 68 стор., 25 рис., 23 формули, 1 додаток, 28 літературних джерел.

**Об’єкт дослідження –** евристичні алгоритми реалізації проблеми пакування об’єктів в контейнери.

**Предмет дослідження** – особливості роботи алгоритмів та їх методи імплементації для реальних задач.

**Мета роботи** — Створення додатку для демонстрації коректної роботи алгоритмів на математичній основі існуючих методів та алгоритмів.

**Методи дослідження –** створення математичної моделі та створення алгоритму LAFF

**Результати —** розроблено програмну реалізацію евристичного алгоритму, який розв’язує задачу трьох вимірного пакування об’єктів в контейнери.

**LAFF, NP-Hard, Bin packing problem, C++**

# ЗМІСТ

# ВСТУП

У сучасних технічних системах важливе місце займають завдання компонування та розміщення тривимірних об'єктів з відомими властивостями. Крім безпосередньо завдання упаковки товарів у контейнери(bin packing), такі питання виникають у медицині під час планування автоматизованого радіохірургічного лікування, у матеріалознавстві при вивченні та моделюванні структури молекул речовини, в логістиці при проектування мережі центрів обслуговування.

З точки зору продуктивності та алгоритмічної складності, проблему “Bin Packing Problem” відносять до категорії NP-Hard задач [1]. Зв’язана задача про перевірку того, що всі елементи можуть бути розташовані в відповідних контейнерах відноситься до NP-Complete категорії задач [1]. Не дивлячись на критичність найгіршого випадку, оптимальне рішення для дуже великих випадків можна отримати з використанням складних алгоритмів. Також існує велика кількість алгоритмів апроксимації. Наприклад алгоритм First-fit забезпечує швидке, але часто неоптимальне рішення, яке включає в себе розміщення кожного елементу в перший контейнер, в яку він може бути розташований [2]. Це потребує $O(n \log n)$ часу, де n – це кількість елементів, які потрібно упакувати. Алгоритми можна модернізувати, для більшої ефективності, відсортувавши елементи від більшого за об’ємом до меншого(алгоритм first-fit decreasing), хоча аналогічно відсутня гарантія оптимального рішення, а для великих об’ємів вхідних даних збільшується час роботи алгоритму.

Існує множина варіантів такої задачі, наприклад двовимірне пакування, лінійне пакування, пакування за вагою, пакування за ціною та ін. Проблему пакування контейнера можна трактувати як окремий випадок проблеми розкрою матеріалів. Коли кількість контейнерів обмежена до 1 і кожен елемент характеризується як об’ємом, так і ціною, проблема максимізації

цінності елементів, які можуть поміститися в контейнер, відома як проблема рюкзака.

Варіант пакування в контейнери, які зустрічаються в реальному світі, - це коли елементи можуть розділяти простір при пакуванні в контейнер. Сукупність елементів може займати менше простору при пакуванні, ніж сума об’ємів індивідуальних елементів. Цей випадок відомий як пакування віртуальних машин [2], бо коли віртуальні машини упаковуються на серверах, їх загальна потреба в пам’яті може зменшуватися за рахунок сторінок пам’яті, які спільно використовуються віртуальними машинами, які потрібно зберегти лише один раз. Якщо елементи можуть ділити простір довільним способом, проблему Bin packing дуже важко вирішити. Однак якщо спільне використання простору інтегрується в ієрархію, як в випадку з розділенням пам’яті в віртуальних машинах, проблема пакування комірок може бути ефективно апроксимована.

Іншим варіантом практичної реалізації Bin packing в реальному світі є “онлайн пакування”. В даному випадку передбачається, що елементи різних розмірів поступають послідовно і сторона, яка приймає рішення, повинна вирішувати процес обробки для кожного випадку в незалежності від інших елементів. Тобто ця керуюча сторона повинна вирішити, чи потрібно обрати и упакувати даний елемент або пропустити. Проте автономне пакування контейнерів дозволяє переставляти елементи для покращення метрик пакування після отримання нових елементів. Звісно, це потребує додаткового сховища для розміщення елементів, які потрібно розташувати.

# 1 ІНФОРМАЦІЙНИЙ ОГЛЯД ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

## 1.1 Задачі типу NP-hard. Асимптотична класифікація задач

Загальноприйнятою мінімальною вимогою для того, щоб алгоритм вважався «ефективним», є те, що час його виконання є поліноміальним: $O(n^c)$ для деякої константи $c$, де $n$ — розмір вхідних даних [3]. Дослідники рано визнали, що не всі проблеми можна вирішити швидко, але було важко зрозуміти, які з них можливо, а які ні. Існує кілька так званих NP-складних задач, які, як вважають більшість людей, неможливо вирішити за поліноміальний час, бо ніхто не може довести суперполіноміальну нижню межу. Виконання схеми є доцільним прикладом проблеми, яку ми не знаємо, як вирішити за поліноміальний час. У цій задачі вхідним є логічний контур: набір елементів І, АБО і НЕ, з'єднаних проводами. Ми будемо вважати, що в схемі немає петель (тому немає ліній затримки або тригерів). Вхідним сигналом для схеми є набір $m$ логічних (TRUE/FALSE) значень $x1,...,xm$. Вихідним є одне булеве значення. Враховуючи конкретні вхідні значення, ми можемо розрахувати вихід схеми за поліноміальний (фактично, лінійний) час, використовуючи пошук у глибину, оскільки ми можемо обчислити вихід $k$-вхідного вентиля за $O(k)$ час.

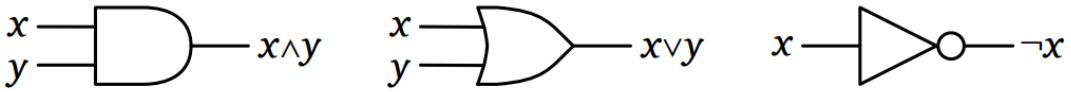



Рисунок 1.1 - Контур І, контур АБО, контур НЕ

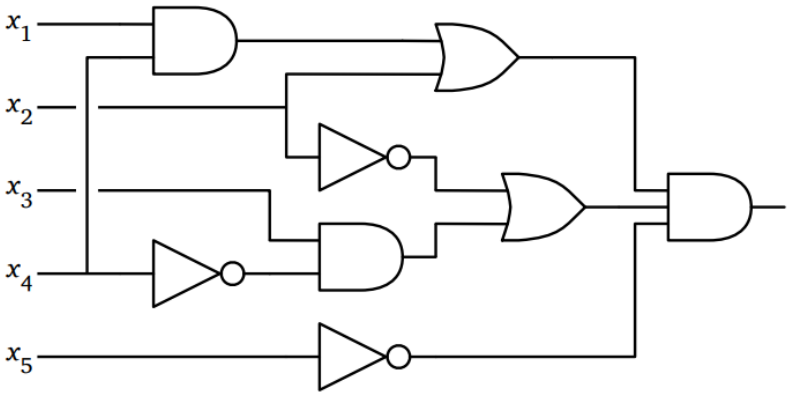



Рисунок 1.2 - Логічна схема. входи входять зліва, а вихід виходить справа

Проблема виконання ланцюга запитує, для даної схеми, чи є вхід, який робить вихід схеми TRUE, або навпаки, чи завжди схема виводить FALSE. Ніхто не знає, як вирішити цю проблему швидше, ніж просто спробувати всі 2 *m* можливих входів у схему, але для цього потрібен експоненційний час. З іншого боку, ніхто ніколи не довів, що це найкраще, що ми можемо зробити; можливо, є розумний алгоритм, якого ще ніхто не відкрив [3].

### *1.1.1 Задачі типу P, NP, і co-NP*

Проблема рішення — це проблема, вихідним результатом якої є одне булеве значення: ТАК або НІ [3]. Визначають три класи задач вирішення:

- P — це набір задач, які можна вирішити за поліноміальний час. Інтуїтивно, P — це набір задач, які можна швидко розв'язати.
- NP — це множина задач вирішення з такою властивістю: Якщо відповідь ТАК, то є доказ цього факту, який можна перевірити за поліноміальний час. Інтуїтивно, NP — це набір проблем вирішення, де ми можемо швидко перевірити відповідь ТАК, якщо перед нами є рішення.
- co-NP є протилежністю NP. Якщо відповідь на задачу в co-NP НІ, то є доказ цього факту, який можна перевірити за поліноміальний час.

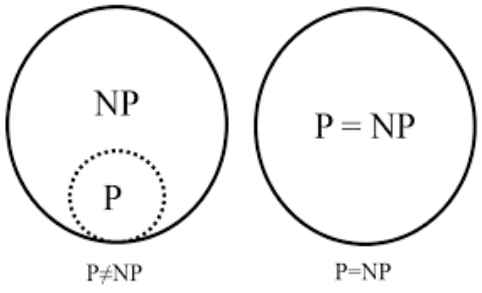



Рисунок 1.3 - Типи складності задач

Наприклад, проблема виконання схеми знаходиться в NP. Якщо відповідь ТАК, то будь-який набір з m вхідних значень, який дає TRUE вихід, є доказом цього факту; ми можемо перевірити доказ, оцінюючи схему за

поліноміальний час. Поширена думка, що ланцюг виконується не в P або в co-NP, але насправді ніхто не знає.

Кожна проблема вирішення в P також знаходиться в NP. Якщо проблема в P, ми можемо перевірити відповіді ТАК за поліноміальний час, перерахувавши відповідь з нуля. Аналогічно, будь-яка проблема в P також знаходиться в co-NP.

Мабуть, найважливіше відкрите питання теоретичної інформатики, якщо не всієї математики, полягає в тому, чи насправді класи складності P і NP відрізняються. Більшості людей здається очевидним, що P != NP; Домашні завдання та іспити в цьому та інших класах переконали у тому, що проблеми може бути неймовірно важко вирішити, навіть якщо рішення очевидні в ретроспективі. Видатний фізик Річард Фейнман, один із перших піонерів квантових обчислень, мабуть, не міг прийняти, що P != NP насправді є відкритим питанням. Але ніхто не знає, як це довести. Інститут математики Клея перераховує P проти NP як першу з семи проблем, що присуджуються премії тисячоліття, пропонуючи винагороду в розмірі 1 000 000 доларів за її вирішення [3].

Більш тонке, але все ще відкрите питання полягає в тому, чи відрізняються класи складності NP і co-NP. Навіть якщо ми зможемо швидко перевірити кожну відповідь ТАК, немає підстав вважати, що ми також зможемо швидко перевірити відповіді НІ. Наприклад, наскільки нам відомо, немає короткого доказу того, що булеву схему неможливо виконати. Вважається, що NP != co-NP, але ніхто не знає, як це довести.

Якби ми могли швидко вирішити одну конкретну NP-складну задачу, то ми могли б швидко вирішити будь-яку проблему, розв’язання якої легко зрозуміти, використовуючи рішення цієї спеціальної проблеми як підпрограму. NP-складні задачі принаймні так само складні, як і будь-яка проблема в NP.

Нарешті, проблема є NP-повною, якщо вона одночасно є NP-складною та елементом NP (або «NP-легкою»). NP повні проблеми є найскладнішими задачами в NP. Якщо хтось знайде алгоритм з поліноміальним часом навіть для однієї NP-повної задачі, то це означатиме алгоритм поліноміального часу для кожної NP-повної задачі. Було показано, що тисячі проблем є NP-повними, тому алгоритм поліноміального часу для однієї (і, отже, для всіх) з них здається неймовірно малоймовірним.

Не відразу зрозуміло, що будь-які проблеми рішення є NP-складними або NP-повними. NP-складність — це вже багато вимог до проблеми; наполягати на тому, що проблема також має недетермінований алгоритм поліноміального часу, здається майже абсолютно нерозумним. Наступна теорема була вперше опублікована Стівом Куком у 1971 році та незалежно Леонідом Левіним у 1973 році [4].

### *1.1.2 Формальне визначення*

Більш формально, задача *П* визначається як NP-складна тоді і тільки тоді, коли для кожної задачі *П'* в NP існує зведення Тьюринга за поліноміальний час від *П'* до *П* — скорочення Тьюринга просто означає скорочення, яке можна виконано на машині Тьюринга. Скорочення Тьюринга за поліноміальним часом також називають скороченнями оракула або скороченнями Кука. Це елементарно, але важко доводити, що будь-який алгоритм, який можна виконати на машині з випадковим доступом [5] за час T(n), можна змоделювати на однострічковій машині Тьюринга за час $O(T(n)2)$, тому скорочення Тьюринга за поліноміальним часом насправді не потрібно описувати за допомогою машин Тьюринга.

Дослідники теорії складності вважають за краще визначати NP-складність у термінах поліноміальних скорочень, які також називають скороченнями Карпа. Скорочення Карпа визначаються так: набори рядків над фіксованим алфавітом Σ. (Без втрати загальності ми можемо вважати, що $\Sigma = \{0, 1\}$.) Зведення з однієї мови $П' \subseteq \Sigma^*$ до іншої мови $П \subseteq \Sigma^*$ є функцією

f:Σ*→Σ* такою що x ∈ *П'* тоді і тільки тоді, коли f (x) ∈ *П*. Тоді ми могли б визначити мову П як NP-складну тоді і тільки тоді, коли для будь-якої мови П'∈ NP існує скорочення на кілька одиниць від П' до П, яке можна обчислити за поліноміальний час.

Кожне скорочення Карпа є скороченням Кука, але не навпаки; зокрема, будь-яке скорочення Карпа від *П* до *П'* еквівалентно перетворенню вхідних даних *П* у вхідні дані для *П'*, виклику оракула (тобто підпрограми) для *П'*, а потім дослівному поверненню відповіді. Однак, наскільки відомо, не кожне скорочення Кука можна моделювати за допомогою скорочення Карпа.

Теоретики складності віддають перевагу скороченням Карпа насамперед тому, що NP закрито за скороченнями Карпа, але не закрите за скороченнями Кука (якщо NP=co-NP, що вважається малоймовірним). Існують природні проблеми, які NP-складні щодо скорочень Кука, але NP-складні щодо скорочень Карпа, лише якщо P=NP. Як тривіальний приклад розглянемо проблему UNSAT: чи завжди вона хибна, якщо врахувати булеву формулу? З іншого боку, скорочення на кілька одиниць стосуються лише проблем прийняття рішень (або більш формально, до мов); формально жодна задача оптимізації чи побудови не є складною по Карпу-NP.

Щоб зробити речі ще більш заплутаними, і Кук, і Карп спочатку визначили «NP-твердість» у термінах скорочень логарифмічного простору. Кожне скорочення логарифмічного простору є скороченням поліноміального часу, але не навпаки. Залишається відкритим питання, чи змінює ослаблення набору дозволених (Кука чи Карпа) скорочень від логарифмічного простору до поліноміального часу набір NP-складних задач.

## 1.2 Bin packing problem. Аналітичний огляд існуючих алгоритмів

Проблема упаковки контейнерів (bin packing problem або BPP) — це NP-складна комбінаторна оптимізаційна задача, яка визначається як розміщення набору предметів різного розміру в контейнери таким чином, щоб кількість використовуваних контейнерів було оптимально мінімізовано [6]. Крім того, на практиці існують різні варіації проблеми залежно від розміру елементу контейнеру(надалі просто елемент), обмежень розміщення та пріоритету. Більше того, існує кілька важливих застосувань BPP в реальному світі, особливо в галузях транспортування, складування та управління шляхами поставок. Через практичну актуальність цієї проблеми дослідники постійно досліджують нові та вдосконалюють методики для оптимального вирішення проблеми. Евристики — це потужні алгоритми, які довели свою неймовірну здатність вирішувати складні та складні оптимізаційні задачі, включаючи кілька варіантів BPP. Однак не існує вичерпного огляду літератури щодо застосування евристичних підходів до вирішення проблем BPP. Тому, щоб заповнити цю прогалину, ця робота представляє огляд останніх досягнень, досягнутих для BPP, з особливим акцентом на евристичних алгоритмах.

Класична 1D BPP(1 dimensional bin packing problem) може бути спроектована як цілочисельна лінійна програма. Для початку визначемо змінну u, як верхню границю мінімальної кількості контейнерів, які потрібні для упакування всіх елементів або “ящиків”. Будемо вважати, що ці контейнери пронумеровані від 1 до u. Змінній n призначемо кількість елементів, які потрібно упакувати. Вводимо дві бінарні змінні

$$y_i = \begin{cases} 1 & \text{якщо контейнер } i \text{ входить в рішення} \\ 0 & \text{в іншому випадку} \end{cases} \quad (1 \le i \le u) \tag{1.1}$$

$$x_{ij} = \begin{cases} 1 & \text{якщо елемент } j \text{ упакованний в контейнері } i \\ 0 & \text{в іншому випадку} \end{cases} \quad (1 \le i \le u;\ 1 \le j \le n)$$

Мінімізувати

$$\sum_{i=1}^{u} y_i \tag{1.2}$$

Тобто

$$\sum_{j=1}^{n} w_j x_{ij} \le c y_i \quad (1 \le i \le u) \tag{1.3}$$

$$\sum_{i=1}^{u} x_{ij} = 1 \qquad (1 \le j \le n)$$

$$y_i \in \{0,1\} \qquad (1 \le i \le u)$$

$$x_{ij} \in \{0,1\} \qquad (1 \le i \le u; 1 \le j \le n)$$

Обмеження дотримуються того, що максимальна місткість контейнера не повинна бути перевищена і що кожен елемент повинен бути упакований тільки в один контейнер. Метою 1D BPP є мінімізувати загальну кількість конейнерів *N*, які використовуються для упаковки всіх n предметів, це можна оцінити наступним чином:

$$N \ge \left\lceil \left( \sum_{i=1}^{n} S_i \right) / C \right\rceil, \tag{1.4}$$

де $S_i$ представляє розмір або вагу кожного елемента в списку з $n$ елементів, а $C$ представляє фіксовану місткість контейнерів.

### *1.2.1 Приблизні алгоритми*

Наближені алгоритми були першим підходом до вирішення класичного 1D BPP. Існує два типи алгоритмів апроксимації: онлайн та офлайн. В онлайн-алгоритмах елементи розглядаються один за одним, не знаючи про наступні елементи. Отже, розміщення поточного предмета залежить виключно від розміру поточного товару та вже запакованих елементів [7].

Переконливою концепцією, яка вводиться в контексті онлайн-алгоритмів, є алгоритми обмеженого простору. Контейнери можуть бути як відкритими, так і закритими, а предмети повинні бути упаковані тільки у відкриті контейнери. Коли додається новий контейнер, йому надається статус

«відкритий». У деяких алгоритмах контейнер може бути закритий через неможливість додати елементи до вмісту (оскільки це може призвести до перевищення максимальної ємності). Тому алгоритм упаковки вважається алгоритмом k-обмеженого простору. Це означає, що якщо відкрито k контейнерів і один предмет запакований в новий контейнер, то один із відкритих ящиків необхідно закрити [8].

Евристика (NF) упаковує максимальну кількість елементів, які може містити поточний кошик, і коли немає вільного місця для наступного елемента, поточний контейнер закривається, відкривається новий контейнер, і цей контейнер стає поточним [9]. Зокрема, $NF(n) \leq 2 * OPT(n) - 1$, і для кожного $N \in Ń$існує такий набір *n, що* $OPT(n) = NiNF(n) = 2 * OPT(n) - 2$

Теореми:

а) Для всіх вхідних послідовностей L:

$$NF(I) \leq 2 * OPT(I). \quad (1.5)$$

б)Існують такі послідовності L, що:

$$NF(I) \geq 2\ OPT(I) - 2. \quad (1.6)$$

Доведення: а

Розглянемо два контейнери $B_{2k-1}$, $B_{2k}$, $2k \leq NF(I)$.

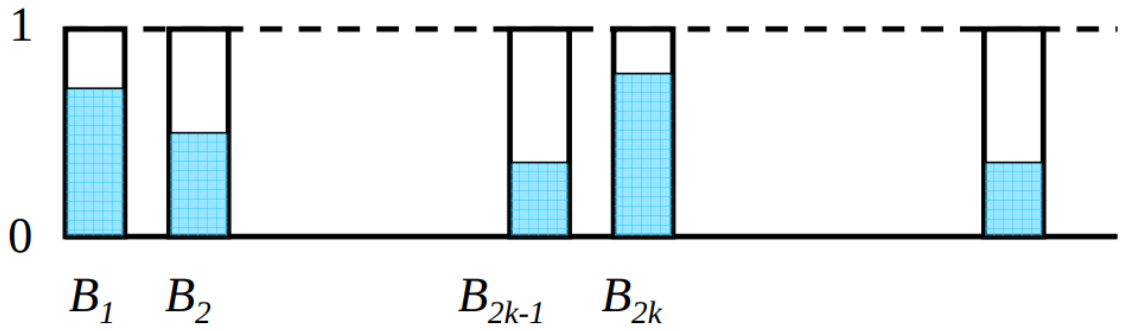



Рисунок 1.4 - Контейнери B2k-1, B2k

Доведення: б

Розглянемо вхідну послідовність *I* розміром *n*

*(n ≡ 0 ( mod 4)):*

*0.5, 2/n, 0.5, 2/n, 0.5, ... , 0.5, 2/n*

Оптимальне пакування:

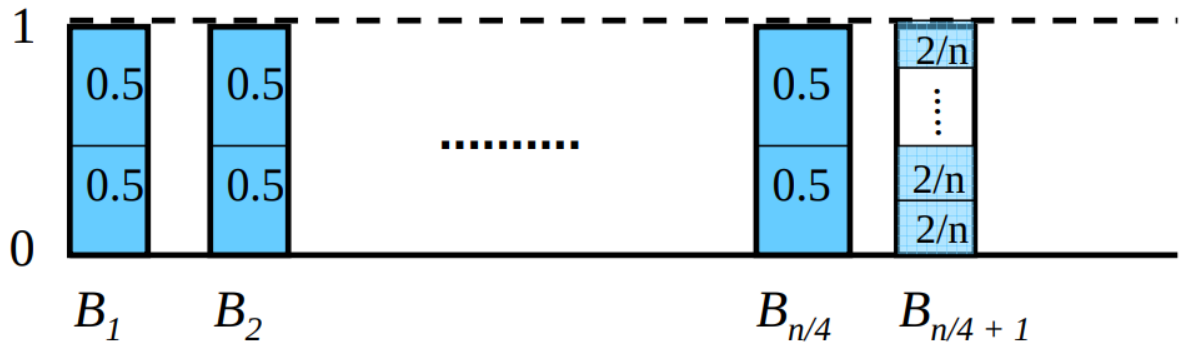



Рисунок 1.5 – Доведення теореми б

Next Fit дає:

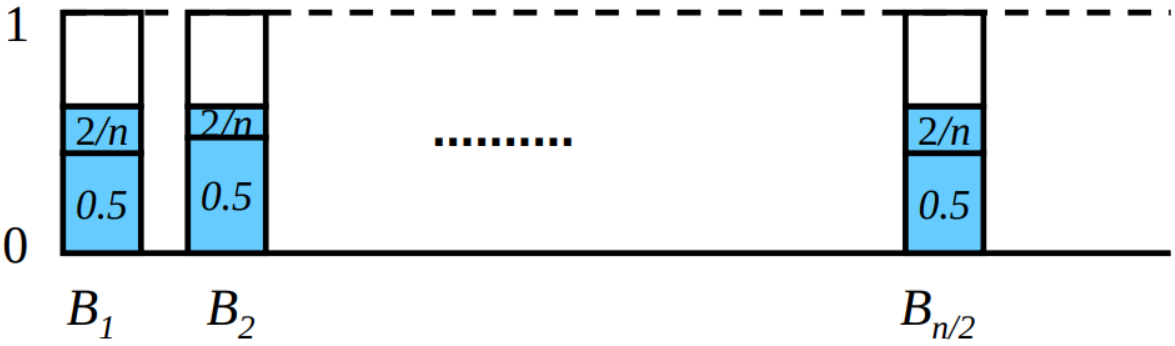



Рисунок 1.6 – Результат Next Fit

Евристика First Fit (FF) відрізняється від NF тим, що контейнери не закриті. Кожен предмет упаковується в перший контейнер, на якому достатньо місця. Якщо предмет не поміститься в жодному з доступних контейнерів, відкривається новий [9]. Евристика Best Fit (BF) упаковує елемент у найкращий або найбільш підходящий контейнер, тобто контейнер з найменшим доступним місцем після включення поточного елемента в нього. Як і у FF, якщо жоден елемент не може поміститися в поточний контейнер, відкривається новий [10]. У кожен момент часу є щонайбільше один контейнер, який заповнений менше ніж наполовину, тобто $FF(I) \leq 2OPT(I)$.

Теорема: Існує така послідовність I, що

$$FF(I) = \frac{10}{6}OPT(I) \tag{1.7}$$

Доведення:

Вхідна послідовність розміром *3 * 6m:*

$$\underbrace{1/7+\varepsilon,\ldots,1/7+\varepsilon}_{6m},\underbrace{1/3+\varepsilon,\ldots,1/3+\varepsilon}_{6m},\underbrace{1/2+\varepsilon,\ldots,1/2+\varepsilon}_{6m}$$

Отримаємо:

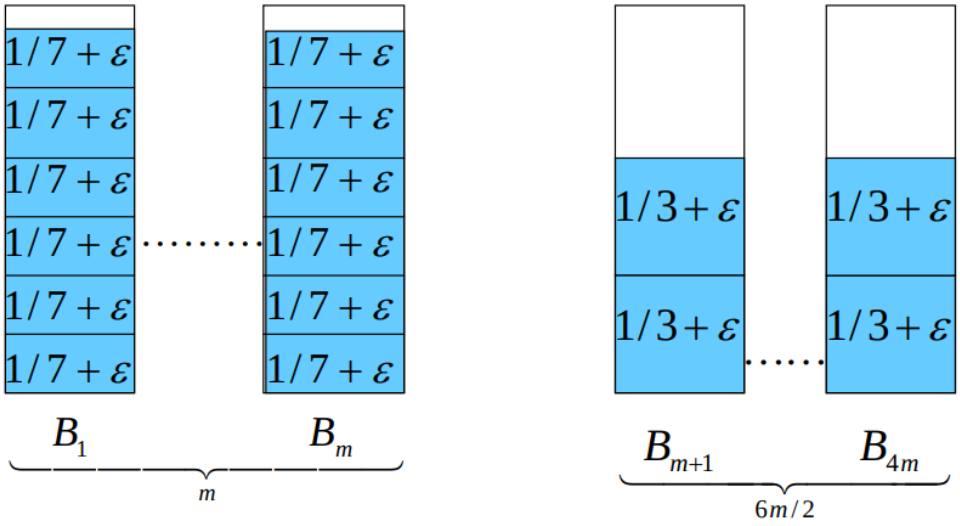



Рисунок 1.7 – Теорема евристики First Fit

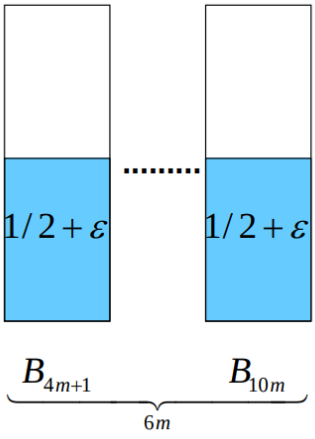



Рисунок 1.8 – Теорема евристики First Fit

З іншого боку, для вирішення цієї проблеми також використовувалися автономні алгоритми. Відомими прикладами є алгоритми NF-Decreasing, BF-

Decreasing і FF. Цей тип евристики обробляє дані перед упаковкою елементів у кошик. Як правило, це означає, що елементи переставляються та сортуються в порядку збільшення або зменшення [11]; після цього ці алгоритми заповнюють контейнери так само, як це роблять оригінальні алгоритми NF, BF і FF. Ці автономні алгоритми не гарантують щоразу повертати оптимальні рішення.

Прикладом алгоритму, який працює як в режимі онлайн, так і в автономному режимі, є евристичний алгоритм Better-Fit (BFH) [12], який видаляє існуючий елемент із кошика та замінює його поточним, якщо поточний елемент краще заповнює кошик. Більше того, якщо упаковка поточного предмета призводить Переконливою концепцією, яка вводиться в контексті онлайн-алгоритмів, є алгоритми обмеженого простору. Контейнери можуть бути як відкритими, так і закритими, а предмети повинні бути упаковані тільки у відкриті контейнери. Коли додається новий контейнер, йому надається статус «відкритий». У деяких алгоритмах контейнер може бути закритий через неможливість додати елементи до вмісту (оскільки це може призвести до перевищення максимальної ємності).Тому алгоритм упаковки вважається алгоритмом k-обмеженого простору. Це означає, що якщо відкрито k контейнерів і один предмет запакований в новий контейнер, то один із відкритих ящиків необхідно закрити. BF-Decreasing, а якщо елементи впорядковано в порядку зростання, алгоритм помістить невеликі елементи в початкові контейнери. Більші предмети не можуть замінити їх. Тому найкращою продуктивністю є середній випадок, коли ваги (або розміри) елементів є випадковими [12].

### *1.2.2 The fitness-dependent optimizer (FDO)*

FDO був розроблений у 2019 році Абдуллою та Ахмедом [13]. Це інтелектуальний алгоритм, який моделює характеристики репродуктивного процесу бджолиних роїв, а також їх колективну поведінку при прийнятті рішень. Хоча FDO вважається алгоритмом на основі PSO, він обчислює

швидкість частинки по-різному – він використовує значення функції пристосованості проблеми для отримання ваг. Ці вагові коефіцієнти потім керують пошуковими агентами алгоритму під час фаз експлуатації та дослідження. Крім того, алгоритм FDO вимагає менше обчислень, ніж алгоритм PSO для оновлення швидкості та положення частинок [13]. Бджоли - це соціальні комахи, які живуть у невеликих печерах або дуплах дерев. Вони працюють у колоніях, які зазвичай називають вуликами. Три типи бджіл у колонії включають матку, робочу та бджоли-розвідницю. Кожна з цих бджіл виконує певні функції відповідно до своїх ознак. Алгоритм FDO фокусується на функції бджіл-розвідників. Ці бджоли досліджують навколишнє середовище, щоб знайти відповідне місце для колонії, щоб побудувати вулик (іншими словами, вони експлуатують бажані вулики). Як тільки це знайдено, бджоли виконують свого роду «танець», щоб спілкуватися з роєм [14]. В алгоритмі кожен вулик, яким користується бджола-розвідниця, розглядається як рішення-кандидат, а найкращий вулик представляє глобальне оптимальне рішення, відповідно до відносної ваги придатності. Для більш формального описання алгоритму використаємо блок-схему.

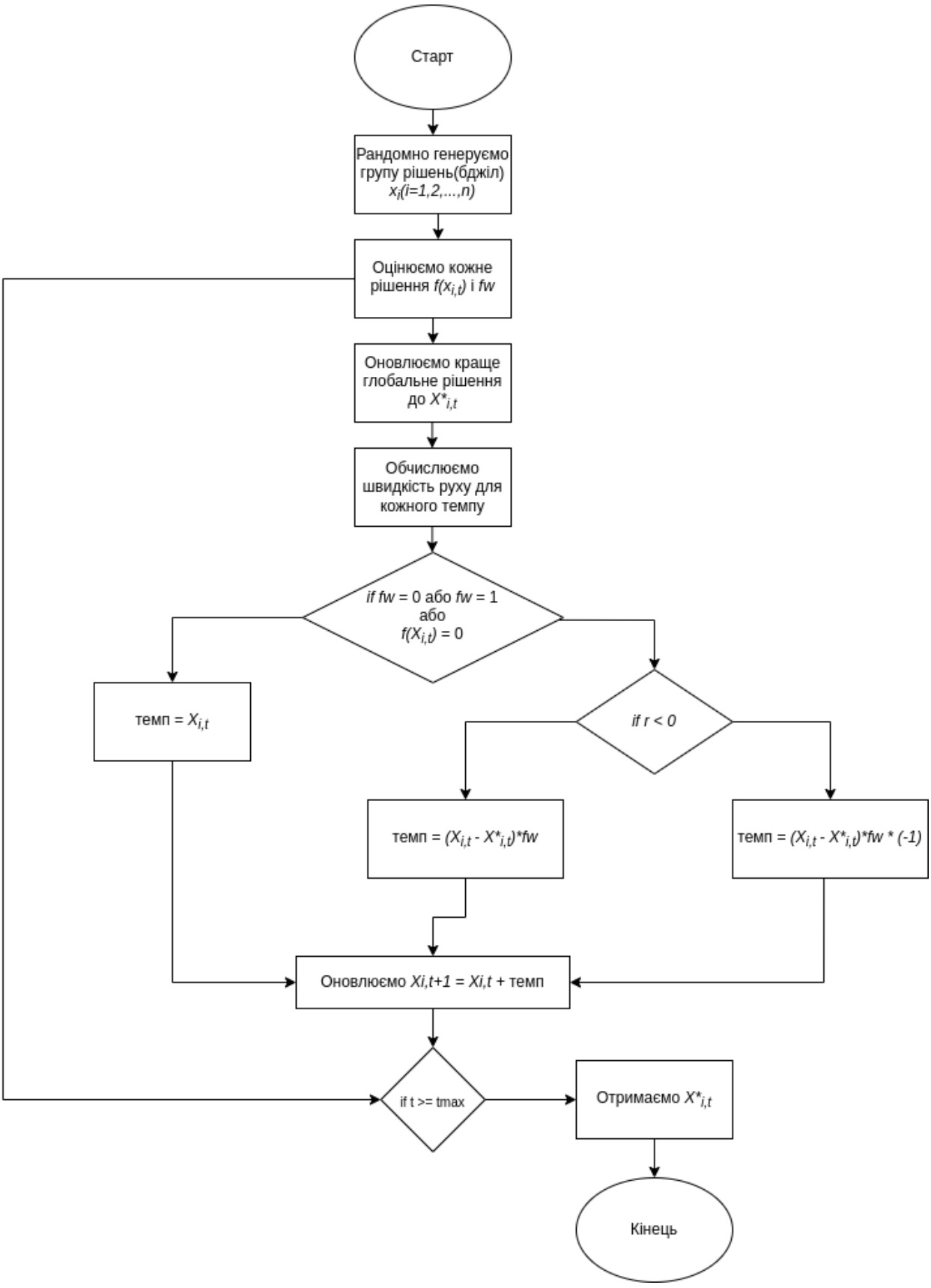



Рисунок 1.9 - Блок-схема алгоритму

Переваги цього алгоритму полягають у тому, що він стабільний як на етапах експлуатації, так і на етапах дослідження, і що він вимагає менше обчислень, ніж інші алгоритми. Недоліком є те, що він використовує ваговий коефіцієнт, щоб контролювати ваги придатності, які в більшості випадків ігноруються [15]. Абдул-Мінаам та ін. [14] врахував ці сильні та слабкі сторони. Вони адаптували FDO для вирішення 1D BPP, замінивши оригінальний спосіб генерації першої сукупності випадковим генеруванням початкової сукупності з використанням покращеної евристики FF та шляхом оновлення операційних стратегій в оригінальному алгоритмі для покращення фаз дослідження та експлуатації.

### *1.2.3 Whale optimization algorithm*

WOA був розроблений в 2016 році Мірджалілі та Льюїсом [16]. Цей алгоритм є метаевристичним методом, заснованим на ройовому інтелекті, і він імітує стратегію полювання на горбатих китів. Ці кити відомі своєю здатністю використовувати незвичайну стратегію полювання, яку також називають стратегією «пузирної сітки». Саме тут кити спритно співпрацюють, щоб захопити свою жертву, рибу, у повітряне кільце. Кити досягають цього, випускаючи потік бульбашок. Ці бульбашки мають дві різні форми: перша — спіраль, а друга — круг, що скорочується. Захопивши свою жертву в цих бульбашках, рибу проковтують горбаті кити. В алгоритмі фаза дослідження представлена полюванням на здобич, тоді як стратегія полювання з використанням стискаючих і спіральних форм представляє фазу експлуатації. Позиції жертви ініціалізуються випадковим чином – це являє собою початкову популяцію, яка оцінюється, щоб знайти поточне найкраще рішення.

До переваг цього алгоритму можна віднести простоту реалізації та низьку обчислювальну вартість. З іншого боку, недоліком є те, що алгоритм іноді призводить до стагнації локального оптимуму, і що фаза експлуатації була б більш вигідною, якби її покращили [13]. Абдель-Бассет та ін. [12] використали ці сильні сторони у своїй адаптації WOA для вирішення 1D BPP,

який називається покращеним алгоритмом оптимізації китів на основі Леві (ILWOA), розробленим для вирішення BPP. Це було досягнуто шляхом інтеграції різних механізмів, включаючи техніку найбільшої вартості замовлення (LOV) (щоб перетворити безперервні результати в дискретні), польоти Леві та хаотичні карти для подолання слабких сторін оригінального WOA.

### *1.2.4 GA*

GA спочатку був запропонований Холландом [15] і був застосований до широкого кола задач оптимізації. Цей алгоритм імітує процес природного відбору еволюції, заснований на вченні Чарльза Дарвіна [10]. У GA рішення моделюються як хромосоми, які можуть приймати різні форми – наприклад, двійкові рядки або вектори реальних чисел. Популяція складається з кількох хромосом. Необхідно визначити функцію придатності, щоб ці хромосоми можна було декодувати та оцінити, а також призначити цій хромосомі значення придатності. Це значення є показником сили цієї особини в поточній популяції та диктує ймовірність бути обраним для створення наступної популяції. Нові популяції генеруються шляхом виконання таких методів на хромосомах, як мутація та кросовер. Більш сильні або більш придатні хромосоми, швидше за все, будуть обрані для перенесення в нову популяцію після застосування генетичних операторів, щоб нові популяції містили кращі якісні рішення.

Переваги цього алгоритму включають те, що він був розроблений для вирішення складних проблем і паралелізму, а також його легко зрозуміти та реалізувати. Однак, природно, у цієї метаевристики є недоліки – це включає складність формулювання правильної функції придатності для даної задачі, визначення найкращого розміру популяції та таких параметрів, як кросовер і швидкість мутації, які визначають, як часто оператори повинні зустрічатися під час ітерації алгоритму. Невідповідний вибір цих параметрів і функцій може призвести до того, що GA виведе результати, які не мають сенсу, або для

GA може бути все важче збігтися. Quiroz та ін. використали GA як основу для їх групування GA, додавши механізми групування та захоплюючий механізм відбору для вирішення BPP.

## 1.3 Методи оптимізації

### *1.3.1 Адаптація метаевристики для 1D BPP*

Рисунок 1.10 ілюструє загальний огляд метаевристичних алгоритмів, які використовуються для вирішення 1D BPP. Алгоритм починається з визначення параметрів, таких як розмір популяції та будь-які інші значення, які необхідно вказати, щоб забезпечити належне функціонування алгоритму. Далі створюється початкова сукупність. Цей крок важливий, і обраний для цього метод сильно впливає на ефективність алгоритму. Далі рішення проходять фазу «підгонки та оцінки». Це стосується використання приблизних алгоритмів для упаковки предметів у контейнери. Ця упаковка потім оцінюється цільовою функцією, щоб розв’язку було присвоєно значення придатності, яке визначатиме його позицію в сукупності. Цей етап додатково допомагає визначити найкраще рішення в популяції. Це найкраще рішення потім зберігається в пам’яті. Потім алгоритм буде виконувати конкретні локальні та глобальні обходи, щоб популяція була покращена, додаючи рішення з більш сприятливими значеннями пристосованості та взагалі відкидаючи несприятливі або слабкі рішення. На цьому етапі необхідно повторити фазу «підгонки та оцінки», щоб можна було перевірити найкраще рішення, і якщо буде знайдено краще рішення, ніж поточне, то це рішення замінює найкраще рішення в пам’яті і стає найкращим на даний момент. Деякі алгоритми можуть вимагати оновлення параметрів, але це не завжди так; тому це необов'язково. Після цього перевіряється критерій припинення. Зазвичай критерій завершення стосується кількості ітерацій (поколінь населення), які пройшов алгоритм. Однак критерієм також може бути перевірка того, чи є найкраще поточне рішення оптимальним. Можливо, що одна або обидві з цих умов перевіряються на цьому етапі алгоритму. Якщо критерій виконано, то

алгоритм може завершити роботу і отримати остаточне рішення. Однак, якщо критерій не виконано, алгоритм повертається, щоб оновити сукупність за допомогою механізмів дослідження та експлуатації, щоб можна було знайти краще рішення, і цикл продовжується.

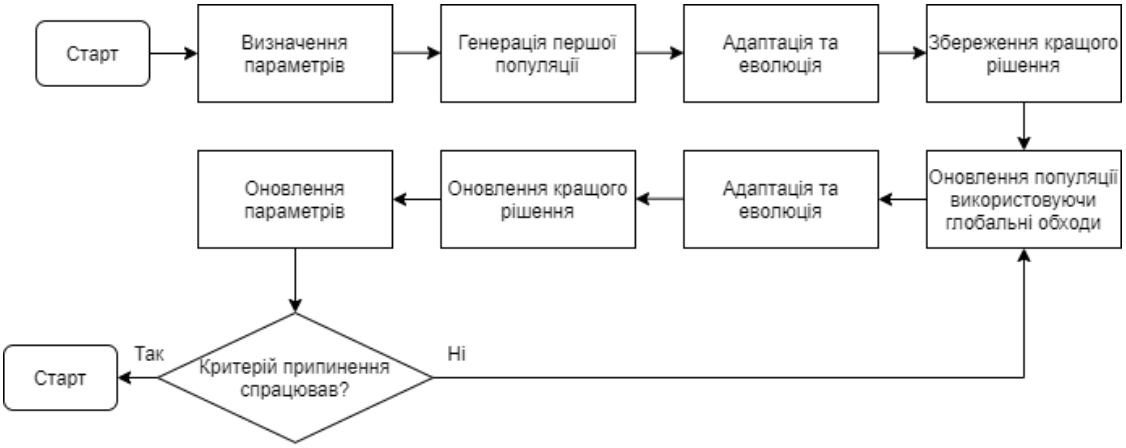



Рисунок 1.10 - Адаптація метаевристики для 1D BPP

### *1.3.2 Адаптована метаевристика для 1D BPP*

Якщо для цієї функції встановлено лише кількість контейнерів, алгоритм в кінцевому підсумку може випробувати стагнацію [16]. Це пов’язано з кількома можливими рішеннями, які мають однакову кількість контейнерів, але різні уявлення, а отже, різну кількість невикористаного простору. Щоб уникнути такого застою і рухатися до пошуку оптимальних рішень, цільова функція повинна враховувати кількість невикористаного простору, а також кількість контейнерів для конкретного представлення. Тому використовуються такі функції:

$$\text{Minimize } f(X) = 1 - \frac{\sum_{k=1}^{N}(\text{fill}_k/C)^z}{N} \tag{1.8}$$

$$\text{Maximize } f(X) = \frac{\sum_{k=1}^{N}(\text{fill}_k/C)^z}{N}, \tag{1.9}$$

де

$$\text{fill}_k = \sum_{i \in X_k} S_i, \tag{1.10}$$

що, по суті, тягне за собою надання більшої цінності розв’язку з використанням найменшої кількості контейнерів і найменшої кількості невикористаного простору. Таким чином, fill k — це місткість контейнера k (це сума розмірів ( S i ) усіх елементів у контейнері), N — кількість контейнерів у рішенні, C — фіксована ємність контейнера, а z — константа, яка використовується для визначення рівноваги заповненого контейнера (зазвичай зберігається на z = 2 ) [17].

### 1.4 Постановка задачі

Отримавши достатній обсяг інформації стосовно теми «актуальності NP повних задач» та після інформаційного дослідження Bin Packing Problem були проаналізовані деякі алгоритми та оптимізаційні процеси, предметна область була опрацьована. Опираючись на теоретичні знання, описані в даній роботі, було виділені такі задачі для реалізації:

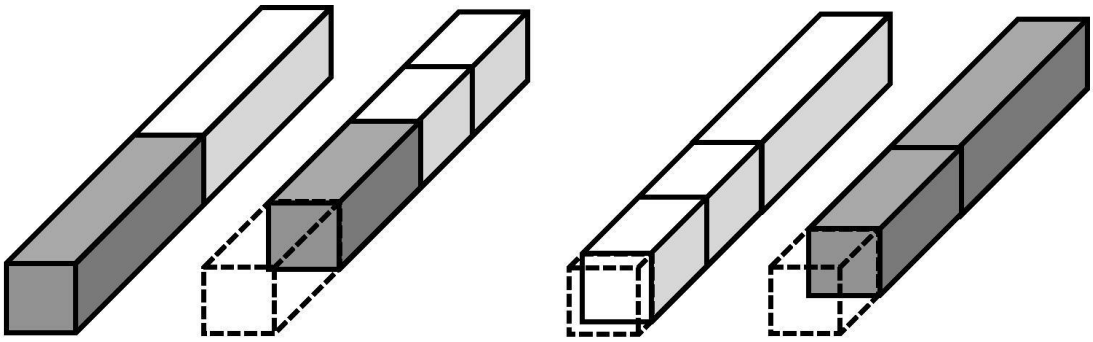
1. Створення додатку для демонстрації коректної роботи алгоритмів на основі існуючих методів.
2. Вибрати одну з реальних проблем, яку відноситься до проблем пакування.
3. Створити алгоритм, який буде знаходити рішення оптимальним методом для конкретної задачі.

# 2 МАТЕМАТИЧНА МОДЕЛЬ ТА МЕТОДИ РІШЕННЯ

У класичній 3D-BPP(3 dimensional bin packing problem) мета полягає в тому, щоб упакувати прямокутні елементи в мінімальну кількість тривимірних прямокутних ящиків(контейнерів) [18]. Додатково до цієї мети на практиці слід враховувати кілька факторів, таких як стабільність завантаження та баланс навантаження, а також інші проблеми, пов’язані з елементами та замовленнями, такими як вимоги до укладання та упаковка предметів в межах одного місця призначення. Стабільність навантаження гарантує, що мінімально допустимий відсоток площі основи кожного елемента підтримується, тобто знаходиться на підлозі або на іншому предметі. Це підтримує стабільність предметів для обробки та транспортування під час внутрішніх і зовнішніх логістичних операцій. Другий фактор, баланс навантаження, пов’язаний з вагою предметів, які повинні бути рівномірно розподілені по нижній частині контейнера. Залежно від способу транспортування, неправильне зміщення від ідеального центру баризонта, який визначається як ідеальна точка, наприклад, центр контейнера, може збільшити ризики для безпеки вантажу [18]. Незважаючи на очевидну важливість, фактори стабільності навантаження та балансу, як правило, не розглядаються явно в задачі упаковки контейнерів. Отже, задача щодо практичного розширення 3D-BPP, яка враховує ці фактори, а саме 3D-BPP з балансуванням навантаження, є відносно мізерною.

Family unty була визначена як нова функція для представлення пов’язаних проблем, які можуть бути додані як характеристика до кожного елемента окремо. Ми вводимо family unity, щоб описати більш повний набір функцій, тоді як product family використовується для представлення елементів такого ж розміру або тієї ж характеристики в задачі. Ця нова концепція стосується предметів, які потрібно запакувати в той самий контейнер через практичні міркування, такі як однакове призначення, вимоги до обробки, умови зберігання тощо. Покращення family unity забезпечує кращі рішення як

для зменшення складності планування вантажно-розвантажувального обладнання, так і для виконання операцій з обробки. На рис. 1 наведено ілюстративний приклад, який показує простий план пакування з двома контейнерами [19]. Хоча для завантаження/розвантаження обох контейнерів на рис. 2.1(a) може знадобитися більше одного типу вантажно-розвантажувального обладнання, для кожного контейнера потрібен один тип вантажно-розвантажувального обладнання на рис. 2.1(b). Крім того, обидва контейнери можна відправити до кінцевого пункту призначення без будь-якої подальшої обробки в центрі передачі, де перше налаштування вимагає перепризначення елементів у контейнери на основі призначення.



1(а)План пакування без Family unit

1(б)План пакування з Family unit

Рисунок 2.1 - Ефект використання Family unit

Зауважимо, що концепція Family unity відрізняється від концепції сумісності, легко доступної в літературі з упаковки контейнера, яка дозволяє або обмежує упаковку предметів в один контейнер. Family unity можна розглядати як специфічну особливість, яка призначається кожному предмету, який потрібно запакувати в певну кількість контейнерів. У літературі проблеми, пов’язані з елементами, розглядаються в рамках обмежень розподілу, які далі можна класифікувати як обмеження з’єднання та обмеження з кількома параметрами. Однак ці обмеження не зовсім відповідають концепції, яку ми вводимо. Обмеження підключення використовуються в проблемах з упаковкою одного контейнера, щоб гарантувати, що предмети, що належать одному клієнту, розташовані близько

один до одного в тому самому кошику. Інша спроба розглянути обмеження, пов’язані з елементами, у проблемі упаковки одного контейнера представлена Егебладом, Гаравеллі, Лісі та Пісінгером і спрямована на упакування неправильних предметів однакової форми близько один до одного [19]. Крім того, у проблемах маршрутизації зустрічаються обмеження з кількома параметрами. Що стосується порядку доставки покупцям і місця, необхідного для доставки, предмети розміщуються в контейнерах так, щоб предмети клієнта стали доступними без переміщення предметів іншого клієнта. Елементи розташовуються на основі замовника з використанням таких стратегій, як LIFO, відповідно до замовлення розповсюдження. У цьому дослідженні ми зосереджуємось як на тому, щоб призначити товари з одного сімейства продуктів до одного кошика, так і запакувати їх у кошик, враховуючи обмеження балансу, тоді як дослідження в роботі, як правило, зосереджені на тісній упаковці продуктів, що належать одному клієнту в межах одного контейнера.

З іншого боку, Famliy unity допомагає неявно врахувати додаткові витрати на розповсюдження, які можуть виникнути, коли товари з однаковим призначенням упаковуються в різні ящики. Таким чином, загальна вартість, ймовірно, зменшиться з додаванням цієї нової практичної концепції, хоча загальна вартість на етапі упаковки контейнера може збільшитися.

Далі ми вводимо багатоцільову математичну модель для збалансованого навантаження 3D-BPP, де дозволена орієнтація елементів і враховуються проблеми, пов’язані з елементами. Ця модель пропонує  метод змішаного цілочисельного програмування (MIP) з повним набором обмежень, що стосуються тривимірної заміни, стабільності та орієнтації, а також набором цільових функцій, що складають мінімізацію кількості ящиків. використовується і відхилення від ідеального барицентру, а також максимізація коефіцієнта єдності сім'ї за допомогою зваженої цільової функції [20].

## 2.1 Математична модель

Проблема упаковки привернула дослідницький інтерес у дослідницькому співтоваристві з пакування. Хоча 3D-BPP можна моделювати та розв’язувати як змішану цілу програму, існує кілька доступних методологій точних рішень (Elhedhli, Gzara, & Yildiz, 2019) [25]. На додаток до точних методів, кілька досліджень були зосереджені на розробці евристики для практичних аспектів проблеми (Chen, Lee, & Shen, 1995) [24]. Крім того, було запропоновано багато евристичних, мета-евристичних та гібридних методів для вирішення проблем упаковки з одним і кількома контейнерами (Elhedhli et al., 2019, Bortfeldt and Wäscher, 2013) [22].

Bortfeldt і Wäscher в 2013 виявили, що практичні міркування щодо проблем упаковки кількох контейнерів майже повністю знехтували в порівнянні з величезними дослідженнями щодо упаковки одного контейнера. Незважаючи на те, що темпи наростають, дослідження множини контейнерів все ще дуже мізерні (Alonso, Alvarez-Valdes, Iori, & Parreño, 2019) [23].

Попередня робота над оптимальним упакуванням предметів у мінімальну кількість контейнерів базується на двовимірній моделі упаковки контейнерів, яку сформулювали Мартелло та Віго (1998) [26]. Liu, Tan, Huang, Goh, and Ho (2008) далі запропонували математичну модель для вирішення двовимірної проблеми упаковки контейнера щодо двох цілей: мінімізації кількості використовуваних контейнерів і балансування навантаження в кожному контейнері [27]. Однак орієнтація та стабільність елементів не враховуються. Хоча для деяких типів вантажів практичне вирішення проблеми двовимірної упаковки контейнерів, моделі 3D-BPP особливо важливі для проблем із завантаженням контейнерів. Однак 3D-BPP є дуже важкою NP-hard і надзвичайно важкою для вирішення на практиці (Martello, Pisinger, & Vigo, 2000) [27].

Перша аналітична модель 3D-BPP, що враховує орієнтацію, кілька розмірів предметів, кілька розмірів ящиків, уникнення перекриття елементів

та обмеження використання простору, представлена Ченом (1995). Pedruzzi, Nunes, Rosa, and Arpini (2016) розширили модель за допомогою визначення додаткових допоміжних обмежень і нової цільової функції, що мінімізує суму координат, щоб отримати компактне розташування елементів у контейнерах. Alonso, Alvarez-Valdes, Iori, Parreño and Tamarit (2017) представляють математичні моделі для 3D-BPP, починаючи від простої моделі до більш складних моделей, адаптуючи кілька розширень, таких як вага та об'єм баз піддонів, положення центру, гравітації та мінімізації кількості піддонів.

У комплексному дослідженні Bortfeldt і Wäscher (2013) [28] було розглянуто 163 публікації, що розглядають практичні обмеження в дослідженнях з упаковки контейнерів. Таблиця 1 підсумовує додаткові обмеження в запропонованих моделях MIP для проблеми з кількома упаковками в літературі.

Таблиця 2.1 - Практичні обмеження в дослідженнях з упаковки контейнерів

| Публікація | Контейнери | | Обмеження балансу | Обмеження стабільності | Обмеження орієнтації |
|---|---|---|---|---|---|
| | Однакові | Різні | | | |
| Chen et al. (1995) | + | + | | + | + |
| Eley (2003) | + | + | | | |
| Trivella and Pisinger (2016) | + | | + | | |
| Pedruzzi et al. (2016) | + | + | | + | + |
| Crainic, Gobbato, Perboli, and Rei (2016) | + | + | | | |

| | | | | | |
|---|---|---|---|---|---|
| Alonso, Alvarez-Valdes, Iori, Parreño, and Tamarit (2017) | + | | + | | |

## 2.2 Моделювання проблеми багатовимірної упаковки контейнерів із збалансованим навантаженням.

Проблема багатовимірної упаковки контейнерів із збалансованим навантаженням полягає в упаковці паралелепіпедних об’єктів у найменшу кількість контейнерів таким чином, щоб центр мас завантажених контейнерів потрапляв якомога ближче до бажаного місця. Тобто, сума зміщень від бажаного розташування барицентру над (найменшою кількістю) використаних контейнерів мінімізується.

### *2.2.1 Балансування одного контейнера*

Припустимо, що ми вже отримали можливе рішення МВР за допомогою точного або евристичного методу, і розглянемо один контейнер. Ми хочемо розташувати набір заданих m ящиків U ={1,2...,m}в D-вимірний контейнер, щоб відстань між бароцентром завантаженого контейнера та ідеальною точкою всередині контейнера, наприклад його геометричним центром, була зведена до мінімуму. За припущенням, у контейнер можна помістити всі предмети.

Припустимо, що коробки однорідні і мають густини ρi, i∈U. За допомогою ρi можна визначити масу mi коробки i за допомогою формули mi = ρi*vol(i), де $vol_i = \Pi_{d \in D} w_{id}$ це об’єм коробки i. Аналогічно, нам можна було б дати безпосередньо маси mi замість густин.

Поняття з фізики, такі як щільність, маса та об’єм, зазвичай відносяться до тривимірних об’єктів, але будуть абстрактно використовуватися також у просторах більш високого виміру.

Оскільки коробка і є однорідним і має прямокутну форму, його барицентр збігається з його геометричним центром, який відповідає Ci=(xid+wid/2)d∈D, а барисцентр B всього завантаженого контейнеру можна обчислити за допомогою виразу:

$$B = \left(B_d\right)_{d\in D} = \frac{\sum_{i\in U} m_i C_i}{\sum_{i\in U} m_i} \tag{2.1}$$

Потім цільова $f(x) = f((x_{id})_{i,d})$ функція являє собою відстань барицентра B від його ідеального розташування, позначену як

$$B^{opt} = (B_d^{opt})_{d\in D}. \tag{2.2}$$

L1, або прямокутна нормаль, тобто сума абсолютних значень різниць компонентів D, використовується для обчислення відстані:

$$f\left(x\right) = \left\|B^{opt} - B\right\|_1 = \sum_{d\in D}\left|B_d^{opt} - B_d\right| \tag{2.3}$$

L1 вибрано, оскільки він краще дозволяє окремо керувати зміщеннями в кожному напрямку і тому, що його можна легко лінеаризувати.

Іноді може бути важливішим балансування в одній координаті, ніж в інших. Наприклад, можна вважати, що більш актуальним є досягнення ідеального балансування на осях x і y 3D-контейнері, не надто піклуючись про висоту барицентра, або навпаки. Як наслідок, деякі вагові коефіцієнти включені у визначення f, щоб задовольнити індивідуальні потреби, щоб налаштувати критерії балансування, і більш загальна цільова функція виглядає так:

$$f\left(x\right) = \sum_{d \in D} k_d \left| B_d^{opt} - B_d \right| \tag{2.4}$$

Наприклад, під час завантаження вантажівки було б розумно вибрати центр основи контейнера як ідеальний барицентр, тобто точку з координатами

$$\left(\tfrac{1}{2} W_x, \tfrac{1}{2} W_y, 0\right) \tag{2.5}$$

Якщо припустити, що низький барицентр вдвічі важливіший за балансування в напрямках x і y, то цільова функція буде виглядати так:

$$\left|\tfrac{1}{2} W_x - B_x\right| + \left|\tfrac{1}{2} W_y - B_y\right| + 2\left|B_z\right| \tag{2.6}$$

Повертаючись до загального випадку і підставляючи компоненти B їх повними виразами, отримуємо:

$$f\left(x\right) = \sum_{d \in D} k_d \left| B_d^{opt} - \frac{1}{M_U} \left( \sum_{i \in U} m_i \left( x_{id} + \frac{w_{id}}{2} \right) \right) \right| \tag{2.7}$$

Функція f не є лінійною, оскільки містить абсолютні значення. Відомий метод для обробки абсолютного значення в цільовій функції полягає у введенні двох додаткових позитивних опорних змінних, які вказують позитивну та негативну частини величини всередині абсолютного значення.

$$r_d - s_d = B_d^{opt} - \frac{1}{M_U} \left( \sum_{i \in U} m_i \left( x_{id} + \frac{w_{id}}{2} \right) \right) \tag{2.8}$$

що дозволяє переписати мету на:

$$f\left(x\right) = \sum_{d\in D} k_d \left(r_d + s_d\right) \tag{2.9}$$

Нарешті, повторно використовуючи набір змінних lijd, уже введений для МВР, і блоки обмежень, що забезпечують відсутність перекриття та порушення кордонів, задача формулюється за допомогою наступної лінійної моделі:

$$\begin{aligned}
&\min \quad \sum_{d\in D} k_d \left(r_d + s_d\right) \\
&\text{s.t.:} \quad r_d - s_d = B_d^{opt} - \frac{1}{M_U}\left(\sum_{i\in U} m_i \left(x_{id} + \frac{w_{id}}{2}\right)\right) \quad \forall d \in D \\
&\sum_{d\in D}\left(l_{ijd} + l_{jid}\right) \geq 1 \quad \forall i < j \in U \\
&x_{id} - x_{jd} + W_d l_{ijd} \leq W_d - w_{id} \quad \forall i \neq j \in U,\ d \in D \\
&x_{id} \leq W_d - w_{id} \quad \forall i \in U,\ d \in D \\
&\text{var:} \quad x_{id}, r_d, s_d \in \mathbb{R}^+ \quad l_{ijd} \in \left\{0,1\right\} \quad \forall i,j \in V,\ d \in D
\end{aligned} \tag{2.10}$$

Будемо вважати (2.10) багатовимірною проблемою балансування одиничного навантаження (SLB).

Структура подібна до загальної МВР, і більшість обмежень по суті однакові. Отже широке використання умовних обмежень і коефіцієнтів великого M ускладнює розв’язання моделі на практиці за допомогою стандартних вирішувачів.

Потрібно звернути увагу, що, за припущенням, ми починаємо з набору предметів, які можна запакувати в один контейнер, отже, можливі рішення існують. Однак ця інформація апріорі відсутня в моделі, і якщо кількість ящиків велика (більше 25–30) з великим заповненням об’єму контейнера, то може бути важко знайти можливе рішення. Більше того, навіть коли можливе

рішення знайдено швидко, різниця від найкращої нижньої межі може зайняти величезну кількість часу.

### 2.2.2 *Пакування та балансування*

Переходячи до загальної проблеми, ми можемо спочатку призначити елементи до мінімальної кількості ящиків, розв’язавши модель MBP, а потім застосувати модель SLB, щоб переставити елементи всередині кожного використаного контейнера, щоб оптимально збалансувати їх навантаження.

На жаль, це не повністю виконує наші поставлені цілі, оскільки обидві моделі не пов’язані між собою, а фази пакування та балансування виконуються асинхронно. MBP, як правило, має багато різних рішень, і можуть існувати інші призначення елементів до найменшої кількості контейнерів, що забезпечує загальне краще балансування. Іншими словами, вихідні дані фази пакування можуть бути неоптимальними для фази балансування, і рішення, яке ми отримуємо після застосування двох послідовних фаз, не гарантовано буде оптимальним для нашої загальної проблеми.

Тепер завдання полягає в тому, щоб об’єднати разом фази пакування та балансування та сформулювати інтегровану модель MILP.

### *2.2.3 Цільова функція*

У загальному випадку задіяно більше контейнерів, і цільова функція міститиме суму зміщень від бажаного барицентру по всіх використаних контейнерах. Позначимо через Uj множину ящиків всередині контейнеру j. Для одного контейнеру j ціль fj моделюється як:

$$f_j\left(x\right) = \sum_{d \in D} k_d \left(r_{jd} + s_{jd}\right) \tag{2.11}$$

де rjd + sjd- абсолютне значення різниці між оптимальним розташуванням B(opt)d і d-й координатою Bjd барицентра контейнера j, тобто:

$$r_{jd} + s_{jd} = \begin{cases} \left| B_d^{opt} - \frac{1}{M_{U_j}} \left( \sum_{i \in U_j} m_i \left( x_{id} + \frac{w_{id}}{2} \right) \right) \right| \\ 0 \end{cases} \quad (2.12)$$

Тоді загальне переміщення по всіх бункерах дорівнює:

$$f\left(x\right) = \sum_{j=1}^{n} f_j\left(x\right) = \sum_{j=1}^{n} \sum_{d \in D} k_d \left( r_{jd} + s_{jd} \right) = \sum_{d \in D} k_d \sum_{j=1}^{n} \left( r_{jd} + s_{jd} \right) \quad (2.13)$$

де сума обчислюється j = 1,..,n, оскільки кількість елементів n є верхньою межею кількості контейнерів, які будуть використовуватися.

За визначенням проблеми, головна мета все ще полягає в тому, щоб мінімізувати кількість контейнерів, і рішення, яке використовує N+1 контейнерів, вважається гіршим, ніж рішення, яке використовує N, незалежно від кращого загального балансу першого. Отже, цільову функцію можна визначити як

$$f\left(x\right) = CN + \sum_{d \in D} k_d \sum_{j=1}^{n} \left( r_{jd} + s_{jd} \right) \quad (2.14)$$

де C обрано таким, що

$$C \geq \sum_{d \in D} k_d \sum_{j=1}^{n} \left( r_{jd} + s_{jd} \right) \quad (2.15)$$

, тобто компонент, пов’язаний з кількістю контейнерів, більший за будь-яке можливе значення балансуючої частини цілі. Таким чином, балансування не буде суперечити пошуку рішення з використанням найменшої кількості контейнерів. Можливе значення C:

$$C = \tfrac{1}{2} n \sum_{d \in D} k_d W_d \tag{2.16}$$

Це значення можна уточнити, щоб зменшити порогове значення. Наприклад, n можна замінити будь-якою верхньою межею для найменшої кількості контейнерів

*2.2.4 Нові змінні та обмеження*

Змінні, які використовуються в моделі МВР, константні. Крім того, ми вводимо двійкові змінні:

$$c_{ij} = \begin{cases} 1 & \text{if } a_i = j, \text{ i.e. item } i \text{ is in bin } j \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

Наслідки можна моделювати за допомогою наступних пар обмежень:

$$a_i - j \leq n\left(1 - c_{ij}\right) \quad \forall i, j \in V a_i - j \geq n\left(c_{ij} - 1\right) \quad \forall i, j \in V \tag{2.18}$$

Щоб обробити зворотну імплікацію, деякі додаткові двійкові змінні γij і δij повинні бути введені разом з обмеженнями:

$$a_i - j \leq -1 + \left(n+1\right)\left(1 - \gamma_{ij}\right) \quad \forall i, j \in V a_i - j \geq 1 - \left(n+1\right)\left(1 - \delta_{ij}\right) \quad \forall i, j \in V c_{ij} + \gamma_{ij} + \delta_{ij} = 1 \quad \forall i, j \in V \tag{2.19}$$

Тепер змінні сij використовуються для визначення неперервних змінних αijd, які відповідають:

$$\alpha_{ijd} = \begin{cases} m_i\left(x_{id} + \dfrac{w_{id}}{2}\right) & \text{if } c_{ij} = 1 \\ 0 & \text{if } c_{ij} = 0 \end{cases} \tag{2.20}$$

Імплікація забезпечується накладенням:

$$\alpha_{ijd} - m_i\left(x_{id} + \frac{w_{id}}{2}\right) + m_i W_d c_{ij} \le m_i W_d \quad \forall i, j \in V,\ d \in D m_i\left(x_{id} + \frac{w_{id}}{2}\right) - \alpha_{ijd} + m_i W_d c_{ij} \le m_i W_d \quad \forall i, j \in V,\ d \in D \tag{2.21}$$

Як і для αijd, ми вводимо змінні βijd так, що:

$$\beta_{ijd} = \begin{cases} m_i\left(B_d^{opt} - r_{jd} + s_{jd}\right) & \text{if } c_{ij} = 1 \\ 0 & \text{if } c_{ij} = 0 \end{cases} \tag{2.22}$$

Це можна змоделювати за допомогою набору обмежень:

$$\beta_{ijd} - m_i\left(B_d^{opt} - r_{jd} + s_{jd}\right) + m_i W_d c_{ij} \le m_i W_d \quad \forall i, j \in V,\ d \in D m_i\left(B_d^{opt} - r_{jd} + s_{jd}\right) - \beta_{ijd} + m_i W_d c_{ij} \le m_i W_d \quad \forall i, j \in V,\ d \in D 0 \le \beta_{ijd} \le c_{ij} W_d m_i \quad \forall i, j \in V,\ d \in D \tag{2.23}$$

Нарешті, наступна важлива рівність:

$$\sum_{i \in V} \alpha_{ijd} = \sum_{i \in V} \beta_{ijd} \quad \forall j \in V,\ d \in D \tag{2.24}$$

представляє балансування навантаження на oci d для bin j, вказуючи, що rij + sij це абсолютне значення різниці між Boptd і d-й координатою барицентра bin j.

### *2.2.5 Фінальна модель*

Підсумовуючи попередні міркування, отримана модель MILP має вигляд:

$$\min \quad NC + \sum_{d\in D} k_d \sum_{j=1}^{n} \left( r_{jd} + s_{jd} \right) \tag{2.25}$$

$$\text{s.t.:} \quad \sum_{d\in D} \left( l_{ijd} + l_{jid} \right) + p_{ij} + p_{ji} \geq 1 \quad \forall i < j \in V$$

$$x_{id} - x_{jd} + W_d l_{ijd} \leq W_d - w_{id} \quad \forall i \neq j \in V,\ d \in D$$

$$x_{id} \leq W_d - w_{id} \quad \forall i \in V,\ d \in D$$

$$a_i - a_j + np_{ij} \leq n - 1 \quad \forall i \neq j \in V$$

$$1 \leq a_i \leq N \quad \forall i \in V$$

$$n\left( c_{ij} - 1 \right) \leq a_i - j \leq n\left( 1 - c_{ij} \right) \quad \forall i, j \in V$$

$$1 - \left( n+1 \right)\left( 1 - \delta_{ij} \right) \leq a_i - j \leq -1 + \left( n+1 \right)\left( 1 - \gamma_{ij} \right) \quad \forall i, j \in V$$

$$c_{ij} + \gamma_{ij} + \delta_{ij} = 1 \quad \forall i, j \in V$$

$$m_i W_d \left( c_{ij} - 1 \right) \leq \alpha_{ijd} - m_i \left( x_{id} + w_{id}/2 \right) \leq m_i W_d \left( 1 - c_{ij} \right) \quad \forall i, j \in V,\ d \in D$$

$$m_i W_d \left( c_{ij} - 1 \right) \leq \beta_{ijd} - m_i \left( B_d^{opt} - r_{jd} + s_{jd} \right) \leq m_i W_d \left( 1 - c_{ij} \right)$$
$$\forall i, j \in V,\ d \in D$$

$$\alpha_{ijd} + \beta_{ijd} \leq 2c_{ij} W_d m_i \quad \forall i, j \in V,\ d \in D$$

$$\sum_{i\in V} \alpha_{ijd} = \sum_{i\in V} \beta_{ijd} \quad \forall j \in V,\ d \in D$$

$$\text{var:} \quad a_j, N \in \mathbb{N},\ x_{id}, r_{jd}, s_{jd}, \alpha_{ijd}, \beta_{ijd} \in \mathbb{R}^+,\ l_{ijd}, p_{ij}, c_{ij}, \gamma_{ij}, \delta_{ij} \in \left\{ 0, 1 \right\}$$
$$\forall i, j \in V,\ d \in D$$

## 2.3 Метод рішення

У цьому розділі ми спочатку визначимо вхідні та вихідні дані для алгоритму з нашими припущеннями, а потім опишемо запропонований алгоритм. Алгоритм Largest Area First-Fit (LAFF). Алгоритм розміщує ящики з найбільшою поверхнею спочатку, мінімізуючи висоту від дна контейнера.

### *2.3.1 Вхідні дані*

Перший вхід – це кількість коробок різного розміру, позначених як N. Другий вхід – це розміри для кожного типу коробок різного розміру. Ми

позначаємо цей вхідний параметр чотирма значеннями (an, bn, cn, kn), де an – ширина, bn – глибина, cn – висота, а kn – кількість блоків для заданого розміру.

Залежно від того, з якого ракурсу дивитись на коробку, кожен вимір можна розглядати як ширину, висоту або глибину. Якщо прийняти bn як ширину, інші розміри an і cn також можна прийняти як висоту або глибину. Тому що коробка тривимірна, її можна обертати та розглядати з різних точок зору. Нижче наведено вхідні параметри запропонованого алгоритму:

Кількість коробок різного розміру : N

Ширина n-ї коробки: an

Глибина n-го коробки: bn

Висота n-ї коробки: cn

Номер n-ї коробки: kn

Вихідні дані

Після виконання алгоритму LAFF програма видає наступні дані:

O1: Об'єм контейнера

O2: Використаний простір (обсяг усіх розміщених коробок)

O3: Витрачений простір

O4: Час роботи

Час роботи роботи виражається в порядку мілісекунд.

*2.3.2 Як працює алгоритм?*

Як згадувалося раніше, проблема 3D-пакування є комбінаторною задачею з NP-hard складністю. Це означає, що оптимальні результати для вирішення проблеми можна знайти, спробувавши всі комбінації можливих різних рішень. Однак, якщо кількість ящиків збільшується, кількість ітерацій зросте настільки, що її неможливо розв’язати за поліноміальний час навіть на найшвидшому комп’ютері, доступному з поточною технологією. За деяких основних припущень і обмежень такі проблеми можуть бути вирішені за допомогою евристичних алгоритмів, які забезпечують рішення, близьке до оптимального.

Запропонований алгоритм називається Largest Area First-Fit. Він використовує евристику, яка розміщує ящики з найбільшою площею поверхні першими, мінімізуючи висоту від дна контейнера. Алгоритм працює наступним чином:

Перш за все, необхідно визначити ширину і глибину ємності. Враховуючи набір коробок різного розміру, ми обчислюємо ширину та глибину контейнера, знаходячи два найдовші краї заданих коробок. Ширина і глибина контейнера визначаються на початку алгоритму і залишаються фіксованими протягом усього виконання алгоритму. Ми вважаємо, що висота необмежена. Висота збільшується в міру виконання алгоритму.

Тобто ширину (ak) і глибину (bk) контейнера визначають шляхом вибору першого та другого найдовшого краю заданих блоків (ai, bi та ci). За ширину (ak) приймають найдовшу кромку, а за глибину (bk) – другу найдовшу кромку. Отже, ak і bk ніколи не змінюються.

Після визначення значень ширини та глибини контейнера дані коробки можна помістити в контейнер. У цьому алгоритмі використовуються два типи методів розміщення. Перший спосіб розміщення виділяє простір для ящика, що збільшує висоту контейнера. Другий метод розміщення розподіляє місце для решти ящиків, якщо є коробка, яка поміщається в доступний простір навколо розміщеної коробки, не перевищуючи висоту розміщеної коробки.

У першому методі розміщення визначаються коробки з найбільшою площею поверхні, і вибрані коробки опрацьовуються, щоб знайти коробку з мінімальною висотою з усіх вибраних квадратів. Потім в контейнер поміщається ящик з мінімальною висотою. Найбільша поверхня обраного ящика повинна бути паралельна дну ємності (рис. 2.2).

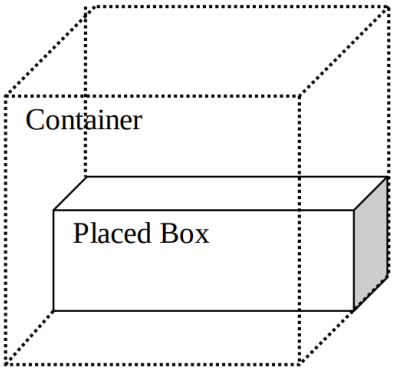
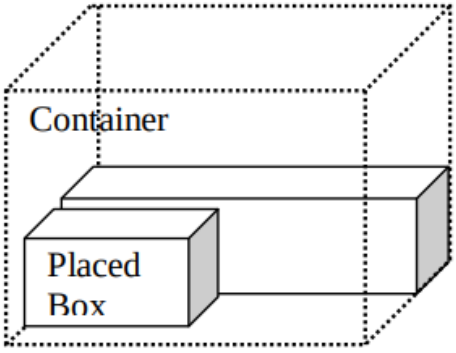



Рисунок 2.2 – Контейнер з коробкою

У другому методі розміщення алгоритм намагається виділити простір для решти коробок навколо коробки, розміщеної за допомогою першого методу. Тобто, якщо навколо розміщеного поля залишається деякий простір, як на рис. 1, алгоритм намагається заповнити цей простір за допомогою другого типу методу розміщення. У цьому методі, враховуючи, що розміщений ящик має розміри ai, bi та ci на рівні, тоді контейнер матиме два порожні місця навколо розміщеного ящика з розмірами ((ak-ai), bk, ci) і (ak, ( bk-bi), ci). Отже, якщо в цих порожніх місцях може бути поміщена будь-яка коробка, яка може бути одна або кілька, ми розміщуємо коробку, яка має максимальний об’єм із коробок, як показано на рис. 2. Ця ітерація триває до тих пір, поки не буде жодних коробок, які зможуть розташуватись в контейнері або всі коробки закінчились.

У другому методі розміщення, якщо навколо розміщеного поля немає місця, то алгоритм продовжує використовувати перший тип методу розміщення.



Щоразу, коли є ящики, які ще не розміщені, алгоритм продовжує використовувати перший тип методу розміщення і так далі, поки не будуть розміщені всі ящики. В кінці алгоритму виходить можливе рішення, показане на рис. 3.

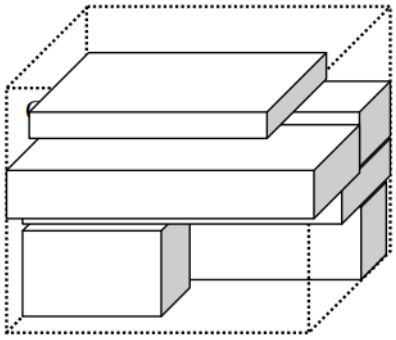


Рисунок 2.4 – Процес пакування
контейнеру

*2.3.3 LAFF алгоритм*

Нижче наведено основні кроки алгоритму LAFF:

Крок 1: введіть розміри та числа.

N: кількість унікальних коробок.

Крок 2: Визначте ширину (ak) і глибину (bk) контейнера.

ak : перший найдовший край усіх коробок.

bk : Другий найдовший край усіх коробок.

Ck : 0.

Крок 3: Виберіть коробку з найбільшою площею поверхні. Якщо їх більше одного, виберіть коробку з мінімальною висотою. Помістіть цю коробку (ith) на найбільшу поверхню паралельно основі контейнера.

Крок 3.1: Визначте висоту контейнера та зменшіть кількість і-го ящика.

ck=ck+ ci

ki=ki-1

Крок 3.2: Якщо кількість ящиків дорівнює нулю, тоді завершуємо алгоритм.

$$\sum_{p} k_p = 0 \tag{2.26}$$

Крок 3.3: Якщо простір (ak-ai)=0 і (bk-bi)=0, то перейдіть до кроку 3. В іншому випадку виберіть поле, яке вписується в цей простір. Якщо немає жодної коробки, яка поміщається в цей простір, перейдіть до кроку 3. Якщо в цей простір є більше одного ящика, виберіть коробку, яка має найбільший об'єм (jth box).

Крок 3.3.1: Визначте розміри простору.

as = ak–ai –aj ve bs = max(bk–bi, bk–bj )

або

as = max(ak–ai, ak–aj ) and bs = bk–bi–bj

Крок 3.3.2: Зменшити номер і-ої коробки.

kj=kj-1

Крок 3.3.3: якщо кількість ящиків дорівнює нулю, тоді завершити роботу.

В іншому випадку перейдіть до кроку 3.3.

$$\sum_{p} k_p = 0 \tag{2.27}$$

### *2.3.4 Складність LAFF алгоритму*

Якщо ми позначимо n як кількість усіх ящиків, які потрібно помістити в контейнер, а k як типи коробок різного розміру, час роботи алгоритму в найгіршому випадку буде виражено нижче:

T(n) = n(nk)

$T(n) = kn^2$, де к – константа

Отже, складність алгоритму є $O(n^2)$

# 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

## 3.1 Програмне забезпечення

При реалізації алгоритму, які вирішує задачу типу NP-Hard і потенційно буде обробляти великі обсяги інформації, потрібно провести аналіз предметної області та обрати технологічний стек для розробоки.

Важливим атрибутом фінального додатку є його продуктивність. Потрібно мінімізувати час обробки даних для випадків, коли обсяг вхідних даних є великим. При імплементації даного додатку до реальних систем, які будуть працювати, швидкість роботи додатку буде позитивно впливати на швидкість бізнес-процесів та автоматизації їх. Тому продуктивність напряму має відношення до якості програмного забезпечення.

Іншим вагомим атрибутом є фактор кросс-платформенності. Для того, щоб реальний користувач міг працювати з даним додатком на будь-якій системі, потрібно обрати мову програмування, яка має можливість компіляції коду для найбільш поширених операційних систем та платформ.

Проаналізувавши мови програмування та фреймфорки, було прийнято рішення використовувати мову програмування C++.

C++ це мова програмування, яка має статичну типізацію. Код написаний на цій мові компілюється в об’єктні файли, котрі зв’язуються з іншими файлами програми за допомогою лінкера і в результаті генерується exe файл. Є підтримка процедурного, загального, функціонального та об’єктно орієнтованого програмування.

Однією з характеристик мови програмування є її рівень абстракції. Наприклад самий нижній рівень це рівень сигналів Hardware. До мов програмування високого рівня можна віднести Python. Чим вище рівень абстракції, там менше розробник програми повинен описувати більш глибокі програмні процеси, але із-за цього транслятор генерує цей код сам і зазвичай, це не є максимально ефективним рішенням. Мова програмування C++ знаходиться по середині цього списку рівнів абстракцій. Він має я можливість

напряму працювати з пам’яттю через покажчики(pointer) та є багато високорівневого функціоналу, який пришвидшує та спрощує процес розробки.

C++ починаючи с 2011 року випускає стандарти кожні 3 роки, які доповнюють функціонал попередніх стандартів та проводить роботу над помилками для створення стабільного версії стандарту. На даний момент існує стандарт C++ 20, який ввів потужний функціонал, який всі розробники довго чекали(модульна система будування програми). Проте новий стандарт ще не став популярним, по пройшло не багато часу з релізу і немає багато відгуків та досвіду роботи з цим стандартом. Тож було вирішено використати стандарт C++ 17.

Як було сказано раніше, C++ - це компільована мова програмування. Існує багато компіляторів для цієї мови, але найпоширенішими є:

MinGW / GCC

Borland c++

Dev C++

Embracadero

Clang

Visual C++

Intel C++

Code Block

Кожен з даних компіляторів має як свої недоліки, так і позитивні сторони. Для наших цілей було обрано компілятор GCC(GNU Compile Collection), а саме GCC 11.1. Важливим є факт, що даний компілятор поширюється за допомогою ліцензії GNU, тобто він є безкоштовним для використання.

Ця версія компілятору встановлює режим дебагу за замовчуванням на DWARF 5 для більшості цілей(target) і змінює версію мови C++ за замовчуванням на -std=gnu++17. Він досягає значного прогресу в підтримці мови C++20, як на стороні компілятора, так і на стороні бібліотеки STL, додає

експериментальну підтримку C++23, деякі покращення C2X, різні покращення оптимізації та виправлення помилок, кілька нових змін і покращень устаткування. на серверні частини компілятора та багато інших змін.

Важливим кроком при розробці додатку є вибір середовища розробки. В нашому випадку був обраний Qt Creator 5.0.3.

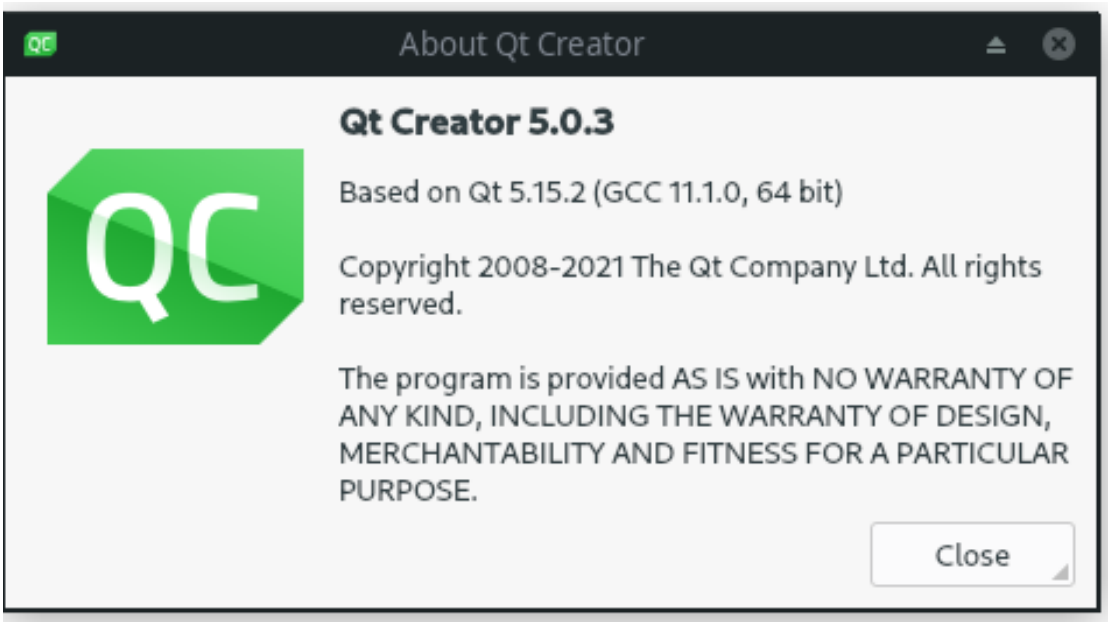
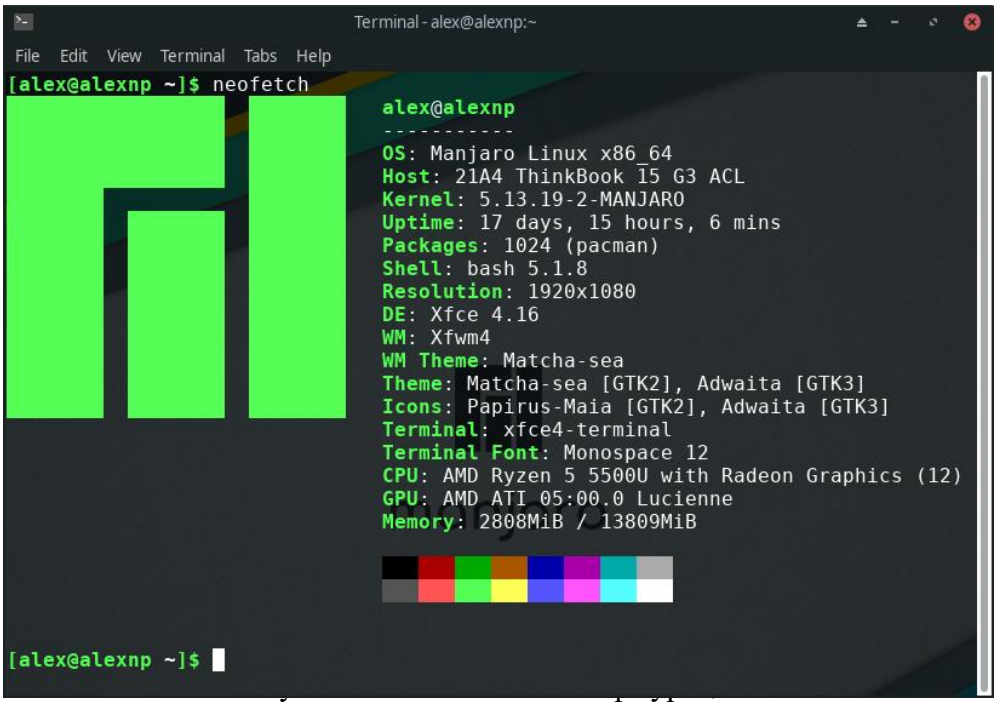



Рисунок 3.1 - Qt Creator 5.0.3

При розробці використовувалась така системна конфігурація :



### 3.2 Проектування архітектури додатку

Після вибору та налаштування середовища розробки, створимо UML діаграму класів для проектування архітектури нашої програми.

Опишемо поля та методи кожного класу окремо.

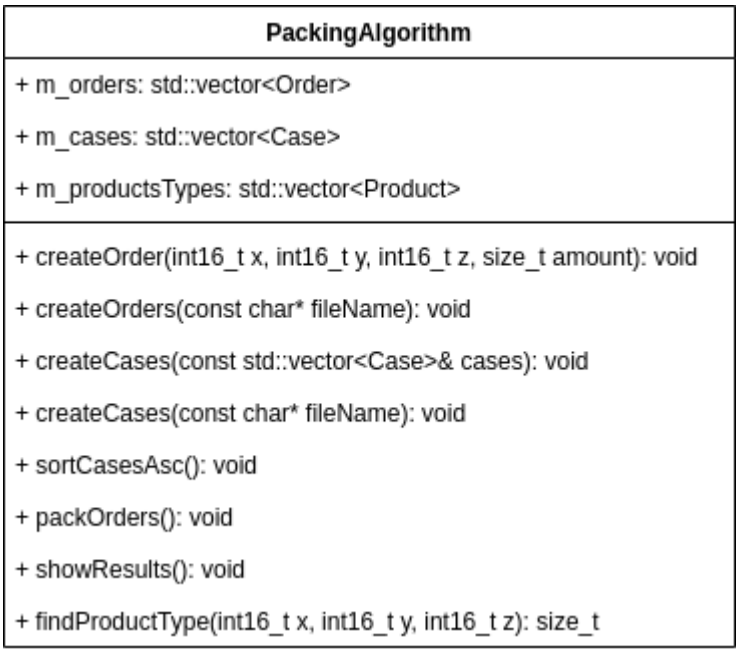



Рисунок 3.3 - Packing algorithm

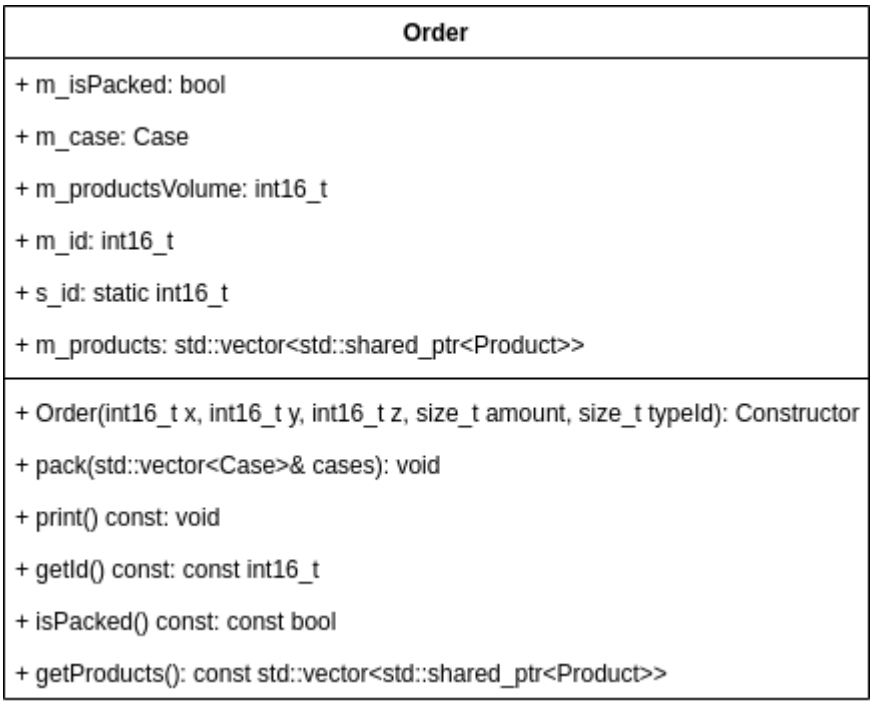



Рисунок 3.4 - Order

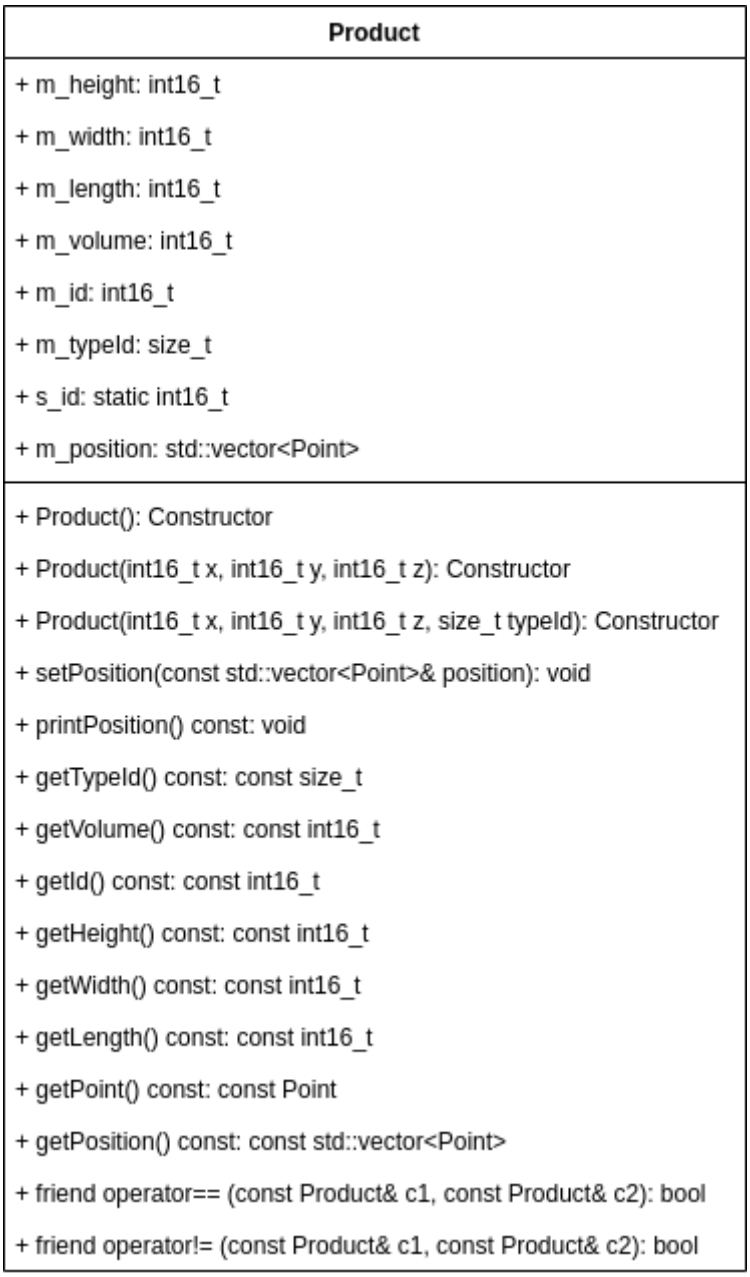



Рисунок 3.5 - Product

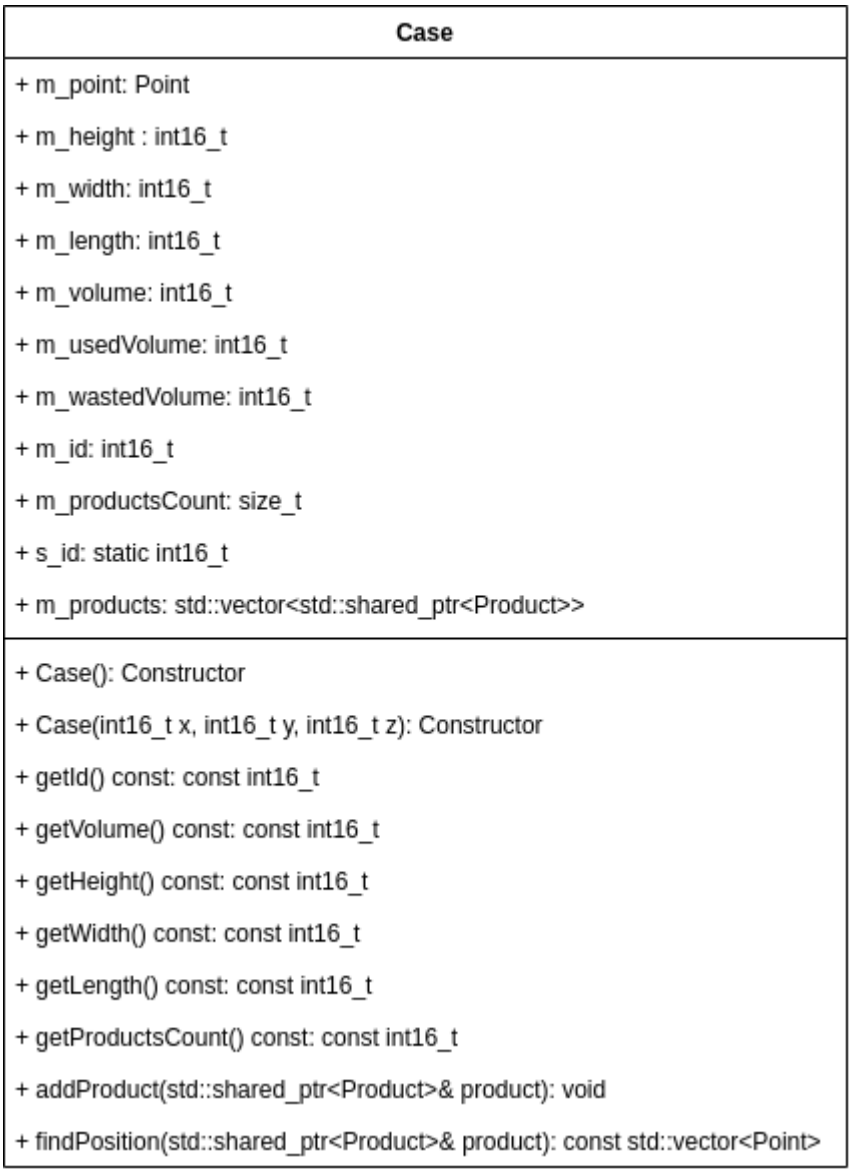



Рисунок 3.6 - Case

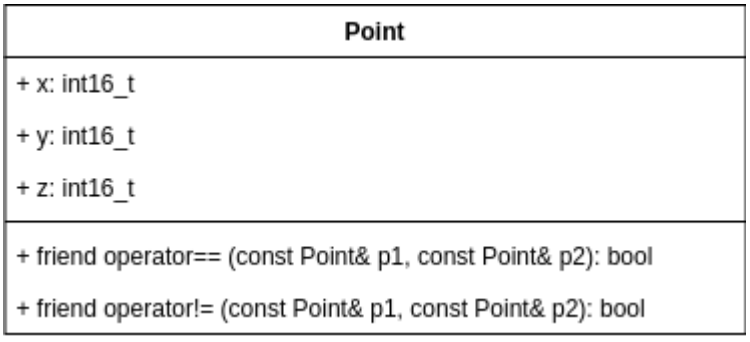



Рисунок 3.7 - Point

Встановимо зв’язки.

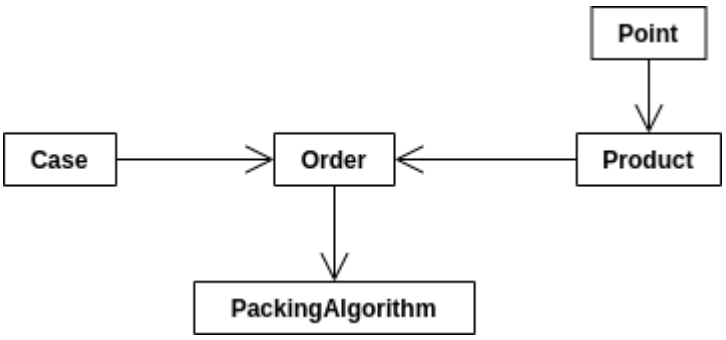



Рисунок 3.8 - Зв’язки

Наступним кроком після проектування класів є етап реалізації.

### 3.3 Реалізація алгоритму

В нашому випадку алгоритм буде отримувати вхідні данні в конструкторі у вигляді текстового файлу, або аргументом з іншого модулю. Для демонстрації роботи програми та тестування будемо використовувати вхідні дані у вигляді передачі конструктору текстових файлів.

Файл з функцією main виглядає так:

```
#include "stdafx.h"
#include "PackingAlgorithm.h"

int main(int argc, char** argv)
{
	namespace bpa = PackingAlgorithm;
	bpa::PackingAlgorithm pa;

	pa.createCases("input_cases.txt");
	pa.createOrders("input_orders.txt");

	pa.packOrders();
	pa.showResults();


	system("pause");
	return 0;
}
```

Підключаємо наш header file «PackingAlgorithm.h». В тілі функції main створюємо аліас для простору імен. Далі створюємо об’єкт нашого класу і визиваємо методи для створення кейсів та замовлень. Після ініціалізації даними нашого об’єкту визиваємо методи packOrders для пакування

замовлення та метод showResults для відображення результатів пакування. Після чого програма завершується.

Функція createOrders з аргументом типу const char* має вигляд:

```
void PackingAlgorithm::createOrders(const char* fileName)
{
	int16_t x = 0;
	int16_t y = 0;
	int16_t z = 0;
	int16_t amount = 0;
	std::ifstream file;
	file.exceptions(std::ifstream::badbit);

	try
	{
		file.open(fileName);
		while (!file.eof())
		{
			file >> x >> y >> z >> amount;
			if (x <= 0 || y <= 0 || z <= 0)
			{
				throw "One or more parameters while reading file <= 0";
			}
			m_orders.push_back(Order(x, y, z, amount, findProductType(x, y, z)));
		}
	}
	catch (const std::ifstream::failure& e)
	{
		std::cerr << "Exception opening/reading file" << std::endl;
	}
	catch (const char* exception)
	{
		std::cerr << "Error: " << exception << std::endl;
	}
	file.close();
}
```

Спочатку створюємо допоміжні змінні для зберігання розмірів, кількості контейнерів, екземпляр класу ifstrem для зчитування даних з файлу та встановлюємо флаг помилок для нього. Далі ми відкриваємо файл та зчитуємо в допоміжні змінні данні з файлу та за допомогою їх створюємо замовлення. Також описані ситуації, коли дані не можуть бути зчитані.

Створення замовлень реалізовано аналогічним способом. Повний лістинг програми знаходиться в додатку А.

Функція packOrders за допомогою циклу викликає відповідну функцію для кожного замовлення.

```
void PackingAlgorithm::packOrders()
{
	for (auto& it : m_orders)
		it.pack(m_cases);
}
```

Алгоритм пакування замовлення представлений функцією pack класу Order:

```
void Order::pack(std::vector<Case>& cases)
{
	try
	{
		for (auto it = cases.begin(); it != cases.end();)
		{
			if (m_productsVolume > it->getVolume())
			{
				++it;
				continue;
			}

			size_t productsInLevel  = it->getLength() / m_products[0]->getLength() *
			it->getWidth() / m_products[0]->getWidth();
			size_t amountOfLevels   = it->getHeight() / m_products[0]->getHeight();
			size_t maxProductsCount = productsInLevel * amountOfLevels;

			if (maxProductsCount >= m_products.size())
			{
				m_case = static_cast<Case>(*it);
				for (auto& product : m_products)
					m_case.addProduct(product);
				if (m_case.getProductsCount() == m_products.size())
					m_isPacked = true;
				cases.erase(it);
				break;
			}
			++it;
		}
		if (!m_isPacked)
			throw "Packing failed";
	}
	catch (const char* exception)
	{
		std::cerr << "Error: " << exception << '\n';
	}
}
```

Важливою функцією є функція addProduct, яка розміщує ящик в контейнер.

```
void Case::addProduct(std::shared_ptr<Product>& product)
{
	if (m_width  - m_point.x >= product->getWidth() &&
		m_length - m_point.y >= product->getLength())
	{
		product->setPosition(findPosition(product));
		m_products.push_back(product);
		m_point.x         += product->getWidth();
		m_usedVolume      += product->getVolume();
		m_wastedVolume    -= product->getVolume();
	}
	else if (m_length - m_point.y - product->getLength() >= product->getLength())
	{
		m_point.x = 0;
		m_point.y += product->getLength();
		product->setPosition(findPosition(product));
		m_products.push_back(product);
		m_point.x         =  product->getWidth();
		m_usedVolume      += product->getVolume();
		m_wastedVolume    -= product->getVolume();
	}
	else if (m_height - m_point.z - product->getHeight() >= product->getHeight())
	{
		product->setPosition(findPosition(product));
		m_products.push_back(product);
		m_point.x         =  product->getWidth();
		m_point.y         =  0;
		m_point.z         += product->getHeight();
		m_usedVolume      += product->getVolume();
		m_wastedVolume    -= product->getVolume();
	}
	++m_productsCount;
}
```

Після запуску тестового варіанту бачимо такий результат:

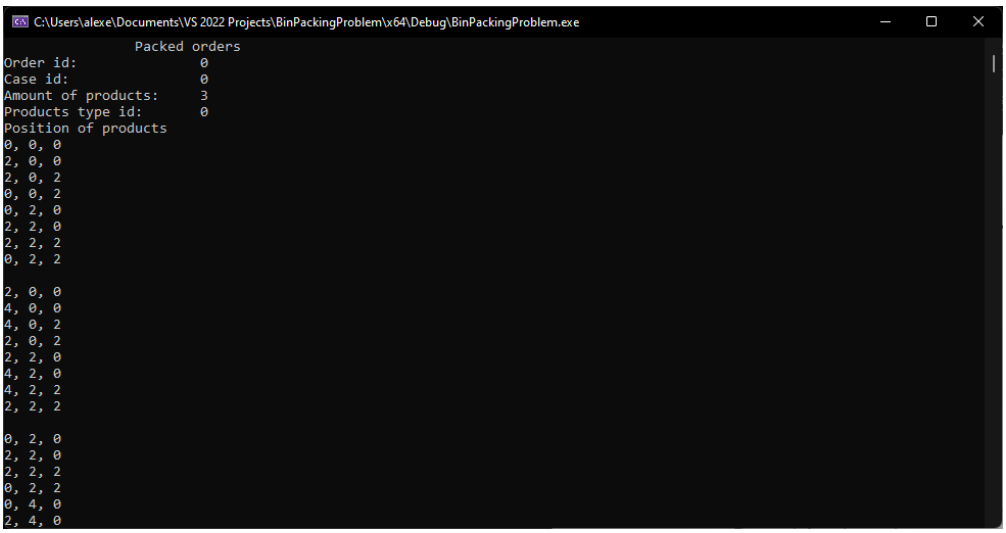



Рисунок 3.9 - Результат запуску

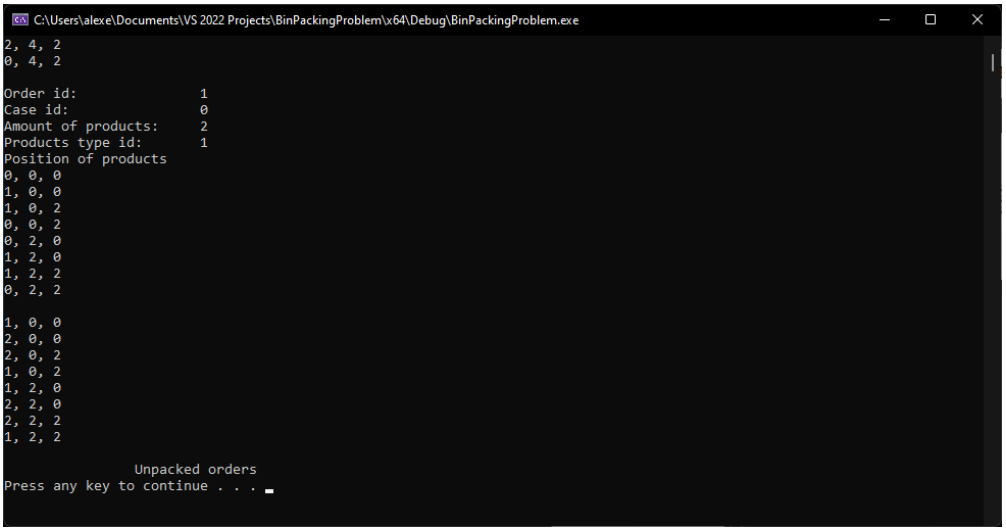



Рисунок 3.10 - Результат запуску

## 3.4 Тестування

Для тестування працездатності програми були написані Unit-тести. Тест для класу Case:

```
#include "pch.h"

using namespace PackingAlgorithm;

TEST(TestCase, ConstructorWithWidthLengthHeiht)
{
      int16_t x = 1, y = 1, z = 1;

      Case* c = new Case(x, y, z);

      ASSERT_EQ(x, c->getWidth());
      ASSERT_EQ(y, c->getLength());
      ASSERT_EQ(z, c->getHeight());
}

TEST(TestCase, AddOneProductToCase)
{
      std::shared_ptr<Product> products(std::make_shared<Product>(1, 1, 1, 0));
      int16_t x = 1, y = 1, z = 1;
      Case* c = new Case(x, y, z);

      c->addProduct(products);

      ASSERT_EQ(1, c->getProductsCount());
}
```

Тест для класу Product.

```
#include "pch.h"

using namespace PackingAlgorithm;

TEST(TestProduct, SetPosition)
{
    Product* product = new Product;
    std::vector<Point> position{ Point{1,1,1}, Point{ 2,2,2 } };

    product->setPosition(position);

    ASSERT_EQ(position, product->getPosition());
}

TEST(TestProduct, ConstructorWithWidthLengthHeightTypeId)
{
    int16_t x = 1, y = 1, z = 1;
    size_t typeId = 0;

    Product* product = new Product(x, y, z, typeId);

    ASSERT_EQ(x , product->getWidth());
    ASSERT_EQ(y , product->getLength());
    ASSERT_EQ(z , product->getHeight());
    ASSERT_EQ(typeId , product->getId());
}

TEST(TestProduct, ConstructorWithWidthLengthHeight)
{
    int16_t x = 1, y = 1, z = 1;

    Product* product = new Product(x, y, z);

    ASSERT_EQ(x, product->getWidth());
    ASSERT_EQ(y, product->getLength());
    ASSERT_EQ(z, product->getHeight());
}
```

Тест для класу Order.

```
TEST(TestOrder, Constructor)
{
    int16_t x = 1, y = 1, z = 1;
    size_t typeId = 0, amount = 3;

    Order* order = new Order(x, y, z, amount, typeId);

    ASSERT_EQ(x, order->getProducts()[0]->getWidth());
    ASSERT_EQ(y, order->getProducts()[0]->getLength());
    ASSERT_EQ(z, order->getProducts()[0]->getHeight());
    ASSERT_EQ(typeId, order->getProducts()[0]->getId());
    ASSERT_EQ(amount, order->getProducts().size());
}

TEST(TestOrder, PackingExpectedSuccessful)
{
    Case c(2, 2, 2);
    std::vector<Case> cases;
    cases.push_back(c);
    Order order(1, 1, 1, 8, 1);

    order.pack(cases);

    EXPECT_TRUE(order.isPacked());
}

TEST(TestOrder, PackingExpectedFailed)
{
    Case c(2, 2, 2);
    std::vector<Case> cases;
    cases.push_back(c);
    Order order(1, 1, 1, 9, 1);

    try
    {
        order.pack(cases);
    }
    catch (std::out_of_range const& err) {
        EXPECT_EQ(err.what(), std::string("Error: Packing failed"));
    }
    catch (...)
    {
        FAIL() << "Expected Error: Packing failed";
    }
```

Результати тестів, показали, що програма працює коректно.

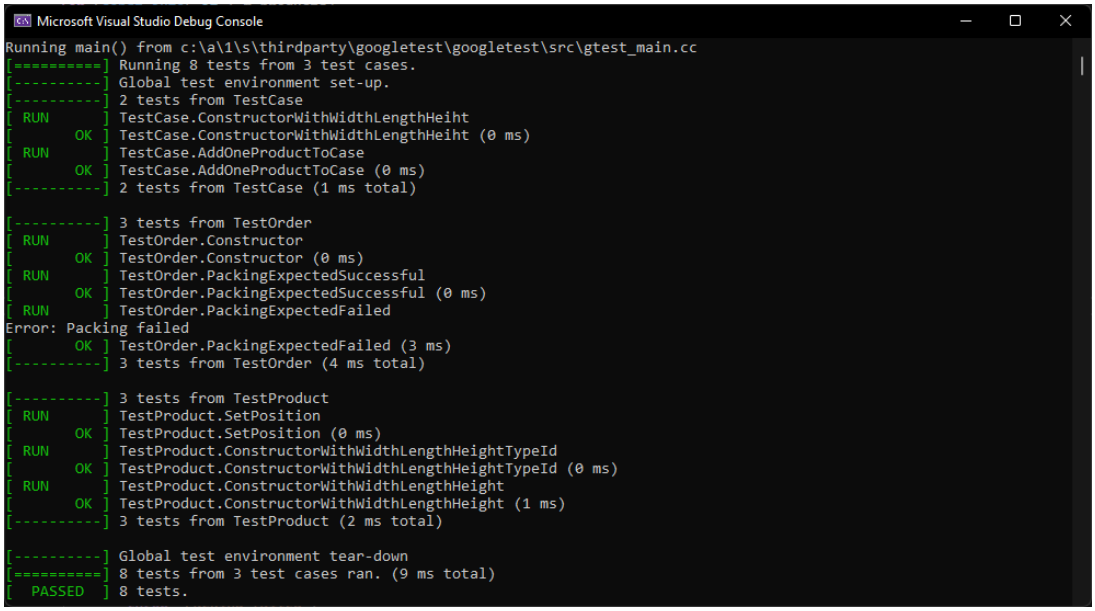



Рисунок 3.11 - Результати тестів

### 3.5 Аналіз результатів

Ми протестували програму з кількома коробками різного розміру, щоб побачити практичну роботу алгоритму. Програма генерує випадкові коробки з різними розмірами як вхідні дані, а потім поміщає коробки в контейнер. У таблиці 1 наведені результати експериментів. Перший стовпець (A) – це кількість різних типів коробок. Другий стовпець (B) – загальна кількість ящиків. Третій стовпець (C) – це об’єм усіх розміщених коробок. Четвертий стовпець (D) - це об'єм побудованого контейнера. Нарешті, останній стовпець (E) показує відсоток втраченого простору. Алгоритм намагається розмістити ящики найкращим чином. Обсяг втраченого простору значно низький. Як видно з таблиці, при збільшенні типів ящиків збільшиться і втрачений простір. Тому що помістити коробки різного розміру в контейнер важче, ніж коробки такого ж розміру.

Таблиця 3.1 - Аналіз результатів

| Кількість типів коробок | Кількість коробок | Об“єм коробок | Об“єм контейнеру | Втрати (%) |
|---|---|---|---|---|
| 1 | 10 | 8400 | 8400 | 0 |

| | | | | |
|---|---|---|---|---|
| 1 | 20 | 12480 | 12480 | 0 |
| 2 | 5 | 2547 | 2610 | 2.41 |
| 2 | 10 | 6252 | 6480 | 3.52 |
| 2 | 15 | 25554 | 25920 | 0.58 |
| 2 | 20 | 49032 | 49320 | 23.3 |
| 5 | 5 | 3910 | 5100 | 16.97 |
| 5 | 10 | 11359 | 13680 | 12.55 |
| 5 | 20 | 22596 | 25840 | 30.11 |
| 10 | 10 | 13419 | 19200 | 21.8 |
| 10 | 20 | 21694 | 27740 | 21.8 |
| 10 | 30 | 12854 | 16800 | 23.49 |

# ВИСНОВКИ

В магістерській роботі було досліджено застосування евристичних алгоритмів для вирішення проблеми оптимального пакування об’єктів в контейнери. Запропонований алгоритм використовує евристичну стратегію, яка розміщує ящики з найбільшою площею поверхні першими за рахунок мінімізації висоти від дна контейнера. Із запропонованого евристичного алгоритму та результатів програмної реалізації можна зробити такі висновки:

- Запропонований алгоритм є ефективним алгоритмом, який має асимптотичну складність порядку $O(n^2)$.
- Кількість втраченого простору також знаходиться в прийнятних межах.
- Ефективність цього алгоритму дає можливість його імплементації для реальних проектів.

В якості майбутньої роботи можна розглянути інші параметри, такі як вага ящиків, розподіл ваги в контейнері, використання кількох контейнерів, порядок відвантаження ящиків для розробки нових алгоритмів. Алгоритм, який був описаний, дає можливість вирішити реальні проблему бізнесу. Основна та наймасштабнішою проблема це пакування товарів для транспортування. Математична модель та сам алгоритм представляє теоретичний інтерес в підході до розв’язання NP-hard задач, які на даний момент не можуть бути розв’язані за поліноміальний.

# СПИСОК ЛІТЕРАТУРИ


1. Valério de Carvalho JM. Lp models for bin packing and cutting stock problems. Eur J Oper Res. 2002;141:253–73.
2. Coffman E, Leung J, Csirik J. Variants of classical one-dimensional bin packing. Handbook of approximation algorithms and metaheuristics. Taylor & Francis; 2007 May. p. 33.
3. Garey MR, Johnson DS. Computers and intractability: a guide to the theory of NP-completeness (series of books in the mathematical sciences). 1st ed., W. H. Freeman, editor. New York: W. H. Freeman and Company; 1979.
4. Gass SI, Harris CM. Encyclopedia of operations research and management science. J Oper Res Soc. 1997;48(7):759–60.
5. Martello S, Pisinger D, Toth P. New trends in exact algorithms for the 0–1 knapsack problem. Eur J Oper Res. 2000;123(2):325–32.
6. Gendreau M, Potvin J-Y. Metaheuristics in combinatorial optimization. Annals OR. 2005 Nov;140:189–213.
7. Zheng T, Luo W. An improved squirrel search algorithm for optimization. Complexity. 2019;2019:6291968. 10.1155/2019/6291968.
8. El-Ashmawi WH, Abd Elminaam DS. A modified squirrel search algorithm based on improved best fit heuristic and operator strategy for bin packing problem. Appl Soft Comput. 2019;82:105565.
9. Holland JH. Adaptation in natural and artificial systems. 2nd ed., Ann Arbor, MI: University of Michigan Press; 1975. p. 1992.
10. Yang XS. Nature-inspired metaheuristic algorithms. United Kingdom: Luniver press; 2010.
11. Quiroz M, Reyes LC, Torres-Jimenez J, Santillán C, Fraire-Huacuja H, Alvim A. A grouping genetic algorithm with controlled gene transmission for the bin packing problem. Comput Operat Res. 2014 Oct;55:52–64.
12. Qian B, Zhou H-B, Hu R, Xiang F-H. Hybrid differential evolution optimization for no-wait flow-shop scheduling with sequence-dependent setup times and

release dates. In: Huang De-S, Gan Y, Bevilacqua V, Figueroa JC, editors. Advanced intelligent computing. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 600–11.

13. Scholl A, Klein R, Jürgens C. Bison: A fast hybrid procedure for exactly solving the 1D BPP. Comput Operat Res. 1997;24(7):627–45.
14. Luo F, Scherson ID, Fuentes J. A novel genetic algorithm for bin packing problem in jmetal. In 2017 IEEE International Conference on Cognitive Computing (ICCC); 2017. p. 17–23.
15. Tlili T, Krichen S. On solving the double loading problem using a modified particle swarm optimization. Theor Comput Sci. 2015;598:118–28.
16. Tadei R, Crainic TG, Perboli G. Ts2pack: A two-level tabu search for the three-dimensional bin packing problem. 2009 Jun;195:744–60.
17. Ezugwu AE, Shukla AK, Nath R, Akinyelu AA, Agushaka JO, Chiroma H, et al. Metaheuristics: a comprehensive overview and classification along with bibliometric analysis. Artif Intell Rev. 2021. 10.1007/s10462-020-09952-0.
18. A. Al-Herz and A. Pothen. A 2/3-approximation algorithm for vertex-weighted matching.iscrete Applied Mathematics, 2019. (in press). J. Balogh, J. Békési, G. Dósa, L. Epstein, and A. Levin. A new and improved algorithm for online bin packing. In Proceedings of the 26th Annual European Symposium on Algorithms (ESA), volume 112 of LIPIcs, pages 5:1–5:14, 2018.
19. J. Balogh, J. Békési, G. Dósa, L. Epstein, and A. Levin. A new lower bound for classic online bin packing. In Approximation and Online Algorithms - 17th International Workshop, (WAOA), volume 11926 of Lecture Notes in Computer Science, pages 18–28. Springer, 2019.
20. J. Boyar, G. Dósa, and L. Epstein. On the absolute approximation ratio for first fit and related results. Discret. Appl. Math., 160(13-14):1914–1923, 2012.
21. M. G. Christ, L. M. Favrholdt, and K. S. Larsen. Online bin covering: Expectations vs. guarantees. Theor. Comput. Sci., 556:71–84, 2014.

22. H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali. Approximation and online algorithms for multidimensional bin packing: A survey. Comput. Sci. Rev., 24:63–79, 2017.

23. E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: survey and classification. In Handbook of combinatorial optimization, pages 455–531. Springer New York, 2013

24. W. F. de la Vega and G. S. Lueker. Bin packing can be solved within 1+epsilon in linear time. Combinatorica, 1(4):349–355, 1981.

25. G. Dósa and J. Sgall. First fit bin packing: A tight analysis. In 30th International Symposium on Theoretical Aspects of Computer Science (STACS), volume 20 of LIPIcs, pages 538–549, 2013.

26. FG. Dósa and J. Sgall. Optimal analysis of best fit bin packing. In Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP), pages 429–441, 2014.

27. C. Fischer and H. Röglin. Probabilistic analysis of the dual next-fit algorithm for bin covering. In LATIN 2016: Theoretical Informatics - 12th Latin American Symposium, pages 469–482, 2016.

28. C. Fischer and H. Röglin. Probabilistic analysis of online (class-constrained) bin packing and bin covering. In LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, volume 10807 of Lecture Notes in Computer Science, pages 461–474. Springer, 2018.

# ДОДАТОК А

stdafx.h

```
#pragma once

#include <iostream>
#include <list>
#include <vector>
#include <cstdint>
#include <algorithm>
#include <fstream>
```

Product.h

```
#pragma once

#include "stdafx.h"

namespace PackingAlgorithm
{
	struct Point
	{
		int16_t x = 0;
		int16_t y = 0;
		int16_t z = 0;
		friend bool operator== (const Point& p1, const Point& p2);
		friend bool operator!= (const Point& p1, const Point& p2);
	};

	class Product
	{
	public:
		Product();
		Product(int16_t x, int16_t y, int16_t z);
		Product(int16_t x, int16_t y, int16_t z, size_t typeId);
		void                    setPosition(const std::vector<Point>& position);
		void                    printPosition() const;
		const   size_t  getTypeId() const;
		const   int16_t getVolume() const;
		const   int16_t getId() const;
		const   int16_t getHeight() const;
		const   int16_t getWidth() const;
		const   int16_t getLength() const;
		const   Point   getPoint() const;
		const   std::vector<Point> getPosition() const;
		friend bool operator== (const Product& c1, const Product& c2);
		friend bool operator!= (const Product& c1, const Product& c2);

	private:
		int16_t m_height        = 0;
		int16_t m_width         = 0;
		int16_t m_length        = 0;
		int16_t m_volume        = 0;
		int16_t m_id            = 0;
		size_t  m_typeId        = 0;
		static int16_t          s_id;
		std::vector<Point>      m_position;
	};
}
```

PackingAlgorithm.h

```
#pragma once

#include "Order.h"

namespace PackingAlgorithm
{
	class PackingAlgorithm
	{
	public:
		void createOrder(int16_t x, int16_t y, int16_t z, size_t amount);
		void createOrders(const char* fileName);
		void createCases(const std::vector<Case>& cases);
		void createCases(const char* fileName);
		void sortCasesAsc();
		void packOrders();
		void showResults();
		size_t findProductType(int16_t x, int16_t y, int16_t z);

	private:
		std::vector<Order>		m_orders;
		std::vector<Case>		m_cases;
		std::vector<Product>	m_productsTypes;
	};
}
```

Order.h

```
#pragma once

#include "stdafx.h"
#include "Case.h"
#include "Product.h"

namespace PackingAlgorithm
{
	class Order
	{
	public:
		Order(int16_t x, int16_t y, int16_t z, size_t amount, size_t typeId);
		void			pack(std::vector<Case>& cases);
		void			print() const;
		const int16_t	getId() const;
		const bool		isPacked() const;
		const std::vector<std::shared_ptr<Product>>	getProducts() const
		{
			return m_products;
		}

	private:
		bool			m_isPacked = false;
		Case			m_case;
		int16_t			m_productsVolume = 0;
		int16_t			m_id = 0;
		static int16_t	s_id;
		std::vector<std::shared_ptr<Product>> m_products;
	};
}
```

## Case.h

```
#pragma once

#include "stdafx.h"
#include "Product.h"

namespace PackingAlgorithm
{
	class Case
	{
	public:
		Case();
		Case(int16_t x, int16_t y, int16_t z);
		const int16_t   getId() const;
		const int16_t   getVolume() const;
		const int16_t   getHeight() const;
		const int16_t   getWidth() const;
		const int16_t   getLength() const;
		const size_t    getProductsCount() const;
		void                    addProduct(std::shared_ptr<Product>& product);
		const std::vector<Point> findPosition(std::shared_ptr<Product>& product);

	private:
		Point   m_point;
		int16_t m_height               = 0;
		int16_t m_width                = 0;
		int16_t m_length               = 0;
		int16_t m_volume               = 0;
		int16_t m_usedVolume    = 0;
		int16_t m_wastedVolume  = 0;
		int16_t m_id                   = 0;
		size_t  m_productsCount = 0;
		static int16_t s_id;
		std::vector<std::shared_ptr<Product>> m_products;
	};
}
```

## Product.cpp

```
#include "Product.h"

namespace PackingAlgorithm
{
	int16_t Product::s_id = 0;

	bool operator== (const Product& p1, const Product& p2)
	{
		return (p1.getPoint() == p2.getPoint());
	}

	bool operator!= (const Product& p1, const Product& p2)
	{
		return !(p1 == p2);
	}

	bool operator== (const Point& p1, const Point& p2)
	{
		return (p1.x == p2.x && p1.y == p2.y && p1.z == p2.z);
	}

	bool operator!= (const Point& p1, const Point& p2)
	{
		return !(p1 == p2);
	}
	Product::Product()
	{
	}
	Product::Product(int16_t x, int16_t y, int16_t z) :
		m_height(z), m_width(x), m_length(y)
	{
	}
	Product::Product(int16_t x, int16_t y, int16_t z, size_t typeId) :
		m_height(z), m_width(x), m_length(y), m_typeId(typeId)
	{
	}
	const size_t Product::getTypeId() const
	{
		return m_typeId;
	}
	const int16_t Product::getVolume() const
	{
		return m_volume;
	}
	const int16_t Product::getId() const
	{
		return m_id;
	}
```

```
	const int16_t Product::getHeight() const
	{
		return m_height;
	}
	const int16_t Product::getWidth() const
	{
		return m_width;
	}
	const int16_t Product::getLength() const
	{
		return m_length;
	}
	const Point Product::getPoint() const
	{
		return Point{ m_width, m_length, m_height };
	}
	const std::vector<Point> Product::getPosition() const
	{
		return m_position;
	}
	void Product::setPosition(const std::vector<Point>& position)
	{
		m_position = position;
	}
	void Product::printPosition() const
	{
		for (const auto& it : m_position)
			std::cout << it.x << ", " << it.y << ", " << it.z << std::endl;
		std::cout << "\n";
	}
}
```

## PackingAlgorithm.cpp

```
#include "PackingAlgorithm.h"

namespace PackingAlgorithm
{
	void PackingAlgorithm::createOrder(int16_t x, int16_t y, int16_t z, size_t amount)
	{
		try
		{
			if (x <= 0 || y <= 0 || z <= 0)
			{
				throw "One or more parameters <= 0";
			}
			m_orders.push_back(Order(x, y, z, amount, findProductType(x, y, z)));
		}
		catch (const char* exception)
		{
			std::cerr << "Error: " << exception << std::endl;
		}
	}

 void PackingAlgorithm::createOrders(const char* fileName)
 {
	int16_t x = 0;
	int16_t y = 0;
	int16_t z = 0;
	int16_t amount = 0;
	std::ifstream file;
	file.exceptions(std::ifstream::badbit);

	try
	{
		file.open(fileName);
		while (!file.eof())
		{
			file >> x >> y >> z >> amount;
			if (x <= 0 || y <= 0 || z <= 0)
			{
				throw "One or more parameters while reading file <= 0";
			}
			m_orders.push_back(Order(x, y, z, amount, findProductType(x, y, z)));
		}
	}
	catch (const std::ifstream::failure& e)
	{
		std::cerr << "Exception opening/reading file" << std::endl;
	}
	catch (const char* exception)
	{
		std::cerr << "Error: " << exception << std::endl;
	}
	file.close();
 }
```

```
void PackingAlgorithm::createCases(const std::vector<Case>& cases)
{
	m_cases = cases;
	sortCasesAsc();
}
void PackingAlgorithm::createCases(const char* fileName)
{
	int16_t x = 0;
	int16_t y = 0;
	int16_t z = 0;
	std::ifstream file;
	file.exceptions(std::ifstream::badbit);

	try
	{
		file.open(fileName);
		while (!file.eof())
		{
			file >> x >> y >> z;
			if (x <= 0 || y <= 0 || z <= 0)
			{
				throw "One or more parameters while reading file <= 0";
			}
			m_cases.push_back(Case(x, y, z));
		}
		sortCasesAsc();
	}
	catch (const std::ifstream::failure& e)
	{
		std::cerr << "Exception opening/reading file" << std::endl;
	}
	catch (const char* exception)
	{
		std::cerr << "Error: " << exception << std::endl;
	}
	file.close();
}
void PackingAlgorithm::sortCasesAsc()
{
	sort(m_cases.begin(), m_cases.end(), [](const auto first, const auto second)
		{
			return first.getVolume() < second.getVolume();
		});
}

	void PackingAlgorithm::packOrders()
	{
		for (auto& it : m_orders)
			it.pack(m_cases);
	}
	void PackingAlgorithm::showResults()
	{
		std::cout << "\t\tPacked orders" << std::endl;
		for (const auto& it : m_orders)
			if (it.isPacked())
				it.print();

		std::cout << "\t\tUnpacked orders" << std::endl;
		for (const auto& it : m_orders)
			if (!(it.isPacked()))
				std::cout << "Order id: \t" << it.getId() << std::endl;
	}
	size_t PackingAlgorithm::findProductType(int16_t x, int16_t y, int16_t z)
	{
		auto it = std::find(m_productsTypes.begin(), m_productsTypes.end(), Product(x, y, z));
		if (it != m_productsTypes.end())
		{
			auto idx = std::distance(m_productsTypes.begin(), it);
			return static_cast<size_t>(idx);
		}
		else
		{
			m_productsTypes.push_back(Product(x, y, z));
			auto it = std::find(m_productsTypes.begin(), m_productsTypes.end(), Product(x, y, z));
			auto idx = std::distance(m_productsTypes.begin(), it);
			return static_cast<size_t>(idx);
		}
	}
}
```