Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи вирішення задачі прямокутного гільйотинного розкрою листового матеріалу
(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-4
Анастасія КОНОНЕНКО
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Георгій ІВАЩЕНКО
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ ______________ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)

«____«_______________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Кононенко Анастасії Ігорівні
(прізвище, ім'я, по батькові)

1. Тема роботи Методи вирішення задачі прямокутного гільйотинного розкрою листового матеріалу

затверджена наказом по університету від « 21 « квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) задача гільйотинного розкрою листового матеріалу;
2) алгоритми комбінаторики;
3) методи обчислювального інтелекту;
4) середовище розробки Visual Studio 2022;
5) мова програмування JavaScript.

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз предметної області;
2) розгляд алгоритмів рішення задачі гільйотинного розкрою;
3) вибір технологій розробки та інструментальних засобів;
4) реалізація програмного застосунку;
5) аналіз результатів досліджень;
6) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 15 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Огляд алгоритмів рішення задачі гільйотин-	22.04.25-29.04.25	
	ного розкрою		
2	Вибір та обґрунтування методики дослідження	30.04.25-05.05.25	
3	Вибір інструментальних засобів	06.05.25-09.05.25	
4	Розробка алгоритмічного забезпечення	10.05.25-21.05.25	
5	Проведення експериментів	22.05.25-02.06.25	
6	Оформлення матеріалів кваліфікаційної роботи	03.06.25-05.06.25	
7	Подання кваліфікаційної роботи керівникові та	06.06.25-09.06.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	10.06.25-12.06.25	
	вання		

Дата видачі завдання « 21 » квітня 20 25 р.

Здобувач ____________________
(підпис)

Керівник роботи ____________________ доц. Георгій ІВАЩЕНКО
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 129 с., 9 рис., 5 табл., 2 дод., 30 джерел.

ЛИСТОВИЙ МАТЕРІАЛ, ГІЛЬЙОТИННИЙ РОЗКРІЙ, КОЕФІЦІЄНТ ВИКОРИСТАННЯ МАТЕРІАЛУ, РІВНЕВІ АЛГОРИТМИ, ЖАДІБНИЙ АЛГОРИТМ, ГЕНЕТИЧНИЙ АЛГОРИТМ, МУРАШИНИЙ АЛГОРИТМ, АЛГОРИТМ ІМІТАЦІЇ ВІДПАЛУ, ЕВРИСТИКА, МЕТАЕВРИСТИКА, СЕЛЕКЦІЯ, КРОСОВЕР, МУТАЦІЯ.

Метою кваліфікаційної роботи є дослідження евристичних та метаевристичних алгоритмів при вирішенні задач гільйотинного розкрою.

У ході виконання кваліфікаційної роботи був проведений аналіз предметної області, існуючих рішень, їх переваг та недоліків. Було розроблено та досліджено методи для вирішення задачі гільйотинного розкрою на основі генетичного, мурашиного та жадібного алгоритму, а також на основі методу симуляції відпалу. Проведено дослідження впливу основних параметрів метаевристичних алгоритмів, таких як кількість ітерацій та кількість мурах у мурашиному алгоритмі, розмір популяції та різні типи кроссоверів у генетичному алгоритмі. Також у ході виконання роботи розроблено тестове програмне забезпечення з графічним інтерфейсом користувача, яке виконує гільйотинний розкрій листового матеріалу на заданих користувачем вхідних даних за допомогою обраних алгоритмів.

ABSTRACT

Master’s thesis: 129 pages, 9 figures, 5 tables, 2 appendices, 30 sources.

SHEET MATERIAL, GUILLOTINE CUTTING, MATERIAL UTILIZATION RATIO, LEVEL ALGORITHMS, GREEDY ALGORITHM, GENETIC ALGORITHM, ANT COLONY ALGORITHM, SIMULATED ANNEALING ALGORITHM, HEURISTIC, METAHEURISTIC, SELECTION, CROSSOVER, MUTATION.

The major goal of this thesis is to investigate heuristic and metaheuristic algorithms for solving guillotine cutting problems.

In order to achieve this goal, the thesis includes an analysis of the subject area, existing solutions, and their advantages and disadvantages. The thesis studied methods based on genetic and ant colony algorithms and greedy algorithms (simulated annealing) to solve the guillotine-cutting problem. A study of the impact of parameters of metaheuristic algorithms, such as the number of iterations and the number of ants in the ant colony algorithm, as well as the population size and different types of crossovers in the genetic algorithm, was performed. Test software with a graphical user interface was also created, which performs guillotine cutting of sheet materials based on user-provided input data using the selected algorithms.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 8
ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ 10
1.1 Опис предметної області 10
1.2 Класифікація задач розкрою 11
1.3 Огляд методів вирішення RGCSP 15
1.4 Аналіз існуючих рішень та наукових досліджень 17
1.5 Математична модель та постановка задачі 19
2 АНАЛІЗ ВИКОРИСТОВУВАНИХ АЛГОРИТМІВ 22
2.1 Евристичний підхід із стратегією Best Fit Decreasing 22
2.2 Метаевристичні алгоритми 24
2.2.1 Генетичний алгоритм 26
2.2.2 Мурашиний алгоритм 28
2.2.3 Симуляція відпалу 30
3 ПРОГРАМНА РЕАЛІЗАЦІЯ 33
3.1 Архітектура проєкту 33
3.2 Модель представлення вихідних даних 34
3.3 Реалізація алгоритмів рішення RGCSP 37
3.3.1 Алгоритм Best Fit Decreasing 37
3.3.2 Генетичний алгоритм 39
3.3.3 Мурашиний алгоритм 42
3.3.4 Симуляція відпалу 43
3.3.5 Гібридний алгоритм 45
4 АНАЛІЗ ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ 46
4.1 Тестові набори та критерії ефективності роботи алгоритмів 46
4.2 Дослідження роботи генетичного алгоритму 47
4.2.1 Дослідження впливу кількості ітерацій 48

4.2.2 Дослідження впливу налаштувань кросоверу 50
4.3 Дослідження роботи мурашиного алгоритму 52
4.3.1 Дослідження впливу кількості ітерацій 53
4.3.2 Дослідження впливу кількості мурах 55
4.4 Порівняння результатів роботи усіх алгоритмів 57
ВИСНОВКИ 61
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 63
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 66
ДОДАТОК Б Вихідний код застосунку 75
Б.1 Реалізація алгоритмів розкрою 75
Б.1.1 geneticAlgorithm.js 75
Б.1.2 BestFit.js 91
Б.1.3 antColonyAlgorithm.js 93
Б.1.4 simulatedAnnealing.js 97
Б.1.5 hybridGeneticAlgorithm.js 103
Б.2 Вихідний код класів 104
Б.2.1 Класс Block 104
Б.2.2 Класс Sheet 105
Б.3 Додаткові модулі 105
Б.3.1 Модуль details.js 105
Б.3.2 Модуль canvas.js 109
Б.3.3 Модуль demo.js 111
Б.4 Код сторінки та її стилізація 121
Б.4.1 Файл index.html 121
Б.4.2 CSS файл 123

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ГА – генетичний алгоритм

ACO – алгоритм оптимізації колоній мурах (англ., Ant Colony Optimization)

BFD – кращий підходящий за спаданням (англ., Best Fit Decreasing)

BPP – задача упаковки в контейнери (англ., Bin Packing Problem)

CSS – каскадні таблиці стилів (англ., Cascading Style Sheets)

C&P – розкрій і упаковка (англ., Cutting & Packing)

HTML – мова гіпертекстової розмітки (англ., HyperText Markup Language)

JS – мова програмування Java Script

MVC – Модель-Відображення-Контролер (англ., Model-View-Controller)

RGCSP – задача прямокутного гільйотинного розкрою листового матеріалу (англ., Rectangular Guillotine Cutting Stock Problem)

SPA – односторінковий застосунок (англ., Single-page application)

VS Code – редактор коду Visual Studio Code

2D-GCSP – двовимірний прямокутний ортогональний гільйотинний розкрій (англ., Two-Dimensional Rectangular Guillotine Cutting Stock Problem)

ВСТУП

Оптимізація виробничих процесів є актуальною проблемою сучасної промисловості, оскільки сприяє підвищенню ефективності використання матеріалів і зниженню витрат. Серед багатьох напрямів оптимізації важливим є вирішення проблеми гільйотинного розкрою листового матеріалу, що виникає у металургійній, деревообробній, текстильній і скляній промисловості. Гільйотинний розкрій передбачає розділення листового матеріалу на прямокутні заготовки за допомогою прямолінійних різів, що забезпечує технологічну зручність процесу, проте потребує пошуку оптимального розкрою.

Задача оптимального розкрою належить до класу NP-складних комбінаторних задач, що унеможливлює отримання точного розв'язку за поліноміальний час для випадків великої розмірності [1]. Це обґрунтовує необхідність використання наближених методів, зокрема евристичних та метаевристичних підходів, серед яких жадібні алгоритми, алгоритми мурашиних колоній, генетичні алгоритми, алгоритми імітації відпалу, ройовий інтелект та еволюційні стратегії. Жоден з цих методів не забезпечує універсального розв'язку для всіх типів задач, тому розробка комбінованих (гібридних) методів є актуальним напрямом досліджень.

Метою дослідження є розробка нового гібридного методу розв'язання задачі прямокутного гільйотинного розкрою листового матеріалу, що забезпечить мінімізацію відходів та покращення якості отриманих розв'язків. Дослідження спрямоване на аналіз існуючих методів, їх порівняльний аналіз та розробку комбінованого підходу на основі синтезу ефективних алгоритмів. Отримані результати можуть бути використані в системах автоматизованого розкрою, що сприятиме підвищенню економічної ефективності та екологічної безпеки виробничих процесів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис предметної області

Прямокутний гільйотинний розкрій листового матеріалу є важливою технологічною задачею у багатьох галузях промисловості, включаючи металообробку, деревообробку, виробництво скла, текстильну та пакувальну індустрії. Суть цього процесу полягає у розділенні вихідного листа матеріалу на менші прямокутні заготовки шляхом послідовних розрізів, які проводяться паралельно сторонам листа. Кожен розріз виконується від одного краю заготовки до іншого, що нагадує принцип роботи гільйотини, звідки й назва методу. Головною метою такого розкрою є максимально ефективне використання площі листа для зменшення відходів виробництва [2].

Задача гільйотинного розкрою є типовою задачею комбінаторної оптимізації, оскільки передбачає знаходження оптимального способу розташування деталей на листі матеріалу з урахуванням заданих обмежень. Основними критеріями ефективності цього процесу є коефіцієнт використання матеріалу, який характеризує ефективність розміщення деталей, швидкість виконання розкрою, яка залежить від оптимальної послідовності різань і впливає на загальну продуктивність, а також кількість використаних аркушів матеріалу, що безпосередньо впливає на собівартість виробництва.

Технологічний процес гільйотинного розкрою має ряд особливостей, які відрізняють його від інших методів розрізання матеріалів. На відміну від довільного розкрою, де деталі можуть розташовуватися під будь-яким кутом, гільйотинний розкрій передбачає строго ортогональне різання, що накладає певні обмеження на можливі варіанти розміщення елементів. Крім того, послідовність розрізів є важливою, оскільки кожен наступний крок залежить від попереднього. Неправильно обраний порядок різань може призвести до значного збільшення відходів або неможливості вирізати всі необхідні деталі.

Основним критерієм оптимальності гільйотинного розкрою є мінімізація відходів матеріалу. Відходи можуть утворюватися у вигляді обрізків, які залишаються після вирізання всіх деталей, або у вигляді неповного використання площі листа. Для досягнення максимальної ефективності необхідно враховувати не тільки геометричні параметри деталей, але й технологічні обмеження, такі як мінімально допустима ширина відходів, можливість повороту деталей на 90 градусів, обмеження на кількість різань тощо. Гільйотинний розкрій знаходить широке застосування у виробництві, де використання матеріалів становить значну частину собівартості продукції. Наприклад, у металообробці він застосовується для розкрою сталевих, алюмінієвих та інших листів при виготовленні деталей для машинобудування, будівництва чи авіаційної промисловості. У деревообробній галузі цей метод використовується для розпилу фанерних плит, ДСП та МДФ при виробництві меблів. У текстильній промисловості гільйотинний розкрій дозволяє ефективно розрізати тканину для пошиття одягу, а у скляній індустрії – нарізати скло за заданими розмірами [3].

Економічний ефект від оптимізації гільйотинного розкрою може бути дуже значним, особливо у масовому виробництві. Навіть невелике підвищення ефективності розміщення деталей дозволяє суттєво зменшити витрати на сировину. Крім того, зниження обсягів відходів сприяє екологічній відповідальності підприємств, оскільки зменшується кількість матеріалів, що підлягають утилізації. Окрім економічних аспектів, важливим фактором є автоматизація процесу розкрою. Сучасні програмні комплекси дозволяють знаходити оптимальні схеми розміщення деталей за допомогою математичних алгоритмів, що значно прискорює процес підготовки виробництва.

1.2 Класифікація задач розкрою

Задачі розкрою листового матеріалу відрізняються між собою за різними характеристиками, що визначають їх складність та підходи до вирішення. Ці

відмінності можна систематизувати за кількома основними критеріями, які охоплюють як геометричні параметри, так і технологічні особливості процесу розкрою.

За розмірністю задачі розкрою поділяються на одновимірні, двовимірні та тривимірні [1, 4]. Одновимірний розкрій призначений для різання матеріалів у вигляді стрічок або прутків, де основним завданням є поділ заготовки на елементи заданої довжини. Такий тип розкрою поширений при нарізці труб, профілів або інших довгих виробів. Двовимірний розкрій передбачає роботу з плоскими листами, коли необхідно розмістити на них двовимірні фігури. Це найбільш поширений випадок у промисловості, що включає розкрій металевих листів, фанери, скла чи тканин. Тривимірний розкрій використовується рідше і пов'язаний з оптимізацією розташування об'єктів у просторі, що актуально для пакування або логістичних задач.

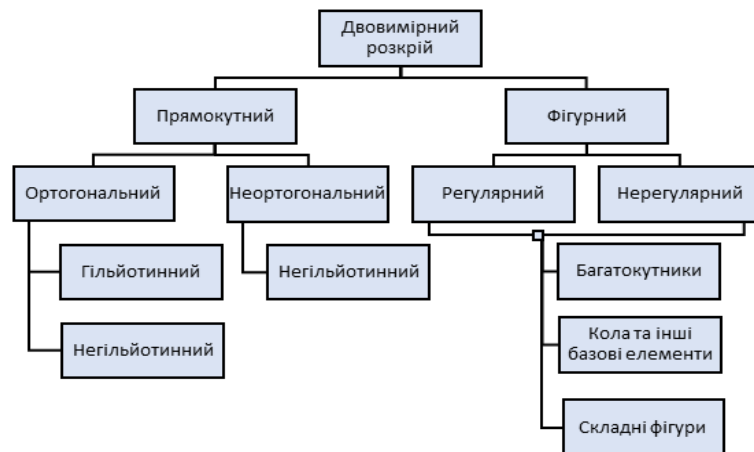

Рисунок 1.1 – Класифікація задачі двовимірного розкрою

Форма деталей є другим важливим критерієм класифікації для двовимірного розкрою (рисунок 1.1) [4]. Прямокутний розкрій, де всі елементи мають прямокутну форму, є найпоширенішим випадком і широко застосовується у металообробці та меблевому виробництві. Фігурний розкрій (рисунок 1.2) передбачає роботу з деталями складної геометрії, такими як багатокутники, круги або їх комбінації. Такий тип поширений у текстильній промисловості або при виробництві деталей зі складними контурами.

Регулярний фігурний розкрій характеризується однаковими або схожими за розміром елементами, тоді як нерегулярний включає деталі різної форми та розмірів, що значно ускладнює процес оптимізації.

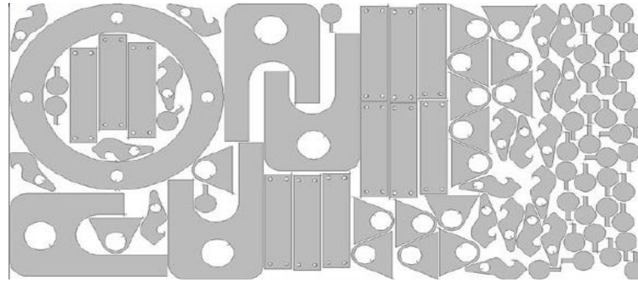

Рисунок 1.2 – Фігурний розкрій

Спосіб розташування деталей також є важливим фактором у класифікації задач розкрою (рисунок 1.3). В ортогональному розкрої всі елементи розміщуються тільки вздовж горизонтальної або вертикальної осі, що значно спрощує процес оптимізації та виконання різань. Такий підхід часто використовується у випадках, коли необхідно мінімізувати відходи матеріалу або прискорити виробництво завдяки простішому плануванню.

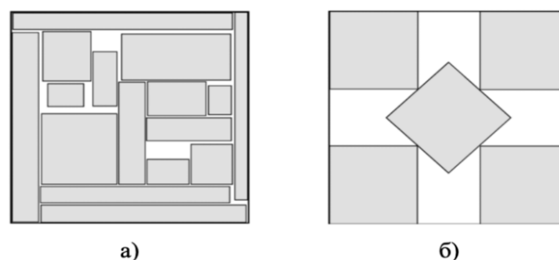

Рисунок 1.3 – Двомірний прямокутний розкрій: а) ортогональний розкрій; б) неортогональний розкрій

У неортогональному розкрої деталі можуть бути розташовані під довільними кутами відносно аркуша, що дозволяє ефективніше використовувати матеріал, особливо при розкрої складних форм. Однак цей метод потребує ретельнішого планування, оскільки врахування кутових поворотів ускладнює процес оптимізації та технічну реалізацію розкрою.

Ортогональний розкрій поділяється на гільйотинний та негільйотинний (рисунок 1.4), залежно від методу розкрою матеріалу [2].

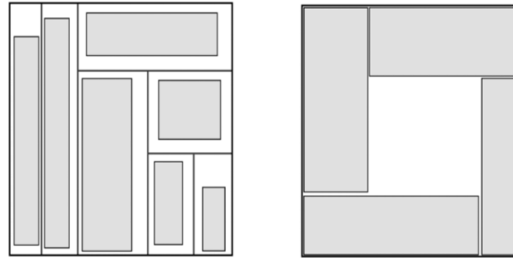

Рисунок 1.4 – Двомірний прямокутний гільйотинний та негільйотинний розкрій

Гільйотинний розкрій передбачає виконання розрізів паралельно сторонам листа (вздовж горизонтальної та вертикальної осі) від одного краю до іншого. Такий розкрій здійснюється за допомогою спеціалізованих верстатів – гільйотинних ножиць або лазерних різаків, які дозволяють виконувати точне розділення матеріалу. Він широко застосовується в промисловості завдяки своїй простоті та ефективності.

Негільйотинний розкрій не має таких обмежень і дозволяє виконувати розрізи між будь-якими точками листа. Це включає довільні розрізи для складних фігур, таких як кола, а також комбіновані методи, що поєднують різні типи різань. Однак такий метод потребує більш складного планування та обчислень, зазвичай із застосуванням алгоритмів оптимізації або динамічного програмування.

Технологічні обмеження становлять окремий клас критеріїв для класифікації. Вони можуть включати обмеження на розмір відходів, кількість допустимих різань або орієнтацію деталей. Наприклад, деякі матеріали, такі як тканина з певним напрямком волокон, вимагають строгої орієнтації елементів під час розкрою. Обмеження на кількість різань часто пов'язані з ресурсом різального інструменту, тоді як вимоги до мінімального розміру відходів впливають на економічну ефективність процесу.

Різноманітність задач розкрою обумовлює необхідність використання різних підходів до їх вирішення. Для прямокутного гільйотинного розкрою існують точні алгоритми оптимізації, тоді як фігурний негільйотинний розкрій часто вимагає застосування евристичних методів. Вибір конкретного підходу залежить від типу задачі, наявних обмежень та вимог до якості кінцевого результату. Усі ці фактори необхідно враховувати при розробці методик вирішення задач розкрою для різних галузей промисловості. У даному дослідженні основна увага зосереджена на двовимірному прямокутному ортогональному гільйотинному розкрої (Two-Dimensional Rectangular Guillotine Cutting Stock Problem, 2D-GCSP), який є одним з найпоширеніших у промислових застосуваннях через свою практичну значущість.

1.3 Огляд методів вирішення RGCSP

Задача прямокутного гільйотиного розкрою листового матеріалу (RGCSP) є різновидом задачі упаковки (C&P) та належить до класу задач із обмеженнями [5]. Особливість цієї задачі полягає у необхідності враховувати умову гільйотиного розрізу, коли кожен розріз повинен повністю проходити через поточний фрагмент матеріалу від одного краю до іншого.

Для вирішення RGCSP використовуються як точні методи, що гарантують знаходження оптимального розв'язку, так і наближені (евристичні) методи, які дозволяють отримати субоптимальний розв'язок за прийнятний час [6]. Точні методи, такі як динамічне програмування та цілочисленне програмування, забезпечують оптимальність рішення, але мають обмежену застосовність через високу обчислювальну складність. Динамічне програмування будує таблицю оптимальних рішень для всіх можливих варіантів розкрою, використовуючи рекурентні співвідношення. Цілочисленне програмування формалізує задачу у вигляді математичної моделі з обмеженнями, що дозволяє застосовувати такі інструменти, як симплекс-метод [7].

Наближені методи включають як прості евристики, так і складні метаевристичні підходи [4]. До класичних евристик належать First Fit (розміщення першого підходящого елемента), Next Fit (розміщення в наступну доступну область), Best Fit (вибір найкращого відповідного місця), Worst Fit (розміщення в найгірший відрізок) та Smallest Piece Method (пріоритет найменших елементів) [8]. Ці методи відрізняються простотою реалізації та швидкістю роботи, але часто дають субоптимальні результати. Оскільки ці евристики спрощують задачу розкрою, вони можуть часто потрапляти в локальні оптимуми, не знаходячи найкращого глобального рішення.

Евристичні методи шукають швидке рішення, часто за рахунок точності, що дозволяє їх застосовувати для задач, де необхідно отримати задовільний результат за короткий час. Проте їх основний недолік полягає в тому, що вони можуть не враховувати всі можливі варіанти і тому можуть застрягати в локальних оптимумах – точках, які є кращими за близькі рішення, але не є глобальними мінімумами.

Потрапляння у локальні оптімуми є актуальною проблемою при вирішенні складних задач, де існує багато варіантів і комбінацій рішень. Для подолання цієї проблеми використовуються метаевристичні методи, які пропонують більш складні підходи до пошуку рішень.

Метаевристики дозволяють покращити ефективність розв’язання задачі, оскільки вони дають змогу уникнути застрягання в локальних оптимумах завдяки гнучкості пошукових стратегій. Один із таких методів – це генетичні алгоритми, які імітують процес природного відбору. Вони працюють за допомогою механізмів мутації, схрещування та відбору, що дозволяє значно підвищити якість рішення, при цьому дозволяючи алгоритму знаходити рішення в складних ситуаціях [9, 10].

Ще одним методом є алгоритм мурашиної колонії, який базується на поведінці мурах, що використовують феромони для позначення найкращих шляхів. Цей метод ефективний для задач, де пошук оптимального рішення вимагає перебору багатьох варіантів і вибору найбільш раціонального шляху,

що є доцільним в контексті задачі розкрою матеріалу, де можливі варіанти розкрою мають взаємозв'язок між собою [11, 12].

Метод симуляції відпалу, з іншого боку, пропонує уникнення локальних мінімумів шляхом тимчасового допущення погіршення якості рішення в процесі пошуку. Це дозволяє алгоритму виходити з локальних мінімумів і знаходити більш глобальні оптимальні рішення [13, 14]. Кожен з цих методів має свої переваги та недоліки, які визначають їх застосування в різних ситуаціях. Наприклад, генетичні алгоритми можуть бути ефективними для складних, багатокрокових задач, таких як задача гільйотинного розкрою, де необхідно враховувати безліч факторів, а також для завдань з великою кількістю варіантів. Мурашині алгоритми дозволяють обробляти великі простори рішень, а симуляція відпалу допомагає уникати застрягання в неефективних рішеннях. Вибір конкретного методу залежить від вимог до точності розв'язку та наявних обчислювальних ресурсів. Для невеликих задач доречні точні методи, тоді як для складних промислових завдань підходять наближені алгоритми. Особливої уваги потребують гібридні підходи, які поєднують переваги різних методів. Наприклад, комбінація генетичних алгоритмів з локальним пошуком або використання евристик для ініціалізації популяції в метаевристиках може значно підвищити ефективність [15].

1.4 Аналіз існуючих рішень та наукових досліджень

Оптимізація процесу розкрою листового матеріалу є обчислювальною задачею, яка належить до класу NP-складних проблем, що призводить до експоненційного зростання обчислювальної складності при збільшенні кількості елементів [1]. Точні методи, такі як лінійне програмування, стають неефективними для таких задач через велику розмірність простору рішень. У зв’язку з цим на практиці широко застосовуються евристичні та метаевристичні підходи, які дозволяють знаходити субоптимальні рішення за прийнятний час.

Одним з відомих евристичних методів є алгоритм BFD , який досліджувався у роботі [16]. Даний підхід полягає у впорядкуванні елементів розкрою за спаданням їх розмірів і розміщенні їх у найбільш підходящий простір на листі. Автори показали, що BFD дозволяє швидко знаходити допустимі розв'язки з досить високим коефіцієнтом використання матеріалу. Проте основним недоліком алгоритму є те, що він не завжди дає оптимальні рішення для складних варіантів задач.

Ще одним популярним підходом є генетичний алгоритм, який належить до класу метаевристичних методів. Цей метод імітує процес еволюції в природі, застосовуючи оператори відбору, схрещування та мутації для створення нових рішень. Зокрема, у роботі [17] представлено застосування генетичного алгоритму прямокутного розміщення для гільйотинного розкрою. Цей метод дозволяє отримати оптимальні результати завдяки здатності досліджувати широкий простір рішень, але його ефективність значно залежить від правильного підбору параметрів, таких як розмір популяції та ймовірність мутацій. Крім того, генетичний алгоритм вимагає значних обчислювальних ресурсів порівняно з простими евристиками.

Мурашині алгоритми є ще одним метаевристичним методом, що моделює процес пошуку оптимальних шляхів у природі, коли мурахи знаходять їжу, залишаючи за собою феромони, що керують пошуком. Застосування конструктивної метаевристики «мурашина колонія» до завдання гільйотинного прямокутного розкрою розглядається у статті [18]. Він є ефективним для задач із багатьма обмеженнями, оскільки використовує принцип колективного інтелекту для пошуку оптимальних шляхів розміщення деталей. Однак його результати сильно залежать від початкових налаштувань, зокрема від швидкості випаровування феромонів, що ускладнює його застосування на практиці.

Симуляція відпалу – це метод глобальної оптимізації, який імітує процес відпалу в металургії. Алгоритм використовує процес поступового зниження температури для досягнення оптимального або наближеного оптимального

результату. У роботі [13] проаналізовано алгоритм симуляції відпалу для двовимірної задачі розкрою. Дослідження показує, що цей метод дозволяє отримати наближені рішення за короткий час, хоча для досягнення оптимального результату потрібне налаштування температури та кількості ітерацій.

1.5 Математична модель та постановка задачі

Дослідження зосереджено на розробці ефективних методів розв’язання задачі ортогонального гільйотинного розкрою листового матеріалу з можливістю повороту деталей 2D-RGCSPR). Основна увага приділяється аналізу та вдосконаленню алгоритмів, здатних знаходити оптимальні схеми розкрою при дотриманні обмежень на характер розрізів та орієнтацію деталей.

Задача має зв’язок з іншими класичними проблемами комбінаторної оптимізації, зокрема з задачею упаковки в контейнери (Bin Packing Problem, BPP) та задачею розміщення прямокутників (Rectangle Packing Problem).

Суть задачі полягає у визначенні оптимального розміщення набору прямокутних деталей із заданими розмірами $w_i \times h_i$ для $i = 1, 2, \dots, N$ на мінімальній кількості листів матеріалу M стандартного розміру $W \times H$ [19]. Допускається ортогональне розміщення деталей, що передбачає можливість їх повороту на 90 градусів, при чому w_i та h_i змінюються місцями. Всі деталі повинні розташовуватися строго паралельно сторонам вихідного листа. Основні вимоги до розкрою включають гільйотинний характер розрізів, коли кожен поділ проводиться по всій ширині або висоті поточного фрагмента матеріалу. Метою є мінімізація кількості використаних листів M при одночасному зменшенні площі матеріальних відходів.

У якості критерію оптимізації використовується коефіцієнт використання матеріалу (КВМ) або коефіцієнт розкрою [20, 30]. Мінімізація витрат матеріалу безпосередньо залежить від максимізації КВМ, оскільки цей показник відображає ефективність використання матеріалу (1.1):

$$\frac{\sum_{i=1}^{N}(w_i' \times h_i')}{M \times W \times H} \rightarrow max. \quad (1.1)$$

Обмеження задачі включають:

$$0 \leq x_i + w_i \leq W, 0 \leq y_i + h_i \leq H, \forall i, \quad (1.2)$$

$$(x_i + w_i \leq x_j) \vee (x_j + w_j \leq x_i) \vee (y_i + h_i \leq y_j) \vee (y_j + h_j \leq y), \forall i \neq j, \quad (1.3)$$

$$\sum_{i=1}^{N}(w_i' \times h_i') \leq M \times W \times H, \quad (1.4)$$

$$w_i' = \begin{cases} w_i, \text{якщо } r_i = 0, \\ h_i, \text{якщо } r_i = 1, \end{cases} \quad (1.5)$$

$$h_i' = \begin{cases} h_i, \text{якщо } r_i = 0, \\ w_i, \text{якщо } r_i = 1, \end{cases} \quad (1.6)$$

$$v_l \in \{0, W\} \cup \{ v_l \mid l' < l \}, h_l \in \{0, H\} \cup \{h_{l'} \mid l' < l\}, \quad (1.7)$$

$$v_l \neq x_i, v_l \neq x_i + w_i, \forall i, h_l \neq y_i, h_l \neq y_i + h_i, \forall i, \quad (1.8)$$

де x_i, y_i – координати лівого верхнього кута і-го елемента на аркуші ($i = 1,2, \dots, N$), $r_i \in \{0, 1\}$ ознака повороту і-го елемента, v_l, h_l координати вертикальних і горизонтальних розрізів, l – індекс розрізу.

Обмеження (1.2) гарантує, що всі елементи розташовуються в межах аркуша. Згідно з (1.3) виключається можливість перекриття елементів, обмежуючи, щоб або правий край одного елемента не заходив за лівий край іншого, або нижній край не перетинався з верхнім. Умова (1.4) обмежує сумарну площу всіх елементів, яка не повинна перевищувати площу аркуша. Вирази (1.5) та (1.6) визначають можливість повороту елементів. Умова (1.7) забезпечує, що гільйотинні розрізи починаються і закінчуються лише на краях

аркуша або на вже існуючих розрізах. Нерівність (1.8) забороняє розрізам проходити через розміщені деталі, запобігаючи їх пошкодженню.

В результаті аналізу існуючих методів розв’язання задачі розкрою (розділи 1.3–1.4), для вирішення поставленої задачі було прийнято рішення розробляти підходи, основані на наближених методах, зокрема, евристичний рівневий алгоритм із стратегією Best Fit Decreasing (BFD), а також метаевристики, зокрема генетичний, мурашиний алгоритми та алгоритм імітації відпалу. Дослідження зосереджено на таких критеріях ефективності як швидкість роботи, коефіцієнт використання матеріалу і кількість використаних аркушів.

Для демонстрації коректної роботи алгоритмів на основі існуючих методів необхідно створити тестове програмне середовище. Застосунок має надавати можливість роботи з розкроєм матеріалів, дозволяючи користувачу задавати параметри вихідного листа, додавати, редагувати та видаляти прямокутні заготовки з індивідуальними розмірами. Функціонал повинен включати вибір алгоритму для тестування, візуалізацію схеми розкрою та відповідних значень критеріїв ефективності.

На даний проєкт поставлені такі завдання:

- огляд та аналіз існуючих підходів до вирішення та наукових досліджень задачі гільйотинного розкрою;
- обґрунтування і вибір засобів вирішення поставленої задачі;
- розробку та детальний опис алгоритмів і технології реалізації застосованих методів вирішення поставленої задачі;
- вибір технологій для розробки тестового програмного середовища;
- розробка та налагодження застосунку у середовищі Visual Studio Code;
- проведення експериментального дослідження на різних наборах даних;
- опис та аналіз результатів експериментів.

2 АНАЛІЗ ВИКОРИСТОВУВАНИХ АЛГОРИТМІВ

2.1 Евристичний підхід із стратегією Best Fit Decreasing

Евристичні алгоритми представляють собою підходи до розв’язання складних оптимізаційних задач, які замість точного розв’язку пропонують наближені, але ефективні за часом рішення. Серед різноманітних евристик найбільш популярними є жадібні алгоритми, які на кожному кроці обирають локально оптимальне рішення, не враховуючи його глобальних наслідків. Відмінністю евристик від точних методів є те, що вони не гарантують знаходження оптимального результату, але зазвичай дають прийнятні рішення за обмежений час, особливо для задач великої розмірності, таких як задача гільйотинного розкрою матеріалів [21].

Best Fit Decreasing (BFD) є класичним представником жадібних евристичних алгоритмів і належить до категорії рівневих алгоритмів розкрою. Він широко застосовується для вирішення задач прямокутного гільйотинного розкрою листового матеріалу, де основним завданням є оптимальне розміщення прямокутних деталей на листі з мінімізацією відходів.

Алгоритм BFD працює за принципом послідовного розміщення деталей, починаючи з найбільших (рисунок 2.1). Спочатку всі прямокутники сортуються за спаданням розміру (за площею або довжиною більшої сторони), що дозволяє ефективніше використовувати простір листа. Після сортування кожна деталь розміщується в позицію, де залишковий простір буде мінімальним, що є характерною рисою жадібного підходу [16, 22, 23].

Як рівневий алгоритм, BFD розміщує деталі рівнями (шарами), де кожен новий рівень починається тільки після заповнення попереднього. Це особливо важливо у гільйотинному розкрою, де розрізи повинні бути суцільними і проходити через весь лист. Такий підхід забезпечує технологічну реалізацію процесу, оскільки відповідає обмеженням на послідовність розрізів.

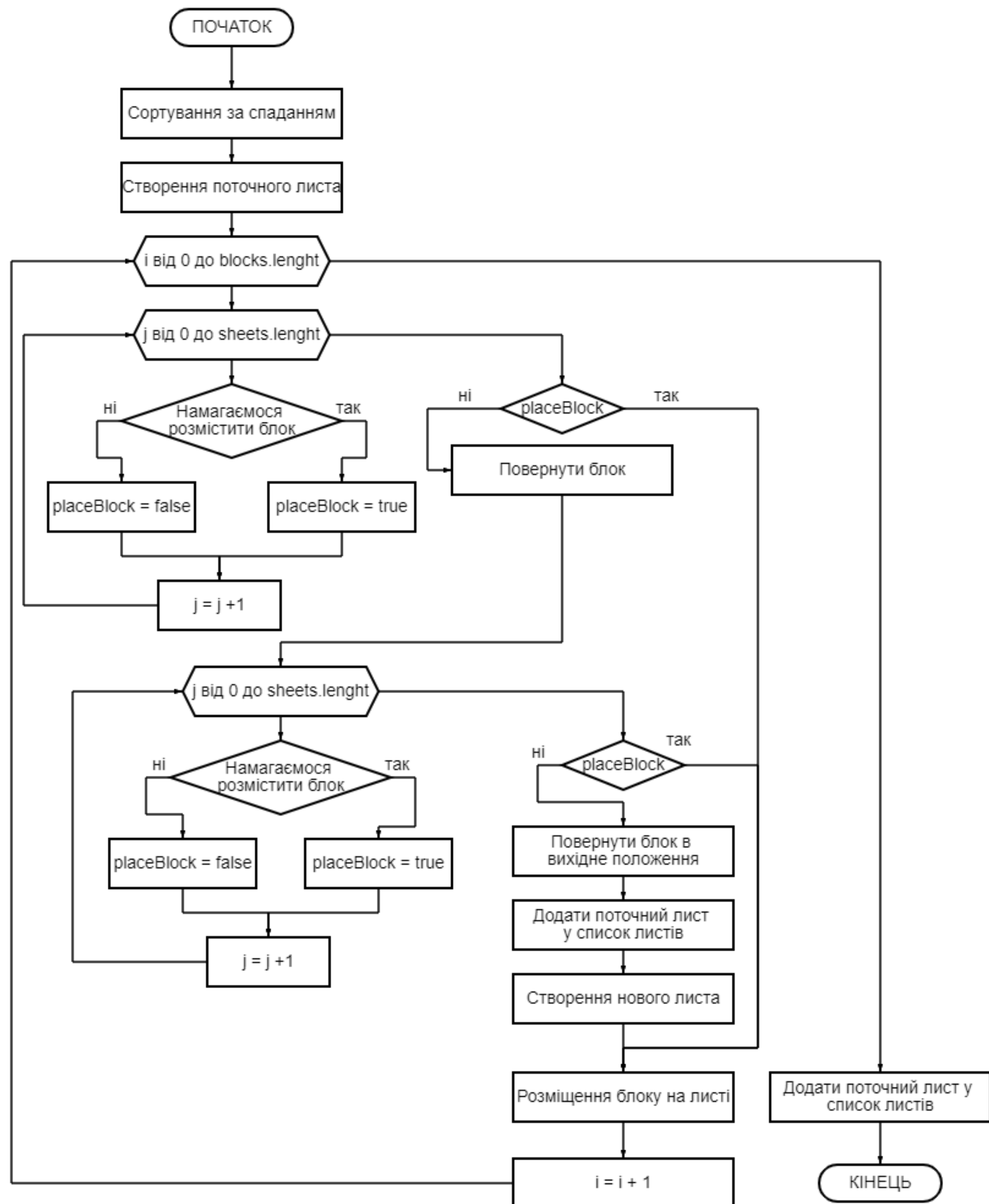

Рисунок 2.1 – Блок-схема алгоритму Best Fit Decreasing

Жадібна природа алгоритму проявляється в тому, що він на кожному кроці приймає локально оптимальне рішення, не враховуючи глобальних наслідків. Це може призводити до певних недоліків, таких як утворення невеликих непридатних фрагментів або субоптимальне використання площі листа. Однак попереднє сортування деталей за спаданням розміру дозволяє значно зменшити ці негативні ефекти. У порівнянні з іншими методами, такими як генетичні алгоритми або методи динамічного програмування, BFD

не гарантує знаходження глобально оптимального розв'язку, але відрізняється простотою реалізації та високою швидкістю роботи. Це робить його особливо корисним у промислових умовах, де важливим фактором є час обчислення.

2.2 Метаевристичні алгоритми

Метаевристичні алгоритми є важливим інструментом для вирішення складних задач оптимізації, зокрема тих, де традиційні методи не здатні забезпечити ефективні результати через великий обсяг простору рішень або високу обчислювальну складність. Метаевристика передбачає використання певної стратегії пошуку, що поєднує декілька евристичних підходів для диверсифікації (дослідження нових ділянок простору рішень) та інтенсифікації (поглиблення пошуку в перспективних областях) процесу оптимізації, забезпечуючи ефективне дослідження можливих рішень і покращення поточного стану [24, 25]. Основною метою метаевристичних алгоритмів є знаходження оптимальних або наближених до оптимальних рішень у випадках, коли точні методи є обчислювально складними або практично непридатними для застосування.

Метаевристичні алгоритми можна поділити на два основні класи: траєкторні та популяційні [25]. Кожен з цих класів має свої характеристики та підходи до пошуку оптимальних рішень.

Траєкторні алгоритми, що також відомі як методи, орієнтовані на одне рішення, передбачають ітеративний процес покращення одного поточного рішення. Пошук здійснюється шляхом поступового переходу від поточного стану до сусіднього, з подальшим оптимізуванням цього стану на кожному етапі. При цьому важливою проблемою є попадання в локальні мінімумі, що може призвести до отримання неоптимальних результатів. Для подолання цієї проблеми застосовуються різноманітні стратегії, такі як використання адаптивних змін ландшафту функції придатності, багатократні запуски пошуку з різних початкових точок або зміна критеріїв оцінки рішень протягом

процесу оптимізації. Траєкторні методи можуть бути ефективними в ситуаціях, де покращення одного рішення може призвести до знаходження оптимального або наближеного до оптимального результату.

Популяційні алгоритми, в свою чергу, працюють з групою рішень, що є частиною популяції, і використовують їх для дослідження простору можливих варіантів. У цих методах кожне рішення в популяції оцінюється з точки зору його якості, і найкращі з них мають більший шанс на «розмноження» (схрещування) з іншими рішеннями, що дозволяє створювати нові варіанти. Це дозволяє ефективно досліджувати більш широкий простір рішень і знаходити оптимальні або наближені до оптимальних варіанти в умовах великої кількості змінних. Популяційні алгоритми включають такі методи, як генетичні алгоритми та алгоритми ройового інтелекту, що імітують природні процеси еволюції та колективної поведінки організмів. Популяційні алгоритми мають перевагу в тому, що вони дозволяють одночасно працювати з кількома рішеннями, що підвищує ймовірність знаходження глобального оптимуму, а також сприяє кращому охопленню простору можливих рішень. Однак вони можуть вимагати більших обчислювальних витрат, оскільки працюють з кількома рішеннями одночасно, на відміну від траєкторних методів, які зосереджені на покращенні одного рішення.

У контексті задачі гільйотинного розкрою матеріалу, де необхідно мінімізувати залишки та втрати при розподілі матеріалу на частини заданих розмірів, метаевристичні методи можуть значно покращити ефективність пошуку оптимальних варіантів. Траєкторні методи, такі як симуляція відпалу, можуть бути застосовані для поступового покращення одного рішення на основі функції придатності, що дозволяє уникнути локальних мінімумів і забезпечити більш ефективне використання матеріалів. Популяційні методи, зокрема еволюційні алгоритми та алгоритми ройового інтелекту, можуть одночасно досліджувати кілька варіантів розкрою, що дозволяє зменшити витрати і максимізувати використання ресурсу.

2.2.1 Генетичний алгоритм

Генетичний алгоритм належить до класу еволюційних методів оптимізації, які знаходять застосування у вирішенні комбінаторних задач високої складності, зокрема задач оптимального розкрою матеріалів. В основу алгоритму покладено принципи природної еволюції, що включають механізми спадкової варіабельності, селекції та адаптації [26].

Основним елементом алгоритму є поняття особини, яка представляє варіант рішення задачі. Кожна особина характеризується хромосомою – структурою даних, що кодує параметри розкрою. У випадку гільйотинного розкрою хромосома може представляти послідовність розміщення деталей або послідовність виконання розрізів [26, 30].

Важливим аспектом є визначення функції пристосованості, яка кількісно оцінює якість кожного рішення на основі критеріїв ефективності використання матеріалу.

Процес оптимізації починається з ініціалізації початкової популяції (рисунок 2.2), де кожна особина генерується випадковим чином з дотриманням заданих обмежень. На кожній ітерації алгоритму здійснюється оцінка якості рішень за допомогою функції пристосованості, після чого відбувається відбір найкращих особин для подальшого розмноження. Вибір батьківських пар може здійснюватися різними методами, включаючи турнірний відбір або пропорційний відбір за значенням функції пристосованості.

Оператор схрещування дозволяє комбінувати характеристики батьківських особин, створюючи нові рішення. Для задачі гільйотинного розкрою особливо важливим є застосування спеціалізованих операторів схрещування, які гарантують отримання допустимих рішень. Оператор мутації вносить невеликі випадкові зміни в хромосоми, що сприяє дослідженню нового простору рішень і запобігає передчасній збіжності алгоритму.

Особливістю застосування генетичного алгоритму для задачі гільйотинного розкрою є необхідність врахування специфічних обмежень, пов'язаних із властивостями процесу розкрою. Зокрема, усі елементи мають розташовуватись повністю в межах аркуша, без виходу за його кордони, та не повинні перетинатися між собою.

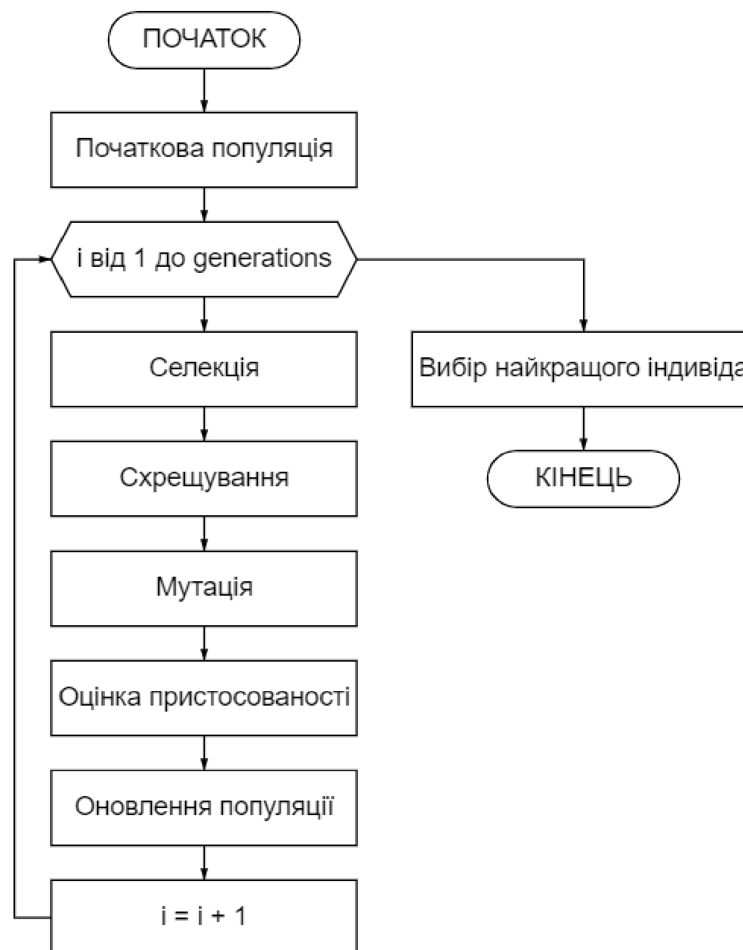

Рисунок 2.2 – Блок-схема генетичного алгоритму

Також враховується можливість повороту деталей, якщо це не суперечить технічним вимогам. Крім того, гільйотинні розрізи мають починатися і закінчуватися лише на краях аркуша або на вже існуючих розрізах. Це вимагає розробки спеціалізованих механізмів генерування початкових рішень, а також відповідних операторів схрещування та мутації, які забезпечують формування лише допустимих варіантів розкрою.

Ефективність алгоритму значною мірою залежить від правильного вибору параметрів, таких як розмір популяції, ймовірність схрещування та мутації, а також критеріїв зупинки обчислювального процесу. Оптимальні значення цих параметрів зазвичай підбираються експериментальним шляхом для конкретного класу задач. Важливою перевагою генетичного алгоритму є його здатність знаходити субоптимальні рішення у складних задачах, де традиційні методи виявляються неефективними. Однак продуктивність методу може значно варіюватись залежно від обраної схеми кодування рішень та реалізації генетичних операторів. Для підвищення ефективності алгоритму часто використовують його поєднання з локальними методами оптимізації або евристичними підходами.

2.2.2 Мурашиний алгоритм

Мурашиний алгоритм відноситься до класу методів роєвого інтелекту та належить до групи метаевристичних методів оптимізації, заснованих на принципах колективної поведінки мурашиних колоній. Він знаходить ефективне застосування для вирішення складних комбінаторних задач, зокрема задач оптимального розкрою матеріалів. В основу алгоритму покладено механізми кооперативного пошуку, що імітують процес пошуку найкоротшого шляху до джерела їжі в мурашнику [27].

У контексті задачі гільйотинного розкрою кожен агент (мураха) представляє потенційний варіант розміщення деталей на листовому матеріалі. Агенти рухаються в просторі рішень, послідовно вибираючи оптимальні позиції для розміщення чергової деталі з урахуванням обмежень гільйотинного розкрою. Кожен такий вибір супроводжується залишенням феромонових слідів, інтенсивність яких залежить від якості знайденого рішення [27, 28].

Процес пошуку оптимального розкрою починається з ініціалізації популяції агентів (рисунок 2.3), кожен з яких будує свій варіант розміщення

деталей. При цьому агенти керуються двома основними факторами: концентрацією феромонів, що вказують на перспективні напрямки пошуку, і евристичною інформацією, яка враховує геометричні характеристики деталей та ступінь заповнення матеріалу.

Після завершення побудови варіантів розкрою всіма агентами відбувається оновлення феромонових слідів. Найкращі рішення отримують більш інтенсивні феромонові позначки, що збільшує ймовірність їх використання в наступних ітераціях. Паралельно відбувається процес випаровування феромонів, який запобігає потраплянню в локальні оптимуми та сприяє дослідженню нових областей простору рішень.

Особливістю застосування мурашиного алгоритму для задачі гільйотинного розкрою є необхідність ретельного визначення правил переміщення агентів, які повинні враховувати специфічні обмеження процесу.

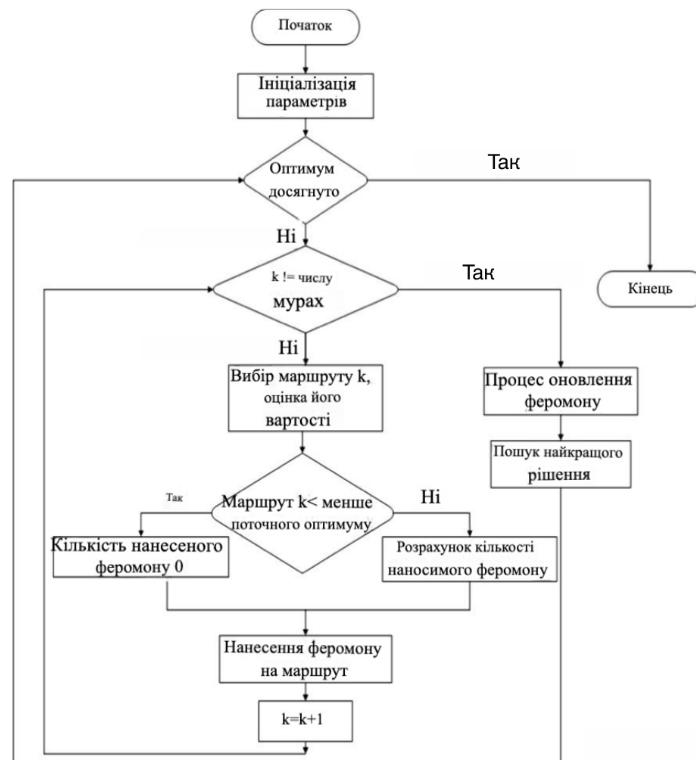

Рисунок 2.3 – Блок-схема алгоритму мурашиної колонії

Кожен крок агенту супроводжується перевіркою на допустимість, що гарантує отримання лише коректних варіантів розкрою.

Ефективність алгоритму в даній задачі значною мірою залежить від правильного балансу між дослідженням нових варіантів і експлуатацією вже знайдених субоптимальних рішень. Цей баланс досягається за рахунок оптимального підбору параметрів, таких як швидкість випаровування феромонів та вагові коефіцієнти для феромонів і евристичної інформації.

В результаті багатоітераційної роботи алгоритму формується набір перспективних варіантів розкрою, серед яких можна вибрати оптимальний з точки зору використання матеріалу. При цьому колективна взаємодія агентів дозволяє ефективно досліджувати великий простір можливих рішень, уникаючи застрягання в локальних оптимумах.

Застосування мурашиного алгоритму для задачі гільйотинного розкрою особливо доречне у випадках, коли необхідно враховувати додаткові технологічні обмеження або коли точні методи вимагають надмірних обчислювальних ресурсів.

Важливим аспектом є правильна настройка параметрів алгоритму, оскільки саме від неї залежить збіжність методу та якість кінцевого рішення. Оптимальний вибір таких параметрів, як кількість агентів, інтенсивність феромонів, швидкість їх випаровування та ваги евристичних функцій, дозволяє досягти найкращих результатів при мінімальних витратах обчислювальних ресурсів.

2.2.3 Симуляція відпалу

Метод симуляції відпалу належить до класу імовірнісних методів оптимізації, заснованих на аналогії з фізичним процесом відпалу металів. Він знаходить ефективне застосування для вирішення складних комбінаторних задач, зокрема задач оптимального розкрою матеріалів. В основу алгоритму покладено принцип поступового «охолодження» системи, що дозволяє уникнути локальні оптимуми [29]. Для задачі гільйотинного розкрою алгоритм починає роботу з генерації початкового рішення (рисунок 2.4), яке

може бути випадковим або створеним за допомогою евристичних методів. Це рішення представляє собою конкретну конфігурацію розміщення деталей на листовому матеріалі з урахуванням обмежень гільйотинного типу. Важливою особливістю є те, що кожен стан системи повинен задовольняти всім технологічним обмеженням процесу розкрою.

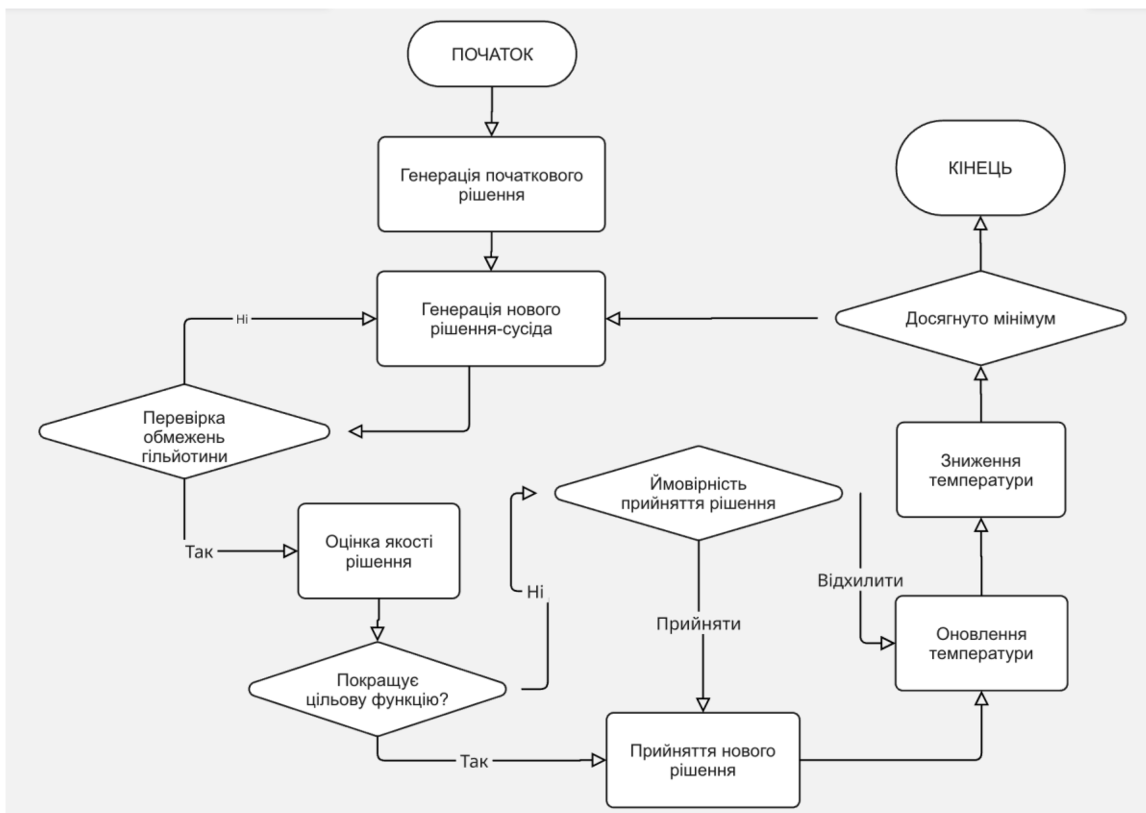

Рисунок 2.4 – Блок-схема алгоритму симуляції відпалу

На кожній ітерації алгоритму виконується послідовність дій. Спочатку генерується нове рішення-сусід шляхом внесення невеликої зміни до поточного стану. Для задачі розкрою такі зміни можуть включати: перестановку двох деталей у послідовності розміщення, зміну орієнтації конкретної деталі або модифікацію напрямку одного з розрізів. Кожна така зміна ретельно перевіряється на відповідність обмеженням гільйотини.

Після генерації нового рішення відбувається оцінка його якості за допомогою цільової функції, яка зазвичай враховує ступінь використання матеріалу. Якщо нове рішення покращує цільову функцію, воно автоматично

приймається. Якщо ж рішення погіршує показник, воно може бути прийняте з певною ймовірністю, яка залежить від поточної «температури» системи. Цей механізм дозволяє алгоритму виходити з локальних оптимумів на ранніх етапах пошуку.

Особливістю застосування методу імітації відпалу для задачі розкрою є необхідність ретельної обробки обмежень. Це включає: спеціальні процедури генерації сусідніх рішень, які гарантують їхню допустимість; введення штрафних функцій для частково допустимих конфігурацій; розробку спеціалізованих операторів зміни стану.

Температурний параметр є дуже важливим у роботі алгоритму. На початкових етапах він встановлюється досить високим, що дозволяє системі вільно досліджувати простір рішень. Поступово температура знижується за вибраною схемою охолодження, що може бути лінійним, геометричним або логарифмічним. Етап охолодження є дуже важливим для успішної роботи алгоритму, адже швидке охолодження може призвести до застрягання в локальному оптимумі, тоді як надто повільне охолодження різко збільшує обчислювальні витрати [25]. Для задач розкрою оптимальний час охолодження зазвичай підбирається експериментально і залежить від розмірності задачі та складності конфігурації деталей.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Архітектура проєкту

Архітектура проєкту побудована за принципами модульності та розділення відповідальностей, що забезпечує зручність підтримки, масштабованість та можливість розширення функціоналу. Основні компоненти системи організовані у вигляді окремих модулів, кожен з яких відповідає за свою частину логіки.

Проєкт реалізовано як клієнтський односторінковий вебзастосунок (SPA), що повністю функціонує у браузері користувача. Вся бізнес-логіка, алгоритми оптимізації, обробка даних, взаємодія з інтерфейсом та візуалізація результатів реалізовані на JavaScript, що забезпечує швидкий відгук, автономність роботи та простоту розгортання без серверної частини.

Файлова структура проекту створеного в VS Code, представлена на рисунку 3.1. За основу архітектури взято патерн MVC (Model-View-Controller), який забезпечує чітке розділення відповідальностей між різними частинами застосунку.

Model (Модель) відповідає за зберігання та обробку даних. У проєкті це модулі, які реалізують різні алгоритми оптимізації розкрою (генетичний, мурашиний, імітація відпалу, Best Fit, гібридний), а також модуль details.js (додаток Б.3.1), що зберігає список блоків, дозволяє їх додавати, редагувати, видаляти та формує звіти за результатами розкрою.

View (Відображення, або Представлення) відповідає за візуалізацію даних та взаємодію з користувачем. Інтерфейс користувача реалізований на основі HTML, CSS та jQuery. Візуалізація результатів розкрою здійснюється через HTML5 Canvas, що дозволяє наочно оцінити розміщення блоків на аркушах у реальному часі. Всі зміни у даних чи параметрах одразу відображаються на сторінці без перезавантаження.

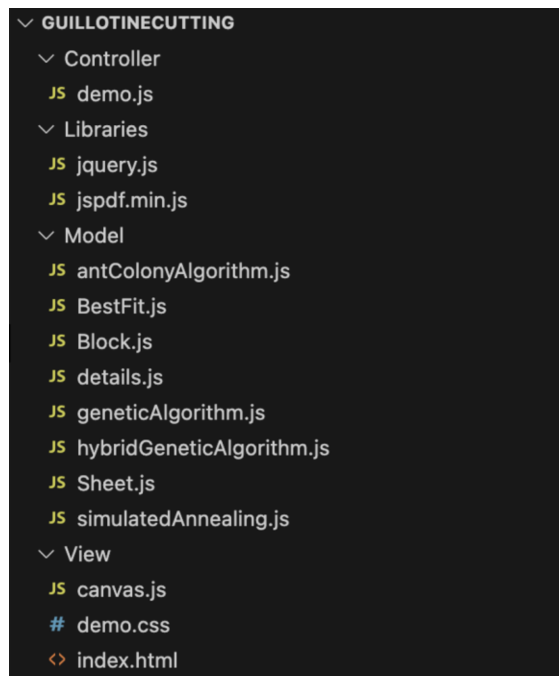

Рисунок 3.1 – Файлова структура програмного застосунку

Controller (Контролер) координує взаємодію між моделлю та відображенням. У файлі demo.js (додаток Б.3.3) контролер ініціалізує елементи інтерфейсу, обробляє події користувача (додавання, редагування блоків, зміну параметрів, вибір алгоритму), зберігає та оновлює дані, керує запуском алгоритмів, вимірює час виконання, розраховує ефективність використання площі та передає ці дані для відображення у View.

У папці Libraries знаходяться дві бібліотеки застосунку:

- jquery для маніпулювання елементами DOM;
- jsPdf для створення PDF-файлів для експорту результатів.

3.2 Модель представлення вихідних даних

У даному програмному застосунку вихідні дані формуються у вигляді структурованих об’єктів, які відображають результат розміщення прямокутних блоків на аркушах матеріалу. Кожен аркуш описується як об’єкт, що містить інформацію про свої розміри, а також масив блоків, які були розміщені на цьому аркуші в результаті роботи алгоритму. Для кожного блоку

зберігаються координати його розташування на аркуші, ширина, висота, а також ідентифікатор, що дозволяє відстежувати його походження серед вхідних даних.

Вихідна структура даних дозволяє не лише зберігати результат розкрою, а й забезпечує можливість подальшої візуалізації на графічному полотні, аналізу ефективності використання площі матеріалу, а також розрахунку додаткових характеристик, таких як кількість використаних аркушів, коефіцієнт ефективності розкрою та час виконання алгоритму. Формат представлення даних є універсальним і може бути легко адаптований для різних алгоритмів оптимізації, що реалізовані у системі.

Модель представлення вихідних даних у системі розкрою базується на двох основних класах: Block і Sheet. Клас Block відповідає за опис кожного прямокутного блоку, який необхідно розмістити на аркуші матеріалу. Клас Sheet, у свою чергу, моделює сам аркуш, на якому розміщуються блоки. Така структура дозволяє не лише зберігати результат розкрою, а й забезпечує можливість подальшої обробки, візуалізації, аналізу ефективності та експорту результатів.

Клас Block (додаток Б.2.1) реалізовано у файлі Model/Block.js. При створенні нового екземпляра класу Block конструктор приймає три основні параметри: ширину блоку, висоту блоку та унікальний ідентифікатор. Ці параметри дозволяють однозначно визначити кожен блок серед усіх вхідних даних, а також забезпечують можливість відстеження його переміщення та стану на різних етапах роботи програми. Властивість w зберігає ширину блоку, а властивість h – його висоту. Ці параметри є основними для подальших розрахунків, зокрема для визначення площі блоку, яка зберігається у властивості area. Площа блоку використовується для аналізу ефективності розміщення, а також для розрахунку коефіцієнта використання площі аркуша. Важливою характеристикою є координати розташування блоку на аркуші: x та y. Вони визначають положення лівого верхнього кута блоку відносно початку координат аркуша. На початковому етапі ці координати встановлюються у

нуль, але після виконання алгоритму розкрою кожен блок отримує конкретні значення x та y, які відображають його фактичне розміщення на аркуші.

Окремо слід відзначити властивість num, яка є унікальним ідентифікатором блоку. Вона дозволяє однозначно ідентифікувати кожен блок серед усіх вхідних даних, що особливо важливо при аналізі результатів, візуалізації та експорту інформації. Ідентифікатор також використовується для зв'язку між вхідними та вихідними даними, що забезпечує прозорість і відстежуваність усіх операцій над блоками. Ще однією важливою властивістю класу Block є rotation. Вона визначає орієнтацію блоку на аркуші: горизонтальну або вертикальну.

У багатьох задачах розкрою допускається поворот блоків для досягнення більшої ефективності використання площі матеріалу. Для цього у класі реалізовано метод rotate, який змінює місцями значення ширини та висоти блоку ([this.w, this.h] = [this.h, this.w]), а також змінює стан rotation між 0 (горизонтальна орієнтація) та 1(вертикальна орієнтація). Таким чином, блок може бути розміщений як у початковій орієнтації, так і у повернутому вигляді, що значно розширює можливості алгоритмів оптимізації.

Після виконання алгоритму розкрою формується масив об'єктів класу Block, кожен з яких містить усю необхідну інформацію для подальшої обробки. Така структура дозволяє легко здійснювати візуалізацію результатів на графічному полотні, аналізувати ефективність розміщення, розраховувати кількість використаних аркушів та коефіцієнт використання площі.

Клас Sheet (додаток Б.2.2), реалізований у файлі Model/Sheet.js, використовується для представлення одного аркуша матеріалу, на якому розміщуються блоки.

При створенні нового екземпляра класу Sheet конструктор приймає два параметри: ширину та висоту аркуша. Властивості w та h зберігають відповідно ширину та висоту аркуша. Додатково у класі Sheet є масив blocks, який містить усі блоки, розміщені на даному аркуші. Кожен блок у цьому масиві є екземпляром класу Block, що дозволяє зберігати повну інформацію

про розташування, розміри, ідентифікатор та орієнтацію кожного елемента. Об’єктно-орієнтований підхід до побудови моделі вихідних даних забезпечує високу гнучкість і масштабованість системи. За потреби можна легко додати нові властивості до класу Block, наприклад, інформацію про колір блоку для візуалізації, додаткові параметри для аналізу або ознаки, що характеризують специфічні властивості матеріалу. Це дозволяє адаптувати програмний комплекс до різних виробничих задач і вимог користувача. Модель представлення вихідних даних, побудована на основі класів Block і Sheet, є ефективною, гнучкою та зручною для подальшої обробки, аналізу й візуалізації результатів розкрою. Вона забезпечує прозорість усіх операцій над блоками, дозволяє легко інтегрувати нові алгоритми та розширювати функціонал системи без суттєвих змін у структурі даних.

Після виконання алгоритму оптимізації формується масив об’єктів класу Sheet, кожен з яких містить масив об’єктів класу Block. Така структура дозволяє легко візуалізувати розміщення блоків, аналізувати ефективність розкрою та експортувати результати.

3.3 Реалізація алгоритмів рішення RGCSP

3.3.1 Алгоритм Best Fit Decreasing

Алгоритм Best Fit Decreasing винесено в окремий модуль з назвою «BestFit.js» (додаток Б.1.2). У цьому модулі реалізовано об’єкт «BestFit», який містить основну функцію «cutBlocks()», а також допоміжні: «placeBlock», «findFittingPosition» та «splitSheet». Нижче наведено детальний опис кожного етапу роботи алгоритму та кожної функції.

Основна функція «cutBlocks(blocks, sheetWidth, sheetHeight)» приймає три параметри. Перший параметр – це масив об’єктів blocks, де кожен об’єкт представляє прямокутний блок із властивостями w (ширина), h (висота) і похідною властивістю area, що дорівнює добутку ширини на висоту. Другий і

третій параметри описують ширину (sheetWidth) та висоту (sheetHeight) аркуша, на якому буде відбуватися розміщення блоків. Як результат, функція повертає масив об’єктів типу «Sheet», кожен з яких містить розміщені блоки та координати вільних ділянок.

На першому етапі, у функції «cutBlocks()», відбувається сортування масиву блоків за спаданням площі. Це необхідно для того, щоб більші блоки займали простір першими, оскільки згодом їм буде складніше знайти відповідну ділянку. Далі створюється перший об’єкт «Sheet», що представляє аркуш із заданими розмірами. Для кожного блоку виконується спроба його розміщення: спочатку перевіряється, чи може блок бути доданий на будь-який з уже створених аркушів; якщо ні – намагаються розмістити його на поточному аркуші. У випадку, якщо і це не вдається, виконується поворот блоку методом «rotate()», який змінює його ширину та висоту місцями. Якщо навіть після повороту блок не може бути розміщений, створюється новий аркуш, і виконується спроби додати блок вже на нього.

Допоміжна функція «placeBlock(block, sheet)» виконує спробу додати блок на заданий аркуш. Вона викликає іншу функцію – «findFittingPosition()», яка шукає серед доступних вільних ділянок на аркуші таку, що дозволяє повністю розмістити блок. У разі успіху викликається функція «splitSheet()», яка розміщує блок та оновлює структуру доступного простору. Якщо жодне місце не підходить – функція повертає false, сигналізуючи про невдале розміщення.

Функція «findFittingPosition(block, sheet)» приймає як параметри сам блок і аркуш, на якому виконується пошук. Кожен аркуш має властивість «availableSpaces» – масив об’єктів, що описують координати на плошині (x, y) та розміри (w, h) вільних ділянок. Якщо список вільних ділянок не ініціалізований, функція створює початкову вільну область, яка відповідає всій площі аркуша. Далі кожна ділянка аналізується на предмет можливості розміщення блоку. Якщо блок може вміститись у межах ділянки, повертається ця ділянка як результат. Якщо блок не входить у ділянку, але може вміститись

після повороту, функція виконує поворот блоку і повертає цю ж ділянку. Якщо жодна ділянка не підходить, то повертається null.

Функція «splitSheet(sheet, block, space)» безпосередньо розміщує блок у знайденому просторі. Координати блоку встановлюються відповідно до координат вільної ділянки space. Блок додається до масиву «sheet.blocks», що містить усі блоки, вже розміщені на аркуші. Потім виконується оновлення вільного простору: якщо після розміщення залишилася частина простору праворуч від блоку або нижче нього – створюються нові вільні ділянки. Стара ділянка space видаляється із масиву, а натомість додаються новостворені ділянки. Таким чином алгоритм динамічно підтримує інформацію про доступні області для розміщення нових блоків.

3.3.2 Генетичний алгоритм

Генетичний алгоритм гільйотинного розкрою реалізовано у модулі «geneticAlgorithm.js» (додаток Б.1.1). Його структура охоплює повний цикл еволюції рішень: від генерації початкової популяції до відбору найкращого розкрою. Кожна функція виконує чітко визначену роль у цьому процесі, приймає відповідні параметри та повертає результат, необхідний для наступних етапів.

Головна функція модуля – «runGeneticAlgorithm». Вона організовує весь процес еволюції. На вхід ця функція приймає розмір популяції, ймовірність мутації, кількість поколінь, масив блоків для розміщення, розміри листа, а також початкову популяцію (якщо вона задана) і тип кроссоверу. Якщо початкова популяція не передана, створюється випадкова за допомогою функції «createInitialPopulation».

Далі для кожного покоління виконується цикл, у якому відбувається відбір батьків, кросовер, мутація та формування нової популяції. Після завершення всіх поколінь функція повертає найкращий знайдений розкрій у вигляді масиву листів із розміщеними блоками.

Функція «isGuillotine» перевіряє, чи є розміщення блоків на листі гільйотинним. Вона приймає масив блоків, а також розміри листа. Для кожного блоку перевіряється, чи розміщений він у лівому верхньому куті однієї з доступних областей, чи не перетинається з іншими блоками, і чи після розміщення область ділиться лише на дві частини. Якщо всі умови виконані, функція повертає true, інакше – false. Для отримання всіх вільних областей після розміщення блоків використовується функція «getFreeGuillotineAreas». Вона приймає масив розміщених блоків і розміри листа. Функція послідовно імітує розміщення кожного блоку, оновлюючи масив доступних областей, і повертає масив прямокутних областей, які залишилися вільними.

Генерація випадкового гільйотинного розкрою реалізована у функції «createRandomGuillotineCutting». На вхід вона отримує масив блоків і розміри листа. Функція намагається розмістити кожен блок у лівому верхньому куті однієї з доступних областей, перебираючи всі можливі орієнтації. Якщо блок можна розмістити без перетинів і з дотриманням гільйотинних обмежень, область ділиться на дві нові, а блок додається до поточного листа. Якщо блок не вдалося розмістити, створюється новий лист. Повертається масив листів із розміщеними блоками.

Початкова популяція створюється функцією «createInitialPopulation». Вона приймає розмір популяції, масив блоків і розміри листа. Для кожної особини блоки перемішуються у випадковому порядку, після чого для них викликається функція створення випадкового розкрою. Результатом є масив особин, кожна з яких містить розкрій у вигляді масиву листів.

Мутації реалізовані у функції «mutate». Вона приймає одну особину (розкрій), ймовірність мутації та розміри листа. Якщо випадкове число менше ймовірності мутації, виконується одна з чотирьох операцій: перестановка двох блоків на одному листі, зміна орієнтації блоку, переміщення блоку у випадкову допустиму область або перенесення блоку між листами. Кожна мутація застосовується лише якщо після неї розкрій залишається гільйотинним. Функція повертає новий масив листів.

Кросовер реалізовано у функції «crossover». Вона приймає дві батьківські особини, розміри листа та тип кросоверу, після чого викликає одну з чотирьох реалізованих схем схрещування: PMX, OX, одноточковий або арифметичний кросовер.

Одноточковий кросовер (функція «onePointCrossover()») працює шляхом вибору випадкової точки розділу генів у хромосомі. Гени до цієї точки беруться від першого батька, а після – від другого. Для кожного блоку зберігається інформація про позицію, орієнтацію та індекс листа. Після формування нового розкрою перевіряється його коректність.

Арифметичний кросовер (функція «arithmeticCrossover()») створює нові нащадки, координати яких обчислюються як зважене середнє між відповідними координатами батьків, а орієнтація та індекс листа вибираються випадково з одного з батьків. Після цього також виконується перевірка коректності розміщення.

У частково відображеному кросовері PMX (функція «pmxCrossover()») відбувається обмін генами хромосом між батьками з побудовою відображення для унікальності блоків. Для кожного блоку визначається його положення, орієнтація та індекс листа. Якщо після декількох спроб не вдається отримати коректний розкрій, повертаються батьківські рішення.

Порядковий кросовер OX (функція «orderCrossover()») формує нову послідовність блоків, копіюючи випадковий ген з одного з батьків, а решту блоків додає у порядку появи з іншого батька, уникаючи повторів. Далі блоки розподіляються по листах згідно з їх індексами, і перевіряється коректність розкрою.

Усі ці функції мають спільний підхід до перевірки результату: якщо новий розкрій не відповідає гільйотинним обмеженням, містить перетини або виходить за межі листа, повертаються батьківські рішення. Якщо ж розкрій коректний, повертається масив із двох нових особин. Така структура дозволяє ефективно комбінувати властивості батьків, зберігаючи при цьому всі технологічні обмеження задачі.

Оцінка якості розкрою здійснюється у функції «evaluateFitness()». Вона приймає масив листів і розміри листа. Функція обчислює сумарну площу розміщених блоків та площу всіх використаних листів, після чого повертає відношення цих величин. Таким чином, алгоритм прагне мінімізувати кількість листів і максимізувати заповненість кожного листа.

Відбір батьківських особин для кросоверу виконується через турнірний відбір у функціях «tournamentSelection()» та «selectParents()». Турнірний відбір приймає популяцію та розмір турніру, випадковим чином обирає декілька особин і повертає найкращу за значенням функції пристосованості. Функція вибору батьків повертає дві різні особини для кросоверу.

Для контролю збіжності алгоритму використовується лічильник спроб без покращення найкращого рішення, що дозволяє завершити роботу достроково, якщо прогрес відсутній протягом певної кількості поколінь.

Після завершення роботи алгоритму повертається найкраща знайдена особина у вигляді масиву листів, для кожного з яких вказано розміри, а також координати, розміри, номери та орієнтацію усіх розміщених на ньому блоків.

3.3.3 Мурашиний алгоритм

Алгоритм мурашиної колонії реалізовано в окремому модулі з назвою antColonyAlgorithm.js (додаток Б.1.3). Центральною функцією є «runACO()», яка організовує повний цикл роботи алгоритму для задачі гільйотинного розкрою. На вхід ця функція приймає масив блоків для розміщення, ширину та висоту листа, а також параметри алгоритму: кількість мурах, кількість ітерацій, коефіцієнти впливу феромонів і евристики (alpha, beta) та коефіцієнт випаровування феромонів. Всі ці параметри дозволяють гнучко налаштовувати поведінку алгоритму з урахуванням вимог конкретної задачі.

На першому етапі відбувається ініціалізація феромонів для всіх можливих позицій і орієнтацій кожного блоку на листі. Для цього координати дискретизуються з певним кроком, щоб обмежити кількість варіантів

розміщення. Далі запускається основний цикл, у якому кожна мураха будує власне рішення. Кожна мураха послідовно розміщує блоки на листах, обираючи позицію та орієнтацію з урахуванням поточного рівня феромонів і евристичної оцінки. Після вибору позиції для блоку перевіряється, чи не перетинається він з уже розміщеними блоками, а також чи дотримано гільйотинних обмежень. Для цього використовується допоміжна функція «areRectanglesIntersecting()», яка приймає два прямокутники (з координатами, розмірами та орієнтацією) і повертає булеве значення, що вказує на наявність перетину. Далі застосовується функція «isGuillotine()», яка перевіряє, чи кожен блок розміщено у лівому верхньому куті однієї з доступних областей, і чи після розміщення область ділиться лише на дві частини. Вона приймає масив блоків, ширину та висоту листа і повертає true або false залежно від дотримання гільйотинних умов.

Після завершення побудови рішень усіма мурахами відбувається етап оновлення феромонів. Спочатку всі феромони частково випаровуються, що реалізує механізм забування неефективних рішень. Далі підсилюються феромони на тих шляхах, які входять до складу найкращих знайдених рішень. Для цього використовується функція «calculateFitness()», яка оцінює якість розкрою як відношення сумарної площі розміщених блоків до площі всіх використаних листів. Вона приймає масив листів, ширину та висоту листа і повертає числове значення, що характеризує ефективність розміщення.

Після завершення всіх ітерацій функція «runACO()» повертає найкраще знайдене рішення у вигляді масиву листів, кожен з яких містить інформацію про розміщені блоки, їх координати, розміри та орієнтацію.

3.3.4 Симуляція відпалу

Алгоритм імітації відпалу реалізовано у модулі simulatedAnnealing.js (додаток Б.1.4). Центральною функцією є «runSimulatedAnnealing()», яка організовує повний цикл пошуку оптимального розкрою. На вхід ця функція

приймає масив блоків для розміщення, ширину та висоту листа, а також об’єкт з параметрами алгоритму: початкова температура, кінцева температура, коефіцієнт охолодження (alpha) та максимальна кількість ітерацій.

Початкове рішення генерується за допомогою функції «generateInitialSolution()», яка приймає масив блоків, ширину та висоту листа. Вона намагається послідовно розмістити всі блоки на листах, перебираючи можливі орієнтації та області, і повертає масив листів із розміщеними блоками. На кожній ітерації алгоритму створюється сусіднє рішення за допомогою функції «generateNeighbor()». Вона приймає поточний розкрій, ширину та висоту листа. Мутація може бути трьох типів: перестановка двох блоків на одному листі, зміна орієнтації одного з блоків, або переміщення блоку у випадкову допустиму область. Для кожної мутації перевіряється гільйотинність розміщення за допомогою функції «isGuillotine()», яка приймає масив блоків, ширину та висоту листа і повертає булеве значення, що вказує на коректність розміщення.

Для оцінки якості рішення використовується функція «calculateEnergy()», яка приймає масив листів, ширину та висоту листа. Вона обчислює сумарну площу розміщених блоків, ділить її на загальну площу всіх використаних листів і повертає від’ємне значення цього відношення.

Під час генерації сусідніх рішень для переміщення блоку у вільну область використовується функція «getFreeGuillotineAreas()». Вона приймає масив блоків, ширину та висоту листа і повертає масив усіх доступних для розміщення областей. Для перевірки перетинів між блоками застосовується функція «areRectanglesIntersecting()», яка приймає два блоки з координатами, розмірами та орієнтацією і повертає булеве значення, що вказує на наявність перетину.

Після завершення ітерацій функція «runSimulatedAnnealing()» повертає найкращий знайдений розкрій у вигляді масиву листів, кожен з яких містить інформацію про розміщені блоки, їх координати, розміри та орієнтацію.

3.3.5 Гібридний алгоритм

Гібридний алгоритм реалізовано у модулі hybridGeneticAlgorithm.js (додаток Б.1.5). Гібридний алгоритм розроблено з метою поєднання переваг евристичних та еволюційних підходів у задачі розкрою прямокутних блоків. Евристичний алгоритм BestFitDecreasing дозволяє швидко отримати якісні стартові рішення, однак він не гарантує знаходження глобального оптимуму, оскільки працює за жадібним принципом і може «застрягати» у локальних мінімумах. З іншого боку, генетичний алгоритм здатен досліджувати значно ширший простір можливих розв’язків, але для ефективної роботи йому необхідна різноманітна початкова популяція. Саме тому раціональнім є створення гібридного підходу, у якому початкова популяція формується за допомогою BestFitDecreasing, а подальший пошук оптимального розкрою здійснюється генетичним алгоритмом. Такий підхід дозволяє підвищити якість рішень, пришвидшити збіжність до оптимального результату та уникнути типових недоліків кожного з методів окремо.

Головна функція алгоритму «runHybridGeneticAlgorithm()» приймає параметри розміру популяції, ймовірності мутації, кількості поколінь, масиву блоків, розмірів листа та типу кросоверу. На першому етапі всередині цієї функції визначається допоміжна функція «createBestFitPopulation()», яка відповідає за створення початкової популяції. Для кожної особини у популяції відбувається випадкове перемішування масиву блоків, що дозволяє уникнути одноманітності рішень. Далі для кожного такого перемішаного набору блоків викликається метод «cutBlocks()» з модуля «BestFit», який повертає масив листів із розташованими на них блоками. Кожен лист трансформується у формат, придатний для подальшої роботи генетичного алгоритму: для кожного блоку зберігаються координати розташування, розміри, номер та орієнтація.

4 АНАЛІЗ ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ

4.1 Тестові набори та критерії ефективності роботи алгоритмів

Для оцінки ефективності запропонованого підходу до вирішення задачі гільйотинного розкрою були сформовані спеціальні тестові набори, які відображають різноманітність реальних виробничих задач. Кожен тестовий набір містить певну кількість блоків: 20, 40, 60, 80 та 100. Розміри блоків у кожному наборі підібрані таким чином, щоб забезпечити як однорідність, так і різноманітність – у наборах присутні як великі, так і малі блоки, а також блоки з різними співвідношеннями сторін. Це дозволяє перевірити, як алгоритми поводяться у випадках, коли блоки легко комбінуються на листі, і у складніших ситуаціях, коли ефективне розміщення ускладнюється через різні розміри та пропорції.

Така наповненість тестових даних дозволяє оцінити стійкість алгоритмів до різних типів вхідних даних (прості та складні конфігурації), різноманітність розмірів і форм блоків імітує реальні виробничі умови, де деталі можуть суттєво відрізнятися одна від одної. Також поступове збільшення кількості блоків у наборах дає змогу дослідити масштабованість алгоритмів, тобто, як змінюється їхня продуктивність при зростанні розміру задачі.

Для кожного тестового набору алгоритми запускалися по 5 разів, щоб усереднити результати та зменшити вплив випадкових факторів. Під час тестування фіксувалися такі показники: час виконання, кількість використаних листів стандартного розміру (300×300) та коефіцієнт ефективності розкрою. Час виконання характеризує швидкодію алгоритму та дозволяє оцінити його придатність для задач різної складності.

Кількість використаних листів відображає економію матеріалу, що є головним критерієм у виробництві. Коефіцієнт ефективності розкрою визначає, наскільки щільно блоки розміщені на листах, і розраховується як

сумарна площа всіх розміщених блоків, поділена на площу охоплюючого прямокутника та помножена на 100 поділене на кількість листів. Площа охоплюючого прямокутника визначається як площа найменшого прямокутника, що містить усі розміщені блоки на всіх листах.

Такий підхід дозволяє оцінити не лише заповненість окремих листів, а й загальну щільність розміщення блоків у межах усієї області розкрою. Чим вище цей показник, тим раціональніше використовується площа матеріалу.

4.2 Дослідження роботи генетичного алгоритму

Генетичні алгоритми (ГА) потребують ретельного підбору численних параметрів для досягнення максимальної ефективності. Розроблений модуль дозволяє гнучко задавати розмір популяції, ймовірність мутації, кількість поколінь, а також початкову популяцію і тип кроссоверу. Найкращі значення цих параметрів визначаються експериментально. Налаштування генетичного алгоритму було здійснене наступним чином:

- початкова популяція створюється випадковим чином, а саме для кожної особини генерується унікальний розкрій, де блоки розміщуються у вільних гільйотинних областях листа;
- кросовер у модулі реалізовано у кількох варіантах: порядковий (Order Crossover), частково відображений (PMX), одноточковий (One-point) та арифметичний;
- селекція батьківських особин здійснюється методом турнірного відбору з двома учасниками;
- використовуються чотири типи мутації: перестановка блоків на листі, зміна орієнтації, переміщення блоку у нову допустиму область або перенесення між листами. Ймовірність застосування кожного типу мутації становить 0.25;
- умовами зупинки є досягнення встановленої кількості ітерацій або 200 ітерацій без покращення найкращого знайденого рішення.

4.2.1 Дослідження впливу кількості ітерацій

У проведеному дослідженні було проаналізовано вплив кількості ітерацій генетичного алгоритму (ГА) на ефективність розв’язання задачі для різної кількості деталей (таблиця 4.1).

Таблиця 4.1 – Вплив кількості ітерацій на застосування ГА

Кількість ітерацій	Критерій ефективності	Кількість деталей				
		20	40	60	80	100
1000	Час, мс	301.31	791.69	1405.65	2070.48	3091.61
	Оцінка	40	36.94	47.39	53.39	43.43
	Кількість листів	1	2	2	2	3
2000	Час, мс	661,07	1545.59	2944.15	4432.73	6556.42
	Оцінка	44,45	41.48	46.20	53.96	43.69
	Кількість листів	1	1,8	2	2	3
3000	Час, мс	958.73	2586.94	4543.96	6835.71	37500,4
	Оцінка	41.96	47.13	41.51	54.61	45,45
	Кількість листів	1	1,6	2	2	3
4000	Час, мс	1435.77	3244.86	6238.01	9408.26	13321.07
	Оцінка	48.47	39.67	44.03	54.71	48.10
	Кількість листів	1	1.80	2	2	2,8
5000	Час, мс	1605.97	4064.81	7885.90	12011.47	16882.59
	Оцінка	49,68	49.47	41.28	56.87	52.00
	Кількість листів	1	1,6	2	2	2,6

З метою виявлення закономірностей зміни параметрів алгоритму при нарощуванні обчислювального ресурсу, було проведено серію експериментів

із кількістю ітерацій від 1000 до 5000 у кроках по 1000. Для кожної конфігурації вимірювалися три показники: час виконання, якість отриманого рішення (оцінка) та кількість використаних листів як умовного ресурсу. З результатів дослідження можна помітити, що збільшення кількості ітерацій призводить до стрімкого зростання часу виконання.

Наприклад, при фіксованій кількості деталей час обробки збільшується від 3091.61 мс при 1000 ітераціях до 16882.59 мс при 5000, що становить зростання більш ніж у п’ять разів. Така тенденція характерна для всіх значень кількості деталей, що свідчить про майже лінійну або навіть експоненціальну залежність часу від кількості ітерацій.

Оцінка розв’язку в більшості випадків також зростає, що є позитивною ознакою – збільшення кількості ітерацій дозволяє алгоритму знаходити кращі варіанти рішення. Найбільший приріст якості спостерігається при 40 деталях, де оцінка зростає з 36.94 до 49.47, що відповідає покращенню на 33.9%. Однак для 60 деталей спостерігається зниження оцінки з 47.39 до 41.28, що може бути наслідком випадкового потрапляння в локальний мінімум або нестабільної поведінки алгоритму при великій складності задачі.

Отримані результати свідчать про те, що підвищення кількості ітерацій не завжди гарантує покращення результату і вимагає додаткового аналізу з точки зору налаштувань ГА.

Кількість використаних листів залишається стабільною або зменшується при збільшенні кількості ітерацій. Наприклад, для 100 деталей вона знижується з 3 до 2.6. Це свідчить про оптимізацію ресурсоємності рішень, навіть за умови зростання обчислювальної складності. Така динаміка є позитивною, оскільки демонструє, що алгоритм, хоч і вимагає більше часу, проте дозволяє економити інші ресурси, які можуть бути критично важливими в практичних умовах. Таким чином, можна зробити висновок, що збільшення кількості ітерацій генетичного алгоритму загалом позитивно впливає на якість рішень і дозволяє знаходити оптимальні рашення, проте це супроводжується значним збільшенням часу виконання.

4.2.2 Дослідження впливу налаштувань кросоверу

У таблиці 4.2 наведено результати порівняння ефективності генетичного алгоритму (ГА) при використанні різних типів кросоверів – One-point, Arithmetic, OX та PMX – залежно від кількості деталей у задачі. Основними критеріями оцінки виступають час виконання (мс), якість отриманого рішення (оцінка) та кількість необхідних листів.

Таблиця 4.2 – Порівняння результатів застосування ГА з різними типами кросоверу

Кроссовер	Критерій ефективності	Кількість деталей				
		20	40	60	80	100
One-point	Час, мс	1559.57	4183.73	7898.16	11983.36	17616.3
	Оцінка	44.49	40.35	45.46	54.10	48.53
	Кількість листів	1	2	2	2	2.80
Arithmetic	Час, мс	1655.18	3318.82	5206.13	6981.40	10472.07
	Оцінка	46.53	44.44	44.64	54.70	43.74
	Кількість листів	1	1,8	2	2	3
OX	Час, мс	2846.50	9227.03	12695.42	19661.12	31994.69
	Оцінка	46.17	40.83	43.12	54.48	51.91
	Кількість листів	1	1,8	2	2	2.60
PMX	Час, мс	22633.05	44803.65	85396.8	125915.97	180799.42
	Оцінка	46.38	47.83	46.51	55.87	44.27
	Кількість листів	1	1.60	2	2	3

За даними результатів дослідження спостерігається тенденція до зростання часу виконання з кількістю деталей незалежно від типу кросовера. Найменший час у більшості випадків демонструє кросовер Arithmetic, зокрема при 100 деталях він виконується за 10472.07 мс, що на 40.6% швидше, ніж One-point (17616.3 мс), на 67.3% швидше, ніж OX (31994.69 мс), і в рази швидше, ніж PMX (180799.42 мс), який демонструє значне обчислювальне навантаження – його час виконання майже у 17.3 разів вищий, ніж у Arithmetic. Це свідчить про те, що PMX, хоча й може демонструвати прийнятні оцінки рішень, має дуже високу обчислювальну складність, що робить його менш доцільним для задач із великою кількістю деталей.

Найвищі значення оцінок рішень у більшості випадків демонструють кроссовери PMX та Arithmetic. Наприклад, при 80 деталях PMX має оцінку 55.87, що лише на 2.1% вище, ніж Arithmetic (54.70) та на 3.3% вище, ніж OX (54.48). У цьому випадку різниця в якості рішень між цими трьома кросоверами є незначною, однак витрати часу при використанні PMX-кроссоверу суттєво вищі. Це свідчить, що обирати PMX доцільно лише у випадках, коли навіть незначне покращення оцінки має критичне значення, а обчислювальні ресурси дозволяють це зробити.

One-point кросовер показує стабільну, хоча й не найвищу, якість рішень. Його оцінка зростає від 44.49 (20 деталей) до 48.53 (100 деталей), що свідчить про стійкість, хоча й поступову деградацію при збільшенні складності задачі. У порівнянні з Arithmetic, One-point при 100 деталях має оцінку, лише на 9.9% вищу, однак вимагає на 68.2% більше часу, що вказує на менш ефективне співвідношення між якістю рішень та обчислювальними витратами.

Кількість листів також показує незначні коливання між методами. Наприклад, при 100 деталях One-point потребує в середньому 2.80 листа, Arithmetic – 3, OX – 2, а PMX також 3. У цьому аспекті відмінності між методами є незначними, і не впливають суттєво на загальну ефективність вибору типу кросовера. Слід відзначити, що в окремих випадках – наприклад, при 40 деталях – Arithmetic і OX мають значення 1.8, що вказує на можливу

нестабільність результату або особливість вибірки. Таким чином, можна зробити висновок, що арифметичний кросовер є збалансованим рішенням баланс між якістю розв’язку та часом його отримання. Незначне зменшення якості (у межах 2–10% у порівнянні з найкращими результатами) компенсується значною економією часу, що є важливим для задач великого обсягу.

Кроссовер PMX забезпечує вищу якість, але має надмірно високу обчислювальну складність, що обмежує його практичне застосування. OX демонструє задовільну якість, проте потребує вдвічі більше часу, ніж Arithmetic. One-point – це компромісне рішення, яке може використовуватись для швидкої реалізації в задачах невеликого масштабу.

4.3 Дослідження роботи мурашиного алгоритму

Робота алгоритму мурашиної колонії значною мірою залежить від правильного налаштування його параметрів. Саме тому головним етапом дослідження є вибір оптимальних значень основних параметрів.

Для дослідження мурашиного алгоритму, були встановлені такі початкові умови експерименту:

- коефіцієнт випаровування феромону встановлено на рівні 0.5, щоб феромон поступово зменшував свій вплив, сприяючи пошуку нових варіантів розміщення;
- параметри α та β мають значення 1 та 2 відповідно, що налаштовує алгоритм на баланс між впливом феромону і евристичною інформацією про розміри блоків;
- умовами зупинки є досягнення встановленої кількості ітерацій або 200 ітерацій без покращення найкращого знайденого рішення.

Значення кількості ітерацій та мурах визначаються експериментальним шляхом, оскільки аналітичне обґрунтування цих параметрів є складним через стохастичну природу алгоритму. У дослідженні розглядалися такі варіанти:

- кількість ітерацій: 1000, 2000, 3000, 4000;
- кількість мурах: 100, 300, 500, 1000.

4.3.1 Дослідження впливу кількості ітерацій

У таблиці 4.3 представлено результати порівняння ефективності застосування алгоритму мурашиної колонії (АСО) для задачі розкрою з різною кількістю ітерацій (1000, 2000, 3000, 4000) та різною кількістю деталей (від 20 до 100). Оцінювались три основні показники: час виконання, якість розв’язку (оцінка) та кількість використаних листів.

Таблиця 4.3 – Порівняння результатів застосування АСО з різною кількістю ітерацій

Кількість ітерацій	Критерій ефективності	Кількість деталей				
		20	40	60	80	100
1000	Час, мс	1663.13	6047.98	13505.43	23031.00	33386.59
	Оцінка	40.63	64.87	44.30	55.30	44.28
	Кількість листів	1	1	2	2	3
2000	Час, мс	1601.92	5953.99	13694.91	22513.37	33245.01
	Оцінка	42.08	64.95	42.75	55.06	44.26
	Кількість листів	1	1	2	2	3
3000	Час, мс	1603.02	5952.39	13622.83	23414.68	33229.24
	Оцінка	48.32	65.62	40.83	53.90	42.43
	Кількість листів	1	1	2	2	3
4000	Час, мс	1591.90	13505.43	13741.65	23364.34	33304.91
	Оцінка	43.81	44.30	41.39	53.88	43.13
	Кількість листів	1	2	2	2	3

Час виконання алгоритму закономірно зростає зі збільшенням кількості деталей, що відповідає логіці збільшення обчислювальної складності задачі. Так, для 20 деталей час коливається в межах від 1591.90 мс до 1663.13 мс, що становить приблизно 4.5% різниці між найменшим і найбільшим значенням. Для 100 деталей час змінюється від 33229.24 мс (3000 ітерацій) до 33386.59 мс (1000 ітерацій), тобто відмінність становить лише близько 0.47%. Такі відхилення є незначними і свідчать про те, що збільшення кількості ітерацій від 1000 до 4000 не призводить до помітного погіршення часової ефективності. У деяких випадках, наприклад, для 20 або 40 деталей, час виконання при 4000 ітераціях менший, ніж при 1000 ітераціях, що вказує на стабільність та передбачуваність алгоритму.

Оцінка якості рішень демонструє незначні зміни у межах кількох відсотків при збільшенні ітерацій. Наприклад, для 60 деталей оцінка змінюється з 44.30 (1000 ітерацій) до 41.39 (4000 ітерацій), тобто показує зниження на близько 6.6%. У випадку 40 деталей спостерігається зростання з 64.87 до 65.62 (3000 ітерацій), але потім знову падіння до 44.30 (4000 ітерацій), що є значним погіршенням – на понад 31.1% порівняно з максимумом. Аналогічно, для 100 деталей оцінка знижується з 44.28 до 43.13, що становить приблизно 2.6% погіршення. У випадку 20 деталей ситуація зворотна – оцінка зростає з 40.63 (1000 ітерацій) до 43.81 (4000 ітерацій), що становить покращення на 7.8%. Отже, вплив кількості ітерацій на оцінку рішень є нестабільним: у деяких випадках спостерігається покращення, в інших – погіршення. Загальна тенденція свідчить, що після певного порогу збільшення кількості ітерацій не гарантує кращої якості рішень, а в окремих випадках навіть може призвести до зниження результатів.

Щодо кількості використаних листів, показник залишається постійним для кожного набору деталей незалежно від кількості ітерацій. Так, для 20 та 40 деталей завжди використовується 1 лист, для 60 і 80 – 2, а для 100 – 3. Це свідчить про те, що алгоритм досить ефективно і швидко знаходить оптимальну або близьку до оптимальної комбінацію розміщення, і подальше

збільшення ітерацій не дає додаткового виграшу в цьому аспекті. Загалом, аналіз результатів свідчить про те, що алгоритм ACO демонструє стабільну роботу вже при 1000–2000 ітераціях. Збільшення до 3000 або 4000 не дає системного покращення ані за часом, ані за якістю рішень. У деяких випадках спостерігаються покращення, однак вони не є сталими і суттєвими. Таким чином, для практичного застосування доцільним виглядає використання меншої кількості ітерацій, що дозволяє досягти балансу між часом виконання та якістю рішень.

4.3.2 Дослідження впливу кількості мурах

На основі таблиці 4.4, що демонструє результати використання алгоритму колонії мурах (ACO) при різній кількості мурах у популяції для задач з різною кількістю деталей, можна зробити аналіз ефективності методу. Основну увагу приділено трьом критеріям: часу виконання (мс), якості отриманого розв’язку (оцінка) та кількості листів.

Зі збільшенням кількості мурах спостерігається помітне зростання часу виконання алгоритму. Наприклад, при розв’язанні задачі зі 100 деталями час зростає від 32 057.98 мс (при 100 мурах) до 250 180.50 мс (при 1000 мурах), що відповідає збільшенню приблизно на 680%. Це вказує на значне навантаження на обчислювальні ресурси при зростанні популяції мурах.

Однак якість рішень покращується не так істотно. Наприклад, при 40 деталях оцінка при 100 мурах становить 65.02, а при 300 – 66.69, тобто покращення лише на 2.6%, тоді як час виконання зростає майже втричі (з 6092.32 до 16217.44 мс). У деяких випадках, навпаки, збільшення кількості мурах призводить до погіршення результату. Для задачі з 60 деталями оцінка змінюється з 44.30 (100 мурах) до 43.10 (300 мурах), тобто зменшується приблизно на 2.7%, при цьому витрати часу зростають у 2.74 разів. Це демонструє, що збільшення кількості мурах не завжди супроводжується покращенням результату і може бути нераціональним.

У задачах з невеликою кількістю деталей (наприклад, 20) ефективність розв’язків майже не змінюється або навіть знижується зі збільшенням кількості мурах. Наприклад, при переході від 300 до 500 мурах оцінка знижується з 49.92 до 42.18, тобто на 15.5%, при цьому час збільшується на 55%. Це свідчить про перенасичення системи надмірною кількістю агентів, що не покращує, а погіршує розв’язання простих задач.

Таблиця 4.4 – Порівняння результатів застосування АСО з різною кількістю мурах у популяції

Кількість мурах	Критерій ефективності	Кількість деталей				
		20	40	60	80	100
100	Час, мс	1633.71	6092.32	13459.86	23720.86	32057.98
	Оцінка	48.99	65.02	44.30	54.30	45.25
	Кількість листів	1	1	2	2	3
300	Час, мс	3877.79	16217.44	36892.30	60343.98	125928.61
	Оцінка	49.92	66.69	43.10	53.11	48.75
	Кількість листів	1	1	2	2	2.8
500	Час, мс	5998.74	24035.86	57389.28	92579.43	183357.19
	Оцінка	42.18	65.03	43.44	54.28	52.30
	Кількість листів	1	1	2	2	2.6
1000	Час, мс	11376.63	43255.34	105432.58	159512.42	250180.50
	Оцінка	37.85	61.42	40.55	53.00	47.67
	Кількість листів	1	1	2	2	2.80

Кількість листів залишається стабільною у межах одного значення або незначно змінюється при зростанні кількості мурах. Наприклад, при 100 деталях цей показник змінюється з 3 до 2.80 при переході від 100 до 1000 мурах, що не є суттєвою різницею і, ймовірно, пов'язане з особливостями повторного запуску алгоритму.

Таким чином, можна зробити висновок, що збільшення кількості мурах понад 300–500 одиниць не дає відчутного виграшу в якості розв'язків, але суттєво підвищує обчислювальні витрати. У ряді випадків покращення результату є незначним (до 2–5%), або навпаки – якість знижується, особливо в задачах меншої складності. Тому при практичному застосуванні ACO доцільно обмежувати кількість мурах у межах 300–500, що дозволяє забезпечити прийнятну якість результатів при помірному часі виконання. Застосування 1000 мурах виправдане лише для задач з великою кількістю деталей, де високий обчислювальний ресурс є допустимим.

4.4 Порівняння результатів роботи усіх алгоритмів

Після проведення серії експериментів із налаштування параметрів алгоритмів були відібрані найкращі конфігурації, які забезпечили найвищу ефективність роботи. На основі цих налаштувань було здійснено фінальне порівняння всіх алгоритмів з метою визначення їхньої продуктивності у різних умовах. Налаштування генетичного алгоритму:

- кількість ітерацій – 5000;
- тип кросоверу – Arithmetic;
- метод селекції – турнірний відбір (2 учасники);
- початкова популяція – випадкова;
- ймовірність мутації – рівномірно розподілена між чотирма типами мутацій (по 0.25);
- умова зупинки – 200 ітерацій без покращення або досягнення граничної кількості ітерацій.

Налаштування алгоритму мурашиної колонії (ACO):

- кількість ітерацій – 1000;
- кількість мурах – 300;
- коефіцієнт випаровування феромону – 0.5;
- параметри $\alpha = 1$, $\beta = 2$;
- умова зупинки – 200 ітерацій без покращення або досягнення граничної кількості ітерацій.

Налаштування алгоритму імітації відпалу:

- кількість ітерацій – 1000;
- початкова температура – 100;
- кінцева температура встановлена 1e-4;
- параметр alpha – 0.98.

Гібридний алгоритм використовує такі ж параметри як і звичайний генетичний алгоритм, за вийнятком початкової популяції, у якості якої використовується рішення BFD.

Жадібний алгоритм BFD (Best Fit Decreasing) без параметрів налаштування так як має фіксовану логіку. У таблиці 4.5 наведено порівняльний аналіз ефективності п’яти алгоритмів (BFD, Генетичний, ACO, Імітація відпалу, Гібридний) за трьома основними критеріями: час виконання (мс), якість рішення (оцінка) та кількість листів, при змінній кількості деталей (від 20 до 100). Результати показують як загальні тенденції, так і локальні відмінності у продуктивності кожного алгоритму, що дозволяє сформулювати важливі висновки для вибору оптимального підходу. Починаючи з найменшого об’єму завдань (20 деталей) можна помітити, що найшвидшим є алгоритм BFD (0.12 мс), натомість алгоритм мурашиної колонії витрачає майже 3878 мс, що в 32315 разів повільніше. У цьому ж випадку гібридний алгоритм демонструє високу якість результату (65.79), що на 10.98% краще за BFD (59.28), проте потребує значно більше часу – 1474.87 мс. Імітація відпалу тут демонструє баланс між швидкістю (1.02 мс) та якістю (48.68), хоча програє за оцінкою найкращим алгоритмам.

При зростанні кількості деталей до 40 зберігається тенденція: BFD залишається найшвидшим (0.08 мс), а мурашина колонія – найповільнішим (16217.44 мс). Водночас, саме мурашина колонія показує найвищу якість рішення – 66.69, що на 1.89% краще за BFD (65.40). Гібридний алгоритм з результатом 64.08 лише на 2.01% гірший за BFD, проте в 33800 разів повільніший.

Таблиця 4.5 – Порівняння результатів роботи всіх алгоритмів

Кількість деталей	Критерій ефективності	Алгоритм				
		BFD	Генетичний	Мурашина колонія	Симуляція відпалу	Гібридний
40	Час, мс	0.08	3318.82	16217.44	0.31	2703.93
	Оцінка	65.40	44.44	66.69	37.13	64.08
	Кількість листів	1	1,8	1	2	1
60	Час, мс	0.06	5206.13	36892.30	0.53	4235.27
	Оцінка	42.18	44.64	43.10	45.53	42.18
	Кількість листів	2	2	2	2	2
80	Час, мс	0.12	6981.40	60343.98	0.59	5505.68
	Оцінка	57.36	54.70	53.11	35.99	56.21
	Кількість листів	2	2	2	3	2
100	Час, мс	0.15	10472.07	125928.61	0.72	6907.31
	Оцінка	45.18	43.74	48.75	43.37	49.58
	Кількість листів	3	2.8	2.8	3	2.8

На прикладі 60 деталей помітно значне вирівнювання оцінок: усі алгоритми демонструють дуже близькі значення (від 42.18 до 45.53), що означає, що різниця в якості рішень між ними є незначною – не перевищує 8%. З точки зору кількості листів, усі алгоритми демонструють однакову ефективність (2 листи), що говорить про досягнення однакового рівня компактності розміщення.

При 80 деталях ГА демонструє баланс між якістю (56.21) та помірним часом (5505.68 мс), тоді як мурашина колонія має найнижчу оцінку (53.11) при найбільшому часі (60343.98 мс). BFD, хоч і залишається найшвидшим (0.12 мс), показує лише 57.36, що на 2.05% краще за гібридний. Тобто різниця у якості між цими двома методами незначна. Водночас алгоритм імітації відпалу хоч і швидкий (0.59 мс), значно поступається за оцінкою (35.99), що робить його менш придатним для цієї задачі при великому обсязі вхідних даних.

При найбільшому розмірі задачі BFD залишається надзвичайно швидким (0.15 мс), але якість його рішень (45.18) поступається мурашиній колонії (48.75) на 7.9%. Гібридний алгоритм показує оцінку 49.58 і за якістю перевищує BFD на 9.7%, при цьому витрачає значно більше часу (6907.31 мс). Щодо компактності розміщення – мурашина колонія, генетичний та гібридний алгоритми використовують у середньому 2.8 листа, а BFD – 3, тобто на 7.14% більше, що вказує на його дещо нижчу ефективність у щільному пакуванні при високому навантаженні.

Таким чином, можна зробити наступні важливі висновки. Алгоритм BFD демонструє найвищу швидкість, але його якість рішень у складніших випадках (від 60 деталей) поступово знижується відносно інших методів. Гібридний підхід демонструє хорошу збалансованість між часом і якістю, особливо при великих обсягах, де дає результати, близькі до найкращих. ACO найчастіше забезпечує найвищу якість, проте його час виконання є критично великим, що значно обмежує його практичну застосовність. Імітація відпалу показує хорошу швидкодію, але нестабільну якість. ГА демонструє загалом посередні результати – він не є лідером ані за часом, ані за оцінкою.

ВИСНОВКИ

У ході роботи було проведено детальний аналіз алгоритмів та методів для вирішення задачі прямокутного гільйотинного розкрою листового матеріалу. Розглянуто основні підходи до оптимізації, зокрема евристичний підхід зі стратегією Best Fit Decreasing, генетичні алгоритми, мурашині алгоритми та метод симуляції відпалу. Кожен із цих підходів продемонстрував свої переваги, зокрема здатність швидко знаходити наближені рішення при високій обчислювальній складності задачі.

Під час наукового дослідження було проаналізовано основні наукові публікації та досвід практичного використання сучасних алгоритмів оптимізації розкрою. Це дозволило систематизувати наявні знання та виокремити перспективні методи для реалізації у ході розробки програмного забезпечення, яке автоматизує процес розкрою. Застосування сучасних алгоритмів значно покращує показники точності та ефективності, зменшуючи кількість відходів та оптимізуючи використання сировини.

У практичній частині роботи було розроблено програмне забезпечення для автоматизації процесу розкрою листового матеріалу. Функціональність програми дозволяє здійснювати додавання нових деталей, видалення або зміну їх розмірів після внесення до системи. Всі введені деталі відображаються у спеціальному інтерфейсі для перегляду та корекції. Після завершення налаштувань програма надає можливість зберегти отримані результати у PDF-форматі. Крім того, перед початком обчислень користувач може обрати один із доступних алгоритмів розкрою для оптимального розташування деталей на листі. У результаті роботи програма демонструє наочну схему розкрою, яка відображає розміщення деталей на листі. Додатково програма виводить три показника ефективності, такі як коефіцієнт використання матеріалу, що відображає раціональність розкладки, час, необхідний для підбору оптимального варіанту, а також загальну кількість листів, які знадобилися для

розміщення всіх деталей. Вибір інструментів розробки, таких як Visual Studio Code та спеціалізовані бібліотеки, забезпечив високу продуктивність та зручність у роботі з алгоритмами.

Проведено комплексне тестування з використанням набору різних даних, що дало змогу оцінити точність і швидкість роботи програми. Результати експериментального дослідження підтвердили ефективність застосованих алгоритмів. Зокрема, рівневий алгоритм BFD демонструє найвищу швидкодію, проте поступається за ефективністю використання матеріалу. Генетичні алгоритми продемонстрували достатній рівень використання матеріалу, проте точність результатів знижується у складних умовах розкрою з великою кількістю обмежень. Мурашині алгоритми забезпечили швидкий пошук наближеного оптимуму та високу адаптивність, тоді як метод симуляції відпалу показав здатність уникати локальних екстремумів.

Гібридний алгоритм демонструє найвищу якість рішень на всіх рівнях складності задачі. У випадках, коли його ефективність виявляється нижчою за показники інших алгоритмів, відхилення в якості рішень не перевищує кількох відсотків. Натомість він значно стабільніший за інші алгоритми та забезпечує найкращу загальну якість при помірних обчислювальних витратах.

Запропоноване рішення забезпечує автоматизацію розкрою листового матеріалу з мінімізацією витрат сировини, що особливо важливо в умовах сучасного виробництва. Гнучкість налаштувань системи дозволяє адаптувати алгоритми під конкретні потреби, що значно розширює можливості її використання в промисловості. Таким чином, розроблена система дозволяє суттєво зменшити витрати на виробництво та підвищити загальну ефективність технологічного процесу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Dykhoff H. A typology of cutting and packing problems. *Evropean Journal of Operational Research*. 1990. № 44 (2). P. 145-159.
2. Лебедєв Б. К., Лебедєв О. Б., Лебедєва Є. М. Модернізований мурашиний алгоритм синтезу ідентифікованого дерева гільйтинного розрізу при плануванні СБІС. *Вісник Південного університету.* 2017. № 7. С. 15-28.
3. Beasley J. E. Algorithms for Unconstrained Two-Dimensional Guillotine Cutting. *The Journal of Operational Res. Society*. Vol. 36, No. 4. 1985. P. 297-306.
4. Косолап А. І., Кодола Г. М. Ефективний метод оптимізації в задачах лінійного розкрою матеріалів. *Математичне моделювання*. 2018. № 1. С. 12.
5. Delorme M., Iori M., Martello S. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*. 2016. Vol. 255, No. 1. P. 1-20.
6. Кононенко А. І., Іващенко Г. С. Аналіз методів вирішення задачі розкрою листового матеріалу. *Науковий простір: аналіз, сучасний стан, тренди та перспективи*: зб. тез доп. III всеукр. студент. наук. конф. (м. Київ, 16 черв. 2023 р.). Київ, 2023. С. 123-124.
7. Haessler R. W., Sweeney P. E. Cutting stock problems and solution procedures. *Evropean Journal of Operational Research*. 1992. P. 141-150.
8. Lodi A., Martello S., Vigo D. Recent advances on two-dimensional bin packing problems. *Discrete Applied Math*. 2002. P. 373-380.
9. Курейчик В. М. Генетичні алгоритми. Огляд та стан. *Новини штучного інтелекту*. 1998. № 3. С. 14-64.
10. Родзін С. І. Проектування самотестованих мікросхем із застосуванням генетичного пошуку. *Звістки ТРТУ*. 1997. № 3. С. 84-86.
11. Валеева А. Ф., Петунін А. А., Файзрахманов Р. І. Застосування конструктивних евристик у завданнях розкрою-упаковки. *Додаток до журналу «Інформаційні технології»*. 2006. № 11. С. 1-24.

12. Chopard B., Tomassini M. The Ant Colony Method. *An Introduction to Metaheuristics for Optimisation*. 2018. P. 81-96.

13. Bonnevay S., Aubertin P., Lazert T. A Simulated Annealing algorithm for real-world 2-D Cutting Stock Problem with Setup Cost. *11th Metaheuristics International Conference*. 2015. P. 1-10.

14. Parada V., Sepúlveda M., Solar M., Gómez A. Solution for the constrained guillotine cutting problem by simulated annealing. *Computers & Operations Research*. 1998. Vol. 25, No 1. P. 37-47.

15. Bortfeldt A., Gehring H. A hybrid genetic algorithm for the container loading problem. *European J. of Operational Research*. 2001. Vol. 131, No 1. P. 143-161. DOI: http://dx.doi.org/10.1016/S0377-2217(00)00055-2

16. Imahori S., Yagiura M. The best-fit heuristic for the rectangular strip packing problem: An efficient implementation and the worst-case approximation ratio. *Computers & Operations Research*. 2010. Vol. 37. P. 325-333.

17. Глибовець М. М., Гулаєва Н. М. Аналіз генетичних алгоритмів розв'язання задачі двовимірної ортогональної упаковки прямокутних об'єктів у напівнескінченну смугу. *Проблеми програмування*. 2016. № 4. С. 104-116.

18. Валеева А. Ф., Петунін А. А., Файзрахманов Р. І. Застосування конструктивної метаевристики «мурашина колонія» до задачі гільйотинного прямокутного розкрою. *Вісник Башкірського університету*. 2007. С. 12-14.

19. Петунін А. А., Полєвов А. В., Куреннов Д. В. Про один підхід до вирішення завдань розкрою-упаковки. *Конструювання та технологія виготовлення машин:* збірник наукових праць. 2005. С. 212-216.

20. Орлов А. Н., Курейчик В. В., Кудрякова Т. Ю. Комбінований алгоритм розв'язання задачі прямокутного розкрою. *Праці Конгресу з інтелектуальних систем та інформаційних технологій*. 2015. С. 212-217.

21. Duan S., Jiang S., Dai H., Wang L., He Z. The applications of hybrid approach combining exact method and evolutionary algorithm in combinatorial optimization. *Journal of Computational Design and Engineering*. 2023. Vol. 10, No 3. P. 934-946. DOI: http://dx.doi.org/10.1093/jcde/qwad029

22. Валіахметова Ю. І., Телицький С. В. Застосування систем автоматизованого проектування карт розкрою в суднобудуванні. *Вісник ВДТУ*. 2012. № 6. С. 38-43.

23. Whitwell G. Novel Heuristic and Metaheuristic Approaches to Cutting and Packing. *Univ of Nottingham, School of Computer Science and Information Technology*. 2004. P. 68-71.

24. Blum C., Roli A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computational Surveys*. 2003. Vol. 35, No 3. P. 268-308.

25. Talbi E.-G. Metaheuristics: From Design to Implementation. Hoboken, NJ : John Wiley & Sons. 2009. 593 p.

26. D'Addona D. M., Teti R. Genetic Algorithm-based Optimization of Cutting Parameters in Turning Processes. *Procedia CIRP*. 2013. Vol. 7. P. 323-328.

27. Gaber A., Elshaer R., Khater M. An Ant Colony Optimization Heuristic for Solving the Two-Dimensional Level Packing Problems. *Proc. of IEOM Conf.* Bandung, 2018. P. 712-722.

28. Levine J., Ducatelle F. Ant colony optimization and local search for bin packing and cutting stock problems. *Journal of the Operational Research Society*. 2004. Vol. 55. P. 705-716.

29. Delahaye D., Chaimatanan S., Mongeau M. Simulated annealing: From basics to applications. *Handbook of Metaheuristics*. Springer, 2019. P. 1-35.

30. Іващенко Г. С., Кононенко А. І., Тимошенко Д. О. Метаевристичні методи вирішення задачі гільйотинного розкрою. *Збірник наукових праць «Системи управління, навігації та зв'язку»*. 2025. № 1(79). С. 91-95. DOI: 10.26906/SUNZ.2025.1.91-95